Symmetric Block-Cyclic Distribution: Fewer Communications Leads to Faster Dense Cholesky Factorization

Olivier Beaumont*, Philippe Duchon†, Lionel Eyraud-Dubois*, Julien Langou‡ and Mathieu Vérité*

* INRIA Centre at the University of Bordeaux
Bordeaux, France
Email: firstname.lastname@inria.fr

† Laboratoire Bordelais de Recherche en Informatique
Bordeaux, France
Email: philippe.duchon@labri.fr

‡ University of Colorado Denver
Denver, Colorado
Email: julien.langou@ucdenver.edu

***Abstract*—We consider the distributed Cholesky factorization on homogeneous nodes. Inspired by recent progress on asymptotic lower bounds on the total communication volume required to perform Cholesky factorization, we present an original data distribution, Symmetric Block Cyclic (SBC), designed to take advantage of the symmetry of the matrix. We prove that SBC reduces the overall communication volume between nodes by a factor of square root of 2 compared to the standard 2D block-cyclic distribution. SBC can easily be implemented within the paradigm of task-based runtime systems. Experiments using the Chameleon library over the StarPU runtime system demonstrate that the SBC distribution reduces the communication volume as expected, and also achieves better performance and scalability than the classical 2D block-cyclic allocation scheme in all configurations. We also propose a 2.5D variant of SBC and prove that it further improves the communication and performance benefits.**

I. Introduction

Matrix operations, and particularly matrix factorizations, are at the heart of many applications and have received considerable interest for many years. In this paper, we focus on the dense Cholesky factorization, which, given a dense symmetric positive definite matrix $\mathbf{A}$, consists in computing $\mathbf{L}$, the lower triangular matrix, such that $\mathbf{A} = \mathbf{L}\mathbf{L}^T$. This factorization is an essential step for solving linear systems of type $\mathbf{A}x = \mathbf{B}$, by reducing it to computing solutions of $\mathbf{L}y = \mathbf{B}$, and then $\mathbf{L}^T x = y$.

Parallel algorithms for matrix operations are often obtained with tile-based approaches, where the matrix is divided in tiles and operations on individual tiles are performed in parallel with optimized sequential implementations. The distribution of the tiles of the matrix over the computing nodes of the platform has a strong influence on the performance, by affecting both the load balancing and the amount of required communications. Available implementations mostly rely on static data partitioning strategies, to avoid a significant communication overhead; these strategies are most of the time simple and regular to facilitate their implementation using traditional programming models such as MPI.

However, the high-level abstraction provided by modern task-based runtime systems such as `StarPU` [1], [2] or `ParSEC` [3] makes it possible to use more optimized strategies, even irregular ones, since all communications are transparently handled by the runtime system. Such systems use dynamic priority-based approaches to schedule the tasks inside a compute node, and rely on static placement decisions to balance the tasks among the nodes.

In the case of Cholesky factorization (and of the nonsymmetric variant LU factorization), the standard approach is to use a $2D$ block-cyclic distribution, as is typically done in the linear algebra algorithms in `ScaLAPACK` [4]. This approach is efficient in terms of load balancing (each node receives the same number of tasks to compute) and even in terms of load balancing over time, in the case where the trailing matrix on which the algorithm is applied decreases over time, such as in Cholesky factorization.

Research works on communication-avoiding algorithms have proposed improved $2.5D$ block-cyclic distributions [5], [6], which trade additional storage for lower communications. Based on lower bounds on the minimal communication volume that must be performed to compute the factorization [7], these algorithms can be proven asymptotically optimal, within a constant factor. Recent works on more refined lower bounds have attempted to improve this factor, relying on compilation techniques like the analysis of loop nests [8], [9]. However, these $2.5D$ distributions are still based on the block-cyclic approach, which does not take advantage of the symmetry of the data in the Cholesky factorization. Our work is orthogonal to these approaches.

In this paper, we propose **SBC**, a *symmetric* version of the block-cyclic distribution tailored for the Cholesky factorization, which achieves the same load-balancing benefits as the usual block-cyclic and results in a smaller communication volume. We start by presenting related works on communication lower bounds and communication-avoiding algorithms in Section II. In Section III, we describe the $2D$ variant of **SBC**, and derive its respective communication cost, showing

an asymptotic improvement of a factor of $\sqrt{2}$ over the $2D$ block-cyclic distribution. We describe and analyze the $2.5D$ variant of **SBC** in Section IV. Finally, we show in Section V that the proposed **SBC** distribution significantly reduces both the communication volume and the computational time for the Cholesky factorization. We also consider the resolution of symmetric linear systems and inversion of symmetric matrices, both based on Cholesky factorization, where **SBC** achieves smaller but promising gains. Concluding remarks are provided in Section VI.

II. Related Work

Research work on communication-avoiding algorithms for linear algebra operations has advanced through the development of lower bounds, which express the minimal communication volume necessary to perform a given operation. Striving to reach these lower bounds has led to the design of more efficient algorithms. In order to obtain these lower bounds, simplified machine models are considered:

1) Two-levels memory model: the machine features one fast and limited memory of size M and one "slow" and unlimited memory. Input required for any computation must reside in fast memory to be performed. This model is also referred to as the "out-of-core" setting.
2) Parallel model: P nodes, all equipped with a memory of size M, can communicate through a network.

The first model can be used to study the volume of communication of a single node in a parallel machine since the set of all other nodes can be viewed as a single "slow" memory with which data transfers occur. The first results on communication lower bounds were obtained for classical matrix multiplication [7], [10]: multiplying two $n \times n$ matrices with a limited memory of size M requires transferring $\Theta(\frac{n^3}{\sqrt{M}})$ elements. This asymptotic result was generalized to LU and Cholesky factorization of an $n \times n$ matrix [6], which also require $\Theta(\frac{n^3}{\sqrt{M}})$ data transfers. The same authors later showed in [11] that those bounds are part of a more general framework. Thus similar bounds of the form $\Omega(\frac{\#\text{arithmetic operation}}{\sqrt{M}})$ can be derived for a wide range of operations.

More recently, automatic *cDAG* analysis led to the refinement of these lower bounds by providing explicit leading terms for several operations, including Cholesky factorization. Based on a discrete version of classical Loomis-Whitney projection argument developed in [12], Olivry *et al.* establish in [8], that Cholesky factorization requires at least $\frac{n^3}{6\sqrt{M}}$ data transfers. A specific analysis adapted to the symmetric nature of this factorization allowed Beaumont *et al.* [13] to further improve the bound and obtain an optimal value of $\frac{n^3}{3\sqrt{2}\sqrt{M}}$.

In 2009, Béreux [14] proposed a sequential out-of-core Cholesky algorithm with "narrow blocks" that performs at most $\frac{n^3}{3\sqrt{M}} + O(n^2)$ data transfers. The authors of [13] have improved this result with an algorithm which performs $\frac{n^3}{3\sqrt{2}\sqrt{M}} + O(n^{5/2})$, showing that the lower bound can not be improved. However, these algorithms are not directly applicable to a parallel setting. We discuss the relationship between the sequential and parallel models in more details in Section III-E.

`ScaLAPACK` library [4] contains parallel implementations of many linear algebra operations, based on the $2D$ block-cyclic distribution. For matrix multiplication, the corresponding algorithm is proven *asymptotically optimal* in the following sense: assuming $M = \Theta(\frac{n^2}{P})$, it performs $O(\frac{n^2}{\sqrt{P}})$ data transfers per node, and Irony *et al.* [7] proved that with such an assumption on M, any algorithm has to perform $\Omega(\frac{n^2}{\sqrt{P}})$ data transfers per node. This shows that the ratio between the algorithm and the lower bound is bounded, without providing an explicit value for the bound. A similar result is proven by Ballard *et al.* [6] for the parallel distributed Cholesky algorithm of `ScaLAPACK`.

When more memory is available, it is possible to design $3D$ and $2.5D$ algorithms, by replicating the input and/or output data on several nodes. Similar asymptotic optimality results are proven for the $3D$ matrix multiplication [7] and $2.5D$ matrix multiplication and LU factorization [15]. An early implementation of fully $3D$ distributed algorithms for triangular solve and LU factorization without pivoting is proposed in [16] that are asymptoticaly optimal for the total volume of communications. The $2.5D$ algorithms bring a continuum between $2D$ and $3D$ algorithms where the trade-off between memory footprint and communication is controlled by a parameter. An implementation of $2.5D$ Cholesky on a Cray system is proposed in [17], introducing some overlap between the iterations to minimize the impact of communications. Experimental results show the superiority of $2.5D$ algorithms over conventional $2D$ algorithms. In 2021, Kwasniewski *et al.* [9] present `COnfLUX` and `COnfCHOX`, parallel distributed $2.5D$ LU and Cholesky algorithms based on a block-cyclic distribution. They prove that these algorithms induce a communication volume $\frac{n^3}{\sqrt{M}} + O(n^2)$.

`Chameleon` [18] is a state-of-the art dense linear algebra library, which features tiled implementations of many operations. `Chameleon` follows a task-based approach: the computation is abstracted as a directed acyclic graph of tasks whose dependencies are handled automatically by the underlying runtime system. In a distributed setting, the runtime system also automatically infers the necessary communications based on the task placement which is provided by the application. As shown in [18], careful management of task submission and pruning of the task graph ensures that the approach remains scalable. In the case of Cholesky factorization, viewing the computation as a task graph with fine-grain dependencies makes it possible to avoid synchronization between iterations[1], tasks of the next iteration can start even if the current iteration is not yet completed. This means that tasks on the critical path can be executed sooner, and in turn it provides more parallelism than static, synchronized, MPI-based implementations. The result is a generalization of the overlap approach proposed in [17], performed automatically by the runtime system. To

[1]These iterations refer to the outer `for` loop of the Cholesky factorization, line 1 of Algorithm 1.

ensure that the computation can be performed asynchronously, each tile is sent to its destination as a separate message, so that the number of messages is proportional to the communication volume. Furthermore, thanks to the asynchronous approach, the latency is hidden by the overlap with computations. For these reasons, in this paper we focus on communication volume and not on number of messages.

In Section III of this paper, we propose **SBC**, a *symmetric* block-cyclic distribution, which can be used either in the $2D$ or $2.5D$ setting. These algorithms perform the Cholesky factorization with a communication volume $\frac{n^3}{2\sqrt{M}} + O(n^2)$, which represents a factor of 2 improvement over the result of COnfCHOX. We also show in Section V that implementing these distributions in the task-based Chameleon library significantly improves running time over the classical block-cyclic distribution.

III. 2D Symmetric Block Cyclic Distribution

We start by describing the $2D$ version of the Symmetric Block Cyclic (**SBC**) distribution. In a $2D$ distribution, each tile of the matrix is assigned to only one node, whereas with the $2.5D$ distribution described in Section IV, tiles are replicated to further reduce the communication volume.

A. Overview

We consider throughout the paper a symmetric positive definite matrix $\mathbf{A}$, and we assume that it is divided into $N \times N$ tiles of size b^2, for a fixed tile size b. We denote by $\mathbf{A}_{i,j}$ the tile of matrix $\mathbf{A}$ in position (i, j), for $0 \leq i \leq j < N$. Algorithm 1 provides the pseudo-code for the tiled Cholesky factorization, based on 4 different types of operations (POTRF, TRSM, SYRK, GEMM) performed at the tile level, which we denote as *tasks* in the rest of the paper.

Algorithm 1 Tiled Cholesky Factorization Algorithm

```
1: for i = 1...N−1 do
2:     A_{i,i} ← POTRF(A_{i,i})
3:     for j = i+1...N−1 do
4:         A_{j,i} ← TRSM(A_{j,i}, A_{i,i})
5:     for k = i+1...N−1 do
6:         A_{k,k} ← SYRK(A_{k,k}, A_{k,i})
7:         for j = k+1...N−1 do
8:             A_{j,k} ← GEMM(A_{j,k}, A_{j,i}, A_{k,i})
```

In this paper, we consider a task-based execution, in which task dependencies and communications are entirely handled by a runtime system. During the execution of a given algorithm, the runtime system ensures that each task is executed only after its input data has been computed. However, tasks which do not share any dependency can be executed in parallel.

All of the tasks referenced in Algorithm 1 are a set of elementary computations performed on the data of the input tile(s). They are executed sequentially on one worker of the node, typically on one core, generally relying on classical low level implementation such as BLAS. Since nodes usually feature manycore CPUs or even GPUs, the orchestration of those computations *within* a node is necessary in order to achieve high performance. From the perspective of this paper, this orchestration is left to the responsibility of the runtime system.

The focus of this paper is rather on the distribution of the data and the computations *between* the nodes. A data distribution for a tiled algorithm like Algorithm 1 is an assignment of the $\frac{N(N+1)}{2}$ tiles of the input matrix $\mathbf{A}$ to the P computing nodes: each node owns the data corresponding to the tiles which are assigned to it. In this paper, we consider that tasks are distributed among nodes according to the standard *owner computes* rule, which means that all the tasks that *modify* a given tile are performed by the node to which that tile is assigned.

Since data chunks are distributed among nodes as tiles, the execution of one task by a given node P_1 often requires accessing to the value of a tile $\mathbf{A}_{i,j}$ that is owned by another node P_2. In that case, the runtime system is in charge of requesting the communication of $\mathbf{A}_{i,j}$ to P_1 so that it can be used as input to perform the task. Nevertheless, the ownership of the tile is not modified: the data is still stored by P_2, and P_2 will still receive all future requests for this tile. An implementation of this kind of model of execution is detailed in Section V-C, using StarPU as runtime system and an MPI library as inter-nodes communication layer.

To motivate the introduction of the **SBC** distribution, let us focus on the dependencies between TRSM and GEMM tasks in Algorithm 1, and consider a given tile $\mathbf{A}_{j_0,i}$ computed by a TRSM task. It can be used as the second parameter of a GEMM task on $\mathbf{A}_{j_0,k}$ or as the third parameter of a GEMM task on $\mathbf{A}_{j,j_0}$. Classical data distributions do not take advantage of this additional data reuse possibility, which does not exist for the LU factorization. Indeed, in that case, matrix $\mathbf{A}$ is not symmetric, and line 8 becomes $\mathbf{A}_{j,k} \leftarrow$ GEMM($\mathbf{A}_{j,k}$, $\mathbf{A}_{j,i}$, $\mathbf{A}_{i,k}$): a given tile $\mathbf{A}_{j_0,i}$ can only be used as a second parameter.

The standard $2D$ block-cyclic distribution (denoted **2DBC** in the rest of the paper) is based on a repeating $p \times q$ pattern, where $P = pq$: each node is associated to a 2-dimensional index (x, y) with $0 \leq x < p$ and $0 \leq y < q$ (see Figure 1). Tile (i, j) is assigned to the node whose index is $(i \mod p, j \mod q)$. With this distribution, a tile $\mathbf{A}_{j_0,i}$ produced by a TRSM task is required by the p nodes which are assigned tiles $\mathbf{A}_{j_0,k}$ for $k \leq j_0$ and the q nodes which are assigned $\mathbf{A}_{j,j_0}$ for $j > j_0$. This involves $p + q - 1$ different nodes, and since one of them performed the TRSM task, this tile $\mathbf{A}_{j_0,i}$ needs to be sent to $p + q - 2$ nodes. This is highlighted on Figure 1: a tile produced by node 1 is sent to nodes 2 and 0 on the same row, and to node 4 on the corresponding column.

In the following, we introduce a new distribution called **SBC**, in which the two sets of nodes (for rows and columns) are always the same, thus reducing the communication volume. This is achieved with a slightly larger pattern, so that each

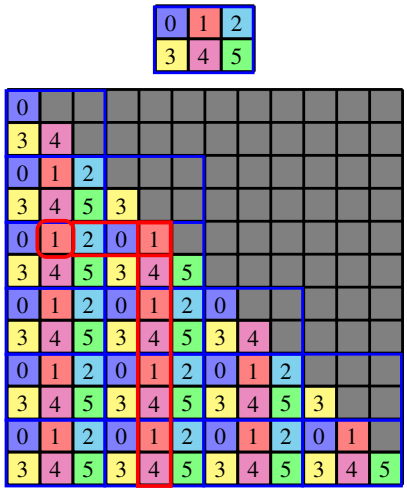

Fig. 1: $2D$ block-cyclic allocation: repetition of the 2×3 pattern (using $P = 6$ nodes; one color per node) over a 12×12 matrix. The communication of one tile is highlighted.

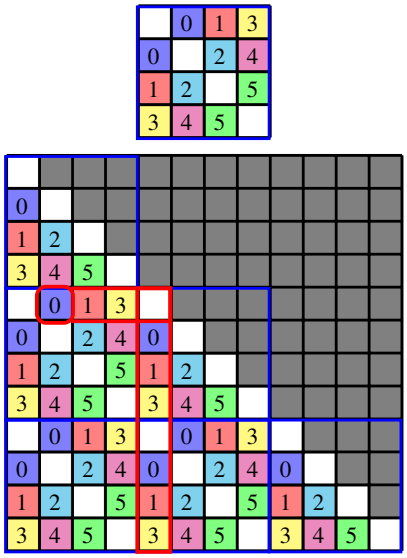

Fig. 2: Symmetric Block Cyclic allocation: repetition of the 4×4 pattern (using $P = 6$ nodes; one color per node) over a 12×12 matrix. Diagonal positions in the pattern are omitted.

Fig. 3: Basic version of **SBC** pattern, with additional nodes, for $r = 4$.

node appears twice in the pattern. We will prove that this reduces the communication volume by a factor of $\sqrt{2}$.

B. Generic *SBC* distribution

Let us consider that the number of nodes is given by $P = \frac{r(r-1)}{2}$ nodes for some integer $r \geq 2$. The **SBC** distribution is built from a $r \times r$ pattern, which is to be repeated over the whole matrix. In the pattern, each node is assigned a pair (x, y) with $x < y$, and is associated with two indices, at coordinates (x, y) and (y, x). This pattern is then repeated across the matrix just like for the block-cyclic distribution: tile (i, j) is assigned to the node associated with the index $(i \mod r, j \mod r)$. The resulting distribution for $r = 4$ and $N = 12$ can be seen on Figure 2.

We provide below two possibilities for the allocation of diagonal indices of the pattern (those with coordinates (x, x)). We first notice, as highlighted on Figure 2, that a tile produced by node 0 is sent to nodes 1 and 3 on the same row, and the nodes on the corresponding column are the same.

C. Allocation of diagonal indices

Let us now discuss how to allocate the diagonal indices of the pattern. To achieve a good load balance, it is important to have each node appear the same number of times on the diagonal (the rest of the pattern is already balanced). We present two possible solutions: the **basic** version of **SBC** is only valid for even values of r, and uses $\frac{r}{2}$ additional nodes on the diagonal, while the **extended** version does not use any additional node and is valid for all r.

*1) Basic version of **SBC**:* For even values of r, we can add $\frac{r}{2}$ nodes to the generic pattern, and assign each of them to two coordinates on the diagonal of the pattern. We assign them in a round-robin fashion, as indicated on Figure 3. The resulting allocation has $P = \frac{r^2}{2}$ nodes, and each tile is communicated to $r - 1$ nodes: all nodes on a row are different, but the tile needs not be communicated to the node that produced it. On the example of Figure 3, we have $r = 4$, thus 8 nodes, and each tile is communicated to 3 nodes. For comparison, the equivalent block-cyclic distribution has $p = 4$ and $q = 2$, thus each tile is communicated to 4 nodes.

*2) Extended version of **SBC**:* Another solution is to keep the same set of $P = \frac{r(r-1)}{2}$ nodes, and assign them on the diagonal. To avoid additional communication, we want to choose nodes that already appear in the same row or column. Since there are only r positions on the diagonal, achieving load balance requires to consider a set of patterns, which will be repeated alternatively over the matrix.

Let us first consider the case of odd r. We build $\frac{r-1}{2}$ patterns (they only differ by their diagonal entries) in the following way: for l from 1 to $\frac{r-1}{2}$, the diagonal entries of the l-th pattern contains the nodes $(1, 1 + l), (2, 2 + l), ..., (r - l, r)$ (first group) and the nodes $(1, r + 1 - l), ..., (l, r)$ (second group). Both patterns are depicted on the right of Figure 4, where nodes of the first group are in blue, nodes of the second

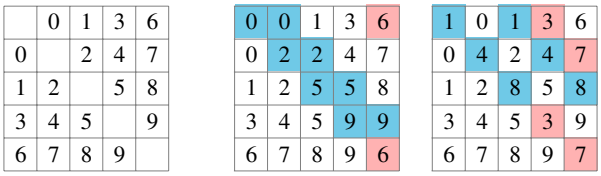

Fig. 4: Extended **SBC** distribution for $r = 5$, $P = 10$.
Left: generic pattern, right: patterns with diagonal nodes.

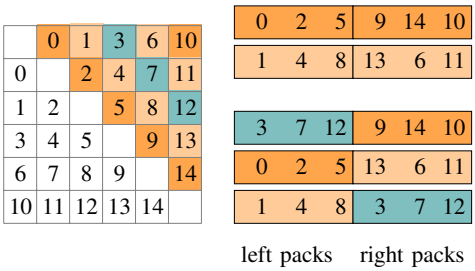

Fig. 5: Example of the **SBC** distribution for $r = 6$, $P = 15$. Left: generic pattern, right: 5 sets of diagonal nodes, with four normal packs and one bonus pack.

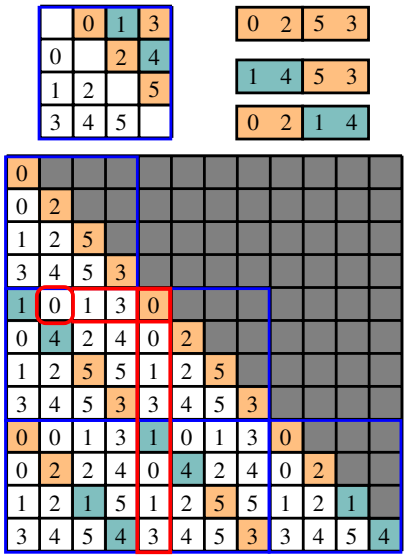

Fig. 6: Diagonal patterns: two normal packs and one bonus pack generate 3 diagonal patterns used in a round-robin column-wise fashion.

group are in red. Each node appears on the diagonal of exactly one pattern, and always on the same row (for the first group) or on the same column (for the second group).

The case of even r is based on the same idea, but with an additional complication: using the same construction leads to both groups of the last pattern containing the same nodes, which would not result in good load balancing. We thus create $r-1$ patterns (only described by the nodes on the diagonal):

- The first $\frac{r}{2}-1$ patterns are obtained just like previously; for each pattern, its diagonal nodes are considered as two packs of $\frac{r}{2}$ nodes, the *left* pack and the *right* pack.
- A last bonus pack is obtained with nodes $(r/2+1,1),...,(r,r/2)$, this pack can be placed both at the beginning (same row) or at the end (same column) of the diagonal. At this point any combination of a left pack and a right pack can be used to create a valid pattern, and the bonus pack can be used either as a left or as a right pack.
- We create $\frac{r}{2}$ additional patterns by shifting the packs of the first $\frac{r}{2}-1$ patterns: we add the bonus pack at the top of the list of the left packs and at the bottom of the list of the right packs, and we combine them together.

The result for $r = 6$ is shown on Figure 5, where packs are represented with different colors. With this construction, each node appears on the diagonal of exactly two patterns, and always on the same row or column. A complete allocation for $r = 4$ and $N = 12$ is depicted in Figure 6. For both cases (even and odd r), the result is an allocation with $P = \frac{r(r-1)}{2}$ nodes, in which each tile is communicated to $r-2$ nodes: each row of the generic pattern contains $r-1$ nodes, and the diagonal node is one of them. We can see on Figure 6 that the tile produced by node 0 is sent to nodes 1 and 3.

D. Analysis of the Communication Volume

The description of the tiled Cholesky factorization given in Algorithm 1 indicates that when using the "owner computes" rule, two types of communication happen:

- the results of POTRF tasks have to be sent along the corresponding column to TRSM tasks;
- the results of TRSM tasks have to be sent along a row and a column to feed GEMM and SYRK tasks.

The result of GEMM and SYRK tasks is not sent over the network because the next task that requires this result (either an intermediate GEMM or a terminal TRSM or POTRF) is performed on the same node. With the **SBC** distribution, each of these two types of communication are sent to either $r-1$ nodes (for the basic version) or $r-2$ nodes (for the extended version). Furthermore, each tile is involved in such a communication exactly once. This yields the following result.

Theorem 1: Let S be the total size required to store matrix $\mathbf{A}$. The total communication volume generated when performing Cholesky factorization with the **SBC** distribution with parameter r is:

- $D_{\text{basic}} = S \cdot (r-1)$ for the basic version,
- $D_{\text{extended}} = S \cdot (r-2)$ for the extended version,

We can compare this result with the square block-cyclic distribution when P grows to infinity. For a given value of P, the parameter r using extended **SBC** is such that $P = \frac{r(r-1)}{2}$, and the parameter p of square **2DBC** is such that $P = p^2$.

When P grows to infinity, $r \sim \sqrt{2P}$ (for both basic and extended versions) and $p \sim \sqrt{P}$.

For a matrix with S elements, **SBC** induces a communication volume of $S \cdot (r-2) \sim S\sqrt{2P}$, whereas square **2DBC** induces a communication volume of $S \cdot (2p-2) \sim 2S\sqrt{P}$. For both weak scaling (where S grows at the same rate as P) and strong scaling (where S remains constant while P grows), **SBC** induces a communication volume $\sqrt{2}$ smaller than $2D$ block-cyclic.

E. Connection between sequential and parallel models

In this section, we provide some insight on the differences between **2DBC** and **SBC**, by comparing them to sequential out-of-core approaches. For ease of presentation, we discuss about the gains on communication volume in terms of *arithmetic intensity*, defined at the level of one computing node. The *arithmetic intensity* of a given node is the ratio between the number of arithmetic operations divided by the communication volume performed by this node. Similarly to the sequential out-of-core methodology, we express how this arithmetic intensity depends on the amount of data M stored on a node.

To highlight the differences between the sequential out-of-core and parallel models, we start with the simpler, non-symmetric case of LU factorization without pivoting. In that context, it is possible to prove that $\sqrt{M}$ is an upper bound on the arithmetic intensity [8]. Béreux's algorithm [14] is an out-of-core sequential algorithm, with a narrow-block, left-looking approach, which achieves $\sqrt{M}$ arithmetic intensity for the Cholesky factorization and for LU factorization (this amounts respectively to $\frac{N^3}{3\sqrt{M}}$ and $\frac{2N^3}{3\sqrt{M}}$ data transfers in total). Asymptotically, for both symmetric and nonsymmetric factorizations, GEMM tasks are dominant in terms of computation, and the dependencies from TRSM to GEMM tasks are dominant in terms of communication.

We first analyze the arithmetic intensity obtained with $2D$ block-cyclic distribution for LU factorization. Assume the optimistic case where $P = p^2$ and $N = p \cdot k$ for some k. There are N^2 tiles in the matrix, so each node stores $M = \frac{N^2}{P} = k^2$ tiles. Let us consider the updates related to the first iteration of the LU factorization. Each node performs $2k^2$ arithmetic operations: one multiplication and one addition for each tile. Each node receives $2k$ tiles: k from the first column, and k from the first row. This yields an arithmetic intensity $\rho = \frac{2k^2}{2k} = k = \sqrt{M}$, reaching the optimal upper bound.

However, in the following iterations, the arithmetic intensity is lowered because the factorization continues on the trailing matrix (the loops on j and k lines 8 and 6 of Algorithm 1 start at $i+1$): the local domain on which the computation is performed shrinks, and parts of the data stored in the node are actually no longer useful for the computations. In total, since each tile is sent to $p-1$ nodes, the number of tiles transferred is $D = N^2 \cdot (p-1) \sim N^2\sqrt{P}$. Since the data stored on each node is $M = \frac{N^2}{P}$, we get $\sqrt{P} = \frac{N}{\sqrt{M}}$ and $D = \frac{N^3}{\sqrt{M}}$. Asymptotically, the total number of arithmetic operations is $\frac{2N^3}{3}$, which yields an average arithmetic intensity of $\frac{2}{3}\sqrt{M}$. This ratio $\frac{2}{3}$ comes from the fact that the local domain shrinks with the iterations.

Let us now consider Cholesky factorization, still with $2D$ block-cyclic distribution. The main difference is that only half of the matrix is stored (thanks to symmetry), so that $M = \frac{N^2}{2P}$, and thus $k = \sqrt{2M}$ since each node stores $\frac{k^2}{2}$ tiles. In the first iteration, each node performs half as many arithmetic operations as for LU (k^2), but receives the same number of tiles ($2k$). This yields an arithmetic intensity of $\frac{k}{2} = \frac{\sqrt{M}}{\sqrt{2}}$. This shows that **2DBC** is well-suited for LU factorization, but not for Cholesky factorization where it cannot reach the maximal arithmetic intensity, even for the first iteration.

For **SBC** distribution, $P \sim \frac{r^2}{2}$ and let us assume that $N = r \cdot k$ for some k. Since the distribution is balanced, we still have $M = \frac{N^2}{2P}$, which means that each node stores $M = k^2$ tiles and that $k = \sqrt{M}$. In the first iteration, each node performs $2k^2$ arithmetic operations, and receives $2k$ tiles: this leads to the same arithmetic intensity $k = \sqrt{M}$ as Béreux's algorithm. Just like for LU factorization above, the shrinking of the local domain induces a lower arithmetic intensity over the whole computation, with the same factor of $\frac{2}{3}$. Indeed, the number of transfers is $D = \frac{N^2}{2} \cdot \sqrt{2P} = \frac{N^3}{2\sqrt{M}}$ for a total number of arithmetic operations of $\frac{N^3}{3}$. This leads to an average arithmetic intensity $\rho = \frac{2}{3}\sqrt{M}$.

In summary, the sequential algorithm of Béreux achieves the same arithmetic intensity for LU and Cholesky factorizations, and is optimal for LU. The $2D$ block-cyclic distribution can match this arithmetic intensity for LU (with a $\frac{2}{3}$ factor), but not for Cholesky. The **SBC** distribution is an adequate adaptation of Béreux's algorithm for the parallel distributed Cholesky factorization. A recent result [13] shows that Béreux's algorithm is actually not optimal for Cholesky: it is possible to achieve $\sqrt{2M}$ arithmetic intensity. However, adapting the algorithm of [13] to the parallel setting is beyond the scope of this work.

IV. 2.5D Symmetric Block-Cyclic Distribution

Similarly to the $2.5D$ block-cyclic distribution (used for example in [9]), it is possible to use **SBC** distribution in a $2.5D$ context. We describe it here with the basic version of **SBC**, but the extended version can be used as well. Assume that we have $P = c\frac{r^2}{2}$ nodes for some value or $r > 1$ and $c > 1$. In the $2.5D$ **SBC** distribution, these P nodes are partitioned into c slices of $\frac{r^2}{2}$ nodes. Each slice stores a copy of the matrix $\mathbf{A}$, distributed with the **SBC** distribution with parameter r.

Each iteration is performed by a different slice, in a round-robin fashion: iteration i is performed on slice $i \mod c$. This means that the GEMM or SYRK updates corresponding to a given tile are accumulated on c different nodes (one in each slice). Before performing the corresponding TRSM or POTRF task on that tile, these updates are aggregated with a reduction operation onto the slice that performs this iteration. This adds a new source of communication: each tile is thus part of $c-1$ communications (whether by a reduce over c nodes or by $c-1$ point-to-point communications, this does not change the total communication volume). However, the result of POTRF and

TRSM tasks are always communicated *within* a slice, and are thus communicated $r-1$ times.

With a synchronized, pure MPI implementation, performing each iteration on a subset of the nodes would result in each slice working one after the other, thus preventing parallelism. But a task-based implementation allows an iteration to be started before all the GEMM updates are finished. Nodes on one slice can thus start working while tasks of the previous slice are still ongoing, and parallelism can be achieved. We show in Section V that our $2.5D$ implementation achieves higher performance compared to the simple $2D$ approach.

A. Communication volume with limited memory

Let us consider a $2.5D$ **SBC** distribution as defined above: c slices, each containing a basic $2D$ **SBC** distribution with parameter r. This distribution involves $\frac{r^2}{2} \cdot c$ nodes. We denote again the total size of matrix $\mathbf{A}$ as $S = \frac{N(N+1)}{2}$. As mentioned in Theorem 1, each tile is sent to $r-1$ nodes, so the communication volume for the intermediate results is $D_1 = S(r-1)$. In addition, each tile of the result is replicated c times, and needs to be aggregated on one node, resulting in $D_2 = S \cdot (c-1)$ communication volume. The total is thus $D = D_1 + D_2 = S(r+c-2)$.

Let us consider that we have P nodes with an amount of memory M per node, and that the dimension N of the matrix grows to infinity while $M = o\left(\frac{N^2}{P^{2/3}}\right)$. We can (in a way similar to [9]) use as many slices as memory size allows, *i.e.* set c to $\frac{PM}{S}$. Since $S \sim \frac{N^2}{2}$, this gives $c \sim \frac{2PM}{N^2}$, and $r = \sqrt{2\frac{P}{c}} \sim \frac{N}{\sqrt{M}}$. We thus have $D_1 \sim \frac{N^3}{2\sqrt{M}}$ and $D_2 \sim \frac{PM}{N^2}$. The assumption on M ensures that the overall data movement is dominated by D_1, and asymptotically, we obtain

$$D \sim \frac{1}{2} \cdot \frac{N^3}{\sqrt{M}} + o(N^3).$$

This can be compared to the previous result by Kwasniewski *et al.* [9], who prove that their $2.5D$ block-cyclic algorithm performs a volume of communication of $\frac{N^3}{\sqrt{M}} + o(N^3)$: our result is a factor of 2 improvement.

B. Number of slices with large memory

If M is large enough, using as many slices as possible may result in a too large communication volume for the reductions. In this section we assume that the P nodes have sufficient memory, and we search for the value of c that achieves the minimum communication volume. Since $D = S(r+c-2)$ and $2P = r^2c$, we want to minimize $r+c$ subject to $r^2 \cdot c \geq 2P$. We can write Karush-Kuhn-Tucker necessary conditions: if (r,c) is a local optimum to this optimization problem, then $\exists u,\ 1 + u \cdot 2rc = 0$ and $1 + u \cdot r^2 = 0$.

We obtain $u = \frac{-1}{r^2}$, and then $r = 2c$. Plugging this into $2P = r^2c$, we get $c \sim \sqrt[3]{P/2}$ and $r \sim 2\sqrt[3]{P/2}$. This yields the following total communication volume

$$D \sim 3\sqrt[3]{1/2} \cdot S\sqrt[3]{P}$$

In the same context, a $2.5D$ block-cyclic distribution has $D = S(p+q+c-3)$ and $P = pqc$, thus the parameters that minimize the communication cost are $p = q = c = \sqrt[3]{P}$, which yields a total communication cost $D_{\text{2.5DBC}} = 3 \cdot S\sqrt[3]{P}$. **SBC** provides an improvement on the communication volume of a factor of $\frac{3}{3\sqrt[3]{1/2}} = \sqrt[3]{2} \simeq 1.26$ over the $2D$ block-cyclic distribution. Furthermore, this is achieved with a lower value of c and thus requires a lower amount of memory, again by a factor of $\sqrt[3]{2}$.

V. Experiments

A. Experimental Setup

All experiments are performed in double precision on the **bora** cluster of PlaFRIM[2], which contains 42 nodes each with 36 *Intel Xeon Skylake Gold 6240* cores, for a total of 1512 cores. The nodes are connected with a 100Gb/s OmniPath network. We use Intel MKL 2020, `Open MPI` version 4.0.3, `StarPU` version 1.3.8, and `Chameleon` version 1.1.0. The hardware peak for one core in this setup is estimated as follows: 2.6 GHz (core base frequency) $\times$ 8 (double-precision Flop per cycle) $\times$ 2 (FMA feature). It yields a maximum rate of 41.6 GFlop/s for each core, equivalent to 1497.6 GFlop/s for one node (36 cores) and 1414.4 GFlop/s for 34 cores.

We consider three different operations: the main one is Cholesky factorization (POTRF), but we also evaluate the performance of **SBC** for two other operations which make use of the Cholesky factorization: linear system solving (POSV) and matrix inversion (POTRI). For each operation and each allocation scheme, a random symmetric positive definite matrix $\mathbf{A}$ is generated (along with a matrix $\mathbf{B}$ as right-hand-side for POSV), and distributed among nodes according to the given allocation scheme. Once the matrix has been generated and distributed, the computing time to perform the operation is measured, and the process is repeated 5 times to ensure a good reproducibility of the results.

We ran 50000×50000 Cholesky factorizations on a single node using different tile sizes from 100 to 1000 to determine the most appropriate for this operation and this hardware. As shown on Figure 7, almost maximum performance is reached as soon as tile size is at least 500×500. Therefore we use tile size $b = 500$ for all subsequent experiments in order to maximize computation throughput and parallelism.

Throughout the rest of the paper and in all experiments with **SBC**, all individual tasks are executed with this block size and on a single core. In the experiments presented, the only parameter to tune on a new architecture is the tile size.

We consider matrix sizes ranging from $N = 25$ tiles (corresponding to $n = 12,500$ elements) to $N = 600$ tiles ($n = 300,000$) for POTRF and POSV operations. The POSV operation is performed using a right-hand-side matrix $\mathbf{B}$ of size $n \times b$: matrix $\mathbf{B}$ is one tile wide.

B. Allocation schemes

We compare different allocation schemes within the `Chameleon` framework:

[2] https://plafrim.fr

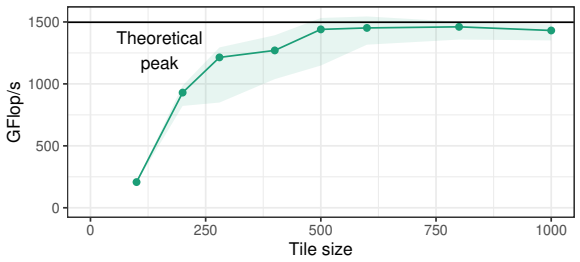

Fig. 7: Performance of Cholesky factorization over a matrix of size $n = 50000$ using `Chameleon-StarPU` on a single node for different tile size.

Symmetric Block Cyclic		2D Block Cyclic		
r	P	p	q	P
6	15	5	3	15
		4	4	16
7	21	5	4	20
		7	3	21
8	28	7	4	28
		6	5	30
9	36	7	5	35
		6	6	36

TABLE I: Sizes of the considered distributions

- The extended **SBC** distribution, with values of r ranging from 6 to 9, which respectively correspond to a number of nodes $P = 15$, 21, 28 and 36. Since each node contains 36 cores, the number of cores ranges from 510 to 1224.
- The $2D$ block-cyclic (**2DBC**) allocation scheme, implemented by default in `Chameleon`, featuring two options with a similar number of nodes, in order to cover the best possible parameters p and q. This ensures that we avoid any unfairness based on a choice of P that would be ill adapted to the **2DBC** scheme (for example for $r = 7$, using $P = 21$ forces to use a 7×3 pattern, whereas a 5×4 pattern is closer to a square and thus generates fewer communications). The parameters used are summarized on Table I.
- The corresponding $2.5D$ variants of both **SBC** and block-cyclic, as described in Section IV.

The Chameleon library was shown to be very competitive with other state-of-the-art implementations by Agullo et al [18]. In these experiments, we focus on the benefits of our newly proposed data distribution, compared to the standard **2DBC** distribution. In addition, we also compare with the recently proposed `COnfCHOX` library. `COnfCHOX` is able to use a $2.5D$ distribution, but in our cases, its automatic parametrization selects a $2D$ distribution. The experiments on our platform confirmed that the $2.5D$ distributions resulted in worse performance than the plain $2D$ distribution. This is why we only present results for `COnfCHOX` with a $2D$ block-cyclic allocation.

C. Task-based implementation

As mentioned above, we have implemented the **SBC** distribution within the `Chameleon` library[3], on top of the `StarPU` runtime system[4] [1]. This combination offers a high level of abstraction to the end user by automatically managing dependencies between tasks and the associated data movements. Our implementation of the $2D$ **SBC** distribution thus simply consists in providing a function to specify the node responsible for each tile of the input matrix. The $2.5D$ variants of **SBC** and **2DBC** are obtained by inserting explicit reduction operations before each TRSM and POTRF tasks in Algorithm 1. When performing an operation such as Cholesky factorization using this distribution, the `Chameleon` library submits the corresponding tasks to the runtime system along with the access mode (read and/or write) associated to each input and output tile. From this description of data dependencies, `StarPU` then infers task dependencies and the necessary communications.

The implementation used for the experiments relies on the MPI library, with one MPI process on each node. Each MPI process uses the `StarPU` runtime system to orchestrate all the tasks performed by this node, so that each node can perform several tasks in parallel.

At the level of one node, `Chameleon` and `StarPU` apply dynamic scheduling strategies to distribute the tasks among workers, and are thus able to manage heterogeneous ressources (CPUs and GPUs). Each elementary task assigned to a node is performed sequentially by a worker (a CPU core or a GPU). In our experiments however, only CPU cores are used, so that all tasks are performed at the scale of a single core. The efficiency of the computation at the node level comes from all those elementary tasks performed in parallel by all cores. Also note that in the following, the possibility of using several cores in parallel for one task (in case of lack of work in particular) is not used.

At the platform level, tasks are assigned to nodes via the *owner computes* rule and inter-node communications that occur are handled asynchronously by `StarPU`. Besides, the current `Chameleon` implementation does not make use of complex collective communication schemes: each inter-node communication uses a point-to-point MPI communication operation. Our theoretical analysis of communication volume thus matches the actual experimental setup. However, the dynamic scheduling policies of `StarPU` result in a large amount of communication time overlapped by computation, which yields a high level of parallelism.

In `StarPU`, the management of tasks (submission and scheduling) and the management of MPI communications are handled by one core reserved for each, so on our platform 34 cores remain available for actual computations. More details about the `Chameleon` library, and in particular the distributed aspects, can be found in [18].

The flexibility and high-level view provided by the runtime system has an additional benefit: when performing several op-

[3] https://project.inria.fr/chameleon/
[4] https://starpu.gitlabpages.inria.fr/

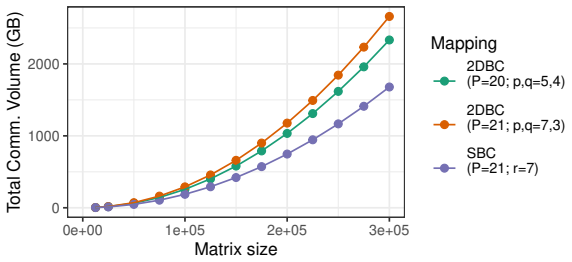

Fig. 8: Measured volume of inter-node communication during POTRF, for $P = 20$ and 21 (one tile of dimension $b \times b$ in double precision is 2 MB).

erations in sequence (for example, the POSV operation is a sequence of POTRF and TRSM operations), no synchronization is needed between the operations. Instead, the task graphs of the operations are merged, and one operation can start working as soon as data is available, while the previous operation is still ongoing. This results in much higher parallelism. The same mechanism allows to transparently perform data redistribution between operations: this is done asynchronously and can be overlapped with the rest of the computation. We use this feature in the experiments for the POTRI operation.

D. Communication reduction

Figure 8 shows the evolution with input matrix size of the overall communication volume (in GBytes) when performing Cholesky factorization using **2DBC** or **SBC** data distribution. The test case used as illustration is $r = 7$. As expected, the Symmetric Block Cyclic distribution induces a significantly smaller communication volume, for all values of n, even for cases where **2DBC** uses fewer nodes.

The results shown in the figure are experimental results, but it is easy to compute analytically, for a given distribution, what volume of data is induced by SBC. Indeed, on the one hand, `Chameleon` and `StarPU` do not modify the initial placement of data and tasks (only their scheduling). On the other hand, they take care of the data movements in an asynchronous way and thus ensure an excellent overlap between calculations and communications. Note that all communications are done at the tile level, without additional optimizations (no detection of collective communications or message aggregation).

E. Performance results for Cholesky

Figure 9 shows the results obtained by all approaches for the $r = 8$ case, which corresponds to $P = 28$, where the $2.5D$ variants use $c = 3$ slices. The `COnfCHOX` library obtains significantly better performance with power-of-two number of nodes, so we present the results with $P = 32$ for this implementation. In all plots in this section, each point represents the average result over the 5 runs for each experiment, in terms of performance per node in GFlop/s (Figures 9, 10, 11, 13 and 14), total volume of inter-node communications in GB (Figure 8) and total running time (Figure 12). For each point the shaded zone shows the actual range of minimum to maximum values over the 5 runs.

Regarding performance evaluation, for an execution of duration t using P nodes, this value is given by $F = \frac{\#\text{flops}}{t \cdot P}$, where #flops is the number of flops for the factorization, that depends on the matrix size n. This allows to fairly compare results obtained by approaches using different numbers of nodes (in Figure 9, some allocations use 28 nodes while others use 30 nodes, so comparing execution times is not relevant). The *theoretical peak* is computed based on hardware specifications of the processors of the platform. A dotted line indicates the theoretical peak achievable by `StarPU`, by counting only the performance of 34 cores.

The `Chameleon` library clearly outperforms the `COnfCHOX` implementation, and manages to approach the peak performance for very large matrix sizes. The dynamic nature of the `StarPU` runtime system also results in a higher variability compared to the static `COnfCHOX` library. The comparison between the allocation schemes in `Chameleon` shows that reducing the communication volume increases the performance further; this is particularly true for intermediate values of N where the communication has the most impact (when N is large, the $O(N^3)$ computation cost overshadows the $O(N^2\sqrt{P})$ communication cost). In this particular case, the benefit gained from the **SBC** distribution over the **2DBC** distribution is similar to the benefit gained by the $2.5D$ approach (which **SBC** achieves without increasing the memory requirements). Furthermore, these benefits are not exclusive, and the $2.5D$ **SBC** distribution yields even better performance than all other schemes. In total, the $2D$ **SBC** distribution obtains up to 23% improvement over **2DBC** and the $2.5D$ **SBC** distribution achieves up to 11% improvement over $2D$ **SBC**. $2.5D$ **SBC** also achieves improvement of up to 27% over the standard **2DBC** distribution.

Figure 10 displays similar results for other values of r, focusing on the relative performance of **SBC** and **2DBC**. We can see that the improvement observed above is valid over all tested values of P. In the end, **SBC** has a much better scalability than **2DBC**: for a matrix size $n = 200,000$, **SBC** with $P = 36$ achieves roughly the same performance per node as **2DBC** with $P = 16$, as can be observed on Figure 11.

The evolution of the total running time against matrix size is depicted Figure 12 for the same values of P. The plots illustrate the overall reduction of running time achieved by **SBC** mapping compared to **2DBC**. Since the performance gain is limited for very large matrices, only results for $n \leq 200000$ are shown to highlight the differences.

F. Performance for other operations

Cholesky factorization is generally used as one stage of a multi-operations workflow applied to matrix $\mathbf{A}$. Common uses include solving linear systems, (POSV operation), and computing the inverse of the matrix (POTRI operation).

1) Solving linear systems: POSV aims at solving the linear system $\mathbf{A}x = \mathbf{B}$ for the unknown x where $\mathbf{A}$ is symmetric definite positive. Multiple right-hand-side can be gathered as

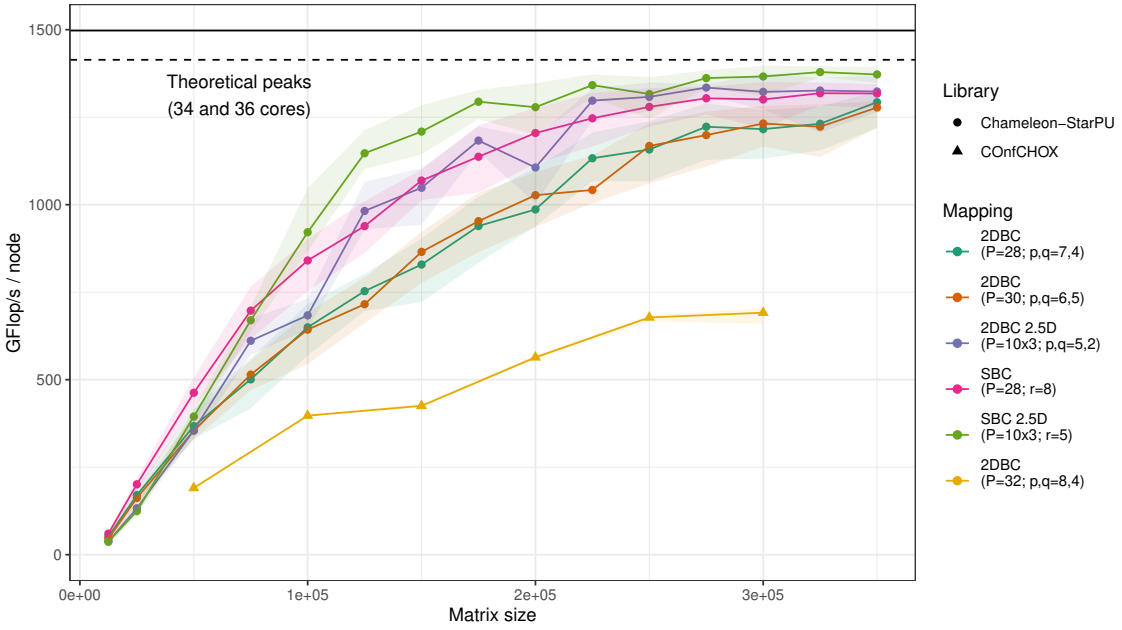

Fig. 9: Performance of Cholesky factorization using $2D$ and $2.5D$ versions of **BC** and **SBC** for $P = 28$, 30 and 32.

columns of $\mathbf{B}$ to perform several resolutions simultaneously. Operations on $\mathbf{B}$ are column-wise independent and thus totally parallel. Our test case uses a right-hand-side of dimensions $N \times 1$ tiles which is customary. POSV involves three steps:

1) Cholesky factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^T$: $\mathbf{A} \leftarrow \text{POTRF}(\mathbf{A})$
2) solve the system $\mathbf{L}y = \mathbf{B}$ in y: $\mathbf{B} \leftarrow \text{TRSM}(\mathbf{B}, \mathbf{A})$
3) solve the system $\mathbf{L}^T x = y$ in x: $\mathbf{B} \leftarrow \text{TRSM}(\mathbf{B}, \mathbf{A}^T)$

Task dependencies of TRSM require two types of data transfers at each iteration i: *(a)* each tile of column i of matrix $\mathbf{A}$ is necessary to compute the tile of $\mathbf{B}$ on the same row, *(b)* in $\mathbf{B}$, the tile on row i is necessary to update the values of all tiles on rows $j > i$.

The communication volume of the first one depends on the number of different nodes that owns tiles of $\mathbf{B}$ on the same row. The cost of the second only depends on the variety of nodes owning tiles in a given column of $\mathbf{B}$. Since $\mathbf{B}$ is one tile wide, the first type of communication dominates. Hence to limit the volume of communication, we use a $1D$ row-cyclic data allocation for the right-hand-side $\mathbf{B}$, regardless of the distribution used for $\mathbf{A}$.

The resulting performance for $r = 8$ is presented in Figure 13. We can observe that **SBC** achieves better performance than **2DBC** as well, however the improvement ratio is lower. This can be explained easily: the additional time (both in terms of communications and computations) induced by TRSM during POSV compared to POTRF alone is independent from the distribution of $\mathbf{A}$, so the benefit of using **SBC** is lower compared to the total execution time of the operation.

2) Inversion operation with data redistribution: POTRI is the operation used to compute the inverse of symmetric definite positive matrix $\mathbf{A}$. It is composed of three steps:
(i) Cholesky factorization $\mathbf{A} = \mathbf{L}\mathbf{L}^T$: $\mathbf{A} \leftarrow \text{POTRF}(\mathbf{A})$
(ii) triangular inversion, compute $\mathbf{L}^{-1}$: $\mathbf{A} \leftarrow \text{TRTRI}(\mathbf{A})$
(iii) symmetric matrix multiplication $\mathbf{A}^{-1} = (\mathbf{L}^{-1})^T\mathbf{L}^{-1}$: $\mathbf{A} \leftarrow \text{LAUUM}(\mathbf{A})$

LAUUM features the same data dependencies pattern as POTRF, so both operations induce the same communication volume for a given distribution scheme. However, the computations involved in TRTRI require nonsymmetric input: at iteration i, the computation for tile (x, y) with $x > i$ and $y < i$ requires data from tiles (i, y) and (x, i). These data dependencies make it necessary to broadcast to all nodes along a row and a column independently. Keeping the same notation as for Theorem 1, the leading term of communication volume when performing TRTRI is:

- $S(p+q-2)$ when using **2DBC** (asymptotically $2S\sqrt{P}$);
- $S(2r-2)$ with extended **SBC** (asymptotically $2\sqrt{2}S\sqrt{P}$).

Since **2DBC** generates a smaller communication volume than **SBC** for this operation, we consider a mixed strategy involving remapping of data between operations, denoted ***SBC remap 2DBC*** on Figure 14: POTRF and LAUUM are performed using **SBC** allocations while TRTRI is done with **2DBC**. Data redistribution of the whole matrix occurs before and after TRTRI to change the allocation. Higher order terms of the communication volume for the whole POTRI are then given by

- $3S(p+q-2) \sim 6S\sqrt{P}$ with **2DBC**;

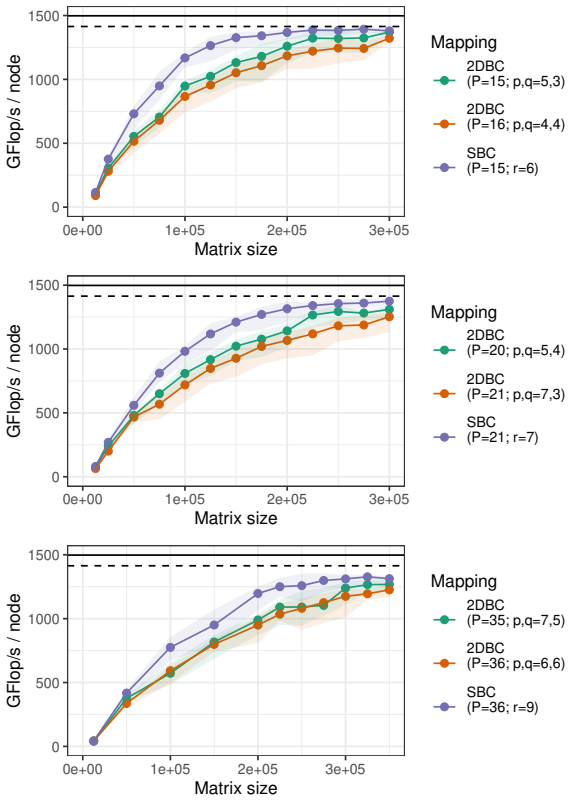

Fig. 10: Performance of Cholesky factorization (GFlop/s per node) using **2DBC** and **SBC**, for P ranging from 15 to 36.

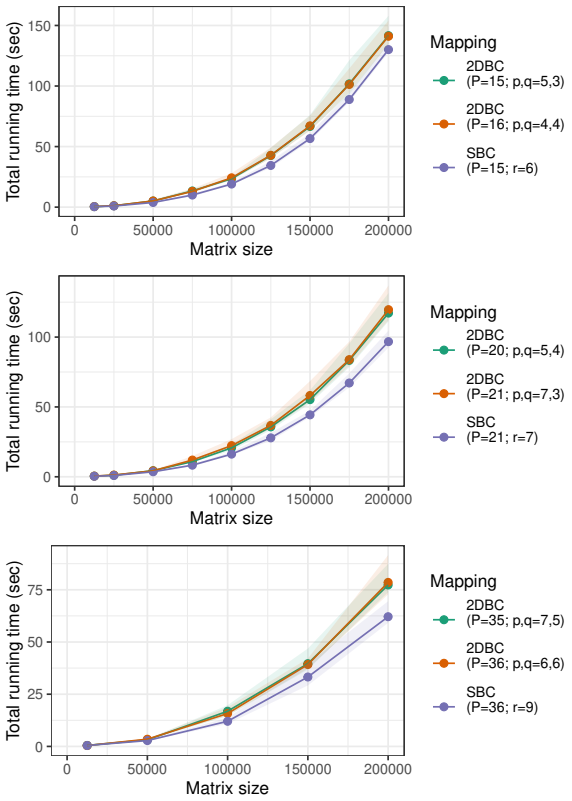

Fig. 12: Total running time of Cholesky factorization using **2DBC** and **SBC** allocations, for P ranging from 15 to 36.

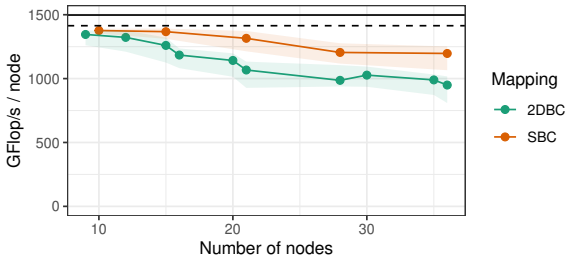

Fig. 11: Strong scaling of **2DBC** and **SBC** for $n = 200000$.

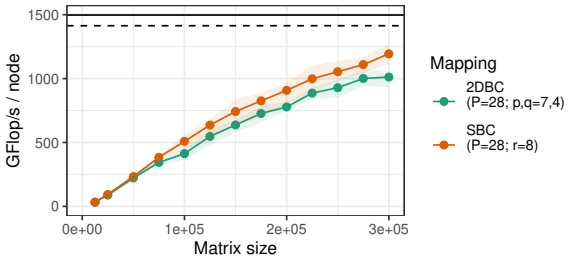

Fig. 13: Performance of POSV with **2DBC** and **SBC**, $P = 28$

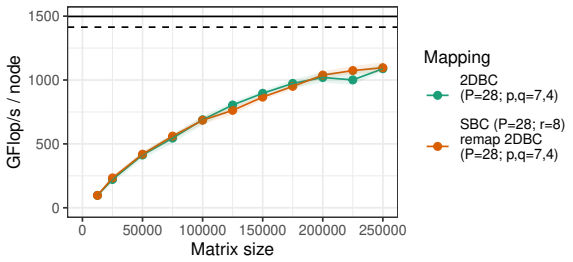

Fig. 14: Performance of POTRI using **2DBC** and **SBC** ($P = 28$).

- $S(2r+p+q-4) \sim 2(\sqrt{2}+1)S\sqrt{P}$ with remapped **SBC**.

The remapped version of **SBC** asymptotically generates a smaller communication volume than **2DBC**, though its relative advantage is smaller than for POTRF alone: the reduction ratio is $\frac{3}{\sqrt{2}+1} \simeq 1.24$ instead of $\sqrt{2}$. A small performance gain can thus be expected when comparing this strategy against **2DBC**. However, this asymptotic performance gain is achieved for values of P which are large enough so that the cost of redistribution (which does not depend on P) is negligible compared to the communication cost (which is proportional to $\sqrt{P}$). Figure 14 presents the case $r = 8$ ($P = 28$), $p = 7$ and $q = 4$, so that the communication volume is reduced by a factor of only $27/23 = 1.17$. In that case, the reduction in communication is too low to obtain a visible improvement in performance. Still, **SBC** reduces the communication volume (and thus energy consumption and contention with other applications) without degrading performance. In addition, this experimental result shows that **SBC** data allocation can be seamlessly integrated to multi-operation workflow via data redistribution, to reduce the communication volume on specific symmetric steps while leaving the others untouched, hence resulting in global better performance.

VI. Conclusion

In this paper, we study the Cholesky factorization and propose **SBC**, a data distribution scheme adapted to the symmetry in its data accesses. We show that **SBC** induces a smaller communication volume than the standard $2D$ block-cyclic distributions, increasing the arithmetic intensity by a factor of $\sqrt{2}$: the arithmetic intensity of Cholesky with **SBC** matches the arithmetic intensity of the LU factorization with **2DBC**. We also propose a $2.5D$ variant of **SBC** and prove that it results in a communication volume of $\frac{1}{2} \cdot \frac{n^3}{\sqrt{M}}$ with limited memory M, which improves over the previous best result by a factor of 2. We present experimental results obtained with the `Chameleon` library over the `StarPU` runtime system, and prove that **SBC** allows to obtain significantly improved performance for the Cholesky factorization. The high-level, task-based approach of the `Chameleon` library allows a seamless integration of this new distribution scheme without changing the Cholesky implementation. Our work highlights the importance to use a data distribution scheme adapted to the data access pattern. Experiments with the solve and inversion operations (POSV and POTRI) show that **SBC** can be used in a wide variety of operations. Furthermore, there is still a $\sqrt{2}$ gap between the arithmetic intensity of **SBC** and the upper bound for Cholesky factorization: it might be possible to design even more efficient data distribution schemes.

Acknowledgments

This work is supported in part by the Région Nouvelle-Aquitaine, under grant 2018-1R50119 "HPC scalable ecosystem", and by the ANR, under grant SOLHARIS - ANR-19-CE46-0009. Julien Langou was supported in part by NSF award #2004850.

References

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier, "StarPU: a unified platform for task scheduling on heterogeneous multicore architectures," *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009*, vol. 23, pp. 187–198, Feb. 2011.

[2] T. Cojean, A. Guermouche, A. Hugo, R. Namyst, and P.-A. Wacrenier, "Resource aggregation for task-based Cholesky factorization on top of modern architectures," *Parallel Computing*, vol. 83, pp. 73–92, 2019.

[3] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra, "DAGuE: A generic distributed DAG engine for high performance computing," *Parallel Computing*, vol. 38, no. (1-2), pp. 37–51, 2012.

[4] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet *et al.*, *ScaLAPACK users' guide*. SIAM, 1997.

[5] E. Hutter and E. Solomonik, "Communication-avoiding Cholesky-QR2 for rectangular matrices," in *2019 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2019, pp. 89–100.

[6] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Communication-optimal parallel and sequential Cholesky decomposition," *SIAM Journal on Scientific Computing*, vol. 32, no. 6, pp. 3495–3523, 2010.

[7] D. Irony, S. Toledo, and A. Tiskin, "Communication lower bounds for distributed-memory matrix multiplication," *Journal of Parallel and Distributed Computing*, vol. 64, no. 9, pp. 1017–1026, 2004.

[8] A. Olivry, J. Langou, L.-N. Pouchet, P. Sadayappan, and F. Rastello, "Automated derivation of parametric data movement lower bounds for affine programs," in *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2020, pp. 808–822.

[9] G. Kwasniewski, M. Kabic, T. Ben-Nun, A. N. Ziogas, J. E. Saethre, A. Gaillard, T. Schneider, M. Besta, A. Kozhevnikov, J. VandeVondele, and T. Hoefler, "On the parallel i/o optimality of linear algebra kernels: Near-optimal matrix factorizations," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, ser. SC '21. New York, NY, USA: Association for Computing Machinery, 2021. [Online]. Available: https://doi.org/10.1145/3458817.3476167

[10] J. Hong and H. T. Kung, "I/O complexity: The red-blue pebble game," in *STOC '81*, 1981.

[11] G. Ballard, J. Demmel, O. Holtz, and O. Schwartz, "Minimizing communication in numerical linear algebra," *SIAM J. Matrix Anal. Appl.*, vol. 32, pp. 866–901, 2011.

[12] M. Christ, J. Demmel, N. Knight, T. Scanlon, and K. A. Yelick, "Communication lower bounds and optimal algorithms for programs that reference arrays - part 1," EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2013-61, May 2013. [Online]. Available: http://www2.eecs.berkeley.edu/Pubs/TechRpts/2013/EECS-2013-61.html

[13] O. Beaumont, L. Eyraud-Dubois, M. Vérité, and J. Langou, "I/O-optimal algorithms for symmetric linear algebra kernels," 2022, arXiv preprint. [Online]. Available: https://arxiv.org/abs/2202.10217

[14] N. Béreux, "Out-of-core implementations of Cholesky factorization: Loop-based versus recursive algorithms," *SIAM Journal on Matrix Analysis and Applications*, vol. 30, no. 4, pp. 1302–1319, 2009. [Online]. Available: https://doi.org/10.1137/06067256X

[15] E. Solomonik and J. Demmel, "Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms," 01 2011, pp. 90–109.

[16] D. Irony and S. Toledo, "Trading replication for communication in parallel distributed-memory dense solvers," *Parallel Processing Letters*, vol. 12, no. 01, pp. 79–94, 2002. [Online]. Available: https://doi.org/10.1142/S0129626402000847

[17] E. Georganas, J. González-Domínguez, E. Solomonik, Y. Zheng, J. Tourino, and K. Yelick, "Communication avoiding and overlapping for numerical linear algebra," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–11.

[18] E. Agullo, O. Aumage, M. Faverge, N. Furmento, F. Pruvost, M. Sergent, and S. P. Thibault, "Achieving high performance on supercomputers with a sequential task-based programming model," *IEEE Transactions on Parallel and Distributed Systems*, 2017.