Task-Based Parallel Programming for Scalable Algorithms: application to Matrix Multiplication

Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Julien Herrmann , Antoine Jego

ISRN INRIA/RR--9461--FR+ENG

ISSN 0249-6399

RESEARCH REPORT

N° 9461

February 2022

Project-Team HiePACS

Task-Based Parallel Programming for Scalable Algorithms: application to Matrix Multiplication

Emmanuel Agullo[*], Alfredo Buttari[†], Abdou Guermouche[‡], Julien Herrmann [§], Antoine Jego[¶]

Project-Team HiePACS

Research Report n° 9461 — February 2022 — 29 pages

Abstract: Task-based programming models have succeeded in gaining the interest of the high-performance mathematical software community thanks to how they relieve part of the burden of developing and implementing distributed-memory parallel algorithms in an efficient and portable way. In increasingly larger, more heterogeneous clusters of computers, these models appear as a way to maintain and enhance more complex algorithms. However, task-based programming models lack the flexibility and the features that are necessary to express in an elegant and compact way scalable algorithms that rely on advanced communication patterns. We show that the Sequential Task Flow paradigm can be extended to write a compact yet efficient and scalable General Matrix Multiplication. This extension required few modifications to the StarPU runtime system. The final implementation is shown to be competitive up to 32,768 cores with state-of-the-art libraries and may outperform them on some specific problem configurations.

Key-words: Mathematical software, parallelism, task-based programming, scalability, communication patterns, matrix multiplication

[*] Inria, France
[†] CNRS-IRIT
[‡] U. Bordeaux
[§] CNRS-IRIT
[¶] CNRS-IRIT

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Programmation parallle base de tches pour algorithmes passant l'chelle: application au produit de matrices

Résumé : Les modles de programmation base de tches ont russi susciter l'intrt de la communaut des logiciels mathmatiques de haute performance grce la manire dont ils soulagent une partie du fardeau que reprsentent le dveloppement et la mise en uvre efficace et portable d'algorithmes parallles mmoire distribue. Dans des grappes d'ordinateurs de plus en plus grandes et htrognes, ces modles apparaissent comme un moyen de dvelopper et maintenir des algorithmes plus complexes. Cependant, les modles de programmation bass sur les tches manquent de flexibilit et les caractristiques ncessaires pour exprimer de manire lgante et compacte des algorithmes passant l'chelle se basant sur des schmas de communication avancs. Nous montrons que le paradigme de flot de tches squentiel (STF) peut tre tendu pour crire une multiplication matricielle passant l'chelle. L'implantation finale est comptitive jusqu' 32 768 curs avec les bibliothques de pointe et peut mme les surpasser dans certaines configurations spcifiques.

Mots-clés : Logiciels mathmatiques, paralllisme, programmation base de tches, passage l'chelle, schmas de communication, multiplication matricielle

1 Introduction

Recent trends in supercomputer architectures have widened the gap between the speed of computations and the speed of data transfers. Modern supercomputers are composed of fat nodes equipped with many computational cores and specialized processing units (such as long vector units or GPUs) that can process operations at extremely fast rates. On the other hand, the speed of networks that interconnect these nodes has increased by a much smaller factor. For this reason, and more than in the past, a considerable effort is being devoted to the development of dense and sparse linear algebra algorithms that communicate better or communicate less. Although the literature of this subject is extremely vast, many of these scalable algorithms share some features. Computations are commonly arranged in such a way that the use of non-blocking or efficient collective communications is maximized van de Geijn and Watts (1997); Schatz et al. (2016). Data reduction patterns are used to increase parallelism and reduce the weight of communications on the critical path Demmel et al. (2012); van de Geijn and Watts (1997); Schatz et al. (2016). In some cases, communications are reduced at the price of a moderate or controllable overhead in terms of memory or operations as in 2.5D or 3D algorithms Schatz et al. (2016); Agarwal et al. (1995); Solomonik and Demmel (2011); Ashcraft (1993); Sao et al. (2019).

With algorithms becoming more and more complex and supercomputers architectures more heterogeneous, the need has raised for programming models that relieve the high performance computing expert from the burden of mixing multiple programming models and interfaces (e.g., MPI, OpenMP and CUDA). While efforts have been made in the community to provide abstraction layers to these programming models, such efforts shift the focus of experts away from the development of efficient and scalable algorithms. Task-based parallelism is one model to help them: it allows for a high level description of the workload in the form of a directed acyclic graph (DAG) where nodes correspond to tasks (i.e., elementary operations) and edges the dependencies among them or, equivalently, data transfers. This model has been widely used in the past; one notable example is the MUMPS Amestoy et al. (2001) sparse direct solver that implements task-based parallelism by means of the MPI programming interface. The task-based parallel programming paradigm has recently known a renewed interest thanks to the emergence of novel programming models that allow for a simpler construction of the DAG. Two well known examples are the sequential task flow (STF) where tasks dependencies are inferred from data access modes (more details on this model are provided in section 2.1) and the parameterized task graph (PTG) where task dependencies are defined by means of rules provided by the programmer. These programming models are available, through dedicated programming interfaces, in various runtime systems (or, simply, runtimes) whose use in the high performance computational linear algebra domain is increasingly popular. Well known runtime systems include StarPU Augonnet et al. (2011) (which provides an interface for the STF programming model), ParSEC Bosilca et al. (2013) (STF, PTG and others) and OpenMP from version 4.0 (STF). The use of task-based parallelism through modern runtime systems has several attractive features. It allows for a relatively easy and portable programming of heterogeneous architectures including distributed memory multinode systems running over multi and manycores as well as accelerators such as GPUs: the runtime system takes care of deploying tasks on the available processing units (according to a defined scheduling policy) and of transferring the data required for their execution through the network or a dedicated bus. This approach presents great modularity. For example, it allows for choosing among pre-defined task scheduling policies or developing new ones; because these are implemented within the runtime system, they are not tied to a specific algorithm or application but can be reused transparently. By the same token, as we will show below, it is easy to switch between different communication libraries. Numerous studies exist that demonstrate the effectiveness of this approach for different applications and algorithms mostly on shared memory systems Agullo et al. (2016) but also, and increasingly so, on distributed memory ones Herault et al. (2019). Questions about the scheduling of complex algorithms in a shared-memory setting Buttari et al. (2009) but also in a distributed-memory setting have been raised in early development of libraries like Plasma or Magma Agullo et al. (2009) or SuperMatrix Igual et al. (2013). Nevertheless, concerns still remain about the expressiveness of task-based parallelism and programming models and their capability to implement distributed memory algorithms with complex features such as those mentioned above.

The purpose of our work is to address these concerns and show that, through a suitable extension of the state-of-the-art STF model for distributed memory machines Agullo et al. (2017a), it is possible to implement complex scalable algorithms; the resulting code is easy to maintain and improve, yet very efficient and portable. As a reference, we will use the dense generic matrix-matrix product (GEMM) and its distributed memory parallel variants such as the SUMMA van de Geijn and Watts (1997) and 2.5D Solomonik and Demmel (2011) algorithms which possess most of the above mentioned algorithmic features. Experimental results will show that the proposed approach provides equivalent, if not better, performance than reference libraries with a code which is only slightly more complex than the classic three nested loops of a basic, sequential, matrix multiplication code.

2 Background

2.1 The sequential task flow programming interface

As explained above, task-based parallelism can be conveniently implemented using specifically designed programming models such as sequential task flow (STF) or parameterized task graph (PTG). In this document we will focus on the STF one, sometimes also referred to as *superscalar* since it mimics the functioning of superscalar processors where instructions are issued sequentially from a single stream but can actually be executed in a different order and, possibly, in parallel depending on their mutual dependencies. The STF model relies on a task insertion or submission primitive which allows for creating a task. The insertion is non blocking, which means that the control is immediately returned to the caller and the execution of the task is deferred. Upon insertion of a task, the caller must specify the data used by the task and whether the task accesses these data in read (`R`), write (`W`) or read-write (`RW`) mode. Based on the order in which tasks are inserted and their data access modes, dependencies between tasks can be easily determined and the DAG of tasks automatically built.

In the case of a shared memory parallel computer, the STF model commonly relies on the use of a *master* process which is in charge of inserting the tasks and multiple *workers* which are in charge of executing them on the available processing units. Multiple types of workers may exist if different processing units are available such as CPU cores and GPUs. In our work we will not cover the case of heterogeneous systems: while a code relying on the STF model can benefit from both CPU cores and GPUs, our work focuses on homogeneous architecture that are challenging enough with regards to the programming model concerns. In the case of a distributed memory machine, multiple masters exist which communicate by exchanging messages; these communications can be internally implemented by the runtime system through the MPI standard but other communication interfaces or libraries can also be used. These communications essentially correspond to dependencies between tasks that are executed by workers associated with different masters. For this reason, in the most basic use of the STF model, all masters must insert all the tasks of the DAG to make sure these communications are correctly detected and executed. This corresponds to the approach based on a concurrent unrolling of the task graph proposed by YarKhan (2012).

This programming model is commonly appreciated because of its simplicity which allows, in a relatively easy way, to transform a sequential code into a parallel one while preserving its readability and maintainability. This advantage must, however, be weighted against potential limitations due to the fact that the DAG must be entirely unrolled by inserting all of its tasks: not only this can be time consuming, but it may require considerable resources for the management of the DAG when it is of large size. Several techniques have been proposed in the literature to alleviate this issue such as the pruning of its traversal Agullo et al. (2017a) or hierarchical tasks Perez et al. (2017); Huang et al. (2021); Kim et al. (2021). The PTG model, on the other hand, has better scalability because the DAG is not explicitly and entirely built but, instead, tasks are efficiently instantiated based on rules defined by the programmer; this, however, comes at the price of a considerably higher programming effort Agullo et al. (2017b).

2.2 Distributed memory scalable GEMM algorithms

The general matrix-matrix multiplication (GEMM) operation, as defined in the BLAS standard, consists in computing

$$C = \alpha \cdot op(A) \cdot op(B) + \beta \cdot C$$

with $C \in \mathbb{R}^{M\times N}$, $op(A) \in \mathbb{R}^{M\times K}$, $op(B) \in \mathbb{R}^{K\times N}$, $op(.)$ being either the identity or the transpose and α and β real scalars in $\mathbb{R}$. The computation can also be executed with complex numbers, replacing the transpose with its conjugate. Without loss of generality, in the remainder of this paper we drop the $op(.)$ operator (and, thus, assume that neither A nor B are transposed) and assume that both α and β are equal to one.

For the purpose of the parallelization, we will suppose that all matrices are partitioned into blocks of size b and that $m = \lceil M/b \rceil$, $n = \lceil N/b \rceil$ and $k = \lceil K/b \rceil$. This will allow us to use efficient sequential BLAS routines for computations on blocks. Based on this assumption, the sequential matrix multiplication can

```
do i=1, m
   do j=1, n
      do l=1, k
         call gemm(A_{i,l}, B_{l,j}, C_{i,j})
      end do
   end do
end do
```

Figure 1: Sequential, blocked GEMM

be simply written as the triply nested loop in Figure 1, where the instruction in the inmost loop computes $C_{i,j} = C_{i,j} + A_{i,l} \cdot B_{l,j}$. Ignoring the data locality issues in NUMA memory configurations, this code can be trivially parallelized for shared memory parallel computers using, for example, loop parallelism or task-based parallelism (more on this will be said in the next section).

When targeting distributed memory parallel computers, the A, B and C matrices must be distributed among the ranks that participate in the computation. Here, and in the remainder of the article, we use the generic word *rank* to denote processes that communicate by exchanging messages; a rank corresponds to a master process in the STF model or to a MPI process in the widely used MPI programming interface. We will assume that a 2D block-cyclic distribution over a $p \times q$ ranks grid is employed because of its wide use in reference dense linear algebra libraries (including ScaLAPACK) and because it complies with the scalable GEMM algorithms described below; for the sake of simplicity, we will also assume that all matrices are aligned, i.e., have a conforming distribution across the ranks grid.

Despite its large arithmetic intensity, the scalability of the GEMM operation on large size supercomputers can be severely limited by the slowness of network communications and many algorithms have been proposed in the literature to overcome this limitation. The Cannon's algorithm Cannon (1969), for example, has been proved to minimize both the communication bandwidth and latency Irony et al. (2004); Ballard et al. (2011). Nevertheless, this algorithm only works on square ranks grids and is, therefore, unpractical. SUMMA Schatz et al. (2016); van de Geijn and Watts (1997); Agarwal et al. (1994) overcomes these limitations of the Cannon's algorithm and has become the most widely adopted algorithm in reference parallel dense linear algebra libraries such as ScaLAPACK Blackford et al. (1997) or PLAPACK van de Geijn (1997). In the SUMMA algorithm, shown in Figure 2, the matrix product is defined as a sequence of outer products where at each iteration $l = 1, \ldots, k$ the l-th column of A is multiplied with the l-th row of B and the result added to C. Each (r, c) rank computes the contribution for the $C_{i,j}$ blocks it owns and, therefore must receive the corresponding $A_{i,l}$ and $B_{l,j}$ blocks; the outer product formulation allows to transfer these blocks using efficient collective communications: the $A_{i,l}$ block is broadcasted to all the ranks in the r-th grid row and the $B_{l,j}$ block is broadcasted to all the ranks in the c-th grid column. A pipelined version of this algorithm was also proposed van de Geijn and Watts (1997) which further reduces the length of the critical path of the parallel matrix product; however, if

```
1  do l=1, k
      forall i, i%p==r :
3        bcast(A_{i,l}, to:(r,:))
      forall j, j%q==c :
5        bcast(B_{l,j}, to:(:,c))
      forall i, i%p==r :
7        forall j, j%q==c :
            call gemm(A_{i,l}, B_{l,j},
   C_{i,j})
9  end do
```

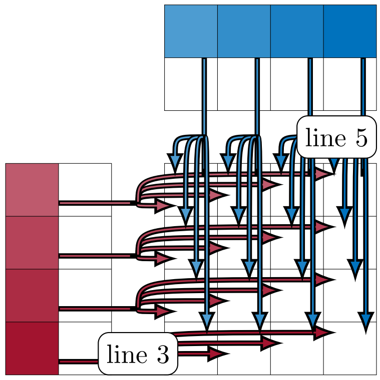

Figure 2: Stationary-C SUMMA algorithm as executed by the (r, c) rank. Pattern of communications are shown for $l = 1$ on an example with a 4×4 ranks grid, $m = n = 4$ and $k = 2$.

non-blocking collective communications are available, the interest of this variant is limited with respect to the basic one.

The SUMMA algorithm presented above is particularly efficient in the case where the C matrix is much larger than A and B because only these two are transferred whereas C stays in place; for this reason we refer to this algorithm as *stationary* C (or stat-C, for short) following the notation proposed by Schatz et al. (2016). Stationary A or stationary B variants can be used in the case where A or B are larger than the other two matrices, respectively; because these two variants behave the same, we only present the first one here. In this algorithm, reported in Figure 3, the matrix-matrix product is defined as a sequence of matrix-panel products where, at each step, the entire A matrix is multiplied by a $B_{*,j}$ block-column producing a $C_{*,j}$ block-column. In this case the A matrix stays in place, the B matrix is transferred using efficient collective communications and locally computed contributions to the C matrix (denoted $C^t_{i,j}$) are assembled using reductions. Note that in this algorithm, because of the cyclic data distribution, the `recv` and `bcast` communications in lines 3 and 5 can be more efficiently implemented using scatter and allgather primitives Schatz et al. (2016).

The scalability of the SUMMA algorithm can be further improved using so-called 2.5D or 3D algorithms Georganas et al. (2012); Schatz et al. (2016). In these algorithms we consider the ranks arranged in a three-dimensional grid of size $p \times q \times s$ and the A, B and C matrices initially distributed among the ranks in the lowest level (0) of this grid, i.e., $(:, :, 0)$. The pseudo-code for the 3D stat-C case executed by the (r, c, h) rank is reported in Figure 4. Here the A and B matrices are partitioned in s parts along the k dimension (columns and rows, respectively) and each part is replicated on one of the higher levels $1, \ldots, s-1$ where a partial stat-C matrix product is computed producing local C^h contributions to the final result. The local contributions are finally assembled into the C matrix using reductions. Clearly, equivalent 3D algorithms can be formulated for stat-A or stat-B SUMMA; we refer the reader to the paper by Schatz et al. (2016) for the related details.

```
do j=1, n
   forall l, l%p==r, l%q==c :
      recv(B_{l,j}, from:(r,j%q))
   forall l, l%p==c :
      bcast(B_{l,j}, to:(:,c))
   forall i, i%p==r :
      forall l, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C^h_{i,j})
      reduce(C^h_{i,j}, to:(i%r,j%q))
end do
```

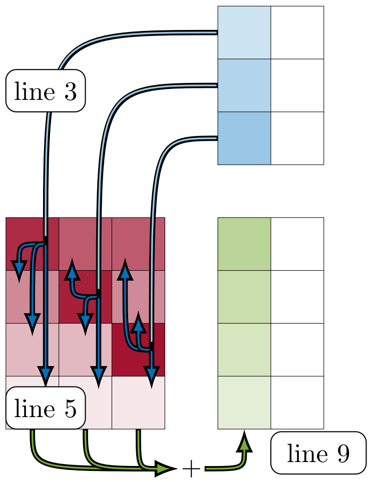

Figure 3: Stationary-A SUMMA algorithm as executed by the (r, c) rank. Pattern of communications are shown for $j = 1$ on an example with a 4×3 ranks grid, $m = 4$,$n = 2$,$k = 3$.

3 STF matrix multiply

3.1 Baseline STF model

The GEMM operation of Figure 1 can be straightforwardly parallelized using the STF model by replacing the calls to the sequential `gemm` on blocks with task insertions through the `insert_task` routine; this delegates the execution of the corresponding operations to the runtime system. The first argument of the `insert_task` method is the operation to be performed upon task execution and is followed by the list of data used by the task (here, the $A_{i,l}$,$B_{l,j}$ and $C_{i,j}$ blocks). For each data, the corresponding access mode is specified though a colon notation: `R` and `RW` denote, read and read-write access modes, respectively. Based on the task insertion order and the tasks data access mode, the runtime system can infer the tasks dependencies, build the corresponding DAG and proceed to schedule its tasks on the available processing units. An example of a generated DAG is shown in Figure 5.

Thanks to the very high arithmetic intensity of the GEMM operation, the code of Figure 5 can achieve very good performance on shared memory, possibly accelerated (e.g., with GPUs) systems, provided that a suitable block size is chosen which provides a good trade-off between parallelism and efficiency of tasks. Additionally, in order to run this code on distributed memory parallel systems, it is enough to make the runtime system aware of the data distribution; for example, in the case of a 2D block-cyclic distribution each (i, j) block of A, B and C is assigned to rank $(i\%p, j\%q)$ of the $p \times q$ ranks grid, where % is the modulo operator. In this case, the runtime system will take care of transferring over the network the blocks needed by a task on the rank where the task is executed. Although this code, based on the baseline model proposed in Agullo et al. (2017a), will be perfectly functional, its performance and scalability can

```
forall j, h*k/s ≤ j < (h+1)*k/s, j%q==c:
   forall i, i%p==r :
      recv(A_{i,j}, from:(r,c,0))

forall i, h*k/s ≤ i < (h+1)*k/s, i%p==r:
   forall j, j%q==c :
      recv(B_{i,j}, from:(r,c,0))

do l=h*k/s, (h+1)*k/s-1
   forall i, i%p==r :
      bcast(A_{i,l}, to:(r,:,h))
   forall j, j%q==c :
      bcast(B_{l,j}, to:(:,c,h))
   forall i, i%p==r :
      forall j, j%q==c :
         call gemm(A_{i,l}, B_{l,j}, C^h_{i,j})
end do

forall i, i%p==r :
  forall j, j%q==c :
     reduce(C^h_{i,j}, to:(r,c,0))
```

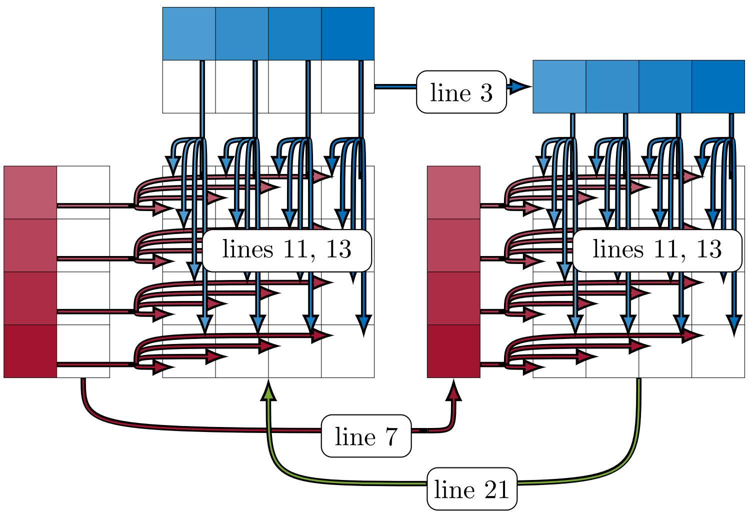

Figure 4: 3D stationary-C SUMMA algorithm as executed by the (r, c, h) rank. Pattern of communications are shown for the complete algorithm on a $4 \times 4 \times 2$ ranks grid with $m = n = 4$ and $k = 2$.

```
do i=1, m
   do j=1, n
      do l=1, k
         call insert_task(gemm, A_{i,l}:R, B_{l,j}:R, C_{i,j}:RW)
      end do
   end do
end do
```

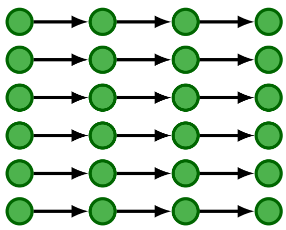

Figure 5: Parallel GEMM using the baseline STF model. A generated DAG is proposed with $m = 2, n = 3, k = 4$ with green circles representing GEMM tasks. Each chain of tasks in the DAG corresponds to contributions to a single block.

be poor compared with what can be achieved with the algorithms described in section 2.2 for a number of reasons.

First of all, this code will not be able to make use of collective communications. In this baseline STF model, communications are expressed by the edges of the DAG which, essentially, define point-to-point data transfers.

Second, this baseline STF model does not allow any control on the mapping of tasks over the $p \times q$ ranks of the grid. Runtime systems implement basic mapping policies where a task is executed on the rank which owns the data that is accessed in read-write mode ($C_{i,j}$, in the case of Figure 5). This can lead to a very large volume of communications, for example, in the case where A and B are much larger than C and will not allow us to use computing ranks that do not own blocks of the C matrix.

Finally, in this baseline STF model it is not possible to take advantage of the commutativity and associativity of certain operations. In our case, it must be noted that all the summations in tasks of the type $C_{i,j} = A_{i,l} \cdot B_{l,j} + C_{i,j}$ for all l can commute or be grouped in any way. This property can be used to improve parallelism or reduce communications. In the baseline STF model, instead these summations are forced to be executed sequentially: because of the order in which tasks are inserted and of the `RW` access mode on the $C_{i,j}$ block, the task that computes $C_{i,j} = A_{i,l+1} \cdot B_{l+1,j} + C_{i,j}$ depends on the one computing $C_{i,j} = A_{i,l} \cdot B_{l,j} + C_{i,j}$.

3.2 Proposed extensions to the STF model

The limitations discussed in the previous section, make the baseline STF model Agullo et al. (2017a) unsuitable for implementing state-of-the-art algorithms for distributed memory systems such as those presented in section 2.2. It must be noted that it is possible to get around some of these limitations through careful programming. For example, it is possible to take advantage of associativity of some operations by declaring temporary data and explicitly inserting tasks to combine partial results; in other words, this amounts to manually implementing reduction operations. This practice, however, leads to complex code which is poorly portable and hard to maintain which, essentially, defeats the purpose of using a high level task-based parallel programming model. The purpose of this section is to present a minimal subset of extensions of the baseline STF model of section 3.1 that allow us to implement scalable algorithms. These features extend both the programming interface and the functionality of a STF-based runtime system while preserving the high level expressiveness of the STF model and, ultimately, the portability and maintainability of the code. In section 4.1 we will discuss the availability of these features in modern runtime systems and possible improvements that lead to better performance.

Reduction tasks The objective of this feature is to provide a mean of taking advantage of the associativity and commutativity of the block-sum operation to improve parallelism. As explained above, this is not possible in the baseline STF model because for all the tasks that compute a $C_{i,j} = A_{i,l} \cdot B_{l,j} + C_{i,j}$ contribution are inserted with `RW` (read-write) access mode on the $C_{i,j}$ block; this induces a chain of dependencies on these tasks according to the order in which they have been inserted. One way to overcome this problem is to introduce a new access mode, which we call `REDUX`. With this access mode, each task will assemble its contribution in a temporary block $D^f_{i,j}$, $f = 1, \ldots, z$; the runtime system takes care of creating all the z temporary blocks and combining their content through dedicated tasks in a transparent way. This reduction phase is carried asynchronously, the only constraint being that it must be completed prior to any other access to the $C_{i,j}$ block with a different access mode; parallelism is also available in this phase which can be used through suitable reduction trees. For this feature to work, it is necessary that the runtime system is informed of how to initialize the temporary blocks and how to combine their values. This can be achieved by declaring to the runtime system two methods, called the *initializer* and the *combiner* (to follow the naming in the OpenMP standard). As temporary blocks are part of a reduction pattern, all tasks that modify a copy of $C_{i,j}$ should be made commutable. It must be noted that a crucial design choice concerns the number z of temporary copies $D^f_{i,j}$ for each $C_{i,j}$ block : this is discussed in section 4.1.

Dynamic collective communications In order to take advantage of efficient and scalable collective communications, whose role is essential in the algorithms presented in section 2.2, we rely on the so-called *dynamic collective communications* feature proposed by Denis et al. (2020). This approach consists in automatically detecting that a data must be transmitted from a source rank to multiple destination ranks; when such a pattern is detected, the corresponding transfers are grouped together and achieved through a collective communication. This feature has a number of interesting properties. First of all, it is completely transparent to the user: no change has to be done at the user-level code but the detection and use of collective communications happen in the communication library underlying the runtime system. Second, these

Algorithm	3D stat-C	3D stat-A	3D stat-B
Executing node of $A_{i,l}B_{l,j}$ (`map`)	$(i\%p, j\%q, \frac{l}{k/s})$	$(i\%p, l\%q, \frac{j}{n/s})$	$(l\%p, j\%q, \frac{i}{m/s})$
Access mode for $C_{i,j}$ (`am`)	$s = 1$: `RW + COMMUTE` $s \neq 1$: `REDUX`	`REDUX`	`REDUX`

Table 1: The mapping of tasks and access modes on the C matrix for the stat-A, stat-B and stat-C 3D GEMM algorithms. They correspond to the values returned by the `map` and `am` methods in Algorithm 6.

collective communications do not rely on the use of subcommunicators which has several advantages as we will explain below. Finally, dynamic collective communications are non-blocking which allows for an effective overlapping of communications and computations.

Tasks mapping This feature amounts to binding one task to one rank, which means that it can be executed by any of the workers associated with that rank. This is simply achieved through an additional `ON_RANK` argument to the `insert_task` routine, which defines the identifier of the rank where the task has to be run on. This feature is essential for the stationary-A and 3D variants where the placement of tasks is not trivially related to the initial data distribution.

3.3 Scalable GEMM with the extended STF model

Using the improved STF model including the features presented in section 3.2, all the GEMM algorithms of section 2.2 can be conveniently implemented as in the pseudo-code of Figure 6. The `map` and `am` function returns the rank where a (i, j, l) task must be executed and the access mode on the $C_{i,j}$ block for this task: depending on the requested variant, specified by the `stat` and `s` (number of grid levels) arguments, this function will return the corresponding mapping and access mode. It must be noted that the same result will be obtained regardless of how the i, j, and l loops are nested because in all cases the DAG of tasks would be, essentially, the same. For the sake of readability, in the pseudo-code of Figure 6 we have omitted the declaration of the initializer and combiner routines for the reductions; these correspond, respectively, to zeroing out all the coefficients of a block and summing two input blocks into an output one.

The central claim of this paper is that the proposed extended STF model allows one to express the three advanced GEMM algorithms (and communication patterns) described in section 2.2 with this extremely compact and simple code (Figure 6), together with an appropriate choice for the mapping and access modes. In Table 1, we present the mapping and access mode corresponding to each of the 3D GEMM algorithms for completeness. In the following paragraphs, we detail such mapping and access mode corresponding to three specific variants : the 2D stat-C, the 2D stat-A and 3D stat-C.

Stationary-C SUMMA In the stationary-C SUMMA algorithm the

rank owning the $C_{i,j}$ block is in charge of all the $C_{i,j} = A_{i,l} \cdot B_{l,j} + C_{i,j}$ tasks for $l = 1, \ldots, k$. Therefore, assuming a 2D block-cyclic distribution on a

```
do i=1, m
   do j=1, n
      do l=1, k
         rank = map(i,j,l,stat,s)
         ACCESS_MODE = am(i, j, l, stat, s)
         call insert_task( gemm,
                    A_{i,l}:R,
                    B_{l,j}:R,
                    C_{i,j}:ACCESS_MODE,
                    rank:ON_RANK)
      end do
   end do
end do
```

Figure 6: Parallel GEMM using the improved STF model. The outputs of function `map` and `am` can be found in Table 1.

$p \times q$ grid, the `RANK` output of the `map` and `am` function will be $(i\%p, j\%q)$. As for the access mode of the $C_{i,j}$ block, this is read-write `RW`; however, to improve parallelism and take advantage of the fact that each rank has multiple workers, we can make all the tasks related to the same $C_{i,j}$ block commutable; this is achieved through the `COMMUTE` access mode which informs the runtime that all of these tasks can be performed in any order. The dynamic collective communications feature will transparently group together all the transfers of a $A_{i,l}$ ($B_{l,j}$) along the $i\%p$ ($j\%q$) ranks row (column) and perform them using an efficient collective communications; this corresponds to the broadcast communications in lines 3 and 5 of the pseudo-code in Figure 2.

Stationary-A SUMMA In the stationary-A SUMMA variant, the rank in charge of computing $C_{i,j} = A_{i,l} \cdot B_{l,j} + C_{i,j}$ is the one that owns the $A_{i,l}$ block; therefore, the `RANK` returned by the `map` and `am` function is $(i\%p, l\%q)$. As for the access mode for the $C_{i,j}$ block, this has to be `REDUX` to achieve the reduction on line 9 of the pseudo-code in Figure 3. The dynamic collective communications feature will detect that the $B_{l,j}$ block has to be sent to all the ranks in the $l\%q$ grid column and achieve these transfers with an efficient broadcast communication corresponding to the `recv` and `bcast` communications in lines 3 and 5 of the pseudo-code in Figure 3. It must be noted that in MPI-based implementations, the broadcasting of $B_{l,j}$ to the $l\%q$ grid column must be done in two steps because the rank owning this block does not necessarily belong the $l\%q$ grid column sub-communicator. Because the dynamic collective communications feature does not rely on the use of sub-communicators, the `recv` communication in line 3 of Figure 3 is not necessary.

3D Stationary-C SUMMA In the 3D stationary-C SUMMA variant, the rank selected by the `map` and `am` function for computing the contribution $A_{i,l} \cdot B_{l,j}$ to the $C_{i,j}$ block is $(i\%p, j\%q, l/(k/s))$. The access mode to the $C_{i,j}$ block is `REDUX` to operate the reductions in line 21 of the pseudo-code in Figure 4. The broadcasts in lines 11 and 13 of Figure 4, are performed straight from the lowest grid level through dynamic collectives without the need for the preliminary point-to-point communications of lines 3 and 7. Similarly to what explained above for the stationary-A variant, these point-to-point communications are necessary to move data on a rank belonging to the sub-communicator where the broadcast happens; because in the dynamic collectives feature the scope of a collective communication is arbitrary and not defined by a sub-communicator, these preliminary copies are unnecessary.

4 Implementation

In this section we discuss the practical implementation of the pseudo-code of Figure 6 which we achieved using the StarPU runtime system and its STF programming API within the qr_mumps Agullo et al. (2016) library. This library has been chosen as it implements parallelism for dense kernels through StarPU for the computation of mathematical routines.

4.1 STF advanced features

The proposed implementation of Figure 6 makes use of the features described in section 3.2. In this section we discuss the availability and use of these features in the StarPU runtime system and, in the case of the reduction tasks feature, some improvements that we have implemented in order to achieve better performance. The task mapping feature is already available in the latest StarPU releases and will not be discussed any further.

The second feature, the dynamic detection of collective communications, is available, through the NewMadeleine library; the reader is referred to the recent work by Denis et al. (2020) for a thorough discussion of this feature. This mechanism is implemented [1] such that the communication pattern used to achieve collective communications can be chosen at run time through an environment variable. In all our experiments we have used a binomial tree. It must be noted, though, that the use of a chain (where each rank forwards the message to only one other rank) essentially leads to an asynchronous implementation of the pipelined SUMMA algorithm (van de Geijn and Watts, 1997, sec. 5.2). We reserve the analysis of this approach for future work.

Regarding the third feature, reduction tasks, recent official releases of StarPU provide the `REDUX` access mode to data. In the available implementation of this feature, every worker executing a task on a data provided with this access mode allocates, initializes and modifies a private copy of it; as soon as another task is submitted that accesses the same data with a different access mode, the runtime system transparently creates tasks that perform a reduction to merge all the private copies into the original one. This design choice might lead to an excessively high, and potentially unnecessary, number of copies of the data; for example, in the case of the stationary-A (3D stationary-C) variant, for each $C_{i,j}$ block, there can be as many copies as q (s) times the number of workers per rank. Another shortcoming of the available StarPU implementation of the `REDUX` access mode is that the reduction step is performed in a sequential fashion, that

[1] The dynamic detection of collective communications is available in a separate branch `nmad-coop-mcast` that, as of writing, is scheduled for merging into the `master` branch

is, all the copies are assembled into the original one sequentially, one after the other. This design maximizes parallelism but may lead to an excessive memory consumption and time consuming reductions when the number of workers is high. Therefore, we improved this feature in two ways. First, we implemented the `RANK_REDUX` access mode in StarPU [2]; here, the number of temporary copies is equal to the number of ranks participating in the reduction which means only one private copy of the data is created per rank and, therefore, shared by all the workers associated with the rank. Although, with this access mode, we do not take advantage of associativity within a rank, we still use commutativity; this means that all the tasks mapped on the same rank that access one data through this access mode can be executed in any order. Second, we extended this implementation in such a way that the reduction tree shape can be chosen (at run time) among multiple shapes; in all our experiments (see section 5) a binary tree was used.

4.2 Handling the general case

The general matrix-matrix multiplication operation includes multiplications by the α and β scalars and the possibility of transposing the A and B matrices as explained in section 2.2. Additionally, in a completely general setting, the matrices may not be aligned on the ranks grid and possibly they can be distributed over different, non-overlapping, sets of ranks. Our concerns lie in implementing all the variants, not in providing the means to choose the most fit one to address a choice of transposition operations, size of the matrices and the grid, etc.. This choice is not any different from other libraries such as ScaLAPACK and is conveyed through the `stat` argument in the pseudo-code of Figure 6.

4.2.1 Transposition and alignment

In MPI the scope of a collective communication is defined by a (sub)communicator. This does not represent a problem in the case where the A, B and C matrices have a conforming distribution over ranks and neither A nor B must be transposed: all blocks of A (respectively, B) already belong to the row (column) subcommunicators where the broadcasts happen in the stat-C SUMMA algorithm (similar observations can be made for the stat-A and B algorithms). In the opposite case, however, a block of A (respectively, B) might reside on a rank which does not belong in the same row (column) subcommunicator as where the broadcast happens. Handling this case requires additional communications and code, as it is the case, for example, in ScaLAPACK. In our approach, though, handling misaligned distributions and matrix transpositions does not require any special care because dynamic collectives do not rely on the use of subcommunicators but are constructed on the fly for any arbitrary set of ranks.

4.2.2 Scaling

Scaling by the α scalar does not require any special handling. Scaling of the C matrix by the β scalar, instead, might be handled in such a way to achieve

[2] The `RANK_REDUX` access mode is coined `STARPU_MPI_REDUX` and is available as part of the master branch of the library

better efficiency and parallelism. Let's assume $k = 2$ in the pseudo-code of Figure 6; this implies that each $C_{i,j}$ block is concerned by the two tasks

$$\begin{aligned} \texttt{task1}: \; & C_{i,j} = \alpha A_{i,1} B_{1,j} + \beta C_{i,j} \\ \texttt{task2}: \; & C_{i,j} = \alpha A_{i,2} B_{1,2} + C_{i,j} \end{aligned}$$

These two tasks, which can be computed using the BLAS GEMM routine, do not commute because of the multiplication by β; this means that the second task can only be performed after the first even if all the data it needs are already available on the computing rank. In our implementation, the scaling by β is performed beforehand through dedicated tasks:

$$\begin{aligned} \texttt{task1}: \; & C_{i,j} = \beta C_{i,j} \\ \texttt{task2}: \; & C_{i,j} = \alpha A_{i,1} B_{1,j} + C_{i,j} \\ \texttt{task3}: \; & C_{i,j} = \alpha A_{i,2} B_{1,2} + C_{i,j} \end{aligned}$$

Here, the first task is relatively cheap and does not require communications and the two other tasks are commutative; this allows for a faster start of computations on all the ranks which might lead to significant performance improvements especially for small size problems.

4.3 Scalability of the STF model

In STF, all the tasks must be created sequentially in order to ensure the dependencies are correctly detected. Although this has some favorable implications (for example it can be used to reliably control the memory consumption of a parallel execution Agullo et al. (2016)), because of this the STF model is commonly regarded as less scalable than other task-based parallel programming models such as PTG, that is, less capable of handling large workloads on large parallel systems. Nevertheless, with some care in the programming and some appropriate techniques, it is possible to overcome this limitation even for very large workloads and computers, as shown by the experimental results in the next section.

4.3.1 DAG pruning

Our implementation employs a DAG pruning technique Agullo et al. (2017a) to prevent each rank from creating all the tasks in the DAG as in the basic concurrent unrolling YarKhan (2012) (see the discussion in section 2.1). Each rank, instead, will create a part of the DAG containing only local tasks (i.e., the tasks it has to execute), remote tasks that use data it owns and remote tasks that produce data needed by local tasks to ensure dependencies are correctly detected at a global scale. In the case of the stat-C 2D SUMMA algorithm, for example this means that a rank has to create a task only if it owns one of the three blocks involved from A, B and C, respectively; in this case the local size of the DAG is $(mnk)/P$ if the A, B and C matrices are aligned over the ranks grid. In the case of the other variants of SUMMA algorithms, nodes contributing to reduction patterns have to be aware of all the contributing nodes in order to properly reduce the results : the local DAG is bigger with a size of $(mnk)/(P/h) = h(mnk)/P$ for the 3D stat-C variant, of up to $(q+1)(mnk)/P$ for the 3D stat-A variant and of up to $(p+1)(mnk)/P$ for the 3D B-stat variant.

The pruning rule for these variants requires that all processes know how the matrix C is distributed in order to map which blocks of C to track in the DAG based on their rank and the data they own. Ruling out unnecessary tasks could be achieved through suitable iterators, as mentioned in the following section, however this more advanced technique requires more assumptions on the complete distribution of the matrices.

4.3.2 Efficient submission of tasks

It must be noted that all the possible nesting orders for the loops of Figure 6 will lead to equivalent DAGs where collective communications and reductions are correctly detected and executed for all the presented variants. Nevertheless, if the nesting order is appropriately chosen, it is possible to ensure some properties of the execution that can be exploited, for example, to control the memory consumption (more on this will be said in section 5.2.3). For the stat-A, stat-B and stat-C variants, the nesting order used in our implementation is, respectively, (n, m, k), (m, n, k) and (k, m, n). Although it is still possible to implement the three variants in a single code by using suitable iterators, this would inevitably render the code less readable and, most importantly, the creation of tasks less efficient. Preliminary experimental results revealed that the StarPU runtime system is particularly sensitive to the efficiency of tasks creation and, for this reason, we decided to have three separate codes for the stat-A, stat-B and stat-C algorithms and their 3D variants.

5 Experiments

In this section we report experimental results that aim at assessing the effectiveness of the proposed approach. Parallel GEMM is an extremely important numerical kernel and a subject that has been the object of numerous research works. As a result, many different implementations exist in many software packages. Among the most recent efforts, we can cite the work by Herault et al. (2019) where remarkable performance is achieved on large, GPU-based systems, with a task-based parallel approach relying on the PaRSEC runtime system and its PTG programming model. An exhaustive comparison with other software packages is out of the scope of this paper. Rather, the main objective of this experimental analysis is to show that the proposed approach can achieve performance that is on par with reference implementations, despite the use of a very high level parallel programming model and the fact that most of the complex work is delegated to a runtime system. For this reason we have chosen to compare with libraries that either are well known references to which most other works compare, or share some features with our approach; these are

- ScaLAPACK (version 2.0.2): this is the *de facto* standard in parallel dense linear algebra. This library implements the stat-A, B and C 2D SUMMA algorithms using MPI; shared memory parallelism is achieved through the use of multithreaded BLAS routines.
- Elemental[3]: this library has been developped to further the MPI+X approach of ScaLAPACK by relying on modern programming language and

[3]commit 4abe4ef0 from `https://github.com/LLNL/Elemental`

abstraction layer. Codes for all the 2D stationary variants are implemented.

- Slate[4]: this recent effort has the objective of producing a ScaLAPACK replacement for modern multicore and accelerator based supercomputers. It implements the stat-A and C 2D SUMMA variants using a hybrid MPI×OpenMP approach and employs a lookahead method to achieve better efficiency by overlapping communications and computations.

- Chameleon[5]: this library provides parallel dense and data-sparse linear algebra subroutines using task-based parallelism through different runtime systems. It implements the pipelined stat-C 2D SUMMA algorithm using the baseline STF model (see section 3.1); therefore, it does not make use of the extended features described in the present work but, rather, communications are explicitly handled through data copies into temporary matrices. It uses a lookahead mechanism to overlap computations and communications. The runtime system chosen for our experiments is StarPU.

5.1 Experimental setup

Our experiments were run on two different partitions of the Joliot-Curie supercomputer of the French Très Grand Centre de Calcul (TGCC) supercomputing center:

- *Skylake*. A partition of 1,656 nodes equipped with two Intel Skylake 8168 @ 2.7 GHz processors, 24 cores each, and 192 GB of DDR4 memory. This partition uses an Infiniband EDR interconnect with a pruned fat tree topology.

- *Rome*. A partition of 2,292 nodes equipped with two AMD Rome (Epyc) @ 2.6 GHz, 64 cores each, and 256 GB of DDR4 memory. This partition uses an Infiniband HDR100 interconnect through a dragonfly topology using 5 cells. Each cell is interconnected through a fat tree topology.

On both computers and for all software packages we used the BLAS routines provided by the Intel MKL (version 21.3.0) library and the GNU (version 9.3.0) compilers suite. OpenMPI (version 4.0.5) was used for ScaLAPACK, Elemental and Slate whereas for qr_mumps and Chameleon we use StarPU[6] with the NewMadeleine communication backend[7] which, in the case of qr_mumps, provides support for the dynamic collective communications. For Slate, Chameleon and qr_mumps experiments were run with one rank per node using as many threads/-workers as the available cores. Although for these three packages slightly better performance could be achieved with multiple ranks per node, this requires a careful placement of processes and threads : while a fine tuning of these parameters is out of the scope of this experimental analysis, one rank per node yielded satisfying results close to the optimum. In the case of ScaLAPACK multiple ranks (MPI processes) per node were used (24 and 64 for Skylake and Rome,

[4]commit bb597ae4 from `https://bitbucket.org/icl/slate`
[5]commit 9825fbf1 from `https://gitlab.inria.fr/solverstack/chameleon`
[6]commit 0afdaeb09 from `https://gitlab.inria.fr/starpu/starpu`
[7]commit fdec689ab from `https://gitlab.inria.fr/pm2/pm2`

respectively) each using two threads as this configuration resulted in the best performance. This is also the case of Elemental which uses 24 ranks per node on Skylake (two threads per rank) and 32 ranks per node on Rome (four threads per rank), those configurations resulting in the best performance.

In all the packages it is possible to tune the block size to achieve a favorable trade-off between parallelism and efficiency of local computations. We have chosen to run all experiments using multiple block sizes, namely, 256, 512 and 1024, and report the best results. Smaller and larger block sizes were found to be suboptimal on all the tested configuration either because of an excessively small granularity of computations or because of an insufficient amount of parallelism.

For the Slate and Chameleon packages, the default lookahead depth of one was used; higher values were not found to improve the performance.

For the qr_mumps tests, because the GEMM routines are benchmarked alone and not as part of a larger application, we enriched the DAG with artificial tasks to make sure that the dynamic collective communications are correctly detected and the reduction operations entirely executed.

Finally, multiple runs were executed for each experiment, including a warm-up run which is not taken into account in the performance measurement; median performance across these runs is reported in the following sections.

5.2 Experimental results

We have chosen 3 matrix problem types ($m = n = k$, $m = n = 8k$, $m = 8n = k$) and, for each type, three sizes of increasing value. The dimensions of the smallest problem may be inadequate to attain the peak performance of the machine in a large-scale setting however it assesses the scalability of the different libraries. On problems large enough, all libraries should be able to attain satisfying performances as the workload from communications and the computing workload are easily intertwined. For each type and size we have conducted experiments using 16, 64 and 256 nodes (i.e., up to $12,288$ and $32,768$ cores on Skylake and Rome, respectively). For the 2D variants, Chameleon, Slate and qr_mumps use square grids with (4×4, 8×8, 16×16) ranks, one rank per node each using all the available cores; ScaLAPACK uses grids with (24×16, 48×32, 96×64) and (32×32, 64×64, 128×128) ranks using two cores each on Skylake and Rome, respectively. Elemental uses the same grids as ScaLAPACK on Skylake, on Rome the grids are of size $(32 \times 16), (64 \times 32), (128 \times 64)$. When using a 3D variant in qr_mumps, we used rank grids with 4 layers ($4 \times 4 \times 4$, $8 \times 8 \times 4$) on 64 and 256 nodes.

5.2.1 Stationary-C

We have evaluated the effectiveness of the stat-C variant on two different problems: the first one uses square matrices ($m = n = k$) and the second one only keeps the C matrix square with A and B being smaller, i.e., ($m = n = 8k$). The first problem is often used in the literature for comparing parallel GEMM algorithms and implementations. In this case, where all matrices are of comparable size, all 2D variants are likely to achieve comparable performance although the stat-C one can be preferred due to its relative simplicity. The stat-C is still the best suited variant for the second problem despite k being small. Although, in this case, a considerable amount of parallelism might still be available when m

and n are large, the latency to exchange blocks of A and B is critical to achieve high execution speed because the number of outer products is reduced.

For these problem types, we can observe in figures 7 and 8 that our method is competitive with all the libraries: it obtains the best median across several configurations. When it is not the most efficient method, our method is able to be on par with other ones as it proves strongly scalable. This is likely due to the use of asynchronous collective communications and the ability of the runtime system to take advantage of the commutativity of the tasks that contribute to the same C_{ij} block. Notably, our implementation is able to achieve over 500 Tflop/s on the largest size of the $m = n = 8k$ problem on the Rome partition, on par with Elemental.

For square matrices, we compared the 2D and 3D stat-C variants of qr_mumps. The 2D variant is better than the 3D in all tests except on Rome for the smallest problem and largest grid. In this case the 3D variant achieves better performance than the 2D one thanks to its ability to achieve better parallelism without using an excessively small block size. This is not inconsistent with results presented in the literature related to 3D matrix multiplication Schatz et al. (2016); Georganas et al. (2012); Demmel et al. (2012) because our approach heavily relies on multithreading for using the cores of each node rather than message passing and employs non-blocking collective communications which were not available in MPI at the time of those works.

5.2.2 Stationary-A

In order to evaluate the effectiveness of the stat-A variant, we have chosen problems where the size of the A matrix is much larger than that of the B and C ones, namely $m = 8n = k$ with $m = \{65536,\ 131072,\ 262144\}$. For qr_mumps and Slate, the stat-A and stat-C routines are directly callable and, thus, we include results for both of them; for Elemental we only report the stat-A variant; for ScaLAPACK it is only possible to call the generic GEMM routine which, internally, chooses the most appropriate variant (which ended up being stat-A for the chosen problem sizes); Chameleon does not implement the stat-A algorithm so we have chosen not to report the performance of this library on this problem type.

For this problem type, we can observe on Figure 9 that our method is significantly better than most libraries – Elemental obtains results similar to ours on both partitions. This is likely due to the use of non-blocking collective communications and the runtime system leveraging commutativity of local tasks as well as the reduction patterns.

Stat-A variants are significantly better than their stat-C counterparts especially on small size problems and large size grids where the execution time is dominated by communications. As the problem size increases, this difference is less remarkable because there's more opportunity for overlapping communications and computations.

5.2.3 Controlling the memory consumption

Task-based parallel runtime systems in general, and StarPU in particular, commonly rely on an eager scheduling: as soon as a task becomes ready, it is scheduled for execution. In the case of StarPU, this also concerns communi-

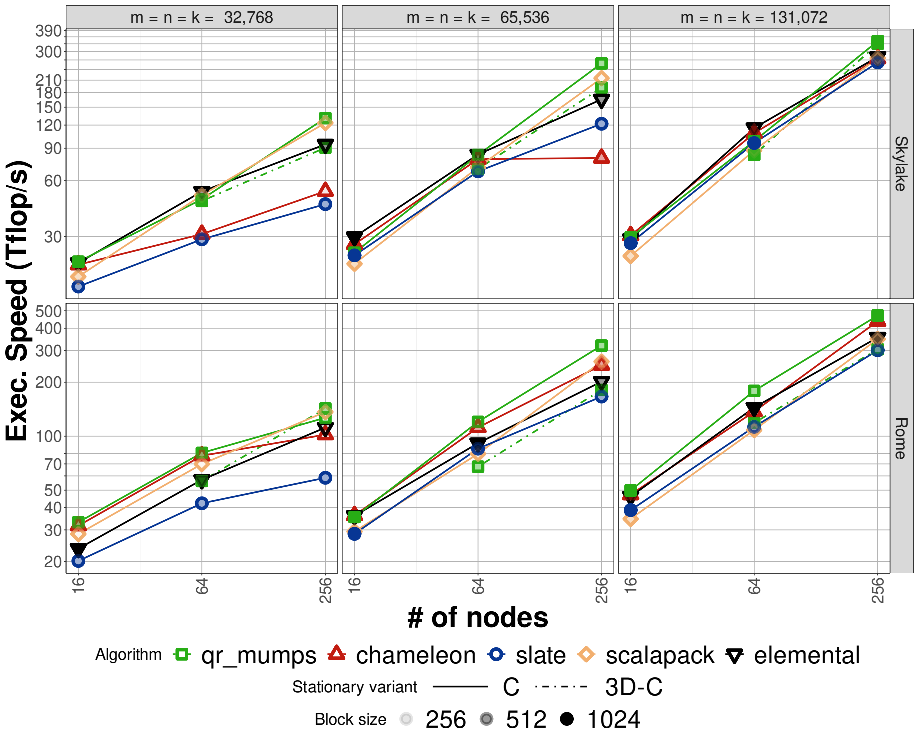

Figure 7: Comparisons of ScaLAPACK, Elemental, Slate, Chameleon and qr_mumps DGEMM on square matrices ($m = n = k$) and an increasing number of nodes. Markers correspond to the best median across runs using a given blocking for an algorithm.

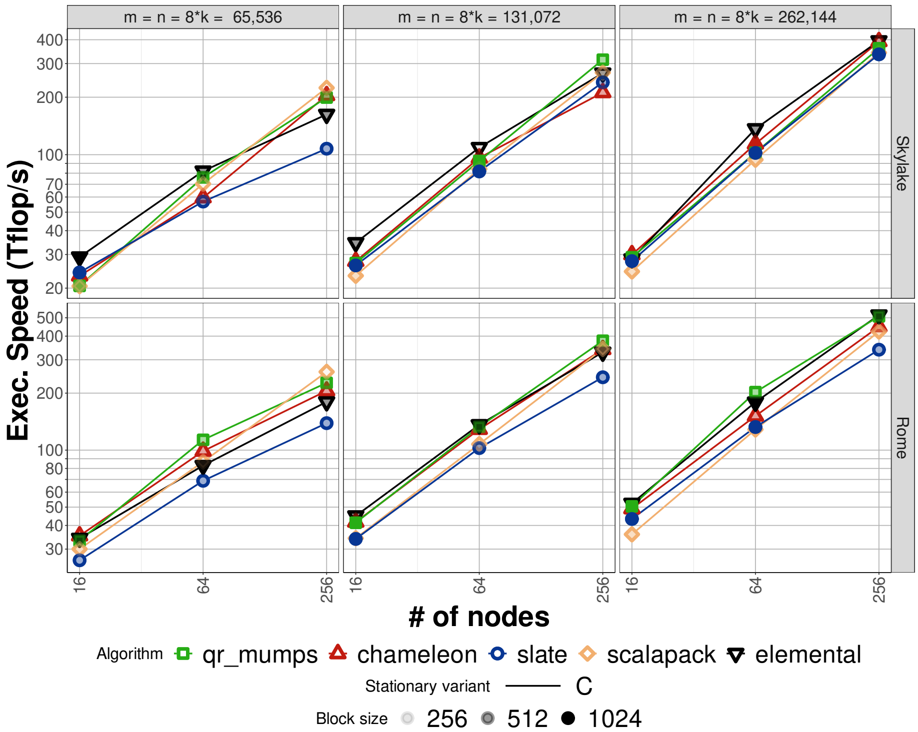

Figure 8: Comparisons of ScaLAPACK, Elemental, Slate, Chameleon and qr_mumps DGEMM involving a large C matrix ($m = n = 8k$) on an increasing number of nodes. Markers correspond to the best median across runs using a given blocking for an algorithm.

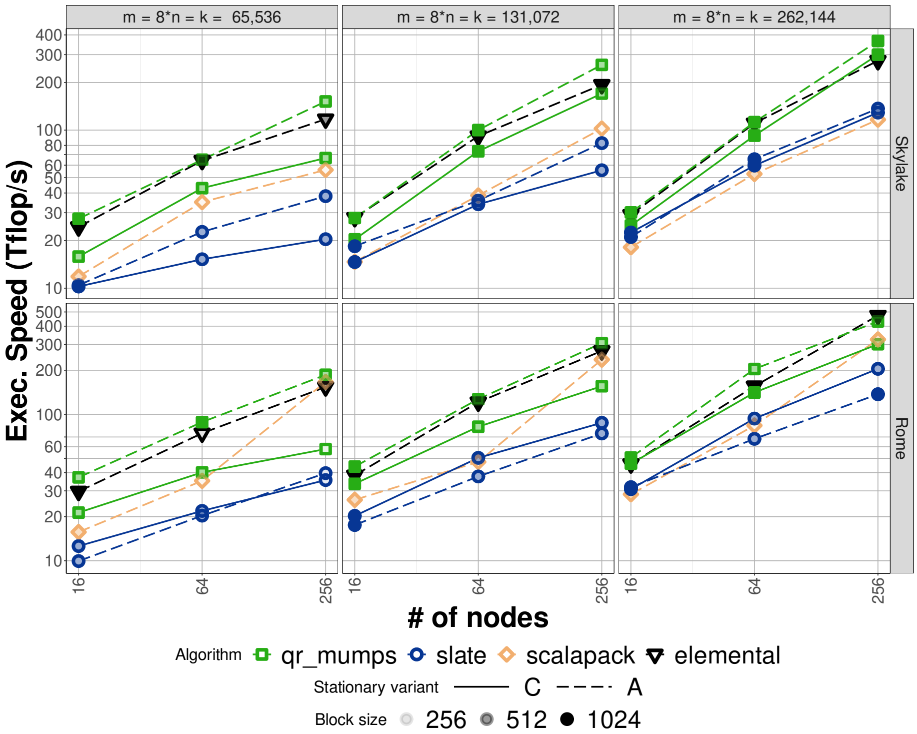

Figure 9: Comparisons of ScaLAPACK, Elemental, Slate and qr_mumps DGEMM involving a large A matrix ($m = 8n = k$) on an increasing number of nodes. Markers correspond to the best median across runs using a given blocking for an algorithm.

cations because they are fulfilled by dedicated tasks which are automatically and transparently created by the runtime system. When the GEMM routine is not evaluated as part of a larger application where the matrices are produced by other operations, all the communication tasks are immediately ready and scheduled for execution and potentially all executed in the very early stages of the matrix product. This has two effects. First it might cause an excessive memory consumption because StarPU must allocate communication buffers for blocks that are received much earlier than when they are actually used. Second, it may reduce performance because of the high contention that it generates on the network.

StarPU does not currently offer a proper feature to control the execution of communication tasks. Nevertheless, it is possible to control the execution of communications indirectly by delaying the submission of tasks. Note that this relies on a fundamental property of the STF programming model: tasks are created sequentially, that is, in exactly the same order in which the corresponding operations would be executed in a sequential code. Based on this assumption it is possible to use a feature of StarPU that allows one to cap the number of submitted tasks through a sliding window mechanism (this feature is already discussed in related work by Agullo et al. (2017a)). This feature provides two environment variables that can be used to define the maximum and minimum number of submitted tasks: when the maximum is reached, the tasks creation is suspended (the task submission routine becomes blocking) and is resumed when, upon execution of already created tasks, the number falls below the minimum. We used this feature to implement a lookahead mechanism in our implementation. The maximum is set to be the number of tasks in a prescribed number of iterations of the outer loop of the product which, essentially, corresponds to the lookahead depth.

The results obtained with this approach on the stat-C variant are presented in Figure 10 (similar results were obtained on the stat-A one). This figure shows the maximum memory consumption over all ranks, including the initial A B and C matrices (represented by the gray dotted line), with respect to performance for multiple values of the lookahead depth compared to Slate and Chameleon (for which the default lookahead of 1 was used). The figure clearly shows that when no memory control mechanism is used in qr_mumps (which corresponds to an infinite depth lookahead), the memory consumption is excessively high and much higher than the other packages. When a fixed-depth lookahead is used, instead, not only the memory consumption becomes comparable to that of Chameleon and Slate, but performance is improved thanks to a reduced pressure on the communication layer.

This analysis suggests that the memory consumption could be reliably controlled through an analogous feature that allows for capping the maximum size of communication buffers rather than the number of tasks. The sequential tasks submission order will ensure that no deadlocking occurs provided that a sufficient amount of memory can be used Agullo et al. (2016). We reserve the implementation and study of such a feature for future work.

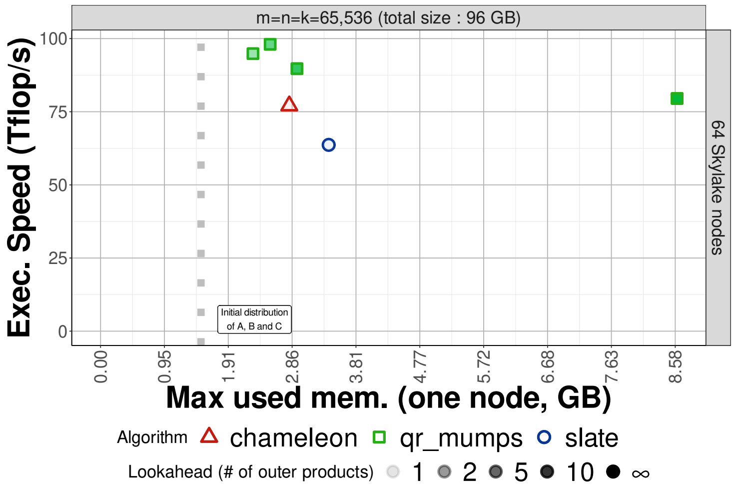

Figure 10: Comparisons of Slate, Chameleon and qr_mumps DGEMM on square matrices of size 65,536 using 64 nodes and a blocking of size 512. qr_mumps uses different tasks window mechanisms.

6 Conclusions and future work

The main conclusion of this work is that it is possible to design state-of-the-art matrix multiplication algorithms for distributed memory machines through a very compact sequential-like code. The programmer can be relieved from the burden of writing low-level complex communication schemes as it the case for instance with MPI-based codes such as ScaLAPACK. This can be achieved through the use of advanced features of the STF, task-based, parallel programming model. In this enhanced STF model we have proposed, we have indeed shown that efficient communication patterns can be effectively inferred from an appropriate choice of the mapping and data access modes. We emphasize also that relying on this extended STF model, the scalabity is achieved with moderate effort from the programmer's point of view: The main philosophy of the STF model (expressing parallel algorithms through a sequential submission process) still holds in this distributed memory context. The second main conclusion is that, as of 2022, the software ecosystem is mature enough to ensure that the resulting code may be competitive against state-of-the-art, finely hand-tuned libraries.

We believe that these conclusions shall motivate the community to further consider task-based programming models, and the STF one in particular, for designing scalable algorithms, while ensuring an enhanced portability and maintenance. Efforts to extend the features of generic, i.e. not necessarily dedicated to numerical linear algebra, runtimes can be beneficial to the community: improvements to such software lead to improvements in the ecosystem of applications relying on it. On our side, we plan to assess the suitability of the model proposed in this paper to design dense and sparse direct methods.

While the performance results are already remarkable, we believe that the proposed programming model still offers additional opportunities for further optimization. Indeed, while state-of-the-art MPI-based libraries rely on communicators to ensure collective communications, on the contrary, multiple collective communications can (be inferred and) progress concurrently Denis et al. (2020) in our model. In particular, a careful study of their impact on state-of-the-art distributed memory matrix multiplication algorithms remains to be conducted.

While we have shown that the memory footprint can be kept under control without performance penalty (or even improving it), the employed technique – a sliding window mechanism – has required some manual tuning regarding the window size. We plan to make it automatic with respect to a prescribed memory footprint, similarly to Agullo et al. (2016).

7 Acknowledgments

We wish to thank Nathalie Furmento, Olivier Aumage and Samuel Thibault of the Inria STORM team for helping us making the most out of the StarPU runtime system. We also thank Alexandre Denis and Philippe Swartvagher for the help they provided in using the NewMadeleine library and its dynamic collectives feature. Work carried out as part of this paper used the PlaFRIM experimental testbed, supported by Inria, CNRS (LABRI and IMB), Universit de Bordeaux, Bordeaux INP and Conseil Rgional dAquitaine (see https://www.plafrim.fr). This work was granted access to the HPC resources of TGCC under the allocation 2021-5063 made by GENCI. This work was supported by the SOLHARIS project (ANR19-CE46-0009) which is operated by the French National Research Agency (ANR).

References

Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. 1995. A three- dimensional approach to parallel matrix multiplication. *IBM Journal of Research and Development* 39, 5 (1995), 575–582. `http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.120.4575&rep=rep1&type=pdf`

Ramesh C. Agarwal, Fred G. Gustavson, and Mohammad Zubair. 1994. A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication. *IBM J. Res. Dev.* 38, 6 (Nov. 1994), 673681. `DOI:http://dx.doi.org/10.1147/rd.386.0673`

Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. 2017a. Achieving high performance on supercomputers with a sequential task-based programming model. *IEEE Transactions on Parallel and Distributed Systems* (2017). `DOI:http://dx.doi.org/10.1109/TPDS.2017.2766064`

Emmanuel Agullo, George Bosilca, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. 2017b. Exploiting a Parametrized Task Graph Model for the Parallelization of a Sparse Direct Multifrontal Solver. In *Euro-Par 2016: Parallel Processing Workshops: Euro-Par 2016 International Workshops, Grenoble, France, August 24-26, 2016, Revised Selected Papers*, Frédéric Desprez, Pierre-François Dutot, Christos Kaklamanis, Loris Marchal, Korbinian Molitorisz, Laura Ricci, Vittorio Scarano, Miguel A. Vega-Rodríguez, Ana Lucia

Varbanescu, Sascha Hunold, Stephen L. Scott, Stefan Lankes, and Josef Weidendorfer (Eds.). Springer International Publishing, Cham, 175–186. DOI: http://dx.doi.org/10.1007/978-3-319-58943-5_14

Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. 2016. Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems. *ACM Trans. Math. Softw.* 43, 2, Article 13 (Aug. 2016), 22 pages. DOI:http://dx.doi.org/10.1145/2898348

Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. 2009. Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects. *Journal of Physics: Conference Series* 180, 1 (2009), 012037. http://stacks.iop.org/1742-6596/180/i=1/a=012037

Patrick R. Amestoy, Iain S. Duff, Jako Koster, and Jean-Yves L'Excellent. 2001. A Fully Asynchronous Multifrontal Solver Using Distributed Dynamic Scheduling. *SIAM J. Matrix Anal. Appl.* 23, 1 (2001), 15–41.

Cleve Ashcraft. 1993. The Fan-Both Family of Column-Based Distributed Cholesky Factorization Algorithms. In *Graph Theory and Sparse Matrix Computation*, Alan George, John R. Gilbert, and Joseph W. H. Liu (Eds.). Springer New York, New York, NY, 159–190.

Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. 2011. StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures. *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (Feb. 2011), 187–198. Issue 2. DOI:http://dx.doi.org/10.1002/cpe.1631

Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing Communication in Numerical Linear Algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901. DOI:http://dx.doi.org/10.1137/090769156

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. 1997. ScaLAPACK: A Linear Algebra Library for Message-Passing Computers. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, Hyatt Regency Minneapolis on Nicollel Mall Hotel, Minneapolis, Minnesota, USA, March 14-17, 1997*. SIAM.

George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. 2013. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science and Engineering* 15, 6 (2013), 36–45. DOI:http://dx.doi.org/10.1109/MCSE.2013.98

Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. 2009. A class of parallel tiled linear algebra algorithms for multicore architectures. *Parallel Comput.* 35 (January 2009), 38–53. Issue 1. DOI:http://dx.doi.org/10.1016/j.parco.2008.10.002

Lynn Elliot Cannon. 1969. *A Cellular Computer to Implement the Kalman Filter Algorithm.* Ph.D. Dissertation. Montana State University, USA. AAI7010025.

James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal Parallel and Sequential QR and LU Factorizations. *SIAM J. Sci. Comput.* 34, 1 (Feb. 2012), 206–239. http://dx.doi.org/10.1137/080731992

Alexandre Denis, Emmanuel Jeannot, Philippe Swartvagher, and Samuel Thibault. 2020. Using Dynamic Broadcasts to Improve Task-Based Runtime Performances. In *Euro-Par 2020: Parallel Processing*, Maciej Malawski and Krzysztof Rzadca (Eds.). Springer International Publishing, Cham, 443–457.

Evangelos Georganas, Jorge Gonzalez-Dominguez, Edgar Solomonik, Yili Zheng, Juan Tourino, and Katherine Yelick. 2012. Communication avoiding and overlapping for numerical linear algebra. In *SC '12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* 1–11. DOI:http://dx.doi.org/10.1109/SC.2012.32

Thomas Herault, Yves Robert, George Bosilca, and Jack Dongarra. 2019. 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.* IEEE, Denver, United States, 33–41. DOI:http://dx.doi.org/10.1109/ScalA49573.2019.00010

Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2021. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. *IEEE Transactions on Parallel and Distributed Systems* (2021), 1–1.

Francisco D. Igual, Gregorio Quintana-Ortí, and Robert A. van de Geijn. 2013. Scheduling algorithmsbyblocks on small clusters. *Concurrency and computation : practice and experience.* 25, 3 (2013), 367–384.

Dror Irony, Sivan Toledo, and Alexander Tiskin. 2004. Communication lower bounds for distributed-memory matrix multiplication. *J. Parallel and Distrib. Comput.* 64, 9 (2004), 1017–1026. DOI:http://dx.doi.org/https://doi.org/10.1016/j.jpdc.2004.03.021

Jungwon Kim, Seyong Lee, Beau Johnston, and Jeffrey S. Vetter. 2021. IRIS: A Portable Runtime System Exploiting Multiple Heterogeneous Programming Systems. In *Proceedings of HPEC'21.* 1–8.

Josep M. Perez, Vicen Beltran, Jesus Labarta, and Eduard Ayguad. 2017. Improving the Integration of Task Nesting and Dependencies in OpenMP. In *Procceeding of IPDPS'17.* 809–818.

Piyush Sao, Xiaoye S. Li, and Richard Vuduc. 2019. A communication-avoiding 3D algorithm for sparse LU factorization on heterogeneous systems. *J. Parallel and Distrib. Comput.* 131 (2019), 218–234. DOI:http://dx.doi.org/https://doi.org/10.1016/j.jpdc.2019.03.004

Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. 2016. Parallel matrix multiplication: a systematic journey. *SIAM Journal on Scientific Computing* 38, 6 (2016), 748–781. `http://www.cs.utexas.edu/users/flame/pubs/2D3DFinal.pdf`

Edgar Solomonik and James Demmel. 2011. 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)*. Springer-Verlag, Berlin, Heidelberg, 90–109. `http://dl.acm.org/citation.cfm?id=2033408.2033420`

Robert van de Geijn and Jerrell Watts. 1997. SUMMA: scalable universal matrix multiplication algorithm. *CONCURRENCY: PRACTICE AND EXPERIENCE* 9, 4 (1997), 255–274. `http://www.netlib.org/lapack/lawnspdf/lawn96.pdf`

Robert A. van de Geijn. 1997. *Using PLAPACK - parallel linear algebra package.* MIT Press.

Asim YarKhan. 2012. *Dynamic task execution on shared and distributed memory architectures.* Ph.D. Dissertation.

RESEARCH CENTRE
BORDEAUX – SUD-OUEST

200 avenue de la Vieille Tour
33405 Talence Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399