The Journal of Supercomputing (2024) 80:16630–16653
https://doi.org/10.1007/s11227-024-06097-7

Performance improvement of the triangular matrix product in commodity clusters

Inmaculada Santamaria-Valenzuela[1] · Rocío Carratalá-Sáez[1] · Yuri Torres[1] · Diego R. Llanos[1] · Arturo Gonzalez-Escribano[1]

Accepted: 21 March 2024 / Published online: 15 April 2024

Abstract

There are many works devoted to improving the matrix product computation, as it is used in a wide variety of scientific applications arising from many different fields. In this work, we propose alternative data distribution policies and communication patterns to reduce the elapsed time when computing triangular matrix products in distributed memory environments. In particular, we focus on commodity clusters, where the number of nodes is limited, proposing alternatives to traditional approaches in order to improve this operation's performance. Our proposal overcomes the performance results associated with the state-of-the-art libraries, such as ScaLAPACK and SLATE, offering execution times that are up to 30% faster.

Keywords Commodity clusters · Triangular matrix product · TRMM · SLATE · ScaLAPACK

Inmaculada Santamaria-Valenzuela and Rocío Carratalá-Sáez contributed equally to this work.

✉ Rocío Carratalá-Sáez
rocio@infor.uva.es

Inmaculada Santamaria-Valenzuela
msantamaria@infor.uva.es

Yuri Torres
yuri.torres@infor.uva.es

Diego R. Llanos
diego.llanos@uva.es

Arturo Gonzalez-Escribano
arturo@infor.uva.es

[1] Dpto. Informática, Universidad de Valladolid, Paseo de Belén 15, 47011 Valladolid, Castilla y León, Spain

1 Introduction

The matrix-matrix product (MM) is an essential linear algebra operation at the core of multiple areas of scientific applications, such as deep learning, fluid dynamics, or image processing. This mathematical operation is usually highly time-consuming; therefore, much effort has been devoted from the high-performance computing (HPC) community to solve it efficiently to reduce its computing time as much as possible.

Compared with single-node systems, distributed HPC environments increase the number of computing resources available for the execution of an MM operation. Distributing the workload among the computational nodes introduces the potential to execute faster MM, or to execute multiplications of bigger matrices. However, developing an efficient implementation in distributed systems for all kinds of matrices and platforms remains a challenge, since it is difficult to find a good balance between the calculations and the communications derived from the data distribution across the involved nodes.

The rise of modern commodity clusters has been driven by the opportunities they provide, such as easy scalable performance, reduced cost, mass storage, the flexibility of configurations, and programming model commonality [1]. Many companies or academic institutions build their own commodity clusters, equipped with a small number of nodes, dedicated to research or production tasks. There is also a significant number of research centers where small commodity clusters are maintained as the main execution platform for experimentation. Even small supercomputing centers have limited shared facilities where users can only launch their jobs in small partitions with no more than, for example, 36 nodes. Moreover, the queue systems in these shared facilities do not prioritize jobs that require a high number of nodes. Therefore, it is important to continue to make progress in improving the efficiency of the distributed MM product for this kind of small commodity clusters.

There are currently libraries that offer efficient implementations of general and triangular MM, called GEMM (General MM) and TRMM (Triangular MM), respectively. In particular, in the case of distributed environments, there are well-known library options to execute efficient MM that have been intensively optimized. For example, the classic ScaLAPACK [2] library and its recent evolution SLATE [3], whose main goal is to replace ScaLAPACK. They are currently the widely accepted standard for performing complex linear algebra operations in distributed memory systems. These libraries are focused on keeping their scalability rate even when using a high number of nodes, like those available in top-end supercomputers. To attain this objective, they provide implementations using tiling and block-cyclic distributions with classic, highly scalable distributed algorithms as the foundation of their execution strategy in distributed systems, such as Summa [4]. However, these tools do not consider different data partitions, communication patterns, or buffer shapes that have the potential to squeeze the parallelism opportunities even more when using commodity clusters for specialized MM cases, such as the TRMM. This routine is used in many scientific operations (see e.g. [5] that presents a novel implementation of Cannon's algorithm for triangular MM, or [6] that uses triangular linear system solvers in leading machine learning frameworks). TRMM operates with a triangular matrix, and because half of its elements are zero, it presents different opportunities and requires different approaches than GEMM or other full matrix operations to distribute data, create load balance, and communicate the matrix parts.

In this work, we propose and evaluate an alternative combination of classic and revised techniques related to the data distribution, communication patterns, and the communication buffer shape, with the objective of reducing the overall cost of TRMM in distributed memory environments of low/mid scale, such as commodity clusters. Our solution provides a good balance between computation and communication by using non-blocking communications adapted to the number of nonzero elements in the triangular matrices, and better overlapping computations and communications.

In this work, we present the following contributions:

1. We propose different combinations of classic and revised techniques for computing the distributed TRMM. These proposals are an alternative to the ones currently adopted in state-of-the-art libraries and are related to data distribution, communication patterns, and the shape of data buffers.
2. We conduct an experimental evaluation of the execution time associated with each of our proposed alternative TRMM implementations in a commodity cluster, using up to 36 nodes.
3. We compare our elapsed execution times with the equivalent ones by the two main primary state-of-the-art libraries that implement the distributed TRMM: ScaLAPACK and SLATE.
4. We derive recommendations that can be applied by the users and developers of the standard libraries, such as ScaLAPACK and SLATE, in order to improve their performance for commodity clusters.
5. We have made our implementations fully Open Source and available, so anyone in the community can access them.

The rest of the paper is structured as follows: In Sect. 2, we describe the state of the art, mentioning the main existing libraries that offer distributed TRMM; in Sect. 3, we describe our proposal, explaining the main axes covered by it (band partitioning, eluding zeros in the communications, balancing the data distribution, and testing a customized and more sophisticated version of a broadcast-based communication pattern); in Sect. 4, we demonstrate our proposal's performance, comparing it to that offered by the state of the art libraries SLATE and ScaLAPACK; and in Sect. 5, we summarize the main conclusions derived from this work.

2 State of the art

There are many libraries available that offer efficient tools for performing complex mathematical calculations involving dense linear algebra operations. These libraries typically contain an MM, as it is a critical operation in many applications.

GotoBLAS [7] was a pioneer library introducing a framework for highly optimized implementations of the MM, specialized for each target platform or architecture. Most of the modern libraries with implementations of the MM are inspired by the concepts and ideas proposed in GotoBLAS, including both Open Source libraries (like BLIS [8] or OpenBLAS [9]) and vendor provided ones (such as Intel MLK [10], ARMPL [11] or AMD AOCL-BLAS [12]). These libraries offer an implementation of one of the most widely used library interfaces that include the MM: LAPACK [13]. LAPACK has been around for nearly 30 years and is considered a standard choice for performing dense linear algebra computations on a single computer. The programs using LAPACK are more portable and can easily access the desired linear algebra library with the best optimized implementation for the chosen target platform. Most LAPACK implementations also support shared-memory parallelism, which allows the exploitation of multiple processor cores for faster calculations. Thus, they are typically used as the foundation of higher-level MM algorithms for distributed memory systems.

ScaLAPACK [2] extends the capabilities of LAPACK by making it suitable for distributed computing environments, like HPC clusters. It achieves this by exploiting highly scalable distributed algorithms for MM, such as Summa [14], implemented using both Parallel BLAS (PBLAS) [15] and explicit distributed-memory communications with, for example, a message-passing library such as MPI [16]. Among the libraries implementing ScaLAPACK, there is Intel MKL [17], which is used in this work as one of our references for performance comparisons. With ScaLAPACK, the user can tune the routines by varying the parameters that specify the data layout, in order to improve the performance on the specific target system. On shared-memory machines, this is controlled by a block size parameter, while on distributed-memory systems it is controlled by both the block size and the configuration of the logical processes grid. ScaLAPACK promotes the use of block-partitioned algorithms with block-cyclic distributions. However, ScaLAPACK does not allow alternative data distributions and it is not capable of avoiding sending blocks with zero values between nodes when using triangular matrices.

There are other libraries that feature similar to ScaLAPACK [18, 19]. However, none of them have become widely accepted or adopted as a suitable replacement for ScaLAPACK. Nevertheless, the SLATE [3] library has recently been released to offer essential tools for performing complex linear algebra calculations on modern high-performance computing systems. It is offered as a replacement for LAPACK and ScaLAPACK, aiming to provide the same functionality with a better performance on modern machines. It supports multicore CPUs and accelerators such as GPUs. For CPUs, it achieves a portable high performance by exploiting the OpenMP [20] technology. Using the OpenMP runtime puts less of a burden on applications to integrate SLATE than adopting a proprietary runtime would. Furthermore, it improves performance and scalability by employing proven techniques in dense linear algebra, such as 2D block-cyclic data distribution and communication-avoiding algorithms, as well as modern parallel programming approaches, such as dynamic scheduling and communication overlapping. SLATE also offers a new feature, the possibility to use data distributions tailored by the user for specific purposes. Nevertheless, the current implementation only provides the usual 2D block-cyclic data distribution.

Many works study how to adapt the data layout to improve performance when using different linear algebra operations in specific contexts. For example, in [21] the authors outline a methodology for the automatic generation and optimization of numerical software tailored to processors featuring deep memory hierarchies and exclusively pipelined functional units; [22] uses it in the context of computations performed by a matrix engine conceptualized as an accumulation of multiple outer products; in [23], the authors investigate a range of algorithmic and programming optimizations for BLAS-3 and LAPACK-level routines leveraging task-based parallelism.

In this work, we study the use of different data distributions for the case of the TRMM in the context of commodity clusters. We use SLATE (together with Intel MKL ScaLAPACK) as a second reference for performance comparisons.

3 Our proposal

With the objective of improving the efficiency of the distributed triangular matrix product in commodity clusters, as compared to that offered by state-of-the-art libraries, our proposal explores four main ideas:

1. Using band partitioning instead of tiling, which is the usual choice of well-established library implementations.
2. Using specific buffering strategies to avoid sending zeros when communicating the matrix data between the different processors.
3. Customizing the partition of the data to balance the number of elements that are processed at each stage on each node.
4. Evaluating a custom version of the broadcast communication strategy to move the data across the nodes in the distributed system faster, thanks to the increase of the degree of the communications overlap.

In this section, we describe each of these ideas in depth, and we also provide some details on how to access our implementation, which is fully available and Open Source.

Regarding the notation, we consider the triangular matrix product $C = A \times B$ where $A \in \mathbb{R}^{m \times m}$, $B \in \mathbb{R}^{n \times n}$, $C \in \mathbb{R}^{m \times n}$, with A being a triangular matrix, and $m, n \in \mathbb{N}$. Moreover, we also use $i \in \{0 \ldots \texttt{n\_procs} - 1\} \in \mathbb{N}$, with `n_procs` being the total number of computing nodes that participate in the computation, among which the matrix portions need to be distributed. In addition, we will use `n_rows` to refer to the number of rows of the A matrix. For simplicity, we discuss the case where A is lower-triangular. The discussion and algorithms for the case where A is upper-triangular are equivalent, and adapting the implementation is straightforward.

3.1 Band partitioning

The matrix product implementations typically use 2D tiles and 2D process grids to partition and distribute the data among the processors. Although using tiles can be the best strategy when scaling up the number of nodes intervening in the computation, our hypothesis is that using panels or bands can offer better results in commodity clusters. When using the term "panel" or "band", we refer to portions of the matrix formed by a contiguous set of full rows (horizontal panel), or a contiguous set of full columns (vertical panel). Even though, in the long term, panel-based partitions lead to a higher number of communication stages per node, there are communication algorithms for band partitions that move the data of only one matrix with bulk movements of contiguous data, thus simplifying the marshalling and buffer management.

In our proposal, at the start of the triangular matrix product algorithm execution, each node receives its A_i and B_i submatrix. In our implementation, we opt for distributing A by horizontal panels of dimension $m'_i \times m$, with $m'_i \leq m$, and B and C by vertical panels of dimension $n \times n'_i$, with $n'_i \leq n$. Note that here, the subindex i serves as a reference to the i-th partition of A and B, that is, the one provided to the computing node i. Figure 1 graphically illustrates this panel-based distribution of the A, B, and C matrices when four nodes are involved in the computation.

With this proposed partition, along the TRMM algorithm execution, the different nodes send their A_i submatrix to the others, always keeping the same B_i and C_i submatrix. The way A_i is moved across the algorithm stages depends on the communication strategy chosen, as explained in detail in Sect. 3.4. At each stage, each node computes a portion of the result of its C_i matrix; in particular, a vertical panel of dimension $m'_k \times n'_i$, where $k \in \{0 \ldots \texttt{n\_procs} - 1\}$ is different at each stage, and m'_k is equal to the m'_i dimension of the particular A_i that is being multiplied. At the end of the algorithm, the whole C_i panel has been computed by each i process. State-of-the-art libraries, such as SLATE, communicate and multiply blocks of a fixed size. To improve utilization they do the local multiplication operations using batch routines, where multiple independent small MM operations are grouped and then multiple threads work on different MM instances within the group. In our case, whole

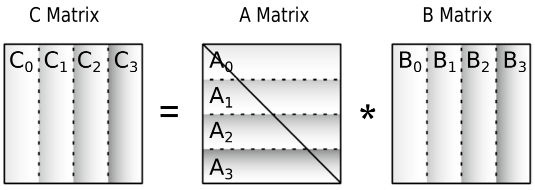

Fig. 1 Sample of a proposed panel-based distribution of A, B and C matrices among four processes

panels are communicated and the local computation on each stage is a single big panel matrix multiplication. Thus, we choose to do these operations by calling classical MKL routines that are designed to internally exploit multiple-thread parallelism in a very optimized and efficient way.

3.2 Eluding zeros in the communications

Data communication usually represents a considerable portion of the total execution time of any distributed memory application. In the case of the TRMM considered in this work, where A is triangular, almost half of the A matrix is known to be filled with zeros, as shown in Eq. (1).

$$\begin{aligned}
number\_of\_elements\_of\_A &= m^2 \\
number\_of\_nonzero\_elements\_of\_A &= \frac{m(m+1)}{2} \\
\frac{number\_of\_nonzero\_elements\_of\_A}{number\_of\_elements\_of\_A} &= \frac{\frac{m(m-1)}{2}}{m^2} = \frac{m+1}{2m} \\
\lim_{m\to\inf} \frac{number\_of\_nonzero\_elements\_of\_A}{number\_of\_elements\_of\_A} &= \lim_{m\to\inf} \frac{m+1}{2m} = \frac{1}{2}
\end{aligned} \tag{1}$$

It is unnecessary to invest time and resources on sending the matrix portions that only contain zeros, as those values are not considered when computing the triangular matrix product. For this reason, we have designed two marshalling strategies that can be implemented by MPI derived datatypes, to reduce or even completely avoid the zero values that are communicated across the computing nodes: Boxes and trapezoids. We propose using these strategies in contrast to communicating the whole matrix panel A_i. We refer to this last option as the "full panel" strategy; see Fig. 2a.

The "box" strategy wraps the elements of the rows in an A panel, up to the column that contains the matrix diagonal element of the last row of the panel. The result is a rectangular region. The elements in the rest of the columns, on the right of the box, are skipped because they contain zeros. In this case, if the panel has more than one row, the last columns of the upper rows of the box still contain some zeros. Figure 2b illustrates this graphically. Analyzing in depth the number of zeros that this data structure contains, we see that:

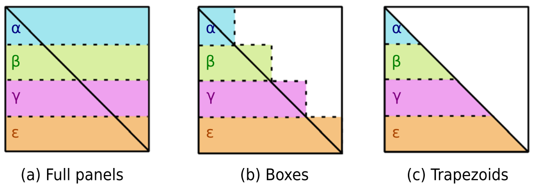

Fig. 2 Full panel (**a**), box (**b**), and trapezoid (**c**) shapes for distributing the data among the processes. Observe that, from left to right, the area of elements forming each matrix portion (α, β, γ, and ϵ) decreases

$$number\_of\_zeros\_per\_box = \frac{\frac{n\_rows}{n\_procs} \cdot \left(\frac{n\_rows}{n\_procs} - 1\right)}{2} \tag{2}$$

When we have a small number of nodes:

$$\lim_{n\_procs \to 1} number\_of\_zeros\_per\_box = \frac{n\_rows(n\_rows - 1)}{2} \tag{3}$$

$$\lim_{n\_procs \to 0} number\_of\_zeros\_per\_box = \infty \tag{4}$$

And, when we have a large number of nodes:

$$\lim_{n\_procs \to \infty} number\_of\_zeros\_per\_box = 0 \tag{5}$$

From Eqs. (3)–(5), we can derive the fact that, in big clusters, the impact of including some of the zeros in the communication is almost negligible. Nevertheless, when the number of nodes is small, these zeros can be representative. For this reason, in the context of commodity clusters, we refine the "box" strategy by creating the "trapezoid" one to skip the extra zeros.

The "trapezoid" strategy wraps the elements of the rows in an *A* panel, choosing for each row the columns up to the diagonal element contained in that particular row. If the *A* panel has several rows, this strategy can be seen as a combination of two strategies: A first one that selects a rectangle, and a second one that selects a triangle. The rectangular part contains the elements on the columns before the diagonal element in the first row. The triangular part contains the nonzero elements on the region defined by the range of rows in the panel and the range of columns that contain a diagonal element of any of those rows. Figure 2c illustrates this graphically.

Although these box/trapezoid marshalling strategies involve copying non-contiguous data sets, and consume some extra time in marshalling logic as compared to buffering the full panel, they involve much fewer data movements, and communicating them is faster. As has been previously commented, this is especially important when the number of nodes intervening in the operation is small. Moreover, these strategies operate in bulk communications for whole panels. We hypothesize that, in commodity clusters, the overall execution time using the bulk "box" or "trapezoid" marshalling strategies will be reduced as compared to that observed when using "full panels", or even classical 2D tiling approaches that require continuous marshalling of small pieces of non-contiguous data.

3.3 Balancing the data distribution at run-time

Regardless of employing marshalling strategies such as those described in the previous section, it is also important to pay attention to the number of rows on each *A* panel, which directly impacts the number of elements involved in the computations executed by each node at each stage. The proposal we describe in this section is related to the number of rows and nonzero elements on each A_i panel, and thus, on the balance of the computations at each stage.

When slicing a regular matrix by panels to distribute it among different nodes, the basic approach consists of dividing the number of rows of the matrix (`n_rows`) by the number of nodes (`n_procs`) to create a regular distribution. We refer to this approach as "regular partition". If the integer division has a remainder, some node(s) could have slightly more rows than others, but this becomes negligible when the size of the matrix grows. For simplicity, for the rest of the discussion, we assume square matrices where `n_rows = n_columns`, with no remainder in the division `n_rows/n_procs`.

In the case of triangular matrices, it is known in advance that a non-despicable amount of their elements are zero, and thus, they do not take part in the computation. If a regular partition is performed on the A matrix, each processor receives an A_i panel with $\texttt{rows\_count} = \frac{\texttt{n\_rows}}{\texttt{n\_procs}}$ rows of the original matrix, starting at row $i \cdot \texttt{rows\_count}$. Considering that A is lower triangular (the result is equivalent when it is upper triangular), each node i receives the following number of nonzero values ($n\_nonzero\_elements\_A_i$):

$$n\_nonzero\_elements\_A_i = n\_elem\_rect\_A_i + n\_elem\_tri\_A_i, \tag{6}$$

where $n\_elem\_rect\_A_i$ refers to the number of elements of the "rectangular" portion of the A_i nonzero data (see the dark gray region in Fig. 3):

$$n\_elem\_rect\_A_i = i \cdot rows\_count^2, \tag{7}$$

and $n\_elem\_tri\_A_i$ refers to the number of elements of the "triangular" portion of the A_i nonzero data (see the light gray region in Fig. 3):

$$n\_elem\_tri\_A_i = \frac{rows\_count \cdot (rows\_count + 1)}{2} \tag{8}$$

Thus, the total number of nonzero values of the panel A_i ($n\_nonzero\_elements\_A_i$) can be expressed with respect to the $rows\_count$ as:

$$n\_nonzero\_elements\_A_i = i \cdot rows\_count^2 + \frac{rows\_count \cdot (rows\_count + 1)}{2} \tag{9}$$

The total number of elements of the panel A_i ($n\_elements\_A_i$) can be expressed with respect to the $rows\_count$ as:

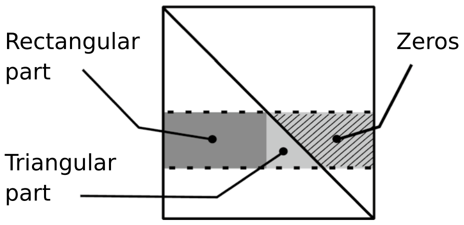

Fig. 3 Graphical representation of the rectangular part (dark gray), triangular part (light gray), and the part filled with zeros (dashed light gray) of a horizontal panel in a lower triangular matrix

$$n\_elements\_A_i = rows\_count \cdot n\_rows \tag{10}$$

If the portion of nonzero values associated with each node is evaluated with respect to the total number of elements of each A_i submatrix, we see that:

$$\begin{aligned}\frac{n\_nonzero\_elements\_A_i}{n\_elements\_A_i} &= \frac{i \cdot rows\_count^2 + rows\_count \cdot (rows\_count + 1)/2}{rows\_count \cdot n\_rows} \\ &= \frac{(2i + 1) \cdot rows\_count + 1}{2 \cdot n\_rows}\end{aligned} \tag{11}$$

When the number of nodes intervening in the computation increases, the *rows_count* gets smaller. In particular, the larger the number of nodes in the computation, the greater the difference between the number of nonzero elements to compute on each node in a given stage. Let us see this in detail.

The first node ($i = 0$) obtains the following quantity of nonzero elements:

$$n\_nonzero\_elements\_A_0 = rows\_count \cdot (rows\_count + 1)/2 \tag{12}$$

The last node ($i = n\_procs - 1$) obtains the following quantity of nonzero elements:

$$\begin{aligned}&n\_nonzero\_elements\_A_{n\_procs-1} \\ &= (n\_procs - 1) \cdot rows\_count^2 + \frac{rows\_count \cdot (rows\_count + 1)}{2} \\ &= \frac{2(n\_procs - 1) \cdot rows\_count^2 + rows\_count \cdot (rows\_count + 1)}{2}\end{aligned} \tag{13}$$

The relationship between the number of nonzero elements received by both nodes is given by:

$$\begin{aligned}\frac{n\_nonzero\_elements\_A_0}{n\_nonzero\_elements\_A_{n\_procs-1}} &= \frac{\frac{rows\_count(rows\_count+1)}{2}}{\frac{2(n\_procs-1)\cdot rows\_count^2+rows\_count(rows\_count+1)}{2}} \\ &= \frac{rows\_count + 1}{2 \cdot rows\_count(n\_procs - 1) + rows\_count + 1}\end{aligned} \tag{14}$$

Thus, the number of nonzero elements provided to the first node is insignificant compared to that of the last node:

$$\lim_{n\_procs \to \inf} \frac{rows\_count + 1}{2 \cdot rows\_count(n\_procs - 1) + rows\_count + 1} = 0 \tag{15}$$

As a consequence, our hypothesis is that it is worth balancing the number of nonzero elements of each A_i in such a way that each node operates over approximately the same number of elements. To attain this, we propose to change the way of calculating the `rows_count`. Instead of a global `rows_count` that is used to determine how many rows form each A_i, we propose to define a collection of row counts, such that $\texttt{rows\_count}_i$ expresses the number of rows mapped to A_i. This lets us balance the number of nonzero elements assigned to each node by supplying a descendant number of rows to each A_i; in other words, $max(rows\_count_0 \dots rows\_count_{n\_procs-1}) = rows\_count_0$, and $min(rows\_count_0 \dots rows\_offset_{n\_procs-1}) = rows\_count_{n\_procs-1}$.

The ideal situation in which the balanced partition is perfect is that in which the number of nonzero elements of the matrix A is divisible by `n_procs` and the quotient of the division is such that the different $\texttt{rows\_count}_i$ can exactly match it. For example, this applies when we have a matrix of dimension 3×3 and distribute it between two processors. In total, that small matrix has nine elements, of which $\frac{3(3+1)}{2} = 6$ are zero values, and the remaining three are zeros. Each processor should get (following a balanced strategy) a total of $\frac{6}{2} = 3$ elements. It turns out that by giving the first two rows to the first processor and the last row to the second one, each has exactly three elements.

Nevertheless, this ideal situation is not frequent. In our proposal, we use Algorithm 1, which in turn relies on Algorithm 2, to determine which rows are assigned to each process to balance as much as possible the number of nonzero elements of each one in the distribution.

Algorithm 1 Calculate_row_count_array

```
Require: $n\_procs,\ m$                                      ▷ m is A dimension
  $row\_counts\_array \leftarrow [\ ]$
  $n\_assigned \leftarrow 0$        ▷ Number of non-zero elements already given to a process
  $n\_remaining \leftarrow \frac{m(m+1)}{2}$        ▷ Number of non-zero elements that remain to assign
  $assgined\_rows \leftarrow 0$
  for $i \leftarrow 1$ to $n\_procs - 1$ do
      $ideal \leftarrow \frac{n\_remaining}{n\_procs-i+1}$
      $rci,\ n\_assigned \leftarrow calculate\_row\_count\_i(ideal,\ assgined\_rows,\ m)$
      $row\_counts\_array.append(rci)$
      $n\_remaining \leftarrow n\_remaining - n\_assigned$
      $assgined\_rows \leftarrow assgined\_rows + rci$
  end for
  $last\_rci \leftarrow m - assigned\_rows$
  $row\_counts\_array.append(last\_rci)$
  return $row\_counts\_array$
```

Algorithm 2 Calculate_row_count_i

```
Require: $ideal,\ assgined\_rows$                              ▷ m is A dimension
  $rci \leftarrow round.ceil(\frac{-2 \cdot assgined\_rows - 1 + \sqrt{4 \cdot assgined\_rows^2 + 4 \cdot assgined\_rows + 1 + 8 \cdot ideal}}{2})$
  $n\_assigned \leftarrow \frac{rci(rci+1)}{2} + rci * assgined\_rows$
  return $rci,\ n\_assigned$
```

Algorithm 2 arises from solving the following equation, where *ideal* refers to $ideal\_n\_nonzero\_elements\_A_i = \frac{n\_nonzero\_elements\_A}{n\_procs}$:

$$n\_nonzero\_elements\_Ai = ideal \rightarrow$$
$$n\_elem\_rect\_A_i + n\_elem\_tri\_A_i - ideal = 0 \rightarrow$$
$$rows\_count_i \cdot assigned\_rows + \frac{rows\_count_i \cdot (rows\_count_i + 1)}{2} - ideal = 0 \rightarrow$$

with *assigned_rows* referring to the number of rows that have already been assigned to other processes. Thus,

$$rows\_count_i^2 + rows\_count_i(2 \cdot assigned\_rows + 1) - 2 \cdot ideal = 0 \rightarrow$$
$$rows\_count_i = \frac{-2 \cdot assigned\_rows - 1 + \sqrt{4 \cdot assigned\_rows^2 + 4 \cdot assigned\_rows + 1 + 8 \cdot ideal}}{2}$$

This regular partitioning (using just one global `rows_count`) versus a balanced partitioning (using a collection of `rows_count`$_i$) is graphically illustrated in Fig. 4.

3.4 Data communication

A simple serial triangular matrix product algorithm (serial TRMM) is described in Algorithm 3. When computing this product in a distributed environment, the data need to be properly spread across nodes, and the results adequately collected. A generic distributed TRMM algorithm has three main parts: (1) initial

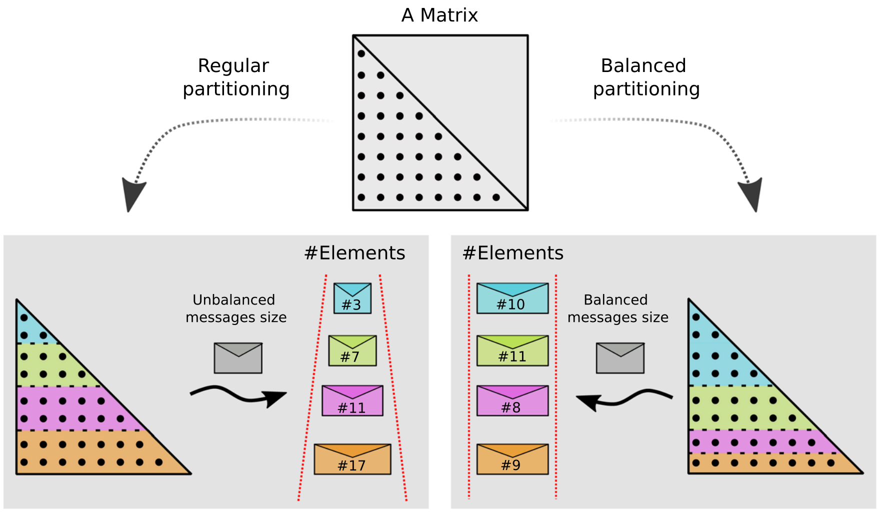

Fig. 4 Sample of a regular (left) and balanced (right) partition of the matrix data among four processes

distribution of the submatrices of A, B and C; (2) a loop to perform as many times as needed the product of the local submatrices A_i and B_i at each node and to interchange the local A submatrices among the nodes for the next loop iteration; and (3) collect the resulting matrix portions C_i calculated by each node to form the result matrix.

Algorithm 3 TRMM (Serial triangular matrix product)

```
Require: $A \in \mathbb{R}^{m \times m}, B \in \mathbb{R}^{m \times n}$ with $m, n \in \mathbb{N}$
  $C \leftarrow \mathbb{R}^{m \times m}$                    ▷ C initially filled with zeros
  for $i \leftarrow 1$ to $m$ do
      for $j \leftarrow 1$ to $n$ do
          for $k \leftarrow 1$ to $i$ do
              $C[i,j] = C[i,j] + A[i,k] \cdot B[k,j]$
          end for
      end for
  end for
  return $C$
```

State-of-the-art libraries, such as SLATE, use OpenMP tasks to manage MPI communications and improve computation and communication overlap of matrix blocks, which are in general of much lower granularity than panels. For whole matrix panels, we can obtain better results by carefully planning the overlapping of computation and communication explicit choosing MPI the proper places of the communication algorithm where MPI communications are issued. After also testing classical pipeline and broadcast approaches, in our proposal we choose

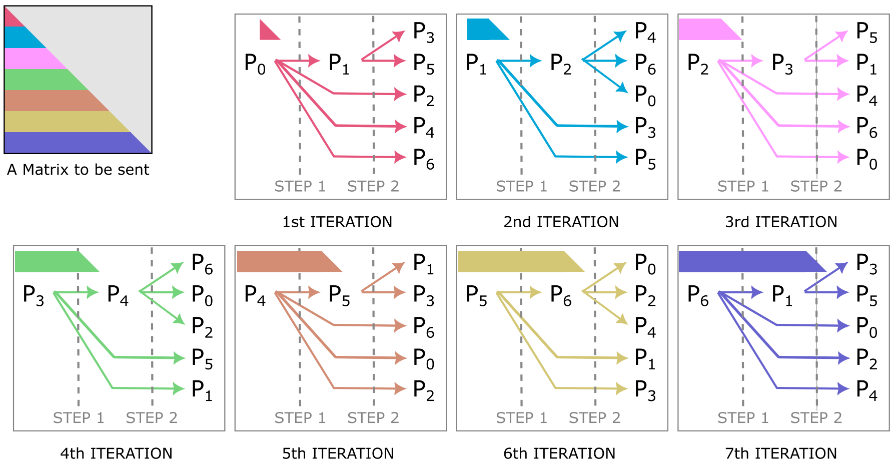

Fig. 5 Sample of our custom broadcast-based communication pattern when seven nodes intervene in the computation. The matrix initial distribution is shown at the top of the figure

a custom version of a broadcast-based communication strategy for the communication and exchange of the local A_i parts across nodes in the main loop. The choice of the communication strategy is orthogonal to the choice of the data distribution.

Algorithm 4 describes our custom broadcast-based communication strategy to communicate the A_i submatrices among the different processes during the distributed TRMM computation. The process is graphically illustrated on Fig. 5. If a classical broadcast strategy were employed, at each iteration, the root node (whose identifier is the same as the iterator) would broadcast its A_{local} matrix to the rest of the nodes, and then all the nodes (including the sender) compute the TRMM with that A_i submatrix and their particular B_{local} matrix. Our approach works in two stages on each iteration, explicitly unrolling the first step of a classical tree-based broadcast algorithm. In the first step, the root node communicates the A_{local} matrix to the process with the next rank. Then, in the second stage, the two processes that have a copy of the data issue a broadcast. Each one broadcasts the data to one half of the rest of nodes. In particular, the sender that has an even rank broadcasts the data to the rest of the even nodes, and the sender with an odd rank does the same for the rest of the odd nodes. For example, if nine nodes are taking part in the process, at the first iteration, the node zero (root node), which is the first sender, will send its A_{local} matrix to the node one (second sender), and then the node zero broadcasts its A_{local} matrix to nodes two, four, six, and eight, while, at the same time, the node one sends the A_{local} matrix received from node zero to nodes three, five, and seven. Due to the big amount of data involved in a whole panel communication, this customized broadcast strategy increases the communications overlap and improves the performance of a single MPI broadcast call.

Algorithm 4 Customized broadcast-based distributed TRMM

```
Require: $A \in \mathbb{R}^{m\times m}, B \in \mathbb{R}^{m\times n}$ $with\ m, n \in \mathbb{N}, p \in \mathbb{N}$ ▷ p corresponds to the node id
  $C \leftarrow \mathbb{R}^{m\times m}$                                ▷ C initially filled with zeros
  $A_{local}, B_{local} \leftarrow initial\_distribution(A, B)$          ▷ Each node gets its $A_i$ and $B_i$
  for $p' \leftarrow 0$ to $(n\_procs - 1)$ do
      $second\_sender \leftarrow p' + 1$
      if $second\_sender = n\_procs$ then
          $second\_sender \leftarrow n\_procs\ \%\ 2\ = 0\ ?\ 0\ :\ 1$
      end if
      if $p' = p$ then
          $A' \leftarrow A_{local}$
          $send(A',\ to : second\_sender)$
      else
          if $p = second\_sender$ then
              $recv(A',\ from : p')$
          end if
      end if
      if $p' = p\ \ OR\ \ second\_sender = p$ then
          $receiver \leftarrow p + 2$
          if $receiver >= n\_procs$ then
              $receiver \leftarrow p\ \%\ 2\ = 0\ ?\ 0\ :\ 1$
          end if
          while $receiver\ != p$ do
              $send(A',\ to : receiver)$
              $receiver \leftarrow receiver + 2$
              if $receiver >= n\_procs$ then
                  $receiver \leftarrow p'\ \%\ 2\ = 0\ ?\ 0\ :\ 1$
              end if
          end while
      else
          $origin \leftarrow p'\ \%\ 2\ =\ p\ \%\ 2\ ?\ p' : second\_sender$
          $recv(A',\ from : origin)$
      end if
      $C_{p',p} \leftarrow TRMM(A', B_{local})$
  end for
  $C = collect\_result()$                 ▷ The $C_i$ submatrices are gathered to form $C$
  return $C$
```

4 Experimental results

In this section, we present the experimental study to evaluate the performance results of the different combinations of the techniques proposed in the previous section. We first describe the platform used for the performance evaluation. Then, we enumerate the different software versions and provide information about the source code availability. Finally, we present the experimental results associated with our proposal, including a comparison with the equivalent performance associated with the ScaLAPACK and SLATE libraries.

4.1 Platform, software versions, and code availability

The tests presented in this section have been conducted in the FinisTerrae III supercomputer at CESGA (Centro de Supercomputación de Galicia) supercomputing center in Spain. It is equipped with 118 TB of memory and 354 computing nodes, each composed of two Intel Xeon Ice Lake 8352Y processors, of 32 cores each, at 2.2 GHz. The nodes are interconnected through a Mellanox Infiniband HDR 100 network. The peak performance of the machine is 4.36 PFLOPS.

As this cluster is a shared resource, the execution of different user jobs is controlled by a queue system. The queue system policies prioritize the shared usage of the whole system. There are usually many jobs in the queues requiring a small number of nodes (from two to eight). Thus, a small number of nodes often become idle when a small job finishes, and other small jobs are eager to occupy them. The higher the number of nodes required in a job, the exponentially longer it waits in the queues to achieve enough privilege to force the system to wait with idle nodes until there are enough free nodes to accommodate it. In our experience, experimenting with more than 36 nodes is a daunting task. We have experienced similar problems when conducting executions with a high number of nodes in other similar supercomputing centers. Thus, we consider this platform a valid example of a generic commodity cluster with up to 36 nodes.

The operating system is Rocky Linux 8.4 (Green Obsidian) for all the nodes. All the codes are compiled using the GCC 11.2.1 compiler. Moreover, we use Intel's oneAPI 2023.2.1 and its Intel(R) MPI Library for Linux OS and MKL Library. For the comparisons, we use the ScaLAPACK version in Intel's oneAPI 2023.2.1 and SLATE v2022.07.00.

All our source codes are publicly available at https://github.com/uva-trasgo/UVaTRMM/.

4.2 Evaluation

To evaluate the performance of our proposal, we have measured the elapsed time when executing the TRMM over two different matrix sets: a small one of dimension 10,000 $\times$ 10,000 and a larger one of dimension 30,000 $\times$ 30,000. For each case, we have tested four different configurations that are associated with all the possible combinations arising from the different data partitioning (regular or balanced) and buffer shapes (boxes or trapezoid), using in all cases our custom broadcast-based communication pattern. Moreover, as we have already anticipated, we have also measured the elapsed time when executing the TRMM from SLATE and ScaLAPACK over the same two matrices, using different block sizes for they block-cyclic distribution; as 256 and 512 were the block sizes offering the best results, those are the ones reflected in this section. Tables 1 and 2 show the results observed

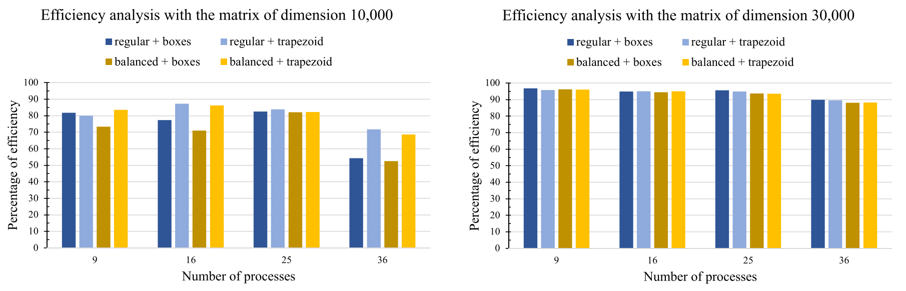

Fig. 6 Efficiency observed when scaling up the number of MPI processes employed to compute the TRMM of the small matrix (left) and the large one (right) with our proposed approaches

Table 1 Elapsed time (in seconds) when executing the TRMM from SLATE, ScaLAPACK, and our proposal over a matrix of dimension $10,000 \times 10,000$ using up to 36 processes

Matrix of dimension 10,000 × 10,000		Number of processes				
		1	9	16	25	36
ScaLAPACK	Block cyclic with nb = 256	14.335	1.969	1.243	0.895	0.679
	Block cyclic with nb = 512	14.368	2.035	1.25	0.900	0.756
SLATE	Block cyclic with nb = 256	15.693	1.948	1.176	0.792	0.603
	Block cyclic with nb = 512	14.862	2.041	1.212	0.852	0.741
Our proposal	reg, boxes	13.231	1.798	1.068	0.641	0.677
	reg, trapezoid	13.243	1.838	0.948	0.631	0.512
	bal, boxes	13.204	1.793	1.162	0.643	0.698
	bal, trapezoid	13.245	1.760	0.959	0.643	0.536

Table 2 Elapsed time (in seconds) when executing the TRMM from SLATE, ScaLAPACK, and our proposal over a matrix of dimension $30,000 \times 30,000$ using up to 36 processes

Matrix of dimension 30,000 × 30,000		Number of processes				
		1	9	16	25	36
ScaLAPACK	Block cyclic with nb = 256	366.059	44.009	25.903	17.756	12.731
	Block cyclic with nb = 512	366.158	45.095	25.878	17.445	12.505
SLATE	Block cyclic with nb=256	421.644	48.739	28.429	18.556	13.097
	Block cyclic with nb = 512	395.306	47.177	27.607	18.480	13.152
Our proposal	reg, boxes	347.098	39.823	22.860	14.522	10.721
	reg, trapezoid	345.230	40.027	22.705	14.552	10.704
	bal, boxes	345.760	39.943	22.881	14.759	10.897
	bal, trapezoid	345.608	39.952	22.730	14.761	10.882

for computing the TRMM over the 10, 000 and the 30, 000 dimension matrices, respectively. It is important to remark that our proposal and the code that uses ScaLAPACK rely on Intel's oneAPI MKL library to execute the GEMM and TRMM kernels in the process of computing the local parts at each stage. We have tested these implementations using Intel threads. Nevertheless, this is beyond our control when using SLATE.

Evaluating our proposal's scalability, we have observed that all of our combinations scale fairly well when increasing the number of processes. Taking as a reference the execution time observed when using one process (with each evaluated configuration), we have computed the ideal efficiency and compared it with the actual one. The results are illustrated in Fig. 6. As can be observed, with the small matrix set, the efficiency is between 80 and 90% with up to 25 processes, and decreases to 70% when using 36 processes, while, with the large matrix, it is between 90 and 95% in any case. The difference observed with the smaller matrix is that, with such a small size (10,000 × 10,000), there is not enough work to compensate for the communication costs when it is distributed among 36 nodes, and it is not worthwhile to use so many processes. Nevertheless, a good efficiency is observed with the larger matrix, which means that any of our proposed approaches present a good efficiency when increasing the number of processes employed if the matrix is big enough, and thus, there is a considerable workload at each node.

In the next sections, we evaluate in detail each of our proposed approaches, determining whether they are beneficial or not, as well as in which cases there are benefits. Finally, we analyze the results of our proposal by comparing them with those obtained with SLATE and ScaLAPACK.

4.2.1 Alternatives to full panels: boxes and trapezoids

An aspect we explore with our proposal is using custom shapes that avoid sending (all or part of) the triangular matrix zeros: Boxes and trapezoids. For comparing our boxes-based against trapezoid-based results, we conducted the following calculation:

$$\%\ of\ difference = \frac{t\_trapezoid - t\_boxes}{t\_boxes} \cdot 100 \tag{16}$$

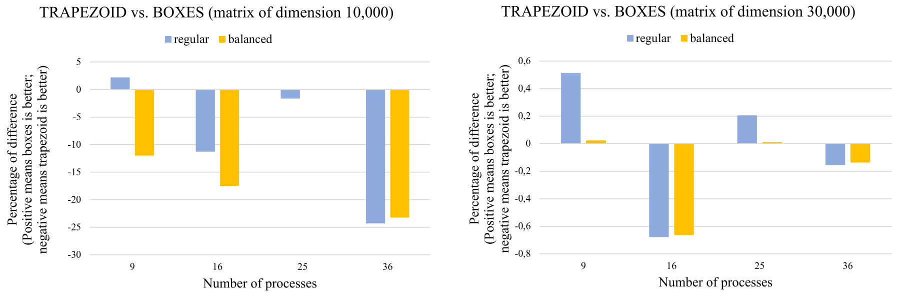

Fig. 7 Comparison between trapezoid and boxes data shapes for the 10K dimension (left) and 30K dimension (right) matrices. Note that the vertical axis range is different at each plot

Figure 7 reflects the results obtained based on Eq. (16). Note that, in each case, we fix regular/balanced to ensure fair comparisons; in other words, we only modify one of the evaluation axes.

In this case, using trapezoid shapes generally reduces the execution time when operating over small matrices (up to 25% improvement, although the execution times are very small and in practice this difference does not represent a large gain). However, the differences observed with the large matrix are negligible (between − 0.67 and 0.51%), so it is not worth the effort of using trapezoid shapes instead of boxes when the matrices are large enough. The reason that justifies this is the fact that, with large matrices, as we are using asynchronous communications, the data transfer overlap with the computation process, and thus the additional effort of creating a trapezoidal data structure instead of simply a rectangular one (box) is not leveraged, as the impact of avoiding sending a small triangle full of zeros is almost non-representative compared to the computations with large matrices and the marshaling of the rest of the big rectangular parts of the panel. In fact, with large matrices, the computation is much more representative than the asynchronous communication, regardless of the data structures that are sent.

4.2.2 Does it pay off to balance the partitioning?

Another aspect that we have also explored with our proposal is the possibility of balancing the number of rows assigned to each process based on the number of elements over which the computations are later performed. To compare our balance-based against regular-based results, we conducted the following calculation:

$$\%\ of\ difference = \frac{t\_balanced - t\_regular}{t\_regular} \cdot 100 \tag{17}$$

Figure 8 shows the results obtained based on Eq. (17). Note that, in each case, we fix boxes/trapezoids to ensure fair comparisons; in other words, we only modify one of the evaluation axes.

The global idea extracted from the evaluation is that, with small matrices, it can be beneficial to balance the partitioning when there is a small number of processes.

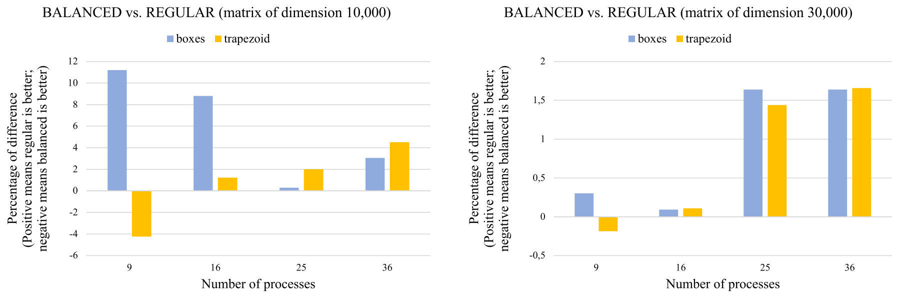

Fig. 8 Comparison between balancing and regular partitioning for the 10K dimension (left) and 30K dimension (right) matrices

Nevertheless, with large matrices, the impact of balancing the partition is negligible, or even negative, so balancing the partition is not worthwhile.

Let us now analyze the small matrix results in more detail. In this case, we see a maximum improvement of 4% due to balancing the partition. When using box shapes instead of trapezoids and up to 9 processes, the impact is greater, as more elements are included in the data structures that are sent. Thus, the imbalance in the regular partitioning is more noticeable. On the contrary, when using more processes, it is not worth balancing the partition, as the differences observed in terms of the number of elements assigned to each process are reduced when using a regular partition (because fewer elements are assigned to each one).

4.3 Comparison with ScaLAPACK and SLATE

In this section, we present a comparison between our results and those observed when executing the TRMM from ScaLAPACK and SLATE, respectively, in Figs. 9 and 10. The metric we use for the comparison against both libraries is calculated as:

$$\%\, of\; improvement\,(library) = \frac{t\_library - t\_our\_proposal}{t\_library} \cdot 100$$

When comparing our proposal results against ScaLAPACK, we observe that we generally offer better results, regardless of the approach from our proposal that we

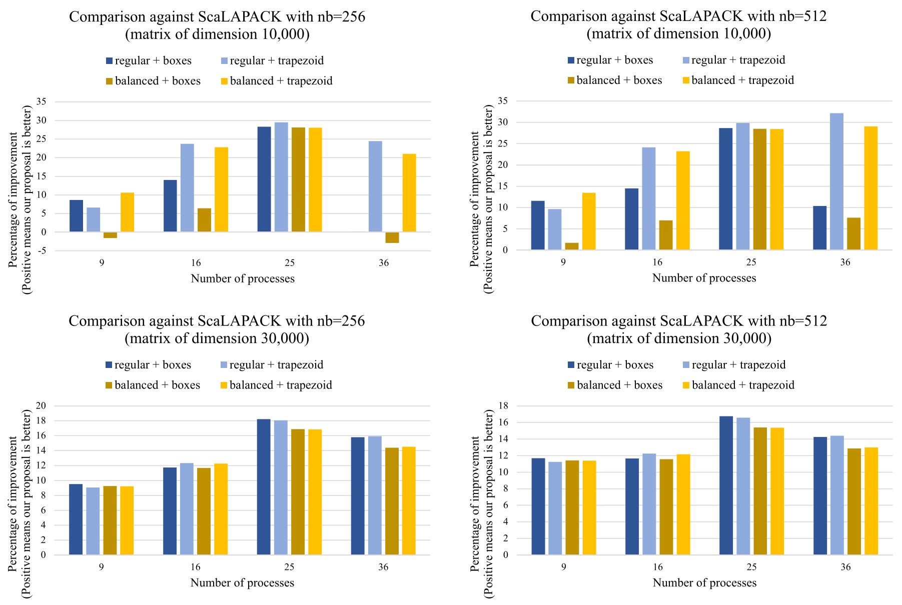

Fig. 9 Comparison between ScaLAPACK and our proposal TRMM for the 10K dimension (top) and 30K dimension (bottom) matrices. Note that, with ScaLAPACK, we have used two different block sizes for the block cyclic distributions; the left plots correspond to nb = 256, the right ones to nb = 512

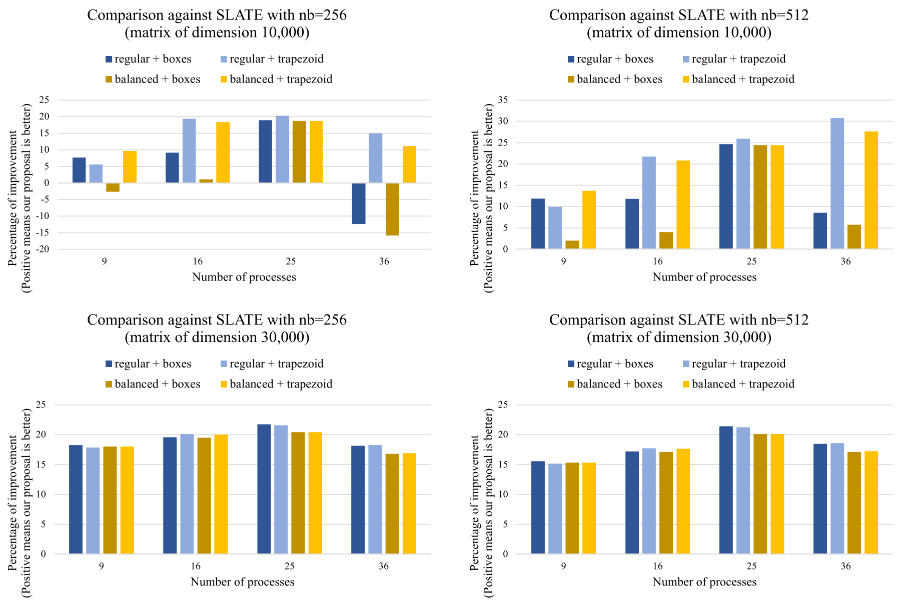

Fig. 10 Comparison between SLATE and our proposal TRMM for the 10K dimension (top) and 30K dimension (bottom) matrices. Note that, with SLATE, we have used two different block sizes for the block-cyclic distributions; the left plots correspond to nb = 256, and the right ones to nb = 512

consider. There are only two cases in which ScaLAPACK (with nb = 256) beats our proposal (by less than 5%). These cases appear when we use balanced + boxes with 9 or 36 processes to compute TRMM of the small matrix. These cases appear when the execution times are around 1 s or less. Thus, it turns out to be almost negligible in terms of execution time difference.

Analyzing in depth the results of the comparison with the small matrix, we observe that our proposal follows a clear trend: With 9 processes there is a 10–15% gain, and with more processes it is between 20 and 30%, considering our best times, which are either using regular or balanced partitioning and trapezoid shapes. In the case of the large matrix, our proposal improvement varies less, offering between 10 and 18% better execution times.

When comparing our proposal results against SLATE, we again observe that we generally offer better results, although the difference with the small matrix is less homogeneous than that observed with respect to ScaLAPACK. In this case, there are three occasions when SLATE (using nb = 256) is faster than our approach with the small matrix. They appear when using 9 and 36 processes with our balanced + boxes approach, and when using 36 processes with our regular + boxes approach. Nevertheless, it is important to highlight again that this happens only with the small matrix, when the execution times are around 1 s or less. Thus, it turns out again to be almost negligible in terms of execution time difference.

A further study of the results with the small matrix reveals that our proposal offers improvements between 10 and 20%, considering our best times, which are either using regular or balanced partitioning and trapezoid shapes. In the case of the large matrix, our proposal improvement varies less, offering between 15 and 22% better execution times.

It is important to highlight that the reason why we observe worse improvement rates when evaluating some of our TRMM with small matrices is that we do not perform any improvement on the linear algebra TRMM and GEMM kernels on which we rely, while the state of the art libraries (specially SLATE) incorporate computational kernel optimizations that are more noticeable in the small matrix case, when the data distribution and communication patterns (the problem that we tackle in this work) have less impact.

5 Conclusion

In this work, we present a proposal that explores four different axes in order to offer new distributed TRMM computation approaches that improve the state-of-the-art performance for the SLATE and ScaLAPACK libraries in commodity clusters equipped with up to 36 nodes.

From the experiments conducted, we extract the following main conclusions:

- Using panel partitions instead of tiling can offer good results in commodity clusters.
- Regarding the communication pattern selection, performance is significantly improved by using a custom broadcast-based communication pattern..
- With respect to the special data shapes to be distributed, we have observed that it is not worth using trapezoid shapes instead of boxes when the matrices are large enough. Nevertheless, with small matrices communicating trapezoids is beneficial.
- The consideration of balancing the partition, so that each process sends/receives and calculates a similar number of elements, is worthwhile only with small matrices.
- Our proposal is capable of computing the distributed TRMM faster than implementations of the current state-of-the-art SLATE and ScaLAPACK libraries. Thus, SLATE and ScaLAPACK users and developers could use the outcomes of this work to enhance their implementations.

We also wish to highlight the fact that, as we have mentioned before, all our implementations are fully available and Open Source.

Future work may include the study of the effect of using partition and communication techniques similar to the ones proposed in this work for GEMM or other routines; introduce the use of GPUs or other accelerator devices to do the coarse-grain local matrix-panel computations; or explore the usage of the proposed partition techniques inside flexible libraries, such as SLATE, that allow the usage of partitions tailored by the user.

Acknowledgements This work was supported in part by the *Spanish Ministerio de Ciencia e Innovación* and by *the European Regional Development Fund (ERDF)* program of the European Union, under Grant

PID2022-142292NB-I00 (NATASHA Project); and in part by *the Junta de Castilla y León - FEDER Grants*, under Grant VA226P20 (PROPHET-2 Project), Junta de Castilla y León, Spain. This work was also supported in part by grant TED2021-130367B-I00, funded by MCIN/AEI/10.13039/501100011033 and by "European Union NextGenerationEU/PRTR". The CESGA - Finisterrae III supercomputing resources were accessed thanks to the project IM-2023-3-0020 from the Red Española de Supercomputación (RES).

Author contributions Inmaculada Santamaria-Valenzuela, Rocío Carratalá-Sáez, Yuri Torres, and Arturo Gonzalez-Escribano designed, implemented, and validated the code. Rocío Carratalá-Sáez and Yuri Torres carried out the experimentation and generated the figures. All authors wrote and reviewed the main manuscript text.

Funding Open Access funding provided thanks to the CRUE-CSIC agreement with Springer Nature. Project PROPHET-2 VA226P20 funded by Consejería de Educación of Junta de Castilla y León, Ministerio de Economía, Industria y Competitividad of Spain, European Regional Development Fund (ERDF) program of the European Union. Project PID2022-142292NB-I00 (NATASHA Project) funded by the Spanish Ministerio de Ciencia e Innovación and the European Regional Development Fund (ERDF) program of the European Union. Project TED2021-130367B-I00, funded by MCIN/AEI/10.13039/501100011033 and by "European Union NextGenerationEU/PRTR". Project IM-2023-3-0020 granted by Red Española de Supercomputación (RES).

Availability of data and materials The source code is available at https://github.com/uva-trasgo/UVaTRMM/.

Declarations

Conflict of interest The authors have no Conflict of interest as defined by Springer, or other interests that might be perceived to influence the results and/or discussion reported in this paper.

Ethical approval Not applicable.

Consent to participate Not applicable.

Consent for publication Not applicable.

References

1. Sterling TL (2011). In: Padua D (ed) Clusters. Springer, Boston, pp 289–297. https://doi.org/10.1007/978-0-387-09766-4_18
2. Blackford LS, Choi J, Cleary A, D'Azevedo E, Demmel J, Dhillon I, Dongarra J, Hammarling S, Henry G, Petitet A, Stanley K, Walker D, Whaley RC (1997) ScaLAPACK Users' Guide. Soc Ind Appl Math. doi 10(1137/1):9780898719642
3. Gates M, Kurzak J, Charara A, YarKhan A, Dongarra J (2019) Slate: design of a modern distributed and accelerated linear algebra library. In: Proceedings of the International Conference for High

Performance Computing, Networking, Storage and Analysis. SC'19. Association for Computing Machinery, New York. https://doi.org/10.1145/3295500.3356223
4. Geijn RA, Watts J (1997) SUMMA: scalable universal matrix multiplication algorithm. Concurr Pract Exp 9(4):255–274
5. Manin V, Lang B (2020) Cannon-type triangular matrix multiplication for the reduction of generalized HPD eigenproblems to standard form. Parallel Comput 91:102597. https://doi.org/10.1016/j.parco.2019.102597
6. Sankaran A, Alashti NA, Psarras C, Bientinesi P (2022) Benchmarking the linear algebra awareness of tensorflow and pytorch. In: 2022 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW), pp 924–933. https://doi.org/10.1109/IPDPSW55747.2022.00150
7. Goto K, Geijn RA (2008) Anatomy of high-performance matrix multiplication. ACM Trans Math Softw 34(3):1–25. https://doi.org/10.1145/1356052.1356053
8. Zee FG, Geijn RA (2015) BLIS: A framework for rapidly instantiating BLAS functionality. ACM Trans Math Softw 41(3):14–11433
9. Wang Q, Zhang X, Zhang Y, Yi Q (2013) Augem: automatically generate high performance dense linear algebra kernels on x86 CPUs. In: SC '13: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis, pp 1–12. https://doi.org/10.1145/2503210.2503219
10. Kalinkin A, Anders A, Anders R (2015) Intel Math Kernel Library PARDISO for Intel Xeon PhiTM Manycore Coprocessor. Appl Math 6:1276–1281. https://doi.org/10.4236/am.2015.68121
11. Limited A. ARM Performance Libraries Reference Guide. https://developer.arm.com/documentation/101004/2030/
12. (AMD) AMD. AMD Optimizing CPU Libraries (AOCL). https://www.amd.com/en/developer/aocl.html#documentation
13. Anderson E, Bai Z, Bischof C, Blackford LS, Demmel J, Dongarra J, Du Croz J, Greenbaum A, Hammarling S, McKenney A, Sorensen D (1999) LAPACK Users' Guide, 3rd edn. Society for Industrial and Applied Mathematics. https://doi.org/10.1137/1.9780898719604
14. Geijn RA, Watts J (1995) Summa: Scalable universal matrix multiplication algorithm. Technical report, USA
15. Dongarra JJ, Du Croz J, Hammarling S, Duff IS (1990) A set of level 3 basic linear algebra subprograms. ACM Trans Math Softw 16(1):1–17. https://doi.org/10.1145/77626.79170
16. Forum MP (1994) MPI: a message-passing interface standard. Technical report, USA
17. Corp I (2023) Intel math kernel library. https://software.intel.com/en-us/mkl
18. Du P, Tomov S, Dongarra J (2012) Providing GPU capability to LU and QR within the scalapack framework. Technical Report UT-CS-12-699 (2012)
19. D'Azevedo E, Hill JC (2012) Parallel LU factorization on GPU cluster. Proc Comput Sci 9:67–75. https://doi.org/10.1016/j.procs.2012.04.008
20. Chandra R, Dagum L, Kohr D, Menon R, Maydan D, McDonald J (2001) Parallel programming in OpenMP. Morgan kaufmann
21. Whaley RC, Dongarra JJ (1999) Automatically tuned linear algebra software. In: Proceedings of the Ninth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1999, San Antonio, Texas, USA, March 22–24, 1999. SIAM
22. Tukanov N, Srinivasaraghavan R, Moreira JE, Low TM (2022) Modeling matrix engines for portability and performance. In: 2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS), pp 1173–1183
23. Valero-Lara P, Catalán S, Martorell X, Usui T, Labarta J (2020) sLASs: A fully automatic auto-tuned linear algebra library based on OpenMP extensions implemented in OmpSs (LASs library). J Parallel Distrib Comput 138:153–171. https://doi.org/10.1016/J.JPDC.2019.12.002