Exploiting system level heterogeneity to improve the performance of a GeoStatistics multi-phase task-based application

Lucas Leandro Nesi, Arnaud Legrand, Lucas Mello Schnorr

Exploiting system level heterogeneity to improve the performance of a GeoStatistics multi-phase task-based application

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

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

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

ABSTRACT

Heterogeneity is part of HPC infrastructures, not only at the intra-node but at the system level. Applications with multiple phases with distinct resource necessities can take advantage of this inter-node heterogeneity to improve performance and reduce resource idleness. Such an application is ExaGeoStat, a task-based machine learning framework specifically designed for geostatistics data. This work presents strategies to efficiently distribute multi-phase applications in system-level heterogeneous resources. We both (1) improve application phase overlap by optimizing runtime and scheduling decisions and (2) compute the optimal distribution for all the phases using a linear program leveraging node heterogeneity while limiting communication overhead. The performance gains of our phase overlap improvements are between 36% and 50% compared to the original base synchronous and homogeneous execution. We show that by adding some slow nodes to a homogeneous set of fast nodes, we can improve the performance by another 25% compared to a standard block-cyclic distribution, thereby harnessing any machine.

CCS CONCEPTS

• **Computing methodologies** → **Massively parallel algorithms**; **Distributed computing methodologies**.

KEYWORDS

Task-Based, Scheduling, Partitioning, Load Balancing

ACM Reference Format:
Lucas Leandro Nesi, Arnaud Legrand, and Lucas Mello Schnorr. 2021. Exploiting system level heterogeneity to improve the performance of a GeoStatistics multi-phase task-based application. In *50th International Conference on Parallel Processing (ICPP '21), August 9–12, 2021, Lemont, IL, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3472456.3472516

1 INTRODUCTION

High-performance computing widely employs intra-node heterogeneity [9]. Most of the supercomputers at TOP500 [10] have at least one type of accelerator that applications can utilize. This heterogeneity enables the acceleration of specific kernels while preserving the other resources (CPUs) to the remaining computational operations. Nevertheless, heterogeneity is also further present at a system level, where different computational node types exist. Supercomputers manifest that heterogeneity usually by dividing the nodes into distinct partitions. Examples of such supercomputer are the French Jean Zay[1], the Swiss Piz Daint[2], and the Brazilian SDumont[3]. Common causes for these diverse nodes are: (i) the design, when the infrastructure possesses such configurations to target diverse workloads; (ii) the financial limitations, when only a subset of nodes receives accelerators because of budget constraints; and (iii) the natural infrastructure upgrades over time. Furthermore, Cloud is another example of system-wide heterogeneity, as the major service providers offer a vast number of virtual machine types that the customers can freely combine [19].

Although most applications suffer from system-wide heterogeneity, some applications can leverage such heterogeneity to enhance their performance. Indeed, many applications have different phases, each one having different computational needs. For example, phases comprising data input and generation are generally more suited to CPUs, while compute-intensive phases, such as classical linear algebra kernels, can efficiently exploit accelerators. An application to correctly use all available resources requires certain freedom to execute the phases concurrently. The challenges to making phase overlapping possible include programming and algorithmic difficulties. From the programmer's perspective, phase overlapping is usually hard to obtain in traditional bulk synchronous parallel applications. The complexity of programming asynchronous and overlapping phases in strongly coupled MPI or MPI+X is high. Even after the phase overlapping gets implemented, the application would still face the algorithmic challenge to find an efficient distribution for the different available nodes. Combining these challenges makes most applications miss an enormous opportunity to use system-level heterogeneousness since the distinct computational demand of phases makes system heterogeneity a natural choice to improve the load partitioning. This work relies on the task-based paradigm, using a runtime, to overcome such difficulties. A Direct Acyclic Graph (DAG), where nodes are tasks and edges are data dependencies, represents the application. The runtime schedule ready tasks while trying to minimize the total makespan. The application developer still needs to give many hints to the runtime to assist

ICPP '21, August 9–12, 2021, Lemont, IL, USA

ACM ISBN 978-1-4503-9068-2/21/08.
https://doi.org/10.1145/3472456.3472516

[1] https://www.top500.org/site/50403/
[2] https://www.top500.org/site/50422/
[3] https://www.top500.org/site/50576/

it during execution. Examples of such runtimes are ParSEC [7], OmpSs [11], and StarPU [3], this last used in this work.

This work focuses on the multi-phase task-based application ExaGeoStat [1], a machine learning framework for computational geostatistics. It uses the task-based dense linear algebra Chameleon solver [2] and the StarPU runtime. ExaGeoStat relies on the Gaussian process and optimizes the likelihood of spatial data, enabling the prediction of missing points. This iterative optimization comprises phases, including generating a positive triangular matrix, a Cholesky decomposition, and a triangular solve. These phases require different computational needs and could overlap among them if the DAG is well structured, the runtime has the right hints, and it uses an adequate data distribution.

In this paper, we study and propose strategies for distributing this multi-phase application over heterogeneous system-level resources. The main contributions of this paper are: (a) Improvements to the task-based asynchronous execution of the ExaGeoStat application that includes a rewriting/adaptation of the solve algorithm, new task priorities to guarantee a smoother phase transition, and identification and addition of hints to help the runtime to take good scheduling decisions; (b) A methodology for generating a heterogeneous system-level static distribution that considers the requirements of the most significant computational phases; (c) A strategy for producing two optimal load distribution for each phase while minimizing the redistribution communication overhead; (d) a comprehensive performance evaluation of the proposed strategies.

Section 2 presents the concepts of the task-based paradigm and the ExaGeoStat application. Section 3 discusses the related work of heterogeneous distributions, asynchronous multi-phases, and multiple distributions. Section 4 explains the strategies for multi-phase distribution in system-level heterogeneous resources. Section 5 relates the performance evaluation of the proposed methods. Finally, Section 6 concludes the paper with a discussion and future remarks.

2 TASK-BASED EXAGEOSTAT APPLICATION

ExaGeoStat is a machine learning application for GeoStatistics data that relies on the Gaussian process and can predict missing observations in data. It is developed using the task-based programming paradigm and relies on the Chameleon task-based dense linear solver and StarPU. In this paradigm, instead of explicitly indicating where and when each computation and communication should occur, the programmer declares discrete computation operations as tasks, expressing the execution flow using data dependencies among these tasks. The final structure is a Direct Acyclic Graph (DAG), where nodes are tasks and edges data dependencies. A task-based runtime is then responsible for dynamically scheduling these tasks over the desired system. ExaGeoStat can rely on several runtimes like ParSEC or StarPU. In this work, we use StarPU, a runtime for heterogeneous (CPU/GPU) systems that can run over multiple nodes using its MPI extension. In StarPU, the scheduling of intra-node tasks is dynamic and managed by one out of the many scheduling algorithms available. The division of data among the nodes is the developer's responsibility and generally follows a static distribution. Consequently, each node will own portions of the data, and StarPU will place tasks on nodes that hold the data they write. This static distribution is not a limitation for flexibility or freedom in task placement since the application can, during execution, change the ownership of memory blocks.

ExaGeoStat interpolates spatial data (X, Z), where X corresponds to the measurement locations and Z corresponds to the measurements, with Gaussian process whose smoothness and scale are controlled by a set of parameters θ that requires adjustments to the data in a Bayesian way. Therefore the application iteratively optimizes the log-likelihood of θ through Equation (1).

$$l(\theta) = -\frac{N}{2}\log(2\pi) - \frac{1}{2}\log|\Sigma_\theta| - \frac{1}{2}Z^T\Sigma_\theta^{-1}Z, \quad (1)$$

where Σ_θ is an $N \times N$ covariance matrix built from X representing the similarity between measurement locations, which is computed through a covariance function K_θ (i.e., $\Sigma_\theta[m,n] = K_\theta(X_m, X_n)$). Note that although Machine Learning commonly uses the squared exponential (Gaussian) covariance function, the Matérn covariance function is more appropriate for geostatistics data which can be relatively rough.

Therefore, this optimization requires computing a large dense symmetric and positive definite matrix Σ_θ at each optimization iteration. This matrix is then decomposed and solved using Cholesky factorization and used through a triangular solve and a dot product to compute the last term of Equation (1). The determinant of the matrix is easily computed from the diagonal blocks of the factorization. One iteration of ExaGeoStat has thus 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 presents the DAG which corresponds to one iteration. One may think that all the distributions and scheduling decisions should be designed around the most computationally intensive phase, the Cholesky factorization. However, the phases have different computational needs with varying affinities for accelerators. While the primary kernel of the Cholesky factorization, `dgemm`, is well suited to GPUs, the Matérn function used in the generation is only available through costly CPU implementation at the moment. Consequently [14], for small and medium cases, the time needed for covariance matrix generation often dominates the Cholesky factorization, even with one order of complexity difference.

In the public ExaGeoStat[4] repository, two execution options are available: (1) Synchronous, with a synchronization point between every phase, and (2) Asynchronous, where the synchronizations between factorization/determinant and solve/dot product disappear. Finally, ExaGeoStat uses the Chameleon library's data distributions, which is the traditional block-cyclic distribution for homogeneous nodes of ScaLAPACK [6].

The ExaGeoStat authors reported excellent performance and scalability results with homogeneous multi-core systems [1]. As we will show in this work, when considering hybrid nodes (CPU+GPU), the Matérn covariance function raises severe load-balancing issues. Indeed, some phases better employ GPUs while others better utilize CPUs. Thus, it is natural to exploit system-level heterogeneity by

[4] https://github.com/ecrc/exageostat

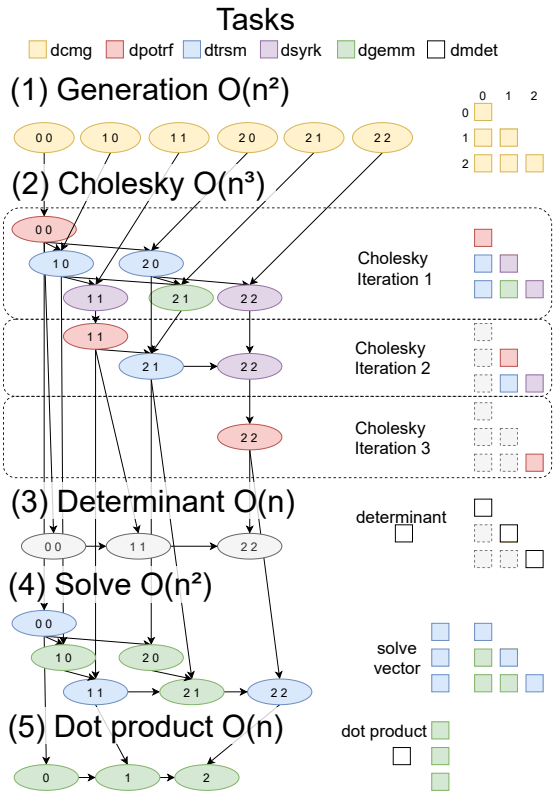

Figure 1: ExaGeoStat iteration DAG for $N = 3$.

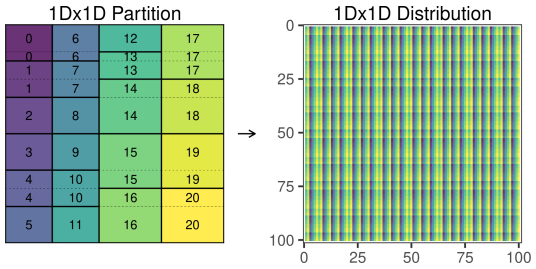

Figure 2: The 1D-1D column-based partition on the left and the distribution generated from shuffling on the right.

mixing different node types to process each phase as efficiently as possible. However, a heterogeneous set of computing nodes will require different distribution strategies than the traditional block-cyclic distribution. Such heterogeneity would even imply different data distributions for each phase, with data redistribution.

3 RELATED WORK

Linear algebra solvers have to deal with factorization algorithms that update a fraction of the matrix that decreases along with the iterations. To ensure a smooth load balancing over iterations and minimize communications with homogeneous nodes, the 2D block-cyclic distribution is one of the most employed strategies [6]. With system-level heterogeneity, the distributions should respect the processing power of each node to balance the load correctly [16]. The equivalent of 2D block-cyclic distributions can be obtained by (1) building a partition of the matrix in rectangles of predefined areas corresponding to node processing powers while minimizing the number of communications (e.g., with the `col-peri-sum` algorithm [4]), as shown in the left of Figure 2, and (2) shuffling rows and columns to ensure a smooth progression of iterations using, for example, the 1D-1D algorithm [5], as shown in the right of Figure 2. The resulting matrix distributions are asymptotically optimal and have been recently implemented [17] in the Chameleon library on which ExaGeoStat relies, providing a solid basis for this work.

Data redistribution is also an old problem since Prylli et al. [18] proposed algorithms to reduce the scheduling redistribution cost between block-cyclic distributions, possibly with different block sizes, used in different application phases. With MPI, such data redistribution typically occurred synchronously between the different phases, which is often inefficient. This is no longer an option with the scale of supercomputers, hence the rising popularity for the data flow algorithms that minimize the number of synchronization points [9]. Linear algebra solvers leverage this asynchronous capability to overlap different tasks and iterations and exploit more parallelism [2, 13]. A possible multi-phase strategy consists of finding the best distribution per phase while minimizing the overall communication cost when changing the application's phases, which can be challenging. Recent work [15] points out that even in simple situations, the target distribution may present many possible permutations (corresponding to similar nodes) and find the ideal distribution (of the many possible permutations) while minimizing both the computation and the total communication cost is NP-hard.

4 MULTI-PHASE PARTITIONING IN HETEROGENEOUS CLUSTERS

This Section presents our contribution that leverages the flexibility of matrix rectangle partitions to build efficient data distributions with low overhead redistribution for the ExaGeoStat application. We first illustrate the application behavior in Section 4.1. Then we present a set of scheduling optimizations that radically improve the phase overlap in Section 4.2. We present in Section 4.3 how to compute an efficient load distribution of all phases at once over a heterogeneous set of nodes. Finally, we present in Section 4.4 how to construct a data distribution for the generation phase that minimizes communications when shifting to the Cholesky distribution.

4.1 Characterizing the Optimization Iteration

The visualization of the application execution behavior can aid in the performance analysis and characterization. Figure 3 presents three panels from the performance analysis tool StarVZ [12] for one iteration of the synchronous version of ExaGeoStat. The X-axis of all panels is the time in milliseconds. The panel in the middle is a Gantt chart of the Node utilization aggregated by the resource type per node (for example, CPU 0 is the aggregated utilization in % of all CPUs in node 0). The rightmost number is the execution makespan. The upper panel is the iteration plot: it depicts the iteration of the Cholesky algorithm on the Y-axis, with the left-black line representing the beginning of the iteration and the right-black line representing the end. This plot allows an understanding of

how the factorization unfolds over time. The generation phase is mapped to iteration 0, and post-Cholesky operations to iteration N (Size of the matrix). The last panel depicts the resource memory utilization per memory node (Each NUMA node and each GPU has a memory node). The three main phases of ExaGeoStat are distinct and visible: the *generation* phase (A annotation) with the yellow dcmg tasks that only run on CPUs; the Cholesky factorization (B) whose predominant dgemm tasks are in green; and the post-factorization operations (determinant, solve and dot product; at C) whose predominant operations are the dgemm tasks from the solve. The resource usage is relatively low, especially at the beginning (where only the CPU cores work) and toward the end (where there is not enough work for all nodes). A large amount of parallelism stemming from the DAG should consolidate most green tasks to the left and reduce idle time. The following subsection will highlight problems and propose strategies to improve phases overlap.

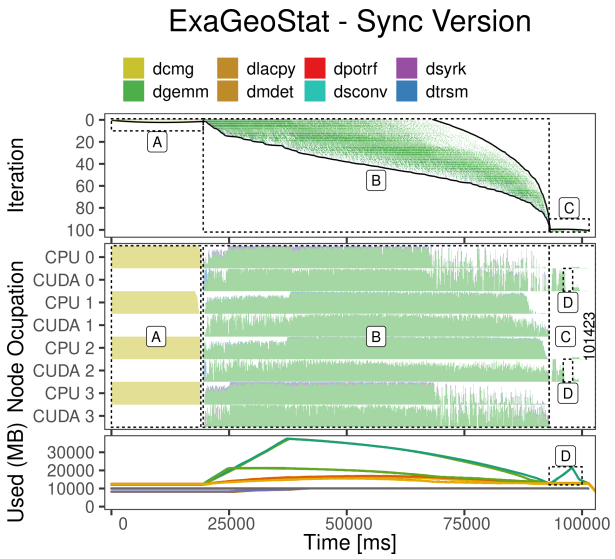

Figure 3: Iteration, Node occupation, and Memory panels for the synchronous version of the ExaGeoStat iteration.

4.2 Improving Application's Phases Overlap

The tasks from different phases do not overlap in the synchronous version, and several resource usage opportunities are lost. For example, the factorization could start when the upper part of the matrix is generated and exploit the GPUs while the generation proceeds. The first optimization is to remove the synchronization points between all inner-operation phases and make a **fully asynchronous** execution, letting StarPU handle the task order and application flow. However, to guarantee a smooth transition between phases, the application should provide hints to the runtime and control collateral behaviors like massive communication or allocations costs.

One of these collateral performance degradations originates from the triangular solve algorithm. The original Chameleon solve performs its dgemm operations on the node that owns the solution vector (the right-hand vector). Consequently, many matrix blocks are moved between nodes to complete a simple dgemv operation. This movement requires extra communication and extra data allocation on the nodes and appears in Figure 3 (D annotation). The combination of heavy communication and allocation induces idle time during the solve step, which can be significant depending on the machines' configuration and matrix size. We replaced the Chameleon solve algorithm with a **local solve algorithm** to settle these problems (Algorithm 1). Instead of performing the dgemv operation on nodes that own the **Z** vector, this algorithm performs and accumulates the dgemv outputs in a local vector **G** of each node. This extra variable breaks some dependencies and leaves the runtime the opportunity to move only **G** to the node that owns the respective **Z** block and reduce it with a dgeadd.

Algorithm 1: The new solve algorithm.

```
for k = 0 up to N do
    dtrsv(M(k, k), Z(k))
    for m = k + 1 up to N do
        dgemv(M(m, k), Z(k), G(m, node of M(m, k)))
    end
    foreach updated G(k+1, n) do
        dgeadd(G(k+1, n), Z(k+1))
    end
end
```

Another exciting aspect when overlapping the phases is memory consumption. We perform four **memory optimizations** by tweaking StarPU options and modifying the runtime and the application to obtain acceptable performance. We remove the RAM allocation from the task submission function and let StarPU treat it as any other memory request. We enable the chunk memory cache system of StarPU for the RAM. This option means that StarPU can reuse memory blocks between phases and optimization iterations. We disallow slow allocation of memory by GPU workers as the CUDA allocation for pinned host memory can be particularly slow and reduce the performance throughput of GPU workers. Finally, we pre-allocate some memory chunks before the execution so that the first optimization iteration can also benefit from the cache.

The runtime considers task priorities to decide how to advance the execution. StarPU prefers higher priority tasks to execute first. The original implementation only prioritized the Chameleon Cholesky factorization tasks, which assumed values from $2N$ to $-N$ with an order following roughly the anti-diagonal. If a task has no specified priority, StarPU considers it to be equal to 0. All generation tasks, specified in ExaGeoStat, and all triangular solve tasks, specified in Chameleon, thereby had a 0 priority, conflicting with the factorization task priorities. We propose **new priority equations** for all phases considering the application DAG and an order inspired by the critical path with a unit execution cost (i.e., starting from the last tasks and going backward to the very first generation tasks) to guarantee a smoother transition among phases. Equations (2) to (11) present these new priorities. Essentially, the base for the priorities originated from the Cholesky DAG. The generation is aligned with the first iteration ($k = 0$) of the dgemm factorization and divides the reduction component (coordinates) by 2 to accelerate it. As the determinant and dot product tasks are leaves of the DAG and do not require any particular order, they receive a priority of 0.

$$[\text{Generation}]\ \text{dcmg} = 3N - \tfrac{n+m}{2} \quad (2)$$
$$[\text{Cholesky}]\ \text{dpotrf} = 3\,(N-k) \quad (3)$$
$$[\text{Cholesky}]\ \text{dtrsm} = 3\,(N-k) - (m-k) \quad (4)$$
$$[\text{Cholesky}]\ \text{dsyrk} = 3\,(N-k) - 2\,(n-k) \quad (5)$$
$$[\text{Cholesky}]\ \text{dgemm} = 3\,(N-k) - (n-k) - (m-k) \quad (6)$$
$$[\text{Solve}]\ \text{dtrsm} = 2\,(N-k) \quad (7)$$
$$[\text{Solve}]\ \text{dgemm} = 2\,(N-k) - m \quad (8)$$
$$[\text{Solve}]\ \text{dgeadd} = 2\,(N-k) \quad (9)$$
$$[\text{Determinant}]\ \text{dmdet} = 0 \quad (10)$$
$$[\text{Dot}]\ \text{dgemm} = 0 \quad (11)$$

In practice, despite these new priorities, a scheduling artifact may arise and lead to earlier scheduling and execution of low-priority tasks, compared to other higher priority tasks submitted when resources are idle. This situation happens because StarPU, like any other scheduler, is incapable of foreseeing the future. So when a low-priority task is submitted and some resources are available, it can start directly. However, if higher priority tasks are submitted right after, the execution may not respect the order of priorities at that moment. To reduce this artifact, we modified the **submission order** of the generation to match the priorities.

Another possible runtime artifact is the delay of very high-priority tasks, like `dpotrf`, that can only execute on CPUs. When these tasks become available, all resources may be working on very long tasks, like the generation ones (`dcmg`). Although the runtime knows that one particular task should execute right now, this high-priority task must wait until a resource becomes available without preemption procedures. The critical path of Cholesky, made of `dpotrf` tasks, is what releases many `dgemm` tasks for already generated blocks. This critical path must advance as fast as possible to release these tasks and populate powerful resources like GPUs. We dedicate a core for non-generation tasks so that the critical path can advance faster. Because usually StarPU reserves a core for the main application thread, we **oversubscribe** a worker to this thread. This configuration keeps the same number of cores for the generation but adds a new worker to these essential tasks.

With all these six phase-overlap optimizations, we expect that all phases will start as soon as possible and smoothly overlap with each other. A fine-tuning of this asynchronous behavior is a critical optimization that precludes any investigation of heterogeneous distributions that the next Section discusses.

4.3 Balancing the Load Over Several Phases

In infrastructures with system-level heterogeneity, groups of nodes with different computational power co-exist. To use the heterogeneous distribution algorithms mentioned in Section 3, it is necessary to compute the processing power for each group. However, this processing power depends on the phase, and since phases overlap, computing the optimal load balance and the corresponding data distribution can be quite complicated. For example, in ExaGeoStat, the two main phases are generation and factorization, and the generation cannot use GPUs. Suppose the factorization distribution considers only the total amount of factorization tasks. Since the two phases overlap, the relative GPU power considered in the distributions would be undersized, as the CPUs are busy processing the generation tasks. The resulting load balancing could also be unequal because the GPUs can start processing factorization tasks earlier. During the generation phase, the GPUs could process other tasks, anticipating work, which means that nodes with faster GPUs should have their relative powers for the distributions increased.

We use a linear programming model to correctly estimate the nodes groups' powers in the ExaGeoStat generation and factorization interaction. Since phases overlap and are dependent on each other, the key idea is to divide the phases into virtual steps and to bound the duration of these virtual steps by resource usage. We can easily extend the model to similar multi-phase applications where phases have different resource power needs.

The linear programming model uses the following notations. t corresponds to a task type (e.g., `dgemm` or `dcmg`). s corresponds to a virtual step that represents a set of independent tasks. In ExaGeoStat, we decide that each *generation step* will correspond to an anti-diagonal in the matrix (all the m and n such as $(m+n)/2 = s$) which corresponds to the priorities in Equations of the previous Section. When referring to *factorization step* s, we consider all the factorization tasks directly dependent on blocks generated at generation step s. The linear programming model requires $Q_{s,t}$, the total number of tasks of type t at step s. r corresponds to a resource group, for example, all CPUs of a homogeneous set of nodes. $w_{t,r}$ denotes the duration that task of type t takes at resource group r (we set $w_{t,r} = \infty$ whenever a given task type t cannot run on a given resource r). Finally, we will denote by $\mathcal{R}$, $\mathcal{S}$, and $\mathcal{T}$ the sets of possible resources, steps, and task types.

We aim at computing $\alpha_{s,t,r}$, the number of tasks of type t of step s placed at resource group r. To account for dependencies, we also need to introduce the variables F_s and G_s, which represent the ending times of each factorization and generation steps s. We can approximate the behavior of ExaGeoStat by the following linear program, where $\alpha_{s,t,r}$, F_s, and G_s are all positive variables:

$$\text{Minimize} \sum_{s \in \mathcal{S}} (G_s + F_s)\ s.t.: \quad (12)$$
$$\forall t \in \mathcal{T}, \forall s \in \mathcal{S} : \sum_{r \in \mathcal{R}} \alpha_{s,t,r} = Q_{s,t} \quad (13)$$
$$\forall s > 1, \forall r \in \mathcal{R} : G_{s-1} + \alpha_{s,\text{dcmg},r} w_{\text{dcmg},r} \leq G_s \quad (14)$$
$$\forall s \in \mathcal{S}, \forall r \in \mathcal{R} : G_s + \sum_{t \neq \text{dcmg}} \alpha_{s,t,r} w_{t,r} \leq F_s \quad (15)$$
$$\forall s > 1, \forall r \in \mathcal{R} : F_{s-1} + \sum_{t \neq \text{dcmg}} \alpha_{s,t,r} w_{t,r} \leq F_s \quad (16)$$
$$\forall r \in \mathcal{R}, \forall s \in \mathcal{S} : \sum_{z \leq s, t \in T} \alpha_{z,t,r} w_{t,r} \leq F_s \quad (17)$$
$$\min_{r \in \mathcal{R}} (w_{\text{dcmg},r}) \leq G_1 \quad (18)$$

The goal is to minimize the ending times of all generation and factorization ending times, in particular, the final factorization ending time F_N that is the application makespan. However, the objective function for the linear program in Equation (12) is more complicated. If the LP used a simple loose objective function like F_N, the ending of the previous factorization steps F_s for $s < N$ could

appear as late as possible when the generation phase is the bottleneck, which is undesirable. Instead, we minimize the sum of all G_s and F_s to drive a simultaneous minimization of steps. Giving more weight to F_N or adopting a recursive minimization fails to bring any practical improvement compared to our simple sum.

The constraints for the linear program are the following. Equation (13) is a conservation equation that ensures that all the tasks are distributed over the resources. In our approximation, all generation (resp. factorization) steps happen one after the other, so Equation (14) enforces that the end of a generation step cannot happen earlier than the previous generation step plus all the associated tasks. The following two constraints express the dependency between generation and factorization steps. Equation (15) ensures that one factorization step cannot end earlier than the generation step end plus all the related factorization tasks. Equation (16) is similar to Equation (14) and enforces that the end of the factorization of blocks of step s cannot happen earlier than the end of the previous factorization step plus all related factorization tasks. This rule is stricter in the model than in real life, as one factorization step could have many iterations that run concurrently with another factorization step. However, it guarantees the correct progression between factorization without penalizing it too much. Equation (17) ensures that resources do not process two tasks at the same time by making the factorization end at a step s be at least the sum of all previous tasks on the resource. Finally, Equation (18) is a simple rule to approximate the beginning of the execution. Because we are using linear programming with rational variables, tasks can be "split" in the model. This Equation guarantees that the first generation step cannot be faster than its best resource implementation.

Despite the number of constraints, less than a second is necessary to solve it. Even if it fails to represent a true lower bound of the makespan because Equation (16) is too strict, it provides an excellent approximation. Besides, the output α of the LP is a guideline to decide how many tasks each phase should execute on every resource group, thereby estimating the relative power that the heterogeneous distribution algorithms should use for each phase.

4.4 Multi-Partitioning for distinct phases

The perfect distribution for each phase of ExaGeoStat is different because each has different computational needs and different tasks. Also, the number of data blocks allotted to each node can vary wildly between phases' distributions. While the main task of factorization is the `dgemm` that GPUs can accelerate, the only task of the generation, `dcmg`, is only implemented on CPUs. Using the linear program output, we can derive the ideal computation load for the factorization and the generation phase. Let us consider a simple situation with four nodes where two of them have fast GPUs. Ideally, the generation would be roughly balanced between the four nodes, while the factorization would mostly use the two faster nodes. Figure 4 shows a possible data distribution for the generation (left: a simple 2D block-cyclic distribution) and for the factorization (middle: a 1D-1D distribution).

The 1D-1D distribution obtained by the shuffling procedure of Section 3 ensures a well-balanced factorization with a minimal amount of communications. However, suppose both distributions are computed independently from each other. In that case, the

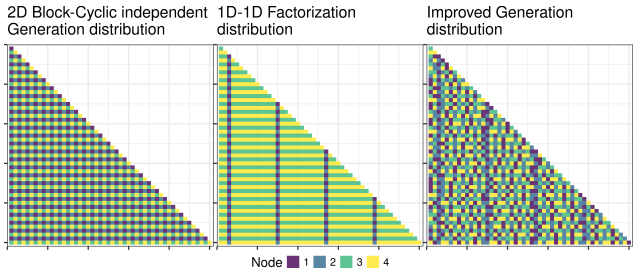

Figure 4: Generation and Factorization distributions for two nodes (1, 2) without and two (3, 4) with GPUs.

chance is that most of the block locations would be different in each distribution, resulting in extra communication in the transition of the phases. Considering a 50x50 matrix with the scenario of Figure 4, using the optimal independent partitions would result in the communication of 890 blocks between the generation and the factorization phase, i.e., 70% of the total number of blocks. However, computing the ideal number of blocks for each phase, the generation has [318, 319, 319, 319] blocks for each node, while the factorization has [60, 60, 565, 590]. That means that the first two nodes should give 517 blocks in total, while the last ones should receive 517 blocks. A transition with 517 communications would be the minimum possible, i.e., 373 (41.91%) fewer transfers than when distributions are independent. Another important aspect is that the generation distribution should also be "cyclic", just like the 1D-1D distributions, to ensure that the beginning of generation is well spread over all the nodes and does not slow down the factorization.

To minimize the redistribution overhead, we propose Algorithm 2, which receives a 1D-1D factorization distribution and a target generation load (the total number of blocks that the generation distribution should have) and computes a generation distribution while minimizing the number of communications. This algorithm goes through the factorization distribution and decides to change the block owner only for the nodes that need to surrender some blocks and based on the ratio between how many blocks that node has and should have. If a node has twice as many blocks as it should have, its ratio is two, and at every two blocks that the algorithms pass through that owner, one block moves to the neediest node. Since the 1D-1D distribution is uniformly spread over the nodes, this cyclic update also ensures a uniform node spread of the generation but respecting processing speeds. The resulting distribution, depicted on the right of Figure 4, minimizes the communications while tries to reach the ideal number per node. We observe similarities with the factorization distribution in the vertical stripes for nodes 1 and 2 and in the horizontal stripes for nodes 3 and 4.

5 PERFORMANCE EVALUATION

This Section presents the evaluation of the strategies to improve phase overlap and the multi-phase heterogeneous distributions.

5.1 Hardware and Software Settings

The performance evaluation uses the Grid5000 platform, more specifically, the Lille site. Table 1 shows the machines used in the experiments. Chifflet and Chifflot have the GTX 1080 and Tesla

Algorithm 2: Generation of a target (generation) distribution (dist2) from a source (factorization) distribution (dist1).

Input: dist1[1...N][1...N]
nblocks_dist2[1...P] *Desired number of blocks per node*
Output: dist2[1...N][1...N]
nblocks_dist1[1...P] = Total number of blocks per node in dist1
diff = nblocks_dist1 - nblocks_dist2
rates, base_rates = $\frac{\text{nblocks\_dist1}}{\text{diff}}$
current_rate[1...P] = (0, ..., 0)
dist2 = dist1
foreach *(m,n) in (1...N, 1...N) by diagonal* **do**
  node = dist1[m, n]
  if *diff[node] > 0* **then**
    current_rate[node] = current_rate[node] + 1
    if *current_rate[node] ≥ rates[node]* **then**
      neediest = which.min(diff)
      dist2[m, n] = neediest
      diff[neediest]++ ; diff[node]- -
      rates[node] = rates[node] + base_rates[node]
      if *diff[neediest] > 1* **then return**
    end
  end
end

P100 GPUs, respectively, while Chetemi does not. The operating system for all the machines is Debian 10 with Linux kernel 4.19. The Chetemi and Chifflet network is a 10Gb Ethernet, while for Chifflot is a 25GB Ethernet. The experiments used the following configurations to control the environment: (a) Intel hyper-threading off; (b) performance frequency governor; (c) NVIDIA GPUs set to persistence mode and max clocks 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.

Table 1: Compute nodes available for our experiments.

Machine	CPU	Memory	GPU
Chetemi	2x Intel Xeon E5-2630 v4	256 GiB	–
Chifflet	2x Intel Xeon E5-2680 v4	768 GiB	GTX 1080
Chifflot	2x Intel Xeon Gold 6126	192 GiB	Tesla P100

The software stack of ExaGeoStat includes the StarPU developer branch commit `015357bd` and the NewMadeleine [8] (the communication layer) master branch commit `d6542d72`. We use the ExaGeoStat master branch commit `9518886` with HiCMA, Chameleon, and Stars-H in the corresponding submodules of the same commit. ExaGeoStat and Chameleon have modifications to accept custom distributions of our strategies. Also, most of the improvements in the phase overlap are modifications on ExaGeoStat that can be enabled or disabled during runtime. The StarPU used 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 the hardware resource (NIC or GPU) is attached to. There is one StarPU process per node. We select two synthetic workloads identified by numbers 8 and 9 with N=57600 and N=96600 from the list of available workloads[5] because they offer a good balance between the generation and the factorization phases for the number of resources we have. As we use 960 as block size, we obtain a matrix size of 60x60 and 101x101 blocks. We use these numbers (60 and 101) to identify each workload.

5.2 Improving Application's Phases Overlap

We evaluate the performance of our strategies to optimize phase overlap using the 60 and 101 workloads in two sets of machines: four or six Chifflet. The goal is to verify if the strategies enable performance improvements by releasing tasks earlier or correcting scheduling and runtime artifacts. Figure 5 presents the results for each workload on the two sets of machines. The X-axis depicts the enabled optimizations, while the Y-axis is the time in seconds. Each configuration has been replicated 11 times, and the error bars represent a 99% confidence interval. The first three strategies (full asynchronous, new solve algorithm, memory optimizations) lead to the bulk of performance improvements. Priority and submission bring minor (with 101 workload) or no (60 workload) performance gains in this homogeneous scenario, but we observed up to ≈10% in heterogeneous scenarios. Finally, the Over-subscription optimization presents a small yet significant and consistent time decrease for both workloads. These results represent performance gains from 36% (for the 101 workload using four machines) to 50% (for the 60 workload using six machines) against the synchronous non-optimized version. We conduct a more detailed analysis of these optimizations using execution traces in what follows.

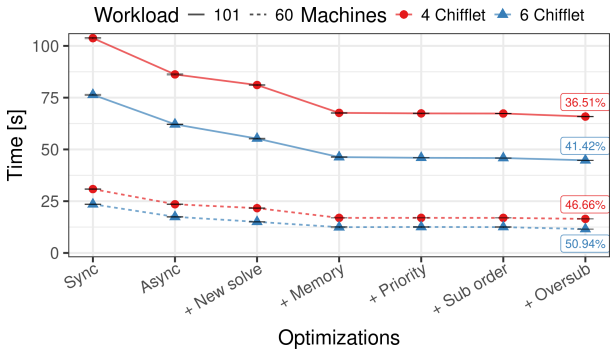

Figure 5: Performance comparison of our phase overlap improvement strategies against the synchronous version.

Figure 6 presents the performance analysis panels (similar to Figure 3) for three versions of cumulative optimizations for the four-machine case with the 101 workload. The left panel (`Async`) shows the **full asynchronous** version. The middle panel (`New Solve + Memory`) shows the asynchronous version plus the **new solve algorithm** and the **memory optimizations** strategies. And finally, the right panel (`All optimizations`) shows the version with all optimizations. The execution behavior with those diverse optimizations is clearly different when analyzing tasks' progression and resource idleness. When compared to Figure 3, Figure 6 Async has

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

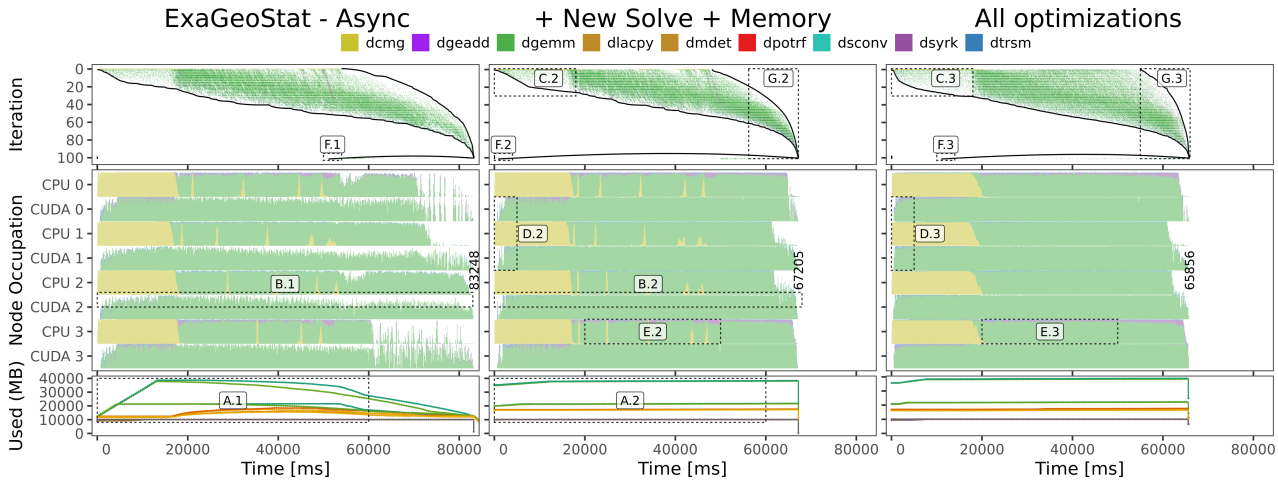

Figure 6: Cholesky Iteration, Node occupation, and Memory utilization panels using 4 Chifflet for three ExaGeoStat iteration cases: Asynchronous, Async + New solve + Memory optimizations, All optimizations.

no more synchronization points, so factorization tasks execute in the GPUs alongside the generation tasks. However, some idle time is clearly visible in the end during the solve phase execution. This idling possibly indicates that the communication and memory utilization may be degrading performance. This hypothesis is further sustained by the behavior of the center panel of Figure 6, when the new solve algorithm and memory optimizations are in place. The memory usage remains high since the beginning of the execution (A.2 annotation when compared to A.1), and resources are almost at 100% utilization (B.2 when compared to B.1). The asynchronous version has a total amount of communications of 11044MB, while with the New Solve version, it decreases to 8886MB.

Although the last three optimizations (Priorities, Submission order, and Over-subscription) provide a modest gain in the iteration makespan, the execution behavior is clearly different, as shown in the visual comparison between the right panel of Figure 6 and the other two panels (left, center). The iteration parallelism is much more pronounced in less time (annotation C.3), showing that the critical path's progression is faster compared to C.2. Also, annotation D.3 demonstrates the absence of a slow start in resource utilization compared to annotation D.2. Nevertheless, we see that all the generation tasks have been completed at the beginning of the iteration (E.3), in contrast to the center panel where some generation tasks are executing when most resources are computing factorization tasks (E.2). The moment when the solve begins changes, as seen in the F.1, F.2, and F.3 annotations, because of the better task prioritization. It is unnecessary to start the solve phase as soon as possible because its results will only be required later. Another behavioral difference appears at the end of the iteration, as shown by the G.2 and G.3 annotations. The iteration ends sooner and presents a more gradual ending (G.2), while all optimizations case provides a more abrupt closure (G.3).

The total resource utilization is also another indication of the performance improvements of our strategies. We can compute this metric as the total amount of time spent in application tasks, divided by the total amount of time (including runtime overhead and pure idle). The total resource utilization for the three executions of Figure 6 are 83.76%, 94.92%, and 95.28% respectively. Moreover, when only considering the first 90% of the iteration, the metric is 93.03%, 99.09%, and 99.13%, respectively. This last fact indicates that the remaining performance improvements are at the end of the execution when parallelism diminishes and working in the critical path becomes more important [17]. In conclusion, the strategies proposed so far are a definitive way to improve phase overlap, resource utilization, and the overall application performance. With these improvements, we evaluate the effectiveness of using distributions suited to heterogeneous clusters.

5.3 Multi-phase partitioning in heterogeneous clusters

We evaluate our proposed distribution strategies for heterogeneous nodes. The experimental cases use the 101 workload and six different sets of machines combining Chetemis, Chifflets, and Chifflots, as depicted by the panels of Figure 7. These sets demonstrate different heterogeneous setup levels, with two (Chetemi and Chifflet) and three types of different machines. For each experiment configuration, we present the makespan as a function of the distribution strategy (colors). The first three bars of each panel are our baselines: the homogeneous block-cyclic distribution using all the resources (red); the homogeneous block-cyclic distribution for the fastest homogeneous subset of nodes (blue); and the heterogeneous 1D-1D distribution using the powers of machines computed considering the `dgemm` speed (green). All these three cases use the same distribution for the factorization and generation phases. The fastest homogeneous subset of nodes (used in the block-cyclic distribution) usually is the Chifflot machines, when they are present. However, in cases 4-4-1 and 6-6-1, the single Chifflot machine is unable to perform this workload well because of high GPU memory utilization. So, for cases 4-4-1 and 6-6-1, the BC Fast Possible Only result indicates the usage of the Chifflet partition. Finally, we depict our result using the linear program to calculate the desirable powers for the factorization partition and generation. Then, 1D-1D uses

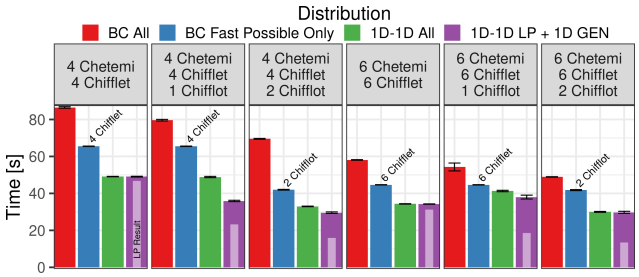

Figure 7: Makespan for homogeneous and heterogeneous distributions in six machine sets configurations.

the factorization partition to produce the respective distribution and the algorithm of Section 4.4 uses this distribution plus the generation partition to compute its distribution (purple).

Figure 7 presents, for each heterogeneous set of machines (panels), the execution time in seconds (Y-axis) as a function of the distribution strategy (X-axis and colored bars). An inner white bar inside our proposed distribution (purple) represents the ideal makespan obtained by the linear program. The block-cyclic distributions are never the best result, neither using all the resources (red) or the fastest homogeneous subset of nodes (blue). The linear program distribution (purple) performs very well in situations 4+4+1, 4+4+2, and 6+6+1. However, in other situations, it presents similar results to the 1D-1D distribution of [17]. This situation happens because the gains obtained by perfectly balancing both phases are minimal, and the phases' imbalance compensates for each other. Furthermore, using a single distribution avoids any redistribution overhead. The very small gap between the ideal makespan obtained by the linear program and the actual execution time for the 4+4 and the 6+6 cases show that the redistribution overhead is perfectly overlapped. However, when using the Chifflot nodes, the actual makespan is much larger than the LP solution, whereas the number of communications does not significantly increase, pointing the source of this problem to the communication library, as we will detail later. These results show that using the LP is beneficial in the best case, and in the worst case, it ties with a single heterogeneous distribution. Compared to the homogeneous cases, the 4-Chifflet case took ≈65s, while 4+4 best-case had a mean makespan of ≈49s (25% faster) and the 4+4+1 best-case took ≈33s (49% faster).

Since adding the very powerful Chifflot nodes fails to provide as much improvement as one would expect, we decided to investigate why. Figure 8 presents the same three panels as before: Iteration (top) and Node occupation (bottom) for three different cases: 4+4 (left), 4+4+1 using all the resources in the factorization (center), and 4+4+1 using only nodes with GPUs in the factorization (right). The execution with only two types of machines is very similar to the one shown in the right of Figure 6, having very low idle times during execution. Furthermore, CPU-only nodes transition between generation and factorization slower and smoother than nodes with GPUs (A.1 annotation). This transition results from high-priority factorization tasks that execute on CPUs alongside generation tasks, while, in GPUs, the same high-priority tasks use GPUs (B.1). The heterogeneous distribution is well balanced, as the generation and factorization in all nodes end almost simultaneously, correctly considering node capabilities (C.1).

When adding a Chifflot node (center of Figure 8), the P100 GPU process the `dgemm` task 10× faster than the Chifflet nodes, adding extra heterogeneity and the need for faster communication. While the overall makespan decreases, a lot of idle time is visible (D.2 annotation), and iterations seem to be limited by the critical path (E.2). Further investigation revealed that communication along the critical path is responsible for such high idle times. Indeed, the factorization is very unbalanced with a very fast Chifflot node helped by slower nodes, and even with a 25GB Ethernet network, data movements are poorly handled since Chifflot is unfortunately on a different subnet of the Lille site. This problem comes from the excessive amount of communication that the fast node has to make, and because of buffering, the block communication ordering does not follow the task priorities strictly and is the object of current developments in NewMadeleine. This problem is also the cause of the performance drop from 6+6 to 6+6+1 of Figure 7. One possible technique to circumvent the communication problem is also to limit the number of nodes during the factorization, which is the phase causing most of the communication operations. This is easily done by excluding the nodes without GPUs from the factorization in the LP constraints. The case in the right of Figure 8 depicts the resulting behavior. The idle time decreases (D.3 annotation), leading to an additional decrease in the makespan. Moreover, the computation is spread again in many iterations (E.3). This case presents a mean makespan of ≈33s. The gap between the actual makespan and the one expected by the linear program remains around 20%, so enhancements in communication should improve this even further. Overall, comparing this result against the original synchronous 4 Chifflet homogeneous execution (≈103s), we have 68% performance improvement.

6 DISCUSSION AND CONCLUSION

This paper presents strategies for distributing multi-phase applications over heterogeneous system-level resources. ExaGeoStat serves as an example, but we believe that most of the techniques we used would apply to similar multi-phase applications, especially ones with generation and factorization phases. We improve the application phase overlap and exploit their different performance needs. The combination of six strategies to enhance scheduling and runtime decisions provides performance gains from 36% to 50% (Section 5.2) on a homogeneous set of nodes. Tackling system-level heterogeneity, we build application phases' data distributions over heterogeneous nodes that ensure a smooth transition between phases while limiting communication overheads and efficiently using the resource compute capabilities for each phase. A linear program that calculates the ideal load for each node, considering shallow phase dependencies, handles the heterogeneity of computation resources and phase requirements. We achieve the communication reduction and the effective transition between phases by devising distinct but tightly coupled distributions for each phase. Altogether, our strategies allow us to exploit system-level heterogeneous setups and to improve performance by up to 68% compared to a simple homogeneous setup (Section 5.3).

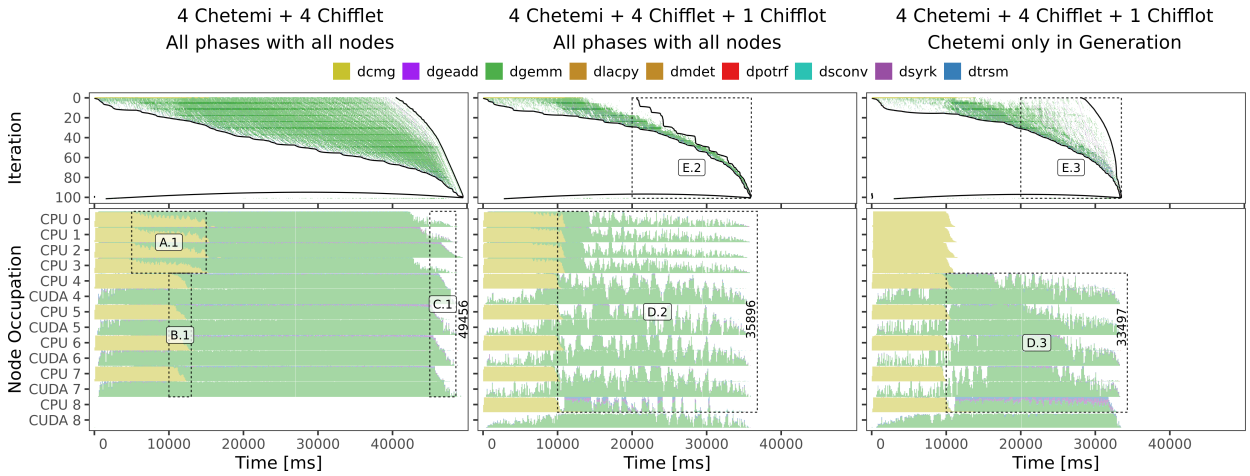

Figure 8: Cholesky Iteration, Node occupation, and Memory utilization panels of the ExaGeoStat iteration using the 1D-1D LP + 1D GEN distributions for three sets of machines: 4+4, 4+4+1, and 4+4+1 restricting factorization to GPU-only nodes.

As future work, we first intend to optimize the communication middleware to improve performance limitation when the compute capabilities between fast and slow nodes are too substantial. Second, we intend to provide a way for ExaGeoStat to decide which set of nodes to use for a given problem size. This capacity planning would be beneficial as throwing more and more nodes is costly and rarely valuable as performance eventually degrades because of communication overheads. However, modeling communication overhead and network contention is complicated, and a possibility could be to use simulation provided by StarPU-SimGrid [17, 20] or machine learning algorithms. Finally, we intend to study how all our improvements behave at scale on large heterogeneous supercomputers like Jean-Zay, Piz Daint, or SDumont.

Software and Data Availability. We endeavor to make our analysis reproducible. A public companion (https://gitlab.com/lnesi/icpp21) contains the data and instructions to reproduce our results.

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, Multi-GPU 16/354-8, and GreenCloud – 16/488-9), CNPq (447311/2014-0), CAPES (Brafitec 182/15, and Cofecub 899/18). 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 would like to thank Hatem Ltaief for his work in ExaGeoStat and the fruitful discussions about this work.

REFERENCES

[1] Sameh Abdulah, Hatem Ltaief, Ying Sun, Marc G. Genton, and David E. Keyes. 2018. ExaGeoStat: A High Performance Unified Software for Geostatistics on Manycore Systems. *IEEE Transactions on Parallel and Distributed Systems* 29, 12 (2018), 2771–2784.

[2] Emmanuel Agullo et al. 2010. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In *GPU Computing Gems*, Wen mei W. Hwu (Ed.). Vol. 2. Morgan Kaufmann.

[3] Cédric Augonnet et al. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Conc. Comp.: Pract. Exp., SI: EuroPar'09* 23 (2011), 187–198.

[4] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, and Yves Robert. 2001. Matrix multiplication on heterogeneous platforms. *IEEE Trans. Parallel Distributed Systems* 12, 10 (2001), 1033–1051.

[5] Olivier Beaumont, Arnaud Legrand, Fabrice Rastello, and Yves Robert. 2001. Static LU Decomposition on Heterogeneous Platforms. *Int. Journal of High Performance Computing Applications* 15 (2001), 310–323.

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

[7] George Bosilca et al. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science Engineering* 15, 6 (2013), 36–45.

[8] Alexandre Denis. 2019. Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests. In *19th IEEE/ACM Int. Symposium in Cluster, Cloud, and Grid Computing*. IEEE, Cyprus, 371–380.

[9] Jack J Dongarra et al. 2017. With Extreme Computing, the Rules Have Changed. *Computing in Science & Engineering* 19, 3 (2017), 52.

[10] Jack J Dongarra, Hans W Meuer, Erich Strohmaier, et al. 1997. TOP500 supercomputer sites. *Supercomputer* 13 (1997), 89–111.

[11] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. 2011. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Paral. Proces. Letters* 21 (2011), 173–193.

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

[13] Mark Gates et al. 2019. SLATE: Design of a Modern Distributed and Accelerated Linear Algebra Library. In *Proceedings of the Int. Conference for High Performance Computing, Networking, Storage and Analysis*. ACM, United States, 18 pages.

[14] Robert B. Gramacy. 2020. *Surrogates: Gaussian Process Modeling, Design, and Optimization for the Applied Sciences*. CRC Press, United States.

[15] Julien Herrmann et al. 2016. Assessing the cost of redistribution followed by a computational kernel: Complexity and performance results. *Par. Comp.* 52 (2016).

[16] Alexey Kalinov and Alexey Lastovetsky. 2001. Heterogeneous Distribution of Computations Solving Linear Algebra Problems on Networks of Heterogeneous Computers. *J. of Par. and Distr. Comp.* 61, 4 (2001), 520.

[17] Lucas Leandro Nesi, Lucas Mello Schnorr, and Arnaud Legrand. 2020. 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, 54–63.

[18] Loïc Prylli and Bernard Tourancheau. 1996. Efficient block cyclic data redistribution. In *Euro-Par'96 Parallel Processing*, Luc Bougé et al. (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 155–164.

[19] Eduardo Roloff, Matthias Diener, Luciano Gaspary, and Philippe Navaux. 2019. Exploring Instance Heterogeneity in Public Cloud Providers for HPC Applications. In *The 9th Intl. Conf. on Cloud Comp. and Services Sci.* SciTePress, 210–222.

[20] Luka Stanisic et al. 2015. 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.