On the Arithmetic Intensity of Distributed-Memory Dense Matrix Multiplication Involving a Symmetric Input Matrix (SYMM)

Emmanuel Agullo, Alfredo Buttari, Olivier Coulaud, Lionel Eyraud-Dubois, Mathieu Faverge, Alain Franc, Abdou Guermouche, Antoine Jego, Romain Peressoni, Florent Pruvost

On the Arithmetic Intensity of Distributed-Memory Dense Matrix Multiplication Involving a Symmetric Input Matrix (SYMM)

Emmanuel Agullo [1] Alfredo Buttari [2] Olivier Coulaud [1]
Lionel Eyraud-Dubois [1] Mathieu Faverge [7] Alain Franc [6]
Abdou Guermouche [4] Antoine Jego [5] Romain Peressoni [1]
Florent Pruvost [1]

Inria[1], Université de Bordeaux[4], Bordeaux INP[7] - LaBRI
{firstname.lastname}@inria.fr
CNRS[2], INPT[5] - IRIT
{firstname.lastname}@irit.fr
INRAE[6] - BioGeCo, Inria
{firstname.lastname}@inrae.fr

Dense matrix multiplication involving a symmetric input matrix (SYMM) is implemented in reference distributed-memory codes with the same data distribution as its general analogue (GEMM). We show that, when the symmetric matrix is dominant, such a 2D block-cyclic (2D BC) scheme leads to a lower arithmetic intensity (AI) of SYMM than that of GEMM by a factor of 2. We propose alternative data distributions preserving the memory benefit of SYMM of storing only half of the matrix while achieving up to the same AI as GEMM. We also show that, in the case we can afford the same memory footprint as GEMM, SYMM can achieve a higher AI. We propose a task-based design of SYMM independent of the data distribution. This design allows for scalable A-stationary SYMM with which all discussed data distributions, may they be very irregular, can be easily assessed. We have integrated the resulting code in a reduction dimension algorithm involving a randomized singular value decomposition dominated by SYMM. An experimental study shows a compelling impact on performance.

1 Introduction

The matrix multiplication involving a symmetric input matrix is a crucial mathematical kernel in many important numerical algorithms. It is, for instance, the dominant kernel when solving symmetric linear systems with multiple right-hand sides [19] or related eigenvalue problems [14] and we refer the reader to [23] and references therein for more details on "block" and "augmented" Krylov subspace methods. It is also the most dominant kernel for randomized algorithms [15, 18], an important class of numerical algorithms which became very popular in the last decade, when dealing with symmetric matrices. In the particular case where the involved matrices are dense, the operation is commonly referred to as the symmetric matrix-matrix (SYMM) product and consists in computing $C \leftarrow \alpha AB + \beta C$, or $C \leftarrow \alpha BA + \beta C$, where α and β are scalars, A is a symmetric m-by-m dense matrix, and B and C are m-by-n dense matrices, or n-by-m dense matrices,

respectively.

While its general (GEMM) counterpart – *i.e.*, not assuming A is symmetric (nor even square) – has been the focal point of many meticulous studies [1, 27, 24, 17], relatively little attention has been devoted to handle the specific features of SYMM in a distributed-memory context. As a consequence, its implementation in reference codes such as ScaLAPACK [10] or Elemental [21] follows the same parallel design as GEMM, consisting in a 2D block-cyclic (2D BC) data distribution. This paper focuses on the case where $m \gg n$, which is the most common case in the important algorithms mentioned above. We show that, though it may seem counter intuitive, the arithmetic intensity (AI) – sometimes also referred to as the operational intensity – of reference implementations of SYMM – following a 2D BC data distribution – is lower than that of GEMM. We then show that alternative data distributions may improve the AI of SYMM up to achieving an equally high AI as GEMM while maintaining its memory advantage of storing only half of the dominant m-by-m matrix. We also show that with the same amount of memory as GEMM, using an extension inspired by the 2.5D algorithms [1, 25, 24] (also referred to as 3D in the literature) allows SYMM to achieve a higher AI than GEMM. In order to assess whether the higher AI may translate into a higher performance, the resulting algorithms have all been implemented in the Chameleon dense linear algebra library [2] following the task-based design proposed in [4]. This work only considers classical algorithms with a cubic computational complexity; Strassen-like variants [7] that can lead to reduced computation and communication volume are out of the scope of this paper.

The present work was initially motivated by a recent study on a dimension reduction algorithm whose dominant step is the matrix multiplication [5]. The study showed that a choice had to be made to perform the matrix multiplication: favor either performance, through a GEMM, or memory, through a SYMM. The new approach proposed in the present manuscript makes it possible to combine both these desirable properties. The remainder of the paper is organized as follows. We present related work in Section 2. We then analyze the AI of reference distributed 2D BC GEMM and SYMM algorithms as well as that of our proposals for new SYMM data distributions in Section 3. We present the implementation of our algorithms in Section 4 and assess the resulting communication volume and performance in Section 5. We eventually show the impact on randomized singular value decomposition (RSVD) and multidimensional scaling (MDS) algorithms – the application that originally motivated this study – in Section 6.

In this paper, we make the following contributions:

- we review several data distribution schemes for the A-stationary SYMM operation and analyze their communication volume;
- we propose an original adaptation of the TBS algorithm from [9] to a distributed-memory setting;
- we propose a scalable task-based implementation of SYMM independent of data distributions; and we experimentally assess the performance gained by lowering the communication volume;
- finally, we validate these gains on a dimension reduction algorithm in the context of a metabarcoding application.

2 Related work, Background

2.1 Multidimensional scaling and randomized singular value decomposition

Dimension reduction algorithms aim at transforming data from a high-dimensional space into a low-dimensional space while keeping the most meaningful properties of the original data. While the most well-known of these algorithms is certainly the principal component analysis (PCA), MDS [26] may be viewed as its analogue when data items are only known through their respective dissimilarities. MDS searches for a low dimensional space, in which points in the space represent the items, one point representing one item, and such that the distances between the points in the space match, as well as possible, the original dissimilarities.

From a numerical point of view, MDS resorts to processing a singular value decomposition (SVD), as PCA does. However, contrary to PCA, MDS uses an input matrix G built from representing dissimilarities between pairs of items and often referred to as the *Gram matrix*. The dissimilarity between pairs of items being a symmetric relation, the input matrix G is itself symmetric. As a consequence, the SVD of G is also its eigenvalue decomposition (EVD) up to the sign of the eigenvalues. We pursue the presentation with the SVD terminology, following [11].

When dealing with large data sets, performing an SVD may be out of reach due to memory or time to solution constraints. A major step forward has been the design of randomized SVD (RSVD) algorithms [22, 15], a fast and probabilistic approach which ensures the quality of the solution via random projections. Its usage within the MDS (RSVD-MDS) [11, 20] has allowed for processing large data sets while preserving the numerical robustness of the standard SVD-MDS [20]. The main idea of the RSVD (here discussed when applied to G) is to approximate the column space of the m-by-m matrix G by only a small number n (such that $m \gg n$) of vectors through a linear combination of the columns. From a computational point of view, this step consists in forming an m-by-n random matrix Ω and perform the $Y \leftarrow G\Omega$ matrix product. After computing an orthonormal basis Q of Y, we compute the $Z \leftarrow GQ$ matrix product and, then, a deterministic SVD only needs to be performed on the tall and skinny m-by-n Z matrix. A complete description is provided in Algorithm 2 in [5].

As discussed in details in [5], the $Y \leftarrow G\Omega$ and $Z \leftarrow GQ$ matrix products are the dominant steps of both the RSVD and the whole RSVD-MDS algorithms. In the remainder of this manuscript, we will refer to them as matrix multiplication 1 (MM1) and matrix multiplication 2 (MM2), respectively. They have the exact same dimensions and can both be viewed in terms of the more common BLAS notations as a $C \leftarrow AB$ matrix product, where A is a symmetric m-by-m dense matrix, and B and C are both m-by-n dense matrices, with $m \gg n$. In order to maximize performance, both matrix products may be performed with a GEMM; however this implies that the dissimilarity matrix, initially stored in a symmetric format, has to be converted to full format, thus doubling the initial memory footprint [5]. The central question we aim at addressing is whether, in a distributed-memory context, we can store such a symmetric matrix A in symmetric format to use SYMM while achieving comparable performance to GEMM.

2.2 Distributed-memory algorithms for matrix multiplication

The general matrix matrix multiplication (GEMM) – *i.e.* not assuming A is symmetric (nor even square) – has been the focal point of many meticulous studies [1, 27, 24, 17].

On the other hand, relatively little attention has been devoted to handling the specificity of SYMM in a distributed-memory context. As a consequence, its implementation in reference codes such as ScaLAPACK [10] or Elemental [21] follows the same parallel design as GEMM, relying on a 2D BC data distribution.

Out of the matrix multiplication context, Beaumont et al. recently proposed to exploit the symmetry of matrices to enhance the distributed-memory Cholesky factorization of dense symmetric positive definite matrices [8]. The main idea is to rely on an alternative data distribution referred to as symmetric block cyclic (SBC). Still in the context of the Cholesky factorization, but in a sequential out-of-core setting, Beaumont et al. proposed in another study a variant referred to as Triangular Block Syrk (TBS) [9], and provided sharp bounds showing that TBS achieves the lowest possible I/O volume for this operation. However, since it does not readily apply to a distributed-memory context in the case of a Cholesky factorization, TBS was only considered in a sequential setting.

The present paper shows that in the case of interest here ($m \gg n$), the state-of-the-art 2D BC SYMM achieves a lower AI than 2D BC GEMM, theoretically confirming the empirical observations reported in [5], which originally motivated our study. It also revisits SBC [8] in the case of the distributed-memory matrix multiplication, and proposes Triangular Block Cyclic (TBC), a distributed-memory adaptation of the ideas behind TBS [9]. Both the theoretical analysis (Section 3) and the experimental evaluation (Section 5) show that these new data distributions significantly improve above the state-of-the-art 2D BC SYMM, eventually allowing us to solve the memory / performance original discrepancy for both the RSVD and RSVD-MDS algorithms from [5] (Section 6).

2.3 Task-based programming model

We will show in Section 3 that our new distributed-memory algorithms based on SBC and TBC are very appealing from a theoretical point of view. However, they lead to irregular data mappings, making their implementation potentially very tedious. Additionally, most widely used dense linear algebra libraries rely on the well known 2D BC data distribution. This data distribution is hardwired in the code and the expression of algorithms is tightly bound to its use. This implies that changing the data distribution in these libraries would essentially require a complete rewriting of algorithms. To overcome these issues, we investigate the use of task-based parallel programming to implement the proposed algorithms with a high level of abstraction, combined with the use of a generic runtime system which is in charge of the data and tasks management on the underlying architecture.

More specifically, we have chosen to rely on the sequential task flow (STF) model [3, 4], where tasks are created sequentially and their mutual dependencies are automatically inferred through an analysis of their data access mode. This model is, for example, available in OpenMP (through the `task` directive and the `depend` clause), OmpSs [13] or StarPU [6], the runtime system we use in this work. The effectiveness of this model on shared memory parallel computers has been proved by numerous works although it has been much more rarely considered in distributed-memory settings; early work on this topic is presented by YarKhan [28] and Agullo et al. [3]. More recently, an extension of the STF model was proposed by Agullo et al. [4] that allows for portable implementations of scalable algorithms for distributed-memory computers; we rely on this approach for the implementation of the proposed SYMM algorithms, as discussed in Section 3.

	Scheme	P	S	Q$/(mn)$	AI
1.	$\mathcal{G}$ 2DBC(p,q)	pq	$\frac{m^2}{P}$	$(p+q-2)$	$\frac{m}{\sqrt{P}} = \sqrt{S}$
2.	$\mathcal{S}$ 2DBC(p,q)	pq	$\frac{m^2}{2P}$	$2(p+q-2)$	$\frac{m}{2\sqrt{P}} = \sqrt{S/2}$
3.	$\mathcal{S}$ SBC(r)	$r^2/2$	$\frac{m^2}{2P}$	$2(r-1)$	$\frac{m}{\sqrt{2P}} = \sqrt{S}$
4.	$\mathcal{S}$ TBC(c)	$c(c+1)$	$\frac{m^2}{2P}$	$2c$	$\frac{m}{\sqrt{P}} = \sqrt{2S}$

Table 1: Discussed GEMM (denoted $\mathcal{G}$ in the table) and SYMM (denoted $\mathcal{S}$) A-stationary schemes together with their communication volume Q and AI. P denotes the number of nodes, S the storage per node. The communication volume Q is expressed as a factor of mn (which is the common size of matrices B and C).

3 Data distributions for SYMM

In this section, we present different data distributions for the $C \leftarrow \alpha AB + \beta C$ matrix product, in increasing order of complexity and AI. These results are summarized in Table 1. We remind that we assume $m \gg n$, i.e., the matrix A is much larger than both B and C, in which case A-stationary schemes – further introduced in Section 3.1 – are the best approaches to minimize communication volume.

We start by the easiest case: if the complete matrix A is stored, the best solution is the 2D BC distribution (line 1 in Table 1, and Section 3.2), and we analyze its communication volume. We then specialize to the case where only half of matrix A is stored (because of symmetry). We use the same analysis to show that the communication volume of the 2D BC distribution (line 2, and Section 3.3.1) is twice larger than in the previous case. We then describe two symmetric distributions: first SBC [8] (line 3, and Section 3.3.2), whose communication volume is lower by a factor of $\sqrt{2}$, then TBC (line 4, and Section 3.3.3), which we adapt from [9], whose communication volume is lower by another factor of $\sqrt{2}$. In total, SYMM with the TBC distribution achieves the same communication volume as 2D BC when the whole matrix is stored, thus saving a factor of 2 on storage. Section 3.4 summarizes these results and also proposes another interpretation when the memory is bounded. Section 3.5 extends the analysis to the 2.5D case [1, 24].

3.1 Generalities

Since A is large, the best solution is to use an A-stationary algorithm: the computations are performed on the node that owns the corresponding block (also referred to as tile or submatrix) of A. In such an algorithm, the blocks of B are broadcast to the nodes that require them, and we denote Q^B the corresponding quantity of data transferred. Several nodes compute updates for a given block of C, and these updates are then reduced to the corresponding node. The communication volume for these reduce operations is denoted Q^C. The total communication volume for the multiplication is $Q = Q^B + Q^C$.

With this total communication volume Q, we can also compute the Arithmetic Intensity (AI), defined as

$$\text{AI} = \text{flop}/Q, \tag{1}$$

where flop is the total number of floating point operations. The number of floating point operations does not depend on the allocation, it is $2m^2n$ in all cases: one multiplication and one addition for each product computed. Hence, the AI is inversely proportional to the communication volume Q, and varies like $\frac{m}{\sqrt{P}}$ for all 2D distributions (see left part

Figure 1: Communications incurred with an A-stationary matrix multiplication, with a 2D BC $(2, 4)$ distribution. **Left:** storing the whole matrix A. One block of B follows the red path and is sent to $p - 1 = 1$ nodes. A result computed on a row of A is involved in a reduction operation on q nodes (blue path), resulting in $q - 1 = 3$ messages sent. **Right:** storing the lower half of A. Both types of communication now involve $p + q - 1 = 5$ nodes, resulting in 4 messages sent. Parts of the matrix where fewer nodes are involved are highlighted in grey.

of the AI column in Table 1). However, we can also express AI as a function of the memory size of one node, denoted as S; this allows one to measure how efficient an algorithm is at using the values stored in memory. For all 2D distributions studied here, $\text{AI} = \Theta\left(\sqrt{S}\right)$; the efficiency of an algorithm is measured by how large the constant is, shown on the right part of the AI column in Table 1.

3.2 A-stationary, general matrix multiplication (GEMM)

We first consider the situation where the whole matrix A is stored, and distributed among the nodes in a 2D BC (p, q) fashion. This situation is depicted on the left of Figure 1.

Consider a given column of matrix A, the corresponding values are owned by a set of p nodes. Each of these nodes must receive all values in the corresponding row of B, which is owned by another set of nodes. The best possible case is that the second set is included in the first one: in that case, each value of B must be sent to $p - 1$ nodes, and this incurs a communication volume of $n(p - 1)$. Since there are m columns in A, in total we get $Q^B = mn(p - 1)$ (in the worst case, the set of nodes that own the blocks of B is disjoint from the set of nodes that own the blocks of A, and we get $Q^B = mnp$).

Similarly, consider a given row of matrix A. Since the nodes that own this row need to perform one reduction per column of C to send the total to the owner of the corresponding block in C, the total communication volume for the blocks of C is $Q^C = mn(q - 1)$ in the best case, and $Q^C = mnq$ in the worst case (when the owner of a block in C never belongs to the corresponding set of nodes in the row of A).

The best case can be achieved if C is distributed with the same (p, q) 2D distribution, and B is distributed with the transpose (q, p) distribution.

In total, the communication volume is $Q^{p,q}_{\text{GEMM}} = mn(p + q - 2)$. In practice, we often choose $p \simeq q \simeq \sqrt{P}$, so that $Q^{\text{2DBC}}_{\text{GEMM}} \simeq 2mn(\sqrt{P} - 1)$. Asymptotically, the AI is $\text{AI}^{\text{2DBC}}_{\text{GEMM}} \simeq \frac{2m^2n}{2mn\sqrt{P}} = \frac{m}{\sqrt{P}}$, with a memory usage $S = \frac{m^2}{P}$, which yields $\text{AI}^{\text{2DBC}}_{\text{GEMM}} \simeq \sqrt{S}$. This result is summarized in Table 1, line 1.

3.3 A-stationary, symmetric case (SYMM)

We now assume that A is symmetric and that we store only (the lower) half of the matrix.

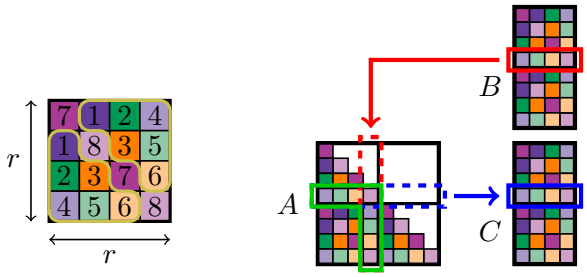

Figure 2: Symmetric Block Cyclic distribution. **Left:** the pattern with $r = 4$, using $P = 8$ nodes. The symmetrical lower and upper parts are highlighted. **Right:** communications induced when using SBC. Communications related to a row of matrices B and C both involve r nodes, resulting in $r - 1 = 3$ messages sent.

3.3.1 Standard 2D block-cyclic distribution

In this case, the 2D BC distribution is only applied to the lower half of the matrix, the upper tiles are not being stored at all. The result is depicted on the right of Figure 1. We can apply the same kind of reasoning as for the previous case. However, now the set of nodes that own a given column of A can be of size[1] up to $p+q-1$: the p nodes that own the (truncated) column, plus the q nodes that own the (truncated) row that completes the column (the total is $p+q-1$ because one node belongs to both the row and the column). Again, the best case is when the set of nodes that own the blocks of B is included in these $p+q$ nodes, and this yields a communication volume $Q^B = mn(p+q-2)$. Similarly, we get $Q^C = mn(p+q-2)$.

In total, $Q^{p,q}_{\text{SYMM}} = 2mn(p+q-2)$. As we can see, the communication volume is twice as large as in the previous case, and can be written as $Q^{\text{2DBC}}_{\text{SYMM}} \simeq 4mn(\sqrt{P}-1)$. Asymptotically, the AI is $\text{AI}^{\text{2DBC}}_{\text{SYMM}} \simeq \frac{2m^2n}{4mn\sqrt{P}} = \frac{m}{2\sqrt{P}}$, with a memory usage $S = \frac{m^2}{2P}$, which yields $\text{AI}^{\text{2DBC}}_{\text{SYMM}} \simeq \sqrt{S/2}$. This result is summarized in Table 1, line 2. As we can see, the AI is twice smaller compared to the previous case, but since the memory usage is also smaller by a factor of 2, the AI expressed as a function of S is only lower by a factor of $\sqrt{2}$. As discussed in more details in Section 3.4, this means that if the memory of the nodes is the limiting factor, storing half the matrix allows to use half as many nodes, which partially offsets the overhead in terms of communication volume.

3.3.2 Symmetric Block Cyclic distribution

In order to reduce the amount of communication, we need to make sure that the nodes that own a truncated column of A are the same as the nodes that own the corresponding truncated row. The Symmetric Block Cyclic distribution (SBC) has been proposed in the context of the Symmetric Rank-k update (SYRK) and the Cholesky factorization [8], where a similar issue appeared. We describe here the basic version of SBC, defined for an even integer $r > 2$ (see Figure 2). It consists of a symmetric $r \times r$ pattern with $P = r^2/2$ nodes: $\frac{r(r-1)}{2}$ nodes are organized arbitrarily in one half of the pattern, and symmetrically on the other half. The remaining $\frac{r}{2}$ nodes are each placed on two locations in the diagonal.

[1]The first q (respectively the last p) columns of A involve a slightly smaller number of nodes, because not all nodes appear in the truncated row (respectively column). The corresponding zones are highlighted in grey on the right of Figure 1. However, since we are interested in large matrices A where $m \gg p, q$, we decide to neglect this effect.

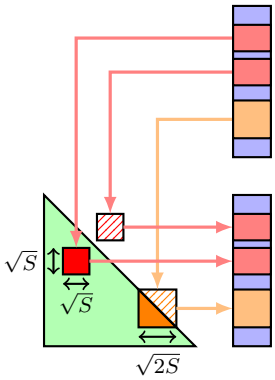

Figure 3: Triangles along the diagonal use fewer communications: both the red square and the orange triangle contain S elements. However, the corresponding operations for the square require $2\sqrt{S}$ rows of B, and only $\sqrt{2S}$ rows for the triangle.

For any i, the row i and the corresponding column i of the pattern contain the same set of r nodes. We can compute the amount of data transferred involved by using this distribution in an A-stationary SYMM operations: each block of B is sent to a set of r nodes, and r nodes are involved in each reduction operation for a given block of C. If we again consider the best case, we get that $Q^B = Q^C = mn(r-1)$, which yields $Q = 2mn(r-1)$. Since $r = \sqrt{2P}$, we can write this as $Q^{\text{SBC}}_{\text{SYMM}} = 2mn(\sqrt{2P}-1)$: this improves over the 2D BC distribution by a factor of $\sqrt{2}$. Since the memory usage is the same, the AI is also improved by a factor of $\sqrt{2}$, which gives $\text{AI}^{\text{SBC}}_{\text{SYMM}} \simeq \sqrt{S}$: SBC obtains the same AI as the 2D BC distribution when storing the whole matrix. This result is summarized in Table 1, line 3.

3.3.3 Triangular Block Cyclic distribution

Another recent work proposed a Triangular Block approach for the SYRK operation [9], which achieves provably the lowest possible quantity of data transferred. This work was presented in the context of sequential out-of-core computations, but we propose here a way to transform it into an allocation for distributed nodes.

We remind that with the SBC distribution, each node is assigned $S = \frac{m^2}{2P} = \frac{m^2}{r^2}$ blocks, and needs to receive 2 rows of matrix B for each repetition of the pattern. The number of required rows of matrix B is thus $h = \frac{2m}{r} = 2\sqrt{S}$. This is similar to assigning a square part of the matrix to a node, as shown on Figure 3: if a node is responsible for the red square of side $\sqrt{S}$, it needs to receive $\sqrt{S}$ rows of B to perform the operations in the lower half, and another $\sqrt{S}$ rows to perform the operations in the upper half.

The idea of the TBS algorithm [9] stems from the observation that, thanks to the symmetry of matrix A, triangular parts along the diagonal of A are involved in operations that require even fewer communications than square parts. Indeed, as shown with the orange triangle on Figure 3, if a node owns a triangle containing S elements along the diagonal of A (its side length is thus $\sqrt{2S}$), it only needs to receive $\sqrt{2S}$ rows from matrix B since the operations on the lower and upper half require the same rows of B. Similarly, this node only needs to participate in reduction operations on $\sqrt{2S}$ rows for matrix C. The TBS algorithm defines a solution where these favorable properties are extended to blocks away from the diagonal by ensuring that each node is assigned a set of blocks which can be gathered into a diagonal triangle using a symmetric permutation.

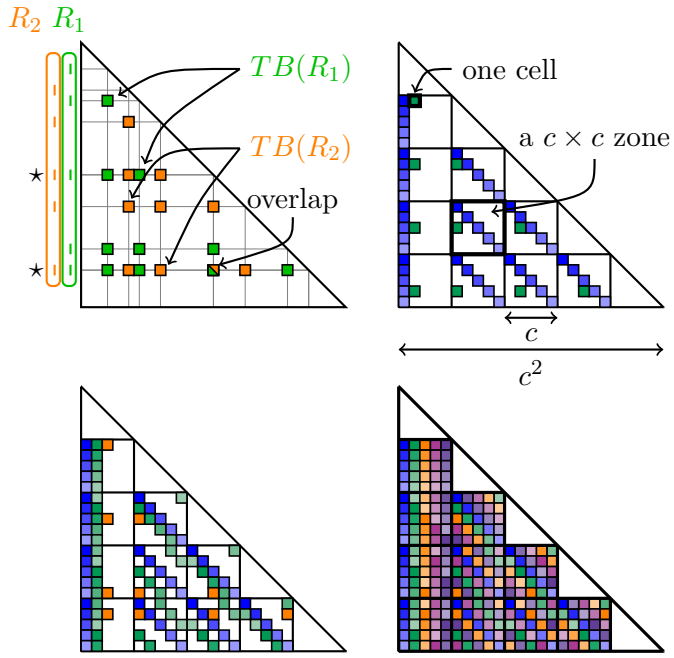

Figure 4: **Top Left:** two triangle blocks. An overlap happens when two triangle blocks have two rows in common ($\star$). **Top Right:** first column of the triangle-blocks in the TBC pattern. Gaps must be introduced for the triangle-block in the next column to avoid overlapping. **Bottom Left:** the gaps in the next column need to be larger, with a "wrap around" when reaching the bottom of the zone. **Bottom Right:** complete pattern for all the zones.

This leads to the notion of *triangle-blocks*, defined, for a given set R of row indices, as the set of blocks of the matrix A that a node can own while only requiring rows of matrix B indexed by R. Formally, the triangle-block associated with R is $TB(R) = \{(i,j) \in R^2 | i > j\}$. Figure 4 (left) shows examples of two triangle blocks. This notion generalizes the "triangle along the diagonal", since a triangle-block with $|R| = h$ contains $\sim \frac{h^2}{2}$ blocks of A, and the corresponding operations involve only h rows of matrices B and C. A triangle-block can indeed be seen as a triangle along the diagonal, up to reordering of the rows and columns of the matrix.

The key contribution of [9] is a method that makes it possible to partition almost all of the matrix in disjoint triangle-blocks. This requires to assign a set of rows R_p to each node, so that any two sets R_p and $R_{p'}$ have at most one row in common. Indeed, as can be seen on the top-left of Figure 4, two triangle blocks overlap if they share two row indices. This implies that the two corresponding nodes will receive the data necessary to perform an operation, however only one of them will actually perform it; communicating that data to the other node was not useful. Finding a distribution with no overlap ensures that we minimize the communication volume.

To apply these ideas in a distributed-memory setting, we propose to build a pattern where each triangle-block is assigned to a different node. Since this pattern is symmetric, for simplicity we only describe its lower half. We fix a prime integer $c > 2$, and build a symmetric pattern of size $c^2 \times c^2$, divided in $c \times c$ square *zones*, each containing c^2 cells. We partition the square zones among triangle-blocks, and the main idea is that each triangle-block has one cell in each zone. The top right of Figure 4 shows how the first c triangle-blocks are organized. The next triangle-block is also shown, and we can see that a gap must be introduced at each new row to ensure that it does not overlap with the previous blocks. The bottom left of Figure 4 shows the next step: the other blocks of the second column can be assigned with the same gaps. However, the next block in the third column needs to have larger gaps at each row to ensure that no overlap happens. With such large gaps, the last row would be outside of the zone, so the actual row is chosen modulo c: this effectively "wraps around" at the boundary of the zone. The final partition with all the triangle-blocks is shown at the bottom right of this figure.

With this partitioning, each node receives $\frac{c(c-1)}{2}$ cells from the lower half of the pattern, but the cells in the triangular zones along the diagonal remain unassigned. The choice of the pattern size ensures that if we exclude the diagonal cells, each of these triangular zones also contain $\frac{c(c-1)}{2}$ cells. We can thus assign them to c additional nodes, which describes the entire lower half of the pattern. By replicating this symmetrically, we get a square pattern where only the non-diagonal cells remain unassigned. A more precise description of the pattern is described in Algorithm 1, and the resulting pattern for $c = 3$ is provided in the left of Figure 5. The iteration (i,j) of the loop in line 4 assigns the triangle-block which contains the cell of coordinates (i,j) of the top-most zone (which is the cell $(i+c,j)$ of the pattern). The idea behind line 5 is that each node should access one row in each zone: the value uc indicates the index of the first row on the u-th zone, and the value $i+(u-1)j \mod c$ is the index of the row within this zone. We can see in this formula that there is a gap of size j between one row and the next (thus for two successive values of u), and that there is a modulo operation to perform the "wrap around", as described in the successive diagrams of Figure 4. The results from [9] (in particular Lemma 5.5), together with the condition that c is prime, ensure that the sets of rows assigned to different nodes overlap exactly once, so the sets of cells assigned to the nodes are disjoint, and each row contains exactly $c+1$ nodes.

This procedure results in a symmetric pattern of size $c^2 \times c^2$, in which the diagonal

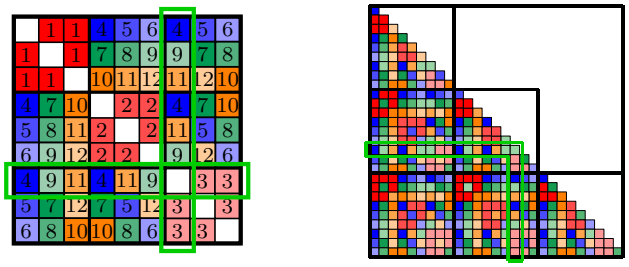

Figure 5: Triangular Block Cyclic distribution. **Left:** the pattern with $c = 3$, using $P = 12$ nodes, with no allocation on the diagonal blocks. **Right:** allocation of this pattern on a 27×27 matrix, where the diagonal blocks of the pattern are filled with the greedy algorithm. As shown in the highlighted part, each communication (related to matrix B or C) involves 4 different nodes (nodes 3, 4, 9, 11 in this example). For comparison, in the $(3, 4)$ 2D BC distribution, each communication involves 6 nodes.

Algorithm 1: TBC(c) pattern, on $P = c(c+1)$ nodes.

// Assign triangular zones (red nodes)
for $i \in \{0, \ldots, c-1\}$ **do**
  $R \leftarrow \{i \cdot c + u \,|\, 0 \leq u \leq c-1\}$
  Assign cells in $\{(x, y) \in R^2 | x \neq y\}$ to a new node
// Assign non-triangular zones (remaining nodes)
for $(i, j) \in \{0, \ldots, c-1\}^2$ **do**
  $R \leftarrow \{uc + (i + (u-1)j \mod c) | 1 \leq u \leq c-1\}$
  Assign cells in $\{(x, y) \in (R \cup \{j\})^2 | x \neq y\}$ to a new node

cells are not allocated. However, there are $c(c+1)$ nodes in total, and only c^2 diagonal cells. Each of these diagonal cells can be allocated to any node already present on the row without increasing the communication volume.

To obtain the final allocation, we replicate this incomplete pattern over the matrix A, and apply a greedy algorithm to allocate the remaining blocks : for each unassigned block, we pick the node with the lowest number of assigned blocks among all the nodes present in the row (and thus in the column, by symmetry of the pattern). The resulting allocation is thus not exactly a *cyclic* allocation, but it can nonetheless be computed very quickly. An example is provided on the right of Figure 5.

This pattern uses a total number of nodes $P = c(c+1)$, and each row and column of matrix A is allocated to a set of $c+1$ nodes. The communication volume can be evaluated just like previously: each communication related to a row of matrix B or C involves $c+1$ nodes, and we obtain $Q^B = Q^C = mnc$ in the best case. Thus, $Q = 2mnc \simeq 2mn\sqrt{P}$. This corresponds to another improvement by a factor of $\sqrt{2}$ over SBC, and asymptotically the same communication volume as with the 2D BC distribution when storing the whole matrix. The AI is also improved by a factor of $\sqrt{2}$: $\text{AI}^{\text{TBC}}_{\text{SYMM}} \simeq \sqrt{2S}$. This is the best of both worlds: the reduced memory storage gained by storing only half the matrix, and the reduced communication volume. This result is summarized in Table 1, line 4.

3.4 Summary

Table 1 summarizes all the results. We propose two interpretations, depending on whether we read the left-hand side or the right-hand side, respectively, of the last column (AI) of the table. A first interpretation, with the left-hand side $\text{AI}(m, P)$, is as follows. Assuming an infinite storage ($S = \infty$), for a given size m of matrix A and a given number of nodes P, 2D BC SYMM has a lower AI by a factor of 2 than 2D BC GEMM. SBC and TBC improve the AI of SYMM by a factor of $\sqrt{2}$ and 2, respectively, thus in particular equaling that of 2D BC GEMM for the latter one while consuming twice less memory.

A second interpretation, with the right-hand side $\text{AI}(S)$, is as follows. Assuming a given storage S and freely choosing the number of nodes P (as low as possible and independently from one method to another), 2D BC SYMM has a lower AI by a factor of $\sqrt{2}$ than 2D BC GEMM. SBC and TBC still improve the AI of SYMM by a factor of $\sqrt{2}$ and 2, respectively. However, with respect to 2D BC GEMM, this means that SBC equals GEMM AI and TBC improves over it by a factor of $\sqrt{2}$.

3.5 2.5D variants

For a fixed number of nodes P, the above discussion hints at a possible strategy for increasing the AI: increase the storage per node S. This is actually the idea behind the 2.5D extensions to 2D BC [1, 24], and it can also be applied to any of the above (A-stationary) distributions. The idea is to split the nodes into s slices (each with $\frac{P}{s}$ nodes) so that, on each slice, the whole matrix A is distributed with one of the above distributions. The matrix A is thus replicated s times. Matrices B and C are accordingly split into s column matrices $B_1, \ldots, B_s$ and $C_1, \ldots, C_s$, so that B_k and C_k are distributed among the nodes of slice k. Then the computation of all the $C_k \leftarrow \alpha AB_k + \beta C_k$ can be performed independently, with no additional communication since matrix A is replicated on each slice. For a fair comparison with the GEMM case where matrix A is assumed to be fully stored, we do not consider the communications involved in replicating A.

Using a 2.5D variant thus multiplies the storage cost S by a factor of s, for a benefit on the communication volume by a factor of $\sqrt{s}$, since the AI grows linearly with $\sqrt{S}$. In

particular, using $s = 2$ slices with the TBC distribution yields the same storage cost as the GEMM solution, with a communication volume lower by a factor of $\sqrt{2}$.

4 SYMM task-based design

Fully-featured distributed-memory dense linear algebra libraries such as ScaLAPACK [10] or Elemental [21] implement SYMM with a 2D BC data distribution. Such a regular data distribution makes it possible to easily set up MPI communicators along rows and columns and ensure collective communications. On the contrary, if we want to consider irregular data distributions, like those discussed in Section 3, it may be challenging to implement the corresponding code directly through the MPI interface. We therefore aim at designing a SYMM routine completely independent of the proposed mappings so that we can then effortlessly implement any non-trivial mapping. Task-based programming allows for such a separation of concerns [16, 4]. In addition, we want to assess whether this is feasible with a code which is easy to write, read and maintain. To this end, we decided, more specifically, to rely on the STF model [13, 6] introduced in Section 2.3, which allows for writing a parallel code which resembles a sequential code one would naturally write. Finally, we do not want to trade off performance with elegance, and we thus rely on the latest developments for the scalability of the STF model [4] to design appropriate communication patterns while preserving the compactness of the expression. Without loss of generality, for the sake of conciseness, we restrict the presentation to the $C \leftarrow AB + C$ case, where A is symmetric and only its lower part explicitly stored. We also assume blocks of size b-by-b, so that A is a M-by-M block matrix ($m = M * b$) and B and C are M-by-N block matrices ($n = N * b$). The sequential algorithm of the SYMM consists of three nested loops where the two innermost loops perform a matrix - block-column product ($C_{*,j} \leftarrow C_{*,j} + AB_{*,j}$), while the outer loop goes through all N block-columns.

Algorithm 2: STF block SYMM

```
1 for j = 1...N do
2     for i = 1...M do
3         for l = 1...M do
4             op ← if i = l then symm else gemm
5             blk_A ← if i ≤ l then A_{i,l} else A^T_{l,i}
6             insert_task(op, blk_A:R, B_{l,j}:R, C_{i,j}:RW)
```

Algorithm 2 shows how to implement this algorithm for distributed-memory machines with the STF model. The first stage (not reported in the pseudo-code) is to register the data to operate on to the runtime system. In our case, the data are the $A_{i,l}$, $B_{l,j}$, and $C_{i,j}$ blocks ($1 \leq j \leq N$, $1 \leq i \leq M$, $1 \leq l \leq M$). When a data is registered, the runtime system is informed of which node owns it. This information can be later retrieved through a rank (`rk()`) function. With this simple paradigm, we are able to instruct the runtime system how to set up the data mappings discussed in Section 3. Other than that, we may observe that the STF pseudo-code is very similar to a sequential one. However, instead of being directly executed, GEMM and SYMM operations on blocks are *inserted* as *tasks* through the `insert_task` primitive. The aim of such an insertion is to make the call asynchronous. Data dependencies between tasks are inferred by the runtime system thanks to the data access modes. The basic access modes being Read (R), Write (W), and Read Write (RW), the most straightforward implementation would be to associate a R access mode to blocks of A and B and a RW mode access to blocks of C. Such

an approach, following the programming models of [28, 3], would be a valid STF code for distributed-memory machines. Indeed, the runtime system can not only infer the dependencies between tasks but it may also trigger the appropriate communications by moving around the data causing the dependencies. Indeed, in this model [28, 3], tasks are executed onto MPI ranks that own data accessed in RW mode ($C_{i,j}$, in our case); the other data are automatically moved there by the runtime system before executing the corresponding task.

Algorithm 3: Scalable STF block A-stationary SYMM

```
1 for j = 1...N do
2     for i = 1...M do
3         for l = 1...M do
4             op ← if i = l then symm else gemm
5             block_A ← if i ≤ l then A_{i,l} else A^T_{l,i}
6             rank ← if i ≤ l then rk(A_{i,l}) else rk(A_{l,i})
7             insert_task(op, block_A:R, B_{l,j}:R, C_{i,j}:DIST_REDUX, ON_RANK=rank)
```

This baseline STF algorithm would however prevent us from implementing the algorithms from Section 3. Indeed, first, performing a task on the MPI ranks owning data accessed in RW access mode implies that the C matrix stays in place and A and B are transferred through the network. The analyses from Section 3 assume that, because the matrix A is the largest one, it is instead preferable to keep A in place and only move around blocks of B and C. To implement such a *A-stationary* scheme, we must be able to explicitly instruct the runtime system where to perform tasks. Algorithm 3 does so through the `ON_RANK` (l. 7) directive which is used to define the rank where to execute a task which, in our case, is the owner of the corresponding block of A (l. 6); this implies that the corresponding blocks of B (and C) will be transparently sent to (and from) that same node by the runtime system prior to the task execution. A second extension is required to achieve the AI of the algorithms devised in Section 3. Indeed, after a task has been computed on the chosen rank, the programming model requires that a data accessed in W or RW mode is sent back to the owner rank [28, 3]. In the SYMM case, $C_{i,j}$ being accessed in RW mode, it must therefore be sent back to the MPI rank that owns it unless it is the same rank as the one where the task is executed. It must be noted that, with such a A-stationary mapping, all the tasks contributing to a $C_{i,j}$ block can be executed by different nodes. With $C_{i,j}$ accessed in RW mode, a single copy exists that will be sent back and forth between the contributing nodes and all the corresponding tasks will be executed sequentially [28, 3]. To overcome this limitation, the `DIST_REDUX` mode [4] (l. 8) can be used: in this case nodes will compute, locally, contributions to $C_{i,j}$ which are stored in a temporary buffer allocated upon executing a task that requires it. All these contributions can be computed concurrently and, in a second step, reduced in the $C_{i,j}$ block. The reduction is transparently detected by the runtime and performed through a binary tree – this is a reasonable choice given the typical number of contributors and the size of the message [4]. Once a contributor has sent its contribution, the corresponding memory buffer is freed. The reduction operation allows for further parallelism and more accurately matches the A-stationary product described in the literature [27]. Altogether, both these extensions allow us to reach our main goal of designing a SYMM routine completely independent of the mappings so that we can then effortlessly implement any non-trivial mapping and achieve the expected associated AI.

5 Experiments

The code designed in Section 4 allows us to assess all the mappings discussed in Section 3. We study their impact on the AI and performance. We have implemented Algorithm 3 on top of the StarPU [6] task-based runtime system and the NewMadeleine communication back-end, which, combined, support the dynamic detection of collective communications [12]. In an applicative setting, where no synchronization is required between filling and computing the matrices, blocks of B are transferred through broadcasts transparently: the runtime detects them through dependencies in the DAG. In our benchmarking setting, artificial tasks are added to mimic the dependencies that allow a similar detection. We conducted our study in double precision on the Skylake partition of the Trs Grand Centre de Calcul (TGCC) computer. It has an Infiniband EDR interconnect. Each node has 192 GB of DDR4 memory and is composed of two 24 cores Intel Skylake 8168 @ 2.7 GHz processors (48 cores per node). Intel MKL v. 19.0.5.281 provides the implementation of GEMM and SYMM (single-core) tasks. All codes have been assessed with a block size b equal to 256, 512 and 1024, preliminary experiments having shown that these values allow for a good efficiency. Each configuration has been executed five times and we retrieve the median performance. GEMM and SYMM algorithms are executed with $(p = 8, q = 7)$ on 56 nodes for 2D BC distributions. 2D SBC ($r = 11$) and 2D TBC ($c = 7$) SYMM are executed on 55 and 56 nodes, respectively. 2.5D SBC ($s = 2$,$r = 8$) and 2.5D TBC ($s = 2$,$c = 5$) SYMM are executed on 56 and 60 nodes. The matrix size (m) of A varies while the number of columns of B and C is constant ($n = 8,192$).

The top plot of Figure 6 presents the AI of the STF algorithms discussed in Section 3 as defined in Equation (1): $\text{AI} = \text{flop}/Q$. The total volume of communication Q is retrieved by StarPU. The results show that the expected theoretical ratios of AI from Section 3 are successfully achieved in practice.

The bottom of Figure 6 presents the resulting per-node performance. The first observation is that the AI gains of SBC and TBC do yield compelling performance benefits on lower size matrices where the AI is not sufficient to ensure a good overlapping between communications and computations. The proposed STF design with SBC and TBC SYMM achieves a performance roughly comparable with 2D BC GEMM, while requiring to store only half of the dominant matrix. The second main observation is that the AI advantage of 2.5D SBC and TBC does not consistently translate into performance improvement. While 2.5D symmetric distributions perform well on small problems, they do not outperform the 2D case on larger ones. This is consistent with the literature [4]. TBC is more impacted by this performance discrepancy despite having an higher AI than GEMM. A preliminary analysis suggests that this is due to contention in the network that happens because, unlike in BC, in TBC any rank participates in multiple broadcast communications.

Figure 7 presents the comparison of the GEMM and SYMM performance of our STF approach with state-of-the-art distributed-memory dense linear algebra libraries proposing a A-stationary implementation of SYMM, namely ScaLAPACK [10] (yellow) and Elemental [21] (black). We also report the GEMM performance of SLATE, a potential successor to ScaLAPACK for which A-stationary SYMM was not available. The first observation is the important gap between SYMM and GEMM performance of both ScaLAPACK and Elemental libraries. These results confirm the empirical observation that SYMM state-of-the-art codes achieve a lower performance than their GEMM counterpart. We recall that both these libraries implement SYMM with a 2D BC data distribution. The second main observation is that the STF algorithms proposed in Section 4 significantly improve over the A-stationary ScaLAPACK and Elemental SYMM reference implementations. This

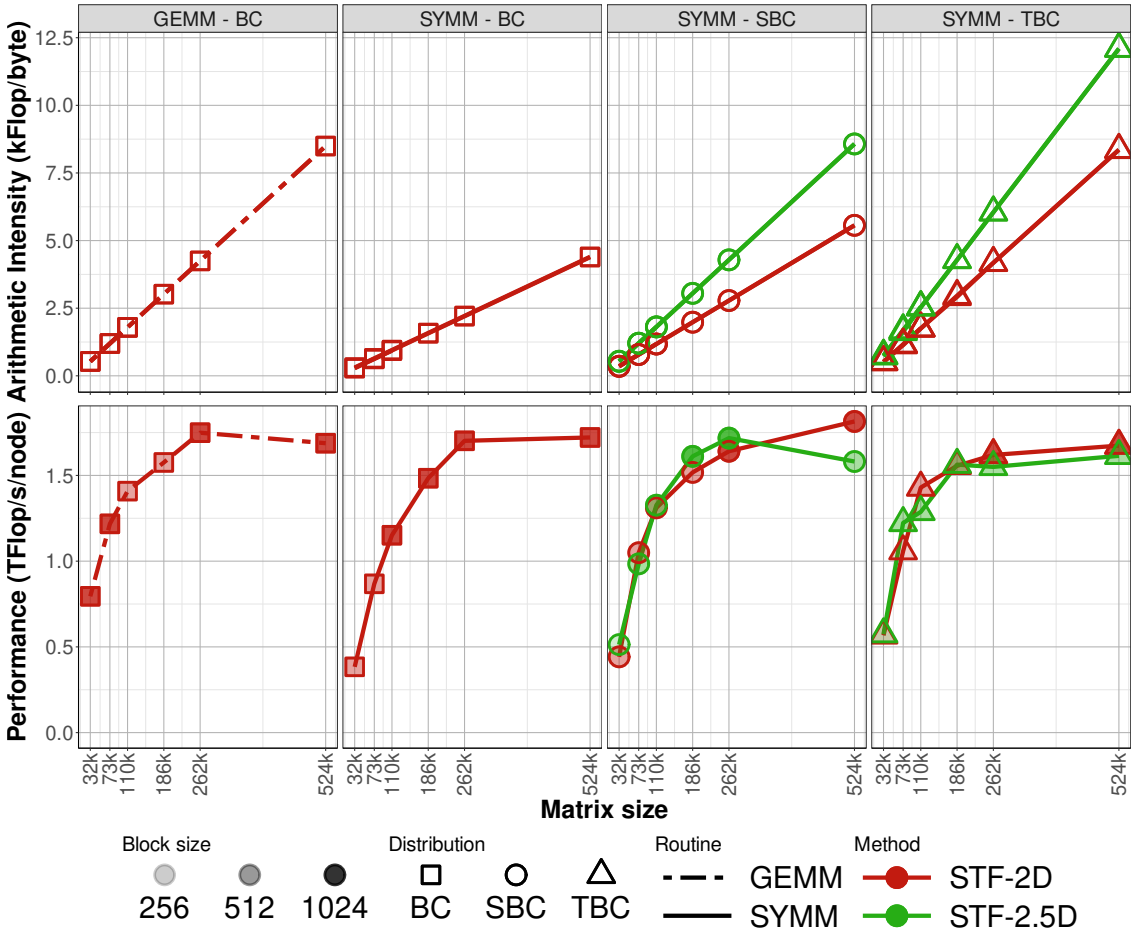

Figure 6: AI (top) and per-node performance (bottom) of the STF algorithm of section 4 with the various distributions of section 3

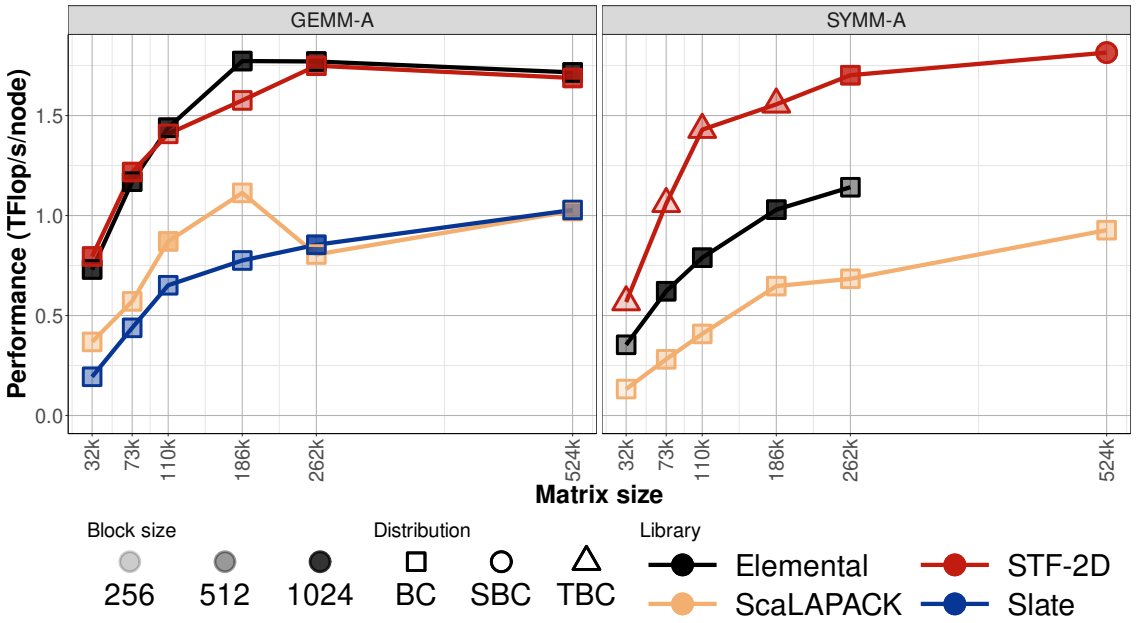

Figure 7: Per-node GEMM (left) and SYMM (right) performance of the proposed STF design [4] compared with state-of-the-art libraries.

Sample name	m	P	(p, q)	c
S1	99,594	1	(1,1)	1
S2	270,983	6	(3,2)	2
S3	426,548	30	(6,5)	5
S4	616,644	56	(8,7)	7
S5	1,043,192	132	(12,11)	11

Table 2: Samples names, matrix size m, number of nodes P (and of MPI processes), parameters (p, q) for 2D BC mappings, and parameter c for TBC mapping.

illustrates the strength of the programming model for designing efficient communication schemes with a high-level expression.

6 Application to RSVD-MDS

The initial point of the present study was that, in the context of an RSVD-MDS dimension reduction algorithm [5], it was necessary to trade off performance, with 2D BC GEMM, with memory, with 2D BC SYMM, during the dominant steps (denoted MM1 and MM2 steps in Section 2.1) of the RSVD algorithm. The central question raised in Section 2.1 was whether we could store only half of the symmetric matrix through SYMM while achieving a performance on par with that of GEMM.

As discussed in Section 2.1, MDS computes a so-called Gram matrix G from an input matrix representing dissimilarities between pairs of items. In the context of our metabarcoding target application, items are diatoms collected in Geneva and dissimilarities between them are their genetic distances. The dataset[2] used as input for the MDS, fully described in [5], is a $10^6 \times 10^6$ matrix of genetic distances between sequences. We may consider either part of the data (*S1*, *S2*, *S3*, *S4*), leading to a matrix of reduced dimension, of the whole data set (*S5*). Table 2 presents the matrix size (m) and parallel setup associated with each sample. The whole MDS algorithm consists of the computation of the Gram matrix G followed by an RSVD (which includes MM1 and MM2 steps). All the tests of this section are performed in single precision and a number $n = 1,000$ of columns for B and C, consistently with [5]. We conducted the study on the Jean Zay supercomputer. Jean Zay is a HPE SGI 8600 machine with an Omni-Path 100 Gb/s interconnect. Each node has 192 GB of memory and is composed of two 20 cores Cascade Lake 6248 @ 2.5 GHz processors (40 cores per node). Intel MKL v. 19.0.4 provides the implementation of single-core kernels. Figure 8 illustrates the impact on performance of the present study. The original code from [5] had been designed following the programming model of [28, 3] in which task mapping was inferred from the data mapping of the RW blocks. This means that it relied on a C-stationary scheme, in which case the optimum GEMM mapping is 1D BC (denoted (0) in Figure 8) as it is both an A- and a C- stationary variant. The application of the new programming model from [4] allows us to employ a 2D BC A-stationary variant (l. 1 of the table and denoted (1) in Figure 8 after this line number). The significant improvement shows the interest of the new programming model when dealing with a task-based approach. Figure 8 furthermore shows that it is possible to store only half of the symmetric matrix through a TBC SYMM (l. 4 in Table 1 and denoted (4) in Figure 8) while achieving a performance competitive with (2D BC) GEMM, which positively answers the question that originally motivated this work. As

[2]Dataset available at https://doi.org/10.57745/NKTRHO.

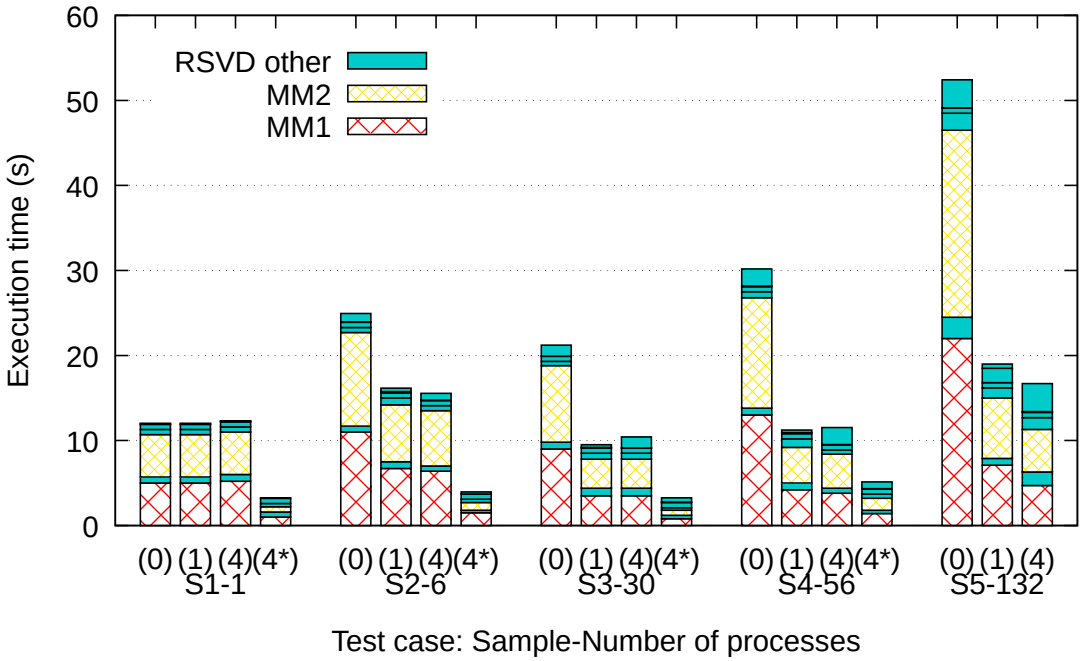

Figure 8: Execution time (s) of the RSVD with $n = 1,000$. MM1 and MM2 are assessed with methods (1) 2D BC GEMM and (4) TBC SYMM, numbered and named after Table 1, in the CPU-only case. Method (0) furthermore denotes 1D BC GEMM. Method (4*) denotes TBC SYMM with GPUs turned on. Execution of (4*) on 132 nodes was not possible because of a *Quality of Service* (QoS) limitation on Jean Zay supercomputer (no more than 512 GPUs per job). Five test cases are assessed, ranging from *S1* on 1 node (denoted *S1*-1) to *S5* on 132 nodes (denoted *S5*-132).

this observation applies to both MM1 and MM2 matrix multiplication steps and since they altogether dominate the RSVD algorithm, this directly translates into a significant improvement for the entire RSVD.

As recalled above, MDS requires to compute the Gram matrix G before applying the RSVD itself. We also redesigned this step, which is mainly a reduction, using the `DIST_REDUX` access mode introduced in [4]. We have not reported detailed figures on the matter, however, the execution time of the entire RSVD-MDS algorithm (Gram computation and RSVD altogether) on the whole data set (*S5*) using 132 nodes (5,280 CPU cores) has been reduced from 70 seconds, with the original code of [5] (denoted (0) here), to 25 seconds, with TBC SYMM (denoted (4) here) together with the new Gram step design, while using about half the memory.

We complete the study with the illustration of the capability of task-based codes to exploit heterogeneous architectures. Without any change in the code (other than providing the CUDA cuBLAS kernels of single-GPU kernels), the runtime system may execute tasks on CPU or GPU [6]. A subset of Jean Zay nodes have the exact same characteristics as described above in CPU-only case but are furthermore enhanced with four NVIDIA Tesla V100 SXM2 GPUs (32 GB). CUDA v. 10.1.2 is used. Bars denoted (4*) in Figure 8 correspond to the execution of the RSVD with GPUs enabled, relying on TBC SYMM for the matrix multiplication. The results show a considerable improvement over the CPU-only case in spite of the relatively low number of columns ($n = 1,000$ only) of B and C, a typical set up for the application.

7 Conclusions

We experimentally confirmed that reference distributed-memory libraries achieve a lower performance with SYMM than with GEMM. We showed that an efficient design of the communication schemes can significantly alleviate this gap. Still, we showed that part of the gap is explained by a lower AI of 2D BC SYMM compared to 2D GEMM (by a factor of 2). We considered two alternative data distributions, SBC and TBC. SBC is a direct adaptation to the matrix multiplication case of a study of the Cholesky decomposition [8]. TBC is a distributed-memory (and even a parallel) adaptation of the ideas behind TBS [9], a sequential out-of-core algorithm. We proved that SBC and TBC improve the AI of SYMM by a factor of $\sqrt{2}$ and 2, respectively, thus in particular equaling that of 2D BC GEMM for the latter one. In the case where we allow SYMM to store an amount of memory equivalent to a full matrix as 2D BC GEMM does, we furthermore showed that 2.5D TBC with $s = 2$ slices achieves a higher AI than 2D BC GEMM by a factor of $\sqrt{2}$. Our experimental study showed that the improvement of the AI translates into a compelling performance enhancement, up to the point of roughly matching GEMM performance. However, the highest AI does not always translate into the best performance. A preliminary analysis, not reported in this paper for a matter of conciseness hints that it is due to the scheduling of the communications that certainly must be reconsidered with the proposed new irregular data distributions. We plan to further investigate it in future work.

The resulting code[3] has been integrated in a metabarcoding application [5]. It consists in a MDS dimension reduction algorithm based on RSVD whose main computational steps are two dense matrix multiplications involving a symmetric input matrix. While one had to trade-off [5] between performance, with GEMM, or memory, with SYMM, we showed that, altogether, the proposed STF design and the new TBC distribution now achieve a

[3]Source code and instructions available at https://doi.org/10.5281/zenodo.7657176

performance competitive with GEMM. This study also showed that algorithms involving very irregular data and task distributions can now be implemented with a code easy to write, read and maintain thanks to the latest developments on the scalability of the STF model [4], while ensuring a competitive performance.

References

[1] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. A three-dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development*, 39(5):575–582, 1995.

[2] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. A hybridization methodology for high-performance linear algebra software for GPUs. In *GPU Computing Gems Jade Edition*, pages 473–484. Elsevier, 2012.

[3] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Paul Thibault. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems*, 2017.

[4] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Julien Herrmann, and Antoine Jego. Task-based parallel programming for scalable algorithms: Application to matrix multiplication. Research Report RR-9461, Inria Bordeaux - Sud-Ouest, 2022.

[5] Emmanuel Agullo, Olivier Coulaud, Alexandre Denis, Mathieu Faverge, Alain A. Franc, Jean-Marc Frigerio, Nathalie Furmento, Samuel Thibault, Adrien Guilbaud, Emmanuel Jeannot, Romain Peressoni, and Florent Pruvost. Task-based randomized singular value decomposition and multidimensional scaling. Research Report 9482, Inria Bordeaux - Sud Ouest ; Inrae - BioGeCo, September 2022.

[6] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU: a unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation: Practice and Experience*, 23(2):187–198, 2011.

[7] Grey Ballard, James Demmel, Olga Holtz, Benjamin Lipshitz, and Oded Schwartz. Communication-optimal parallel algorithm for strassen's matrix multiplication. In *Proceedings of the Twenty-Fourth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, SPAA '12, page 193204, New York, NY, USA, 2012. ISBN 9781450312134. doi: 10.1145/2312005.2312044.

[8] Olivier Beaumont, Philippe Duchon, Lionel Eyraud-Dubois, Julien Langou, and Mathieu Vérité. Symmetric block-cyclic distribution: Fewer communications leads to faster dense cholesky factorization. In *Supercomputing*, 2022.

[9] Olivier Beaumont, Lionel Eyraud-Dubois, Julien Langou, and Mathieu Vérité. I/O-optimal algorithms for symmetric linear algebra kernels. In *Proceedings of the 34th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 423–433, 2022.

[10] L. Susan Blackford, Jaeyoung Choi, Andrew J. Cleary, Eduardo F. D'Azevedo, James Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clinton Whaley. ScaLAPACK:

A linear algebra library for message-passing computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC*. SIAM, 1997.

[11] Pierre Blanchard, Olivier Coulaud, Eric Darve, and Alain Franc. Fmr: Fast randomized algorithms for covariance matrix computations. In *Platform for Advanced Scientific Computing (PASC)*, 2016.

[12] Alexandre Denis, Emmanuel Jeannot, Philippe Swartvagher, and Samuel Thibault. Using dynamic broadcasts to improve task-based runtime performances. In Maciej Malawski and Krzysztof Rzadca, editors, *Euro-Par 2020: Parallel Processing*, pages 443–457, Cham, 2020. Springer International Publishing. ISBN 978-3-030-57675-2.

[13] Alejandro Duran, Eduard Ayguadé, Rosa M Badia, Jesús Labarta, Luis Martinell, Xavier Martorell, and Judit Planas. OmpSs: a proposal for programming heterogeneous multi-core architectures. *Parallel processing letters*, 2011.

[14] Gene H Golub, R Underwood, and James H Wilkinson. The Lanczos algorithm for the symmetric Ax= λBx problem. *Techn. Rep. STAN-CS-72–270, Stanford University*, 1972.

[15] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions. *SIAM review*, 53(2):217–288, 2011.

[16] Thomas Herault, Yves Robert, George Bosilca, and Jack Dongarra. Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC. In *ScalA 2019 - IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*, pages 33–41, Denver, United States, November 2019. IEEE. doi: 10.1109/ScalA49573.2019.00010.

[17] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. Red-blue pebbling revisited: Near optimal parallel matrix-matrix multiplication. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '19, New York, NY, USA, 2019. ISBN 9781450362290. doi: 10.1145/3295500.3356181.

[18] Martinsson, Per Gunnar, Vladimir Rokhlin, and Mark Tygert. A randomized algorithm for the decomposition of matrices. *Applied and Computational Harmonic Analysis*, 30(1):47–68, 2011. doi: 10.1016/j.acha.2010.02.003.

[19] Dianne P O'Leary. The block conjugate gradient algorithm and related methods. *Linear algebra and its applications*, 29:293–322, 1980.

[20] Emmanuel Paradis. Multidimensional scaling with very large datasets. *Journal of Computational and Graphical Statistics*, 27(4):935–939, 2018.

[21] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A new framework for distributed memory dense matrix computations. *ACM Trans. Math. Softw.*, 39(2), feb 2013. ISSN 0098-3500. doi: 10.1145/2427023.2427030.

[22] Vladimir Rokhlin, Arthur Szlam, and Mark Tygert. A Randomized Algorithm for Principal Component Analysis. *SIAM Journal on Matrix Analysis and Applications*, 31(3):1100–1124, jan 2010. ISSN 0895-4798. doi: 10.1137/080736417.

[23] Yousef Saad. Analysis of augmented Krylov subspace methods. *SIAM Journal on Matrix Analysis and Applications*, 18(2):435–449, 1997.

[24] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. Parallel matrix multiplication: a systematic journey. *SIAM Journal on Scientific Computing*, 38(6):748–781, 2016.

[25] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5d matrix multiplication and lu factorization algorithms. In *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*, Euro-Par'11, pages 90–109. Springer-Verlag, 2011. ISBN 978-3-642-23396-8.

[26] W. S. Torgerson. Multidimensional Scaling: I. Theory and Method. *Psychometrika*, 17(4):401–419, 1952.

[27] Robert van de Geijn and Jerrell Watts. SUMMA: scalable universal matrix multiplication algorithm. *CONCURRENCY: PRACTICE AND EXPERIENCE*, 9(4): 255–274, 1997.

[28] Asim YarKhan. *Dynamic task execution on shared and distributed memory architectures*. PhD thesis, University of Tennesee, 2012.

Acknowledgment

This work was granted access to the HPC resources of TGCC under the allocation 2021-5063 made by GENCI. This work was performed using HPC resources from GENCI-IDRIS. This work was supported by the SOLHARIS project (ANR19-CE46-0009) which is operated by the French National Research Agency (ANR). This work has been supported by the Rgion Nouvelle-Aquitaine, under grant 2018-1R50119 HPC scalable ecosystem. We thank the StarPU and NewMadeleine development teams for their great support all along this work.