Advanced task-based programming models for scalable linear algebra operations

Antoine Jego

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Mathématiques Appliquées

Présentée et soutenue par :

M. ANTOINE JEGO

le vendredi 15 décembre 2023

Titre :

Modèles de programmation avancés à base de tâches pour les algorithmes d'algèbre linéaire qui passent à l'échelle

---

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeur(s) de Thèse :

M. ALFREDO BUTTARI

Rapporteurs :

M. EMANUEL RUBENSSON, UPPSALA UNIVERSITET SUEDE

M. GEORGES BOSILCA, UNIVERSITE DU TENNESSEE

Membre(s) du jury :

M. PATRICK AMESTOY, TOULOUSE INP, Président

M. ALFREDO BUTTARI, TOULOUSE INP, Membre

M. EMMANUEL AGULLO, INRIA CENTRE DE RECHERCHE DE BORDEAUX, Invité(e)

MME AURELIE HURAULT, TOULOUSE INP, Membre

M. NICOLAS RENON, MESOCENTRE CALMIP, Invité(e)

M. SAMUEL THIBAULT, UNIVERSITE BORDEAUX 1, Membre

Abstract

Writing high-performance computing packages is not an easy task given the ever-burgeoning supercomputing ecosystems. In the last decade the landscape of supercomputers has become more complex to maintain low energy consumption and high computing power throughput. To achieve this feat groundbreaking computing chips are used such as GPUs. This hardware comes with specific low-level tools to program it and, as such, requires much expertise when building computing-intensive applications.

This thesis is focused on task-based programming models that make the adaptation of the software stack more productive and more resilient to hardware breakthroughs.

The sequential task flow (STF) model is given special care as it proposes polished interfaces that has been widely-adopted in scientific computing packages executed on shared-memory parallel machines. The adoption of this model is more disputable on distributed-memory machines; the mechanisms put in place by state-of-the-art scalable algorithms are well-established in low-level programming models such as the message passing interface however they are not transparently supported by the STF model. By reviewing linear algebra scalable algorithms we have identified missing features in STF that are pivotal to obtain scalability by avoiding communications.

We have implemented and validated them in the StarPU runtime system that support the STF model. The resulting extended programming model makes it possible to express state-of-the-art scalable algorithms in a portable and compact way. The first one is distributed-memory matrix-matrix multiplication for which we deliver a single elegant code that can span multiple algorithmic variants. The second one is matrix decomposition for which our implementation compactly exhibits state-of-the-art designs. These algorithms have been added to the dense routines of the `qr_mumps` package.

A large experimental campaign has been carried out to validate the performance of our implementations. Performance measurements indicate compelling results with regard to other dense linear algebra packages especially on smaller problems typically harder to parallelize or when input matrices dimensions are unbalanced.

The flexibility of our implementations was instrumental to enable the use of layouts tailored for symmetric matrix multiplication when the symmetric matrix is the largest one. The resulting operation now performs comparably well to general matrix multiplication while using half the memory.

Résumé

L'écriture de bibliothèques de calcul haute performance n'est pas une tâche aisée surtout dans des environnements de calcul en perpétuelle évolution. La dernière décennie a vu le paysage des supercalculateurs se complexifier pour maintenir une consommation d'énergie faible et une puissance de calcul élevée. Pour y parvenir des technologies avancées comme les GPU sont mises en œuvre. Ce matériel peut être programmé à l'aide d'outils bas-niveau et à ce titre une expertise est requise pour développer des applications de calcul intensif.

Cette thèse étudie les modèles de programmation à base de tâches rendant l'adaptation logicielle productive et flexible aux innovations matérielles.

On s'intéresse en particulier au modèle de programmation séquentiel en flots de tâches (STF) qui propose une interface largement adoptée par la communauté du calcul scientifique dans l'utilisation des machines en mémoire partagée. Cette adoption est cependant moins univoque dans le cadre des machines à mémoire distribuée; les méchanismes mis en place par les algorithmes de l'état de l'art qui passent à l'échelle s'appuyant plus souvent sur des interfaces bas-niveau comme celle proposée par MPI, ils ne sont pas disponibles de manière transparente dans le STF. En passant en revue les algorithmes d'algèbre linéaire qui passent à l'échelle on a pu identifié des fonctionnalités manquantes au modèle qui permettent l'évitement des communications.

On a implémenté et validé ces fonctionnalités dans le moteur d'exécution StarPU qui prend en charge le modèle STF. Le modèle étendu qui en résulte rend possible l'expression portable et compacte d'algorithmes de l'état de l'art qui passent à l'échelle. Le premier algorithme d'intérêt est la multiplication de matrices pour laquelle on fournit un unique code élégant qui balaie différentes variantes. Le second est la factorisation de matrice dont notre implémentation expose compactement des choix de conception de l'état de l'art. Ces algorithms ont été intégrés à la suite de routines denses de `qr_mumps`.

Une campagne expérimentale importante a été mise en place pour valider les gains de performance réalisés. Les résultats indiquent que notre approche est compétitive face à celle mise en place par des bibliothèques de l'état de l'art, en particullier sur des problèmes de petites tailles difficiles à paralléliser ou bien des problèmes pour lesquels les dimensions sont déséquilibrées.

La flexibilité de nos implémentations a été déterminante pour permettre l'utilisation de distrbutions de données avancées quand la matrice symmétrique est la plus grande. L'opération qui en résulte est aussi performante que la multiplication de générale tout en utilisant moitié moins de mémoire.

Remerciements

I would like to first thank George and Emanuel for reporting on this manuscript. Not only did they give it a thorough read which helped me improve its quality further, they have also taken the time to come to Toulouse to listen to my defense and discuss the achieved work further.

J'aimerais aussi remercier les autres membres du Jury : Aurélie, Nicolas, Patrick et Samuel. Vos perspectives multiples ont participé à améliorer la clarté de ce manuscrit. Quand bien même vous ne venez pas d'aussi loin, c'est tout autant un honneur pour moi que vous ayez pu jugé ces travaux. While the defense may supposedly be a dreadful moment, I felt really happy to answer each and every one of your questions and discuss with all of you.

Évidemment, et parce que sans eux il n'y aurait même pas une once de manuscrit à commencer à améliorer, je souhaite remercier abondament Alfredo, Abdou et Emmanuel. C'a été trois années et demi bizarroïdes en terme de rhythme avec la pandémie (en tout cas quand les institutions essayaient d'organiser sa mitigation), mais ç'a été une période pendant laquelle j'ai eu beaucoup de plaisir à faire de la recherche, à comprendre des papiers et bidouiller des logiciels. C'est un travail qui n'aurait pas été possible sans tout le temps que vous avez donné pour discuter, m'aiguiller, me modérer aussi quand j'ai pu partir dans tous les sens. Vous avez parfois dit que vous m'embêtiez pour pas grand chose mais je pense que la somme des "pas grand chose" a été très formatrice : je suis persuadé d'être extrêmement chanceux d'avoir bénéficié d'un tel cadre.

Un travail de recherche comme une thèse c'est des encadrants mais c'est aussi des équipes de recherche qui savent être accueillantes. À Toulouse au sein de l'équipe APO, j'ai eu la chance de partager la F321 avec Jean-Paul "Boris", Antoine B., Bastien, Sophie, Baptiste: même si mes horaires de sommeil ont pas toujours bien correspondu avec les horaires d'ouverture du labo je garderai de très bons souvenirs avec vous mais aussi avec le reste des doctorants dont Théo, Sadok, Peter, Rémy, Valentin, Alexandre et Matthis et des permanents

de l'équipe et de l'étage. J'espére pouvoir assister, au moins à distance, à toutes vos soutenances. Merci aussi à SAM et Vanessa parce qu'une partie de l'accueil c'est aussi un super secrétariat. Quand j'ai eu l'occasion de faire des détours à Bordeaux, j'ai eu le plaisir de partager les locaux de l'équipe HIEPACS (maintenant Concace et Topal si j'ai tout suivi): ça m'a fait très plaisir d'être accueilli à 3 reprises par les doctorants là-bas. Je voudrais spécifiquement remercier Philippe, Romain, Mathieu, Maxime, Gwénolé et Marek qui font aussi partie du projet SOLHARIS de près ou de loin. J'aimerais aussi remercier Florent Pruvost, Nathalie Furmento et Alexandre Denis qui ont été de supers guides autour de Chameleon, StarPU et NewMadeleine respectivement.

J'aimerais aussi remercier les membres du projet SOLHARIS dans leur totalité pour avoir créé un environnement qui soit propice aux échanges et aux collaborations. C'a été très plaisant de découvrir le monde de la recherche dans cet écosystème et l'ensemble de ces membres peut largement se féliciter d'avoir créer un tel terreau.

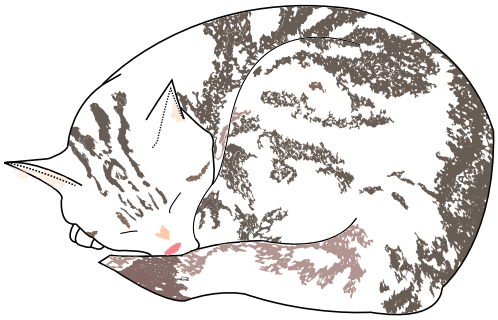
Dans une période où y a eu moins de présentiel que d'habitude, je pense que c'est très pertinent de remercier le discord PhD students: ç'a été un endroit super sympa qui m'a permis de découvrir des horizons scientifiques diamétralement opposés. C'est un super espace sur Internet pour celleux qui préparent une thèse et je pense que quiconque y passe saura saluer l'investissement de cette communauté pour maintenir un tel endroit.

Parce qu'ils comptent beaucoup, je souhaite remercier tous les camarades toulousains et parisiens qui sont parfois eux aussi en train de préparer une thèse: Jérémy, Yann, Robin, Louis, Yohan, Grégoire et Maxime. Vous êtes de très chouettes personnes et ça me fait plaisir qu'on ait gardé notre clique depuis l'ENSEEIHT.

Même si je suis définitivement pas assez monté en Bretagne, ça me paraît très important de remercier mes parents, et mes frère et soeur Agathe et Édouard. Je suis un gros ronchon têtu et casanier mais avant de grandir scientifiquement ou quoi que ce soit il faut grandir tout court et ça c'est définitivement grâce à vous.

Pour leur soutien moral constant mais éternellement dédaigneux, je me dois de remercier Bijoux (qui a l'honneur d'être en bas de page) et Cyrano (qui aura l'honneur d'une poignée de croquettes supplémentaire). Bijoux a une santé si fragile que j'ai eu peur qu'elle ne soit plus là pour la soutenance mais elle est encore là quand j'écris ces remerciements et c'est une chose très précieuse.

Nos chats sont probablement très précieux mais il y a quelque chose de plus précieux et d'absolu encore à mon avis et c'est la famille et la maison qu'on est avec Valentine – que j'ai eu l'honneur d'épouser le lendemain de ma soutenance. Valentine a supporté les infinités de répétitions bourrées d'anglais approximatif que j'ai fait depuis le début de la thèse, et avant ça elle m'a supporté 2 ans encore. Ca fait quand même 5 années de vie commune qui ont



su être magiques ; et je peux que te remercier d'avoir reconnu, même devant l'odieux État Français – tout souverein qu'il est – qu'on est faits l'un pour l'autre. Je ne pourrai probablement jamais te remercier suffisamment pour ta patience et tout ton amour dans un seul paragraphe, je vais donc essayer dans toute une vie.

Il y a très certainement beaucoup de personnes que j'oublie dans ces remerciements ; et la profondeur de ma désolation est probablement pas assez grande pour rattraper les hauteurs à laquelle ils – les remerciements – devraient culminer. Dans le doute, pour tout le monde : merci.

Contents

List of Figures

List of Tables

List of Algorithms

Experimental platforms

A. **Irène-Rome** partition of dual processor AMD Rome 7H12 2.6 GHz (64 cores each, 128 cores per node) hosted by the CEA for GENCI.

   - 2,292 nodes with Infiniband HDR100 interconnect
   - $R_{\text{peak}} = 11.75$ PFlop/s
   - 256GB DDR4 RAM per node
   - Intel MKL 21.3.0, GCC 12.2.0, OpenMPI 4.1.1

B. **Irène-Skylake** partition of dual processor Intel Skylake 8168 2.7 GHz (24 cores each, 48 cores per node) hosted by the CEA for GENCI.

   - 1,656 nodes with Infiniband EDR interconnect
   - $R_{\text{peak}} = 6.86$ PFlop/s
   - 192GB DDR4 RAM per node
   - Intel MKL 21.3.0, GCC 12.2.0, OpenMPI 4.1.1

C. **Jean Zay** partition of dual processor Intel Cascade Lake 6248 2.5 GHz (20 cores each, 40 cores per node) hosted by IDRIS for GENCI

   - 2,140 nodes, 351 of them with quad GPU Nvidia Tesla V100 16GB, 261 of them with quad GPU Nvidia Tesla V100 32GB
   - $R_{\text{peak}}^{\text{CPU}} = 4.87$ PFlop/s $R_{\text{peak}}^{\text{GPU}} = 17.14$ PFlop/s
   - 192GB DDR4 RAM per node
   - Intel MKL 19.0.5

Chapter 1

Introduction

The cost of developing software is high and even more so when the objectives are to deliver performance, portability and ease of maintenance at the same time. Supercomputers are commonly employed in a large variety of applications and algorithms that require considerable computing power; these range from classical numerical simulations to data analytics and are widely-used in numerous scientific and industrial fields. The landscape of supercomputer architectures, however, is ever-changing as constructors strive to adopt state-of-the-art technological advances to obtain unprecedented performance. This makes the task of developing high-performance, portable code for high-performance computing a daunting challenge for developers.

In this thesis we are interested in task-based programming models that aim at making the production, porting and maintenance of high-performance computing software easier on existing and future supercomputers.

If typical supercomputers in the 1990s mostly consisted of clusters of single-core machines, the machines used in the 2020s are vastly different. They host multiple manycore Central Processing Units (CPUs) which are often – but not always – accelerated with other processors such as Graphical Processing Units (GPUs). Writing software that is portable across these different machines is not simply a compilation issue. The correct sets of tools and frameworks which vary between supercomputers have to properly be used to target and extract the performance out of the hardware. One way to observe the evolution of supercomputers is to follow the Top500 semestrial reports. Every semester, the Top500 ranks the most high-performing supercomputers in the world: in 2020 the Japanese Fugaku supercomputer equipped with ARM CPU chips was the fastest supercomputer in History with 442 PFlop/s – for two years. In 2022 it was superseded by the American Frontier equipped with GPUs that reached

1.2 EFlop/s. These two supercomputers have very different architectures that seek to deliver a vast quantity of raw computing power. Their differences *partly* come from how they account for electricity supply, energy cost of cooling, *etc.*: the landscape of supercomputers keeps changing in part to accommodate these physical limits. The related emerging hardware has an impact on the software that is written to use it.

As the hardware gets increasingly complex and varied, porting scientific software over multiple architectures while maintaining performance becomes a challenging task. The programmer has to manage data transfers across processing units and across the network as well as orchestrate the scheduling of operations on the (heterogeneous) processing units. Most parts of the development efforts end up being devoted to these tasks rather than to the implementation of the application. Low-level programming models are used by the programmer to abstract the computer's components and to simplify its use. Community-wide efforts have even made it possible to structure these programming models into standards. For distributed-memory architectures, the Message-Passing Interface (MPI) provides tools to describe data transfers [107]; it is implemented both as free software through OpenMPI, MPICH, *etc.* as well as by several vendors like Intel. When orchestrating computation on shared-memory architectures, the OpenMP standard is favored because the modification of the code can come off as relatively inexpensive through its directive programming [34]. These standards demand a certain level of expertise to be used efficiently. Additionally their impact on the codebase can become considerable. The same standardization effort has not yet settled for the use of GPUs as many actors are involved (Nvidia [47], AMD [16]) and getting involved (Intel) in GPU manufacturing. Adapting a codebase to GPUs often takes the form of a thorough rewrite to handle data transfers between processing units.

Modernizing software cannot be sustainably achieved by incrementally updating entire codebases. A more durable option is to rely on higher-level abstractions of the computer. Using such programming models make the effort of the programmer less focused on low-level architectural details and more time can be spent expressing computation that are prone to parallelism. Among these programming models, task-based ones are becoming increasingly popular because oftentimes algorithms can be easily expressed as a collection of tasks, that is, elementary operations. Runtime systems such as StarPU [20] or PaRSEC [38] (formerly DAGue) expose an interface of these programming models to effectively discharge the programmer of data management, scheduling, *etc.*. Among the models proposed by runtime systems, the Sequential Task Flow (STF) model has been well integrated in multiple scientific computing packages. The sequential aspect of this model is an integral part of its accessibility. A vast body of recent literature and software demonstrates that task-based parallel programming models and runtimes allow achieving great performance and portability on shared memory, heterogeneous systems;

among these efforts, many focus on the use of the STF model for implementing mathematical software. Nonetheless, this model has seen a less pervasive adoption when dealing with distributed-memory architectures. Data management at the shared memory level is often entirely delegated to the runtime system, however this hardly translates to the distributed memory case. Managing the distributed memory comes with more challenges regarding the coherency of the system; therefore the efficient mechanisms to transfer data are more intrusive, especially when the algorithms are presented through a sequential flow of tasks. Ultimately the adoption of general purpose runtime systems is more disputable in distributed-memory environment.

Dense and sparse linear algebra software packages are extensively used in numerous high-performance computing applications. Among these, one of the most well-known and widely-used is ScaLAPACK [29]; this library that provides a wide range of dense linear algorithms has been developed at the end of the 1990s using a message passing parallel programming model relying on the MPI standard. Although ScaLAPACK is still the reference in distributed-memory parallel dense linear algebra software its rigid programming model makes it unsuitable for modern supercomputers as it lacks asynchronism, and native support for multicores and accelerators. This has prompted the High-Performance Computing (HPC) community to develop modern ScaLAPACK replacements such as SLATE [57], Elemental [95] or Chameleon [3]; these have taken over ScaLAPACK from a performance and portability point of view although they do not cover an equally large panel of algorithms. Additionally all of the above packages do not support state-of-the-art scalable algorithms. Many recent research efforts from the HPC community focused on the development of scalable algorithms for large-scale supercomputers; these are often referred to as "communication-avoiding" because they increase parallelism by carefully reducing either the volume of data transfers (i.e., the bandwidth usage) or the number of exchanged messages (i.e., the communications latency), possibly at the cost of a modest increase in the operational complexity. In some cases, for example, this is done by rearranging the computing nodes into a 3D logical grid which requires more intricate communication patterns than the standard 2D logical grids [18, 105, 102, 82]. In other cases, the latency of data transfers can be reduced by grouping more messages together which can create additional, redundant computation [70, 45]. These advanced algorithms cannot be straightforwardly implemented in the basic STF model because it lacks the necessary features; this issue can be partially overcome through carefully designed coding where data and communications are manually handled by the programmer but this, in essence, defeats the purpose of using a high productivity programming model like STF. **In this work, we focus our attention on relieving programmers from such effort by providing them with a suitable programming model to develop scalable algorithms.** Such a model is expected to help the programmer devote more time to the numerical intricacies of the computation they wish

to implement.

The objective of this thesis is to extend the task-based programming models capabilities to improve their use in a highly-parallel distributed-memory context. The implementation of such programming models by runtime systems should grant a practical tool to programmers that seek to harness the computing power of supercomputers with millions of heterogeneous cores interconnected by a relatively slow network. The proposed work could be helpful to modernize the computational routines provided by *e.g.* ScaLAPACK. It could also be instrumental in helping algorithm designers conceive new, innovative algorithms that help users make the most out of any supercomputer. Although StarPU is the runtime system used in this thesis, the scope of our work is much wider since the proposed solutions are applicable virtually in any runtime system relying on the STF model in distributed-memory environments. The contributions presented in this manuscript are as follows.

First, we have focused on the implementation of scalable matrix multiplication algorithms through the STF model. We have demonstrated that through the use of three advanced features – reduction operations, tasks mapping and dynamic collective communications – of the STF model it is possible to implement, in a single parameterized code, six different variants of the well-known SUMMA algorithm, namely, stationary-A, -B and -C both in a 2D and a 3D setting. This code is barely more complex than the canonical three nested loops of a sequential matrix multiplication; this ensures great ease of development and maintenance and great portability because this implementation is completely agnostic of the underlying architecture and data distribution. The above-mentioned features were already available in the StarPU runtime system; nevertheless, we have proposed an improvement of the reduction operations features that better suits large distributed-memory supercomputers. We have conducted an extensive experimental campaign on a large supercomputer (up to 256 nodes and 32,768 cores) demonstrating that not only our code is more readable and portable but it achieves performance that is on par with reference dense linear algebra libraries for the same algorithmic variants. Furthermore, because our implementations covers a wider panel of algorithmic variants, overall it achieves better performance when dealing with matrices of unbalanced dimensions.

Second, we have turned our attention to matrix factorization algorithms. In addition to the STF features presented at the previous point, scalable factorization algorithms require two novel features which we have designed and implemented within StarPU. The first is data write replication which enables multiple tasks to concurrently update multiple copies of the same data. Prior to using this feature in a matrix factorization algorithm, we have assessed its effectiveness on a communication-avoiding stencil computation algorithm showing that it can provide considerable performance benefits with only little additions to a classical STF implementation. The second feature is allreduce operations. The combined use of these two features and those in the

previous point allowed us to develop portable and compact implementations of 3D matrix factorization algorithms. We have conducted experiments on large platforms to demonstrate that the obtained implementation achieves comparable or better performance than existing reference libraries.

Third, we have further validated the extended STF programming model developed in the previous two points. Although the data distribution certainly plays a fundamental role in the performance and scalability, as we explained above, this programming model allows implementing algorithms regardless of the way data is distributed because the runtime will transparently move data where the corresponding tasks are computed. This makes it very easy to experiment with novel data distributions because only very little modification in the code is needed. Based on this, we have worked in collaboration with experts of scheduling to develop data distribution schemes that are capable of reducing the communication volume in parallel algorithms for symmetric matrices. We have experimented with these novel data distributions on the symmetric matrix multiplication and the Cholesky factorization demonstrating that they can practically reduce the communication volume and, consequently, improve performance especially on relatively small-size matrices when the cost of communications is dominant. Finally, we have assessed the effectiveness of this approach in the Diodon data analysis package [1] which makes a heavy use of symmetric matrix-matrix multiplication in its most computing-intensive parts: we assess the performance of several symmetric layouts analyzing real-world datasets over computing nodes accelerated with GPUs. Note that to target accelerated platforms we do not need to adapt our STF expression but we simply need to provide the runtime with accelerated kernels which are readily available in GPU-specialized BLAS and LAPACK libraries.

The rest of the manuscript is structured as follows: key concepts and elements found in the scientific literature are detailed in Chapter 2 with an emphasis on hardware, how to program it efficiently and what algorithms are programmed over it. In Part I of this manuscript, the ability of the STF programming model to express scalable algorithms is assessed and enhanced. More precisely, Chapter 3 focuses on expressing scalable General Matrix-Matrix multiplication variants. While this routine is ubiquitous in numerical linear algebra, the chapter focuses on bringing the most studied – but not necessarily implemented – scalable state-of-the-art routines under a versatile, efficient expression. This routine requires the definition of essential features: the mapping of tasks, the dynamic detection of collective operations and the use of distributed-memory reduction patterns. While these features are instrumental to scalable algorithms, they do not allow the expression of the presented decomposition algorithms. Thus Chapter 4 focuses on some additional features: one is the replication of task computation explored through

[1] diodon git repository `https://gitlab.inria.fr/diodon/cppdiodon`

the example of 2D Stencil computation, the other is the allreduce collective operation. The sum of these features becomes an extended STF programming model suitable to express scalable matrix decompositions. Part II of this manuscript is composed of a single chapter. Chapter 5 builds on the use of the features proposed in Part I for the development of data distribution schemes for algorithms on symmetric matrices. Chapter 6 concludes this thesis and enumerates some perspectives of the accomplished work.

Chapter 2

Background

Plenty of scientific fields have reveled in the use of numerical simulations because they are both confident in the equations that govern their systems and their systems are too big to fit in laboratories. Not only are numerical simulations able to tackle problems that may be hard to set up physically, the precision and scale at which they are able to do so is staggering and ever-increasing. However this sustained growth exposes researchers, engineers and the scientific computing community at large to challenges to maintain, port, and efficiently execute their applications on top of the best computers.

To benefit from the latest breakthroughs in hardware, the past trends for software implementations have been to adapt the code by combining programming models together. Section 2.1 presents the leading architectures that have been manufactured over the three past decades. Porting code over these architectures is an enterprise that becomes more challenging as the software stack gets unreasonably complex and hard to maintain, therefore difficult to expand. To address this issue runtime systems that act as a layer of abstraction between applications and both the hardware and some specialized libraries are recently being favored. These runtime systems and corresponding programming models are described in Section 2.2. They meet most requirements to address the efficient use of parallel supercomputers, especially at the shared memory level. Indeed, they commonly provide programming models that have proven relatively easy to use, productive and efficient in many contexts. Section 2.3 focuses on two omnipresent algorithms in numerical linear algebra: matrix-matrix multiplication and matrix factorization. Being able to design and implement algorithms that are scalable across platforms is paramount to use supercomputers properly. Nonetheless, the latest iterations of these scalable algorithms are still developed through home-brewed, special-

ized abstraction layers despite the existence of general runtime systems. It appears runtime systems lack features in their programming model that make optimizations for scalability easy to leverage.

2.1 An ever-complexifying ecosystem

The need for computing power has been ever-increasing as faster computers can execute programs over larger inputs, sometimes in a shorter time. The programs ran on supercomputers are used in computing-intensive scientific fields to apprehend large-scale phenomenons or in industrial settings to optimize the design of new products. Breakthroughs in hardware manufacture have been a constant driver of scientific computing performance. Figure 2.1 (`top`) illustrates how, before 2005, processors' capabilities were growing exponentially. From 2005 onwards, Moore's law keeps proving itself valid as the number of transistors grew exponentially yet the performance of a single computing thread has stalled as the frequency of the CPU clock stagnates.

Nonetheless, one can observe that the growth in computing power for supercomputers keeps its exponential pace in the bottom of Figure 2.1. Parallelism has been the key principle to maintain such trends. The exaflop barrier was breached in 2022 with the Frontier supercomputer by assembling a system where parallelism can be leveraged at many levels from the Central Processing Units (CPUs) all the way up to the Network Interface Controllers (NICs). When using modern computing centers, a programmer wants to take advantage of all the different levels of parallelism to run their program efficiently. To achieve this goal they typically rely on a great deal of abstractions. The programming language they use to write their source code is a first abstraction – as it provides a translation to machine code – but what they often rely on are specialized libraries or middlewares that leverage the actual architectural properties of the computer they use. Such libraries typically focus on a specific component or technology in the computer: from the efficient use of a single core to the data transfers over various channels, from the orchestration of computation over many cores to the use of dedicated hardware. However, because of the expertise required to use these libraries and the very large scope they encompass, the source code resulting in their usage may come off as tough to read, troublesome to profile and prone to maintenance issues.

2.1.1 Evolution of supercomputers hardware

Supercomputer constructors have sought after ways to maintain a steady increase in computing power. In doing so, they evolved from the canonical single-core machine to clusters of multicore accelerated nodes. This modification of supercomputers has been incremental, with prominent architectures often lasting for about a decade. Latest supercomputers hold thousands of computing nodes each housing hundreds of computing cores, amounting in

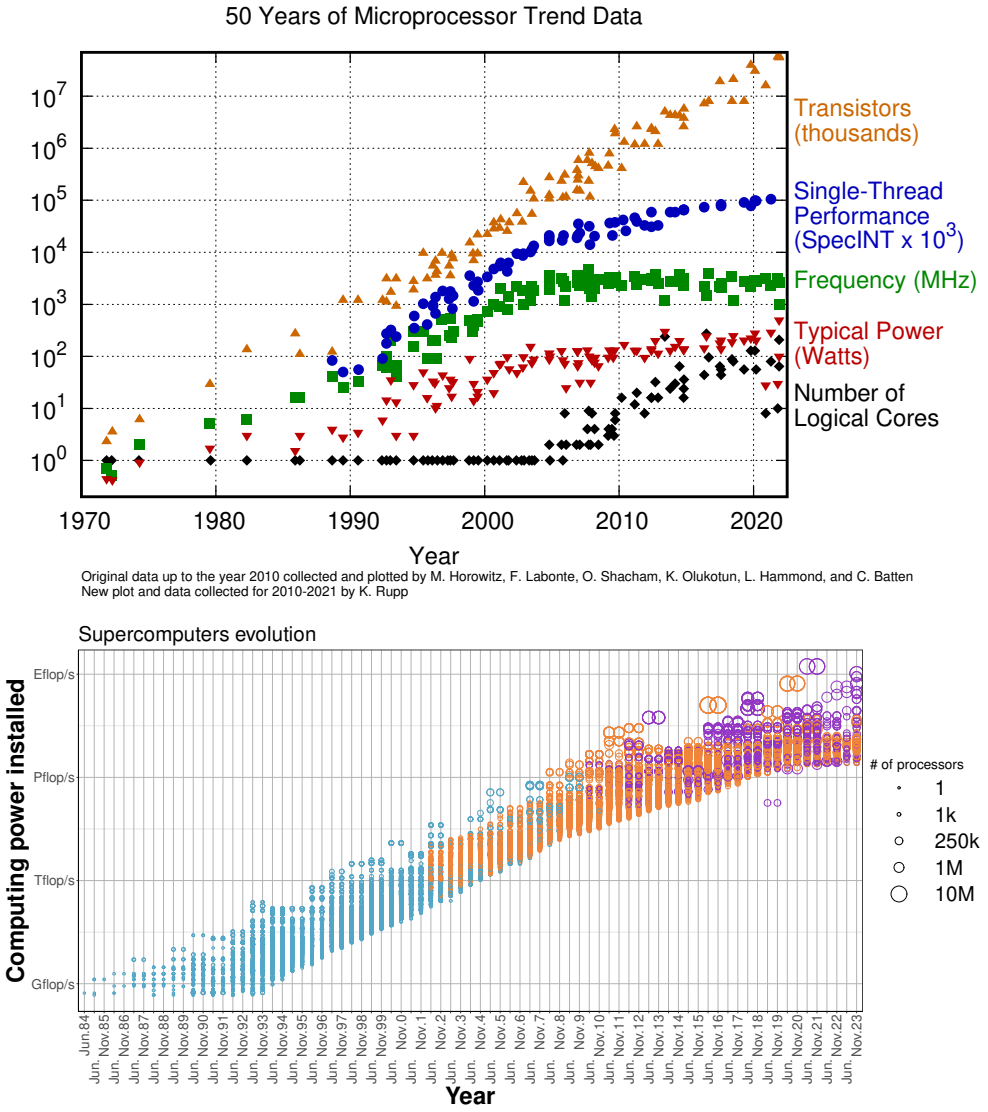


Figure 2.1: Computing trends. **Top:** Microprocessor trends between 1970 and 2020 `https://github.com/karlrupp/microprocessor-trend-data`; **Bottom:** Top 500 between 1984 and 2023 `https://top500.org/`

petaFlop/s or even exaFlop/s of computing power. In this section we will briefly discuss the main technologies that are still in use on modern supercomputers.

2.1.1.1 Clusters

Starting in the 1990s, as a single computing node became a commodity, supercomputers were built as the aggregate of several computing nodes linked through a high-performance network. The ability to efficiently distribute computations over these *clusters* of computing nodes – often referred to as *distributed-memory* parallel computers – was pivotal to increase supercomputers' computing power. This decade of supercomputing saw a fierce competition in vendors' products as well as a prolific literature to study distributed-memory algorithms.

Linking up computing nodes through a network is a high-stake endeavor since the channels used to communicate data between CPUs are orders of magnitude slower than transfers made inside the CPU chip. Therefore, the organization of interconnections between computing nodes has been a key concern when building supercomputers. Indeed one of the metrics used to measure the cost of a running supercomputer is the number of network links it requires: it would be expensive to link every compute node in a cluster with one another but it is possible to link them indirectly. These topological questions rely on dedicated hardware that routes data transfers – *switches*. These switches interconnect parts of a single cluster with one another.

2.1.1.2 Multicores

The so-called golden age of computer architecture began around the 60s and, for many years, thanks to an effective interplay between Moore's law [87] and Dennard scaling [53] allowed for producing processors that were increasingly denser, i.e., more capable, and with a higher frequency with no or little increase in the energy consumption. During this period, simply upgrading the processors of a computer lead to satisfactory performance improvements without further efforts from programmers.

In the beginning of the years 00s the Dennard scaling came to an end: the microprocessors' frequency could not be pushed any further without encountering thermal dissipation and energy consumption issues. Despite Moore's law being still valid, improving processors' performance under these constraints became more challenging. Instruction level parallelism (ILP) techniques such as vectorization or deep pipelining certainly offered a way to push performance a bit further but certainly not at the exponential rate observed in the previous decades.

It is at this moment that computing architectures switched to CPUs equipped with multiple computing cores. Instead of linking computing cores through

the network, a single processor was now built equipped with multiple cores. This design makes it possible to parallelize instructions at a higher level than ILP: sets of instructions accessing different memory addresses can be executed by different cores. This essentially leads to thread-level parallelism (TLP) where a thread is a lightweight process run over a single core. Because the buses inside a computing node are orders of magnitude faster than links over the network, this design is relatively efficient to move data between fast memory such as Random-Access Memory (RAM) and computing units.

This major technological shift lead to a sharp discontinuity with the past which can be easily seen in Figure 2.1 (*top*). The multiplication of computing cores came with newer hardware challenges. It must also be noted that, if multicore systems are able to process instructions concurrently, the rate at which each core in the system performs stagnates and, in general, is lower compared to an old-generation single-core; the reason for this trend lies in the need to respect a power consumption budget which is shared across cores.

CPUs are equipped with cache memories that store some data closer to the Arithmetic and Logical Units (ALUs) than the relatively slow RAM. In multicore computers, this cache is typically shared among multiple cores at some level (typically the farthest from CPU). Although this allows for faster communication between cores, as a shared resource, memory accesses become the subject of contention and may create bottlenecks that degrade performance. Furthermore, modern supercomputer nodes are often equipped with multiple multicore processors possibly in a non-uniform memory architecture (NUMA) setting which makes the speed access to data irregular.

Inside modern supercomputers, a single computing node can hold as many as hundreds of cores. Figure 2.2 shows the result of the `lstopo` utility program delivered by `hwloc` when executed on Platform A. This visualization makes the hierarchy and complexity of the machine apparent. On such a platform, memory accesses are non-uniform between two cores selected randomly. Non-Uniform Memory Accesses (NUMA) domains have to be taken into consideration when allocating memory over such a machine.

This technological rupture with the past came with many daunting challenges for programmers and users of intensive computing. Indeed, upgrading to a new processor does not lead to a performance improvement for sequential codes anymore; many codes and libraries had to be adapted or rewritten with parallelism in mind.

2.1.1.3 PU specialization

At the beginning of the 2010s, the supercomputers started to be equipped with accelerators and, more specifically, with Graphical Processing Units (GPUs) to increase their computing power. This can be seen as a consequence of the end or the slowdown of Moore's law – an issue that can be mitigated through specialization of processing units. These specialized processing units take

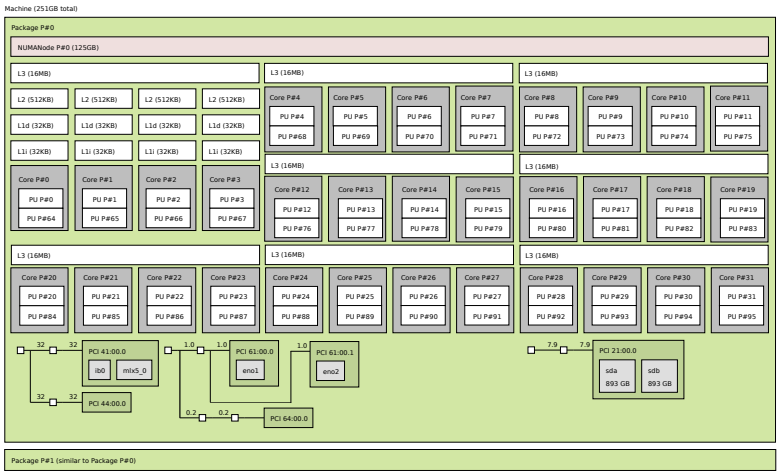


Figure 2.2: The topology of a single computing node of Platform A.

advantage of data parallelism to map instructions over numerous ALUs. A GPU architecture is designed to process massively parallel workloads. Because of their design, GPUs are especially suited for operations that deal with heavy data streams while requiring a low amount of branching and conditions. This hardware specialization is of great interest for scientific computing because applications such as simulations rely on the processing of large data inputs such as arrays of floating-point numbers. GPUs are fundamentally well suited to deal with these regular problem formulations where dense numerical linear algebra is used. However, they often fail to obtain as impressive speedups over CPUs when dealing with irregular or sparse computations. In most cases, GPUs extend the reach of tractable numerical problems but they have to be used in combination with CPUs.

GPUs are often equipped with their own dedicated RAM – often called Video RAM or VRAM – and they compute instructions using data stored therein: CPUs are used to command the data transfers from the RAM to the VRAM or command the allocation of memory in the VRAM. While the GPU memory is typically smaller than RAM, the available bandwidth between this dedicated memory and the GPU cores is higher than the bandwidth between RAM and CPUs. GPUs require such dedicated memory and high memory bandwidth because of the acceleration they provide over large chunks of memory: large bandwidth is a requirement to move data across the multiple ALUs found on the GPU chip.

Because of their large computing power and their design that target massive data parallelism, GPUs are better suited to operate on larger chunks of data. This difference with CPUs is often referred to as the granularity of the operations. A single CPU core is equipped to efficiently process fine-grain operations while a GPU chip is equipped to efficiently process coarse-grain op-

erations. When multiplying matrices, a modern CPU core can reach peak performance with hundreds of millions of flop (equivalent to multiplying square matrices of size 512) however a GPU would require a significantly larger workload to perform at its peak – about three orders of magnitude more.

Nvidia has been a key manufacturer in the beginning of the 2000s however major manufacturers are trying to seize the GPU market. Two of the principal actors are AMD and Intel.

2.1.1.4 Summary

In the course of the last three or four decades, three main hardware technologies have become dominant and widely-adopted in the domain of high-performance computing. The first is distributed-memory parallel computing achieved by connecting multiple computing nodes through a high-performance network. The second is represented by multicore processors. The third corresponds to the advent of accelerators (most commonly GPUs). In order to achieve high-performance at a very large scale, all these technologies are often combined in modern supercomputers. Although world-grade supercomputers do not use dedicated processing units such as the Fugaku. Nodes of a modern supercomputer, illustrated in Figure 2.3, are commonly equipped with multiple multicore processors, often in a NUMA setting, and multiple accelerators or GPUs. These processing units are connected to each other using different types of interconnects and are attached to different memories. These nodes are assembled in very large numbers thanks to dedicated, high-performance networks in order to achieve great performance and scalability. As a result, modern supercomputers are extremely *heterogeneous*: they are equipped with numerous processing units that have different speeds and capabilities, memories with different capacities, bandwidths and latencies and interconnects with different bandwidths and latencies. Therefore, although all these technologies allow increasing performance on paper, in practice they throw an incredible burden on the programmers who have to deal with all this complexity.

2.1.2 Parallel programming paradigms

Because of the incremental development of hardware architectures, the software stack used to program supercomputers has become incrementally complex over the years. To face this complexity, many approaches have been proposed that address different layers of the computer architecture. Through community-wide effort, some of these approaches were standardized and became adopted across scientific computing applications. MPI and OpenMP are such open standards which have been widely-implemented by vendors of hardware to deliver high performance across different machines. The use of GPUs to accelerate computers primarily relies on proprietary libraries that required more involvement from their users than standardized frameworks. Nat-

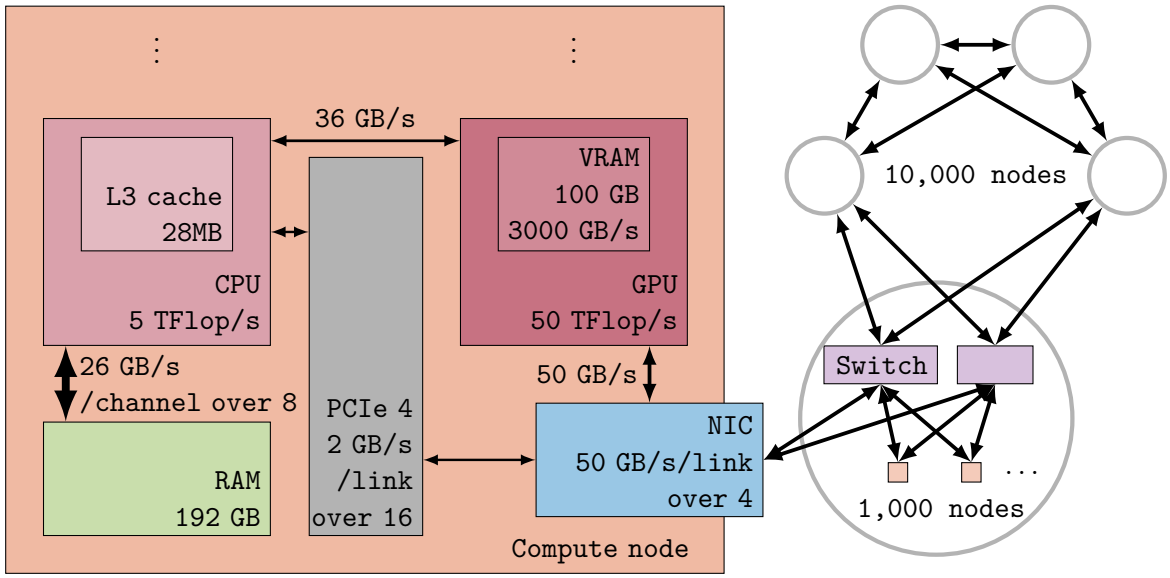


Figure 2.3: A typical HPC computing node inside a cluster of thousands of nodes. Data transfer speeds and computing speeds are indicative and only reflect adopted technologies such as PCIe4, DDR4, Infiniband HDR, etc.

urally when an application undertakes refactoring to use one single vendor's hardware, its efforts are not rewarded with performance portability across all GPUs. Ongoing initiatives aim to provide open, common standards but they are not yet widely adopted. The large combination of hardware technologies means that programmers face a challenge when trying to port their code over a large range of computer architectures. In this exercise, they must rely on the parallel programming paradigms presented in the following sections.

2.1.2.1 Message-passing paradigms

The Single Program Multiple Data streams (SPMD) approach has been popularized widely as it provides a way to tackle the implementation of a distributed-memory program. By using the proper distributed-memory paradigms, programmers can write one single program using various function calls to transfer data across the network. When they compile their program and command its execution over multiple instances, each instance can behave as either the receiver or the sender of the expressed data transfers – or even both when collective operations are invoked. Each computing node equipped with a single computing core would execute the compiled code concurrently. For a long time, SPMD approaches were largely platform-dependent and a plethora of ways to use the network existed. Because of the relative speed of the network, it was often satisfactory to rely on Bulk Synchronous Parallel (BSP) execution model to design algorithms: applications were divided into syn-

chronization, communication and computation phases and each was achieved simultaneously.

The Message-Passing Interface (MPI) standard emerged in the early 90s in a community-wide effort to design software libraries to level the implementation of distributed-memory programs [107]. The MPI standard consists of routines wrapping-up essential communication primitives, including point-to-point as well as collective data transfers, that can be called with a certain level of finesse if the user desires – for instance, explicit support of asynchronism. The MPI standard exposes several key concepts that programmers can harness. One of these concepts is the *communicator* which is a set of processes that can exchange with one another, possibly using collective communication operations such as reduce, gather, allreduce, etc. Within a communicator, each process is given a numerical identifier which we call *rank* in order to express communications more conveniently. Communicators can be split into subcommunicators of fewer processes. Through MPI, users essentially design ways to split ranks into various, hopefully independent, communicators to express the data transfers they require. Over the years, a large literature was produced to provide scalable algorithms for the implementation of efficient communication schemes: their goal was to give the ability to execute them, however complex they could be, at an arbitrary large scale over any network topology.

Other paradigms were instrumental in leveraging distributed-memory computing power such as Partitioned Global Address Space (PGAS) – which has been implemented for C through the Unified Parallel C (UPC) extension [43] or for Fortran through coarrays (since the 2008 standard). The main difference between MPI and PGAS is that PGAS focuses on providing a logical partitioning of the memory such that retrieving a remote piece of data is easily expressed as a one-sided communication. As such, the approach is easier to manage for the programmer because it hides the need to describe communication patterns in a precise manner where the receiver and the sender have to express the same communication on their respective sides. Note that the one-sided communications were eventually incorporated into MPI – without the same focus on logical partitioning that PGAS typically provides.

Many algorithms, including numerical linear algebra algorithms, were designed in terms of MPI or PGAS paradigms: this is detailed in Section 2.3.2.

2.1.2.2 OpenMP

Although it must be noted that shared-memory parallel computers existed already before multicores, it is only with the advent of this technology that shared-memory parallelism became ubiquitous and unavoidable. Therefore, since the mid 00s, many existing *multithreading* solutions were revamped and many new ones were proposed to make efficient use of this new technology. Some existing approaches such as Posix threads (or pThreads) became widely

adopted but turned out to be too low-level to allow for high productivity; proprietary solutions were also proposed, such as Intel TBB [96] but did not meet the broad interest of the HPC community. Among the other existing approaches, we can cite Cilk [33] or Charm++ [79] which are still being used although only by a limited number of programmers.

Among all the existing options, certainly the one that is the most widely adopted is OpenMP, a standard that was first submitted in 1997 [35] and considerably improved and extended over the years. While there is an inherent difficulty to exhibiting parallelism in applications, programming models such as the one offered in the OpenMP standard allow users to simply decorate their code through directives to take advantage of the multiple cores. Such a directive programming approach is convenient for the programmer because they get to point out what statements are prone to parallelism – not modifying their initial serial code much – while letting the compiler produce machine code that will be efficiently dispatched across many cores.

OpenMP relies on a *fork-join* programming model where a code is made of alternating sequential and parallel sections; within a parallel section multiple threads exist which can share work, commonly, through the use of worksharing constructs. The most popular worksharing construct, especially in the early OpenMP standards, is certainly the parallel loops directive which became so widely used that OpenMP was often, although incorrectly, referred to as "loop parallelism". Through the years, the OpenMP standard has been considerably extended and improved. One major step is represented by the introduction of task parallelism in the 3.0 version of the standard, a feature that we will deeply discuss in the remainder of this document. More recently, support for accelerators was also introduced in OpenMP v4.0 as we will explain in the next section.

Although, as explained, many sequential codes had to be rewritten using some multithreading approach in order to take advantage of multicores, parallel codes based on message passing could take advantage of this new technology right away: communications between cores sitting in the same node do not correspond to messages sent through the network but, rather, to copies in the shared memory. Nevertheless, because of these relatively expensive and avoidable copies, and because of its essentially synchronous nature, the MPI model does not allow taking full advantage of shared-memory parallelism. For this reason, many, if not most, codes designed for large scale supercomputers are based on a combination of MPI and some multithreading technology, often OpenMP. This combination is not always easy to achieve and multiple approaches exist. It must be noted that the MPI standard was recently (in MPI 3.0) extended with features that are specifically designed to achieve higher performance on shared-memory parallel computers. Nevertheless, the MPI+OpenMP approach still seems to be the most widely adopted.

2.1.2.3 Programming accelerators

Programming accelerators such as GPUs, often relies on the use of proprietary solutions. For example, one strong actor in the GPU sector is Nvidia that provides the CUDA programming toolkit to program their GPUs [47]. This toolkit includes compilers that provide an extension of the C/C++ language for distributing computations over the GPU cores, debugging and profiling tools and performance libraries, such as cuBLAS, a GPU-optimized implementation of the Basic Linear Algebra Subprograms (BLAS) library. As explained earlier, GPUs commonly have their dedicated memory which is faster than the main memory but of limited size; furthermore, some operations cannot be run efficiently on GPUs or are very hard to implement. Therefore, GPUs are typically used in combination with CPUs, which means that data has to be moved, more or less frequently, from CPU to GPU memory. Cuda comes with data management functions that allow allocating memory on the GPU and moving data from CPU to GPU and the other way around. It must be noted that these data movements make coding for GPUs relatively hard and, despite they happen through fast interconnects, their cost can, at times, overcome the benefit of offloading computations to GPUs.

Cuda is property of Nvidia and, therefore, does not work on GPUs produced by other vendors. Despite this lack in portability, there has been a strong motivation from the HPC community members to adapt their own application using Cuda which has been, for many years, the most widely, if not the only, adopted technology. Other prominent industrial actors such as AMD have recently jumped into the general-purpose GPU (GPGPU) market and have produced boards that are used (and increasingly so) on supercomputers among the most powerful ones. AMD offers the ROCm platform [16] that includes the Heterogeneous Interface for Portability (HIP) layer to target its GPUs as well as other ones. Intel recently delivered a series of consumer-grade GPUs: it is easy to conceive that computing-intensive applications may need to rely and port on a plethora of specialized hardware in a relatively near future. For this reason efforts have been pursued, such as OpenACC [89] or OpenCL [88], to provide a standard and portable programming interface for GPUs. More recently, the OpenMP standard (since v4.0) was extended with directives for programming accelerators but support from compilers is still lacking or incomplete. Other notable efforts to improve the portability of codes across GPUs include portability/abstractions layers such as SYCL [64] or Kokkos [110]. Although all these solutions provide greater portability than proprietary options, they fail to achieve the same performance.

Large code bases have often been rewritten to target trending architectures that deliver an increasingly large amount of computing power. These new architectures come with components that can only be used through specialized packages that limit the portability when adapting applications to supercomputers. Portability may require considerable effort because archi-

tectural breakthroughs force programmers to reconsider the algorithms they implement. The diversity of supercomputers ecosystem has not been interrupted yet, so software packages will need to adapt further. The next section details the middleware – specifically runtime system – that is being considered to mitigate the cost of this adaptation.

2.2 Runtime systems and their programming models

In the same vein as supercomputers have become an assembly of different technologies (i.e., distributed memory, multicores and accelerators), typical modern high-performance computing codes are implemented through a complex combination of programming models and paradigms such as MPI+OpenMP+X (where X refers to any technology used to program accelerators). This mostly happened for two reasons. The first is historical: as new hardware technologies were introduced, existing codes were incrementally extended with the corresponding programming solutions. The second lies in performance: in an attempt to squeeze the last GFlop/s out of their expensive machines, HPC practitioners often prefer to have direct access to all the hardware features through dedicated programming solutions.

As the landscape of HPC architectures becomes increasingly varied and heterogeneous, the inherent difficulty to maintain a software stack that weaves multiple paradigms addressing each component of a modern supercomputer becomes unsustainable. As a consequence, the HPC community has recently leaned toward runtime systems as a way of achieving high productivity and performance portably across many architectures. Runtime systems, or more simply *runtime*, act as a layer between the application and the hardware as pictured in Figure 2.4; by hiding most of the low-level architectural details, they provide the programmer with the vision of an abstract parallel machine and a unified programming model and interface to write code independently of the hardware features. This allows for a much higher productivity and better separation of concerns; applications or mathematics specialists can focus on the development of efficient and scalable algorithms without the burden of dealing with low-level architectural details, whereas runtime experts can develop sophisticated mechanisms and methods to deploy the workload efficiently on the available computing resources.

Numerous runtime systems have been developed over the years; an extensive review of runtime systems is given by Thibault [108]. Our work focuses on runtime systems that rely on task-based parallelism, specifically through the use of the sequential task flow (STF) programming model, which we describe in the next sections.

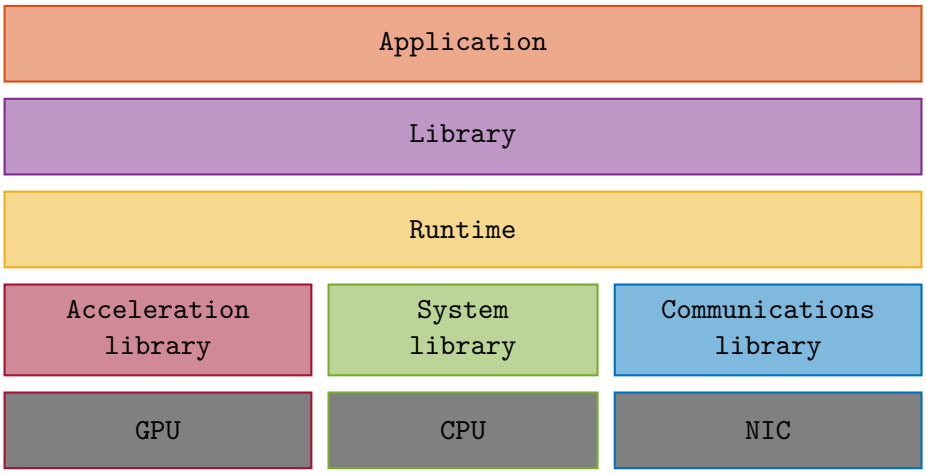


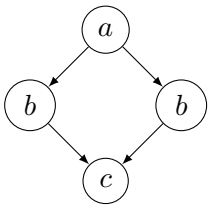
Figure 2.4: Typical HPC application software stack.

2.2.1 Features of runtime systems implementing task-based programming models

Although many, high-level, unified, parallel programming models exist, task-based parallelism and runtime systems are becoming increasingly popular in the high-performance computing domain due to their effectiveness and relative ease of use. In this programming model, the workload is expressed as a collection of *tasks*, that is, elementary operations on data. These tasks are arranged in a directed acyclic graph (DAG) that expresses their mutual dependencies and, consequently, the available parallelism: tasks lying on different paths of the dependency graph are independent and can thus be executed in any order and, possibly, concurrently. The main advantage of this programming model is that porting a code on a new architecture mostly consists in producing optimized (sequential) implementations of tasks, provided that the runtime has been extended to support the new machine. In many cases, these optimized implementation of tasks are available off the shelf within vendor libraries such as BLAS or LAPACK. For a given task type the programmer can provide several implementations, one for each type of worker that is entitled to execute tasks of this type; as such, depending on the scheduling policy and the availability of resources, the runtime can schedule the execution of a task on either worker for which an implementation was provided.

A very simple example of a DAG is provided by the diamond graph illustrated in Figure 2.5 (*left*) which represents the workload associated with the code in the right part of the figure. This DAG includes four tasks of types a, b and c. Tasks b are two different instances of the same operation but on different data and lie on the same level in the graph. From the DAG it is clear that once a is completed both b tasks can be executed and c can only be executed when both b tasks are completed themselves. In this example b

could have a Nvidia GPU implementation provided by a `cuda_b` function and a classical CPU implementation provided by `core_b`; in this case the runtime can choose to execute the two b tasks simultaneously on a CPU core and on a GPU.




```
1 A0,A1 = a()
2 A0 = b(A0)
3 A1 = b(A1)
4 c(A0,A1)
```

Figure 2.5: Diamond graph with 4 tasks. **Left:** the DAG that can be scheduled with each task assigned a letter. **Right:** a pseudocode of a sequential execution. Note that tasks b which are on the same level in the DAG can commute.

Numerous runtime systems rely on task-based parallelism, the most popular among them certainly being OpenMP. Although they might offer different features and achieve different performance and scalability, they all have to face the same challenges and, therefore, present the same, or comparable, components which we briefly describe below.

Programming model and interface The programming model and interface are certainly two key ingredients of any runtime including task-based ones. Through the programming model and interface, the runtime provides programmers with a way of expressing their workload in the form of a DAG of tasks. They must be sufficiently expressive to offer high productivity which means that the programmer must be able to express complex algorithms simply and independently of the low-level architectural details to achieve great portability.

One trivial way of describing the DAG of tasks is by explicit enumeration of all the tasks which it is composed of, along with the corresponding dependencies – an approach that clearly becomes unfeasible at very large scales. Therefore, programming models have been developed that allow for an easier description of the tasks and for the automatic detection of their dependencies. The most widely-employed programming models for task-based parallelism are certainly sequential task flow (STF) and parameterized task graph (PTG) which we describe in greater details in sections 2.2.3 and 2.2.2, respectively. These programming models can be implemented through different interfaces that rely, for example, on the use of directives (as in the case of OpenMP), functions (as in the case of StarPU) or domain specific languages (DSL) (as in the case of PaRSEC). In some case the interface not only allows describing the DAG of tasks but also allows the programmer to provide the runtime sys-

tem with hints to help it achieve better optimizations and, ultimately, better performance.

It must be noted that describing the DAG of tasks has a certain cost which can be more or less important depending on the chosen programming model and interface; modern runtime systems strive to offer the best trade-off between this overhead and expressiveness and ease of use.

Scheduling Once the DAG of tasks is described, the runtime is in charge of deploying the corresponding tasks on the available computing resources. This duty can be extremely challenging considering that, in real-life uses, DAGs can be extremely large, tasks can be of different nature and size and the underlying architecture can be large and heterogeneous. Achieving an efficient distribution of the tasks under all these constraints and taking all these parameters into account, basically amounts to a complex *scheduling* problem. Because the DAG is often generated dynamically and in order to better take the machine status into account, most of the time dynamic scheduling approaches are preferred to static ones. The literature around dynamic scheduling of DAGs of tasks on parallel architectures has been very prolific in recent years. A scheduling policy can be extremely eager and try to assign a ready task to the first available processing unit. On the other side, complex policies may be much more insightful and take many parameters into account such as the critical path, the affinity of tasks and processing units, the placement of data and the cost of data transfers through interconnects. In some cases performance models are automatically built or hints may be provided by the programmer to guide scheduling decisions. Obviously, the more complex the policy is, the higher is the overhead associated with taking scheduling decisions; therefore the granularity of tasks must be large enough that this overhead does not exceed the advantage of using a careful scheduling policy.

Modern runtime systems often include a scheduling engine which implements one or more scheduling policies that aim at maximizing the use of the available computing resources. Runtime users can easily choose the option which is better suited to their workload and computer; some runtime systems also offer the possibility to develop and plug in custom scheduling policies.

Data management In the case were a code is executed on a system that includes multiple incoherent memories, not only the execution of tasks is delegated to the runtime system but also the handling of data. In essence, the runtime will make sure that whenever a task is executed on a given processing unit, the data it needs will be available on the associated memory. In the diamond DAG presented in Figure 2.5, assuming that task a is executed on the CPU and the left task b on a GPU, the runtime system will automatically transfer data A_0 from the CPU memory to the GPU memory after the execution of task a and prior to the execution of task b. An optimization technique

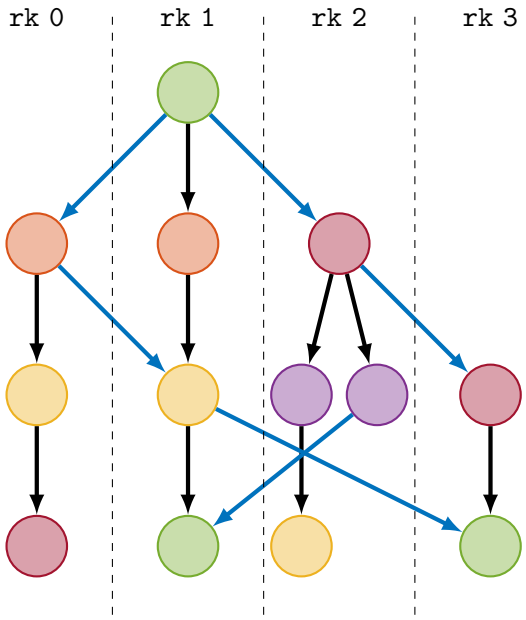


Figure 2.6: A DAG distributed over four ranks to execute a distributed memory operation. The transfers between ranks are highlighted in blue.

commonly employed by modern runtime systems, referred to as *prefetch* consists in executing these transfers in advance so that when a task is scheduled for execution, all the data it needs is already in place and the task execution can start right away without any latency.

Because of these data transfers, at any time during the execution, multiple copies of a single data may exist in the various memories; it is the duty of the runtime to ensure coherency among the existing copies. As long as one data is only read by tasks, these multiple copies can co-exist and be used simultaneously but runtime systems normally prevent multiple tasks from modifying the same data concurrently. In Chapter 4, we will propose a feature that allows relaxing this constraint temporarily.

The coherence that is ensured at the shared memory level should be propagated when working in a distributed-memory setting. Figure 2.6 presents a DAG distributed across four computing nodes; in this figure, dependencies (in solid blue) that cross the border between nodes (in dashed black) inherently describe data transfers happening over the network. Therefore, all nodes must have a sufficiently detailed knowledge of the entire DAG (more on this will be said later) in order to not only schedule the locally-executed tasks but also the necessary remote data transfers. The runtime system should enforce coherency locally such that communications are performed before the local task is executed – communications can be interpreted as tasks reading pieces of data, and being executed by a special worker process that handles the NIC.

2.2.2 The parameterized task graph (PTG) programming model

In the PTG programming model, the DAG is broken down into task classes. A class of tasks is defined by its inputs and outputs – either some instance of a task class or a given piece of data – as well as ranges of values its instances can take. This description allows for writing an algebraic representation of the DAG: it is entirely known from this high-level, implicit description. This provides the PTG model with great scalability because the DAG is not explicitly and entirely built but, instead, tasks are efficiently instantiated based on the rules defined by the programmer. However, this comes at the price of a considerably higher programming effort than the STF programming model presented in Section 2.2.3 [7].

The PTG expression for the diamond DAG presented in Figure 2.5 would result in the code presented in Algorithm 1. The task class b describes two separate instances b_0 and b_1. Nodes `a, b` and `c` are described with their input nodes stated before the first $\rightarrow$ symbol. The output nodes are stated after the second $\rightarrow$ symbol. For each input or output node, the position of the variable in the list of arguments can be deduced as it appears only once – for instance, `C` the second argument of task `a(0)` is the first argument of task `b(1)`.

Algorithm 1: PTG expression of a diamond DAG with 4 tasks.

```
1 a(i=0,B:W,C:W)▷
2     A_0 → B → b(i,B)
3     A_1 → C → b(i+1,B)
4 b(i={0,1},B:RW)▷
5     if i=0 then a(0,B,∼) → B  → c(0,B,∼)
6     else a(0,∼,C) → B  → c(0,∼,C)
7 c(i=0,B:R,C:R)▷
8     b(i,B) → B
9     b(i+1,B) → C
```

PaRSEC [38] is a runtime that implements interfaces to offer the PTG programming model. PaRSEC delivers a wider ecosystem of development tools to handle debugging, visualization, etc.. PaRSEC provides the Job Data Flow (JDF) DSL which can be compiled to source code that uses PaRSEC interfaces. The JDF format simplifies the job of the application developers as PaRSEC can translate the representation of the application to an optimized program. Once an application is described in terms of the PaRSEC interfaces, the runtime system is able to schedule its execution over available processing units in a shared memory as well as a distributed-memory environment. The PTG approach has recently been generalized into the Template Task Graph (TTG) programming model [37]. TTG is focused on the portability and performance of irregular, sparse workflows which often prove challenging because of conditional execution or very low granularity.

2.2.3 The sequential task flow (STF) programming model

The focus of this manuscript is the STF programming model, 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. This programming model does not deal with the issuing of instructions but rather the submission of tasks: the user describes computation to the runtime and let it schedule them accordingly. The STF model relies on a task insertion or submission primitive which allows creating said tasks: `insert_task`. The insertion is non-blocking in the sense 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 not only specify the operation that the task must execute but also 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.

The automatic detection of dependencies from task submission can be achieved by the runtime system by following Bernstein's conditions [28]. In a task-based paradigm, task b depends on a previously-submitted task a if:

- Tasks a and b write over the same datum (Write After Write, "WAW")
- Task a writes over data task b reads (Read After Write, "RAW")
- Task b writes over data task a reads (Write After Read, "WAR")

The diamond DAG presented in Figure 2.5 is expressed through the STF programming model as in Algorithm 2. The reader can appreciate how the expression is very similar to the one of the sequential code: the role of the programmer is simply to delegate function calls to the runtime system.

Algorithm 2: STF expression of a diamond DAG with 4 tasks.

```
1 insert_task(a, A0:W, A1:W)
2 insert_task(b, A0:RW)
3 insert_task(b, A1:RW)
4 insert_task(c, A0:R, A1:R)
```

In the case of a shared-memory parallel computer, the STF model commonly relies on the use of a *manager* process in charge of inserting the tasks and multiple *workers* 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 the case of a distributed-memory machine, multiple managers (at least one per computing node) 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 in charge of distributed-memory data transfers. For this reason, in the most basic use of the STF model, all managers 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 (Figure 4.2 in his dissertation): he outlined a SPMD approach where a single basic program is run across processes to handle the distributed-memory task insertion [114]. This program, introduced in Algorithm 3, indicates how to process a task T in a distributed-memory environment. First, all processes have to nominate the same processor P_{exe} that will execute the task. As P_{exe} requires valid copies of the task's inputs A_i, communications may need to be scheduled between processes owning A_i and P_{exe}. The runtime should also track dependencies between tasks to avoid redundant communications – A_i should not be sent back if it has not been modified. Because every process unrolls the same DAG it is indeed possible to associate a cache record with any data A_i. Agullo et al. implemented a distributed-memory cache mechanism to avoid these redundant communications [6]. The required update of this cache is omitted from the algorithm.

Algorithm 3: Generic statements executed by the rank `me` in a task-based distributed runtime system processing a task `T`.

```
1 task = build_task(T)
2 P_exe ← task.executing_rank
3 for A_i in task.inputs if me = A_i.owner and A_i invalid in P_exe's memory
    do
4 |  insert_mpi_exchange(A_i,to: P_exe)
5 if me = P_exe then
6 |  insert_local_task(task)
```

The STF 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 can this be time-consuming, but it may require considerable resources for the management of the DAG when it is sizeable. Several techniques have been recommended in the literature to alleviate this issue. One such technique is the pruning of the DAG traversal as depicted in Figure 2.7 which reduces the DAG generation time on each rank by only inserting the subset of tasks that are relevant to this rank [6]; these include tasks that are meant to be executed locally as well as remote tasks that are connected to local ones through an inbound or an outbound dependency. A technique that also limits

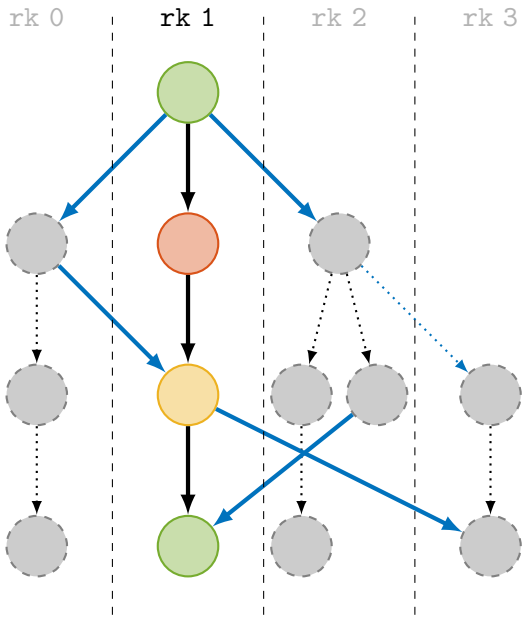


Figure 2.7: A pruned DAG focusing on the operation executed by a single rank. Discontinued elements are not necessary and as such can be removed from the DAG inserted at rank 1.

the processing of irrelevant tasks is hierarchical tasks: parts of the DAG may be discarded at a high-level in the hierarchy such that a given rank does not explore the entirety of the DAG [93, 74, 80]. The pruning of the DAG can be put in place by replicating the control of the memory regions managed by the runtime system as studied in Legion [104, 23].

The STF model is, for example, available in OpenMP (through the `task` directive and the `depend` clause), OmpSs [54] or StarPU [20], the runtime system we use in this work. PaRSEC also provides an STF interface called Dynamic Task Discovery in reference to the DAG being built at runtime rather than symbolically [72]. Chunks and Tasks is a lightweight library that provides an STF programming model where *tasks* get registered sequentially to use the logically partitioned *chunks* of memory [99], its authors have also proposed a related linear algebra library [100]. In the Legion runtime, tasks can be dispatched dynamically to be executed over the relevant memory regions [24]; like PaRSEC it provides a DSL to ease the implementation of applications [103].

2.2.4 StarPU

StarPU is a runtime system that delivers an interface for the STF programming model. This interface has been refined and extended since the work achieved by Augonnet during his PhD thesis [19]. Because StarPU is central to this thesis – extensions to the STF model are implemented and tested with

StarPU – this section presents its architecture and main concepts. StarPU is open-source and available as free software [1]. It has been used to implement numerous parallel codes.

2.2.4.1 StarPU architecture and concepts

At the level of a manager thread, StarPU provides two large components in its infrastructure: a Virtual Shared Memory (VSM) and a modular scheduling engine. The VSM is a way for users to delegate the handling of their data to StarPU. Algorithm 4 shows how a simple array of 10 single-precision elements can be registered into the VSM; from this moment on, StarPU has full control over this data and the user must not attempt to use it directly although it can reclaim the data at any moment through dedicated methods. While the presented example registers `array` as a generic space in memory, StarPU makes it possible to describe more complex data structures. Some of these data structures – vector, matrix, etc. are already described by StarPU. Datatypes are useful to describe how data can be manipulated and transferred across memories. Users may create their own by providing key routines to describe packing/unpacking of a piece of data. Pieces of data managed by StarPU are referred to as *handles*. These handles are used when submitting tasks through the usual `insert_task` function.

Algorithm 4: Registering data with StarPU.

```
1 starpu_data_handle_t handle;
2 array = malloc(10*sizeof(float));
3 starpu_data_register(&handle, array, 10*sizeof(float));
```

In StarPU, the submission of a task relies on the use of a so-called *codelet*. A codelet is, essentially, a task prototype that describes all the important information needed by the runtime to instantiate actual tasks of a given type. The codelet must describe the code that will be run when the task execution is triggered; specifically, one code variant may be provided for each PU type available on the machine so that the runtime can pick whatever PU is most suitable based on the scheduling policy. Once a codelet is defined, tasks can be inserted through the `starpu_insert_task` routine. This routine takes the codelet itself, handles referencing data along with their access modes (`STARPU_R`, `STARPU_W`, `STARPU_RW` and more) and, possibly, other data passed "by value" through a `STARPU_VALUE` access mode. Other information can be provided at the task insertion such as a priority, a scratchpad memory, scheduling hints. An example of codelet declaration and task submission is described in Algorithm 5.

When a task is submitted to StarPU, it becomes ready once all the required handles are available. In Algorithm 5, this may happen once all the previously-

[1] https://starpu.gitlabpages.inria.fr/

Algorithm 5: Inserting user-defined tasks with StarPU.

```
1  starpu_codelet work_cl ← {
2      .cpu_func = work_cpu,
3      .cuda_func = work_cuda,
4      .opencl_func = work_opencl,
5      .fpga_func = work_fpga,
6      .nhandles = 1
7  };
8  starpu_data_handle_t handle;
9  int value = 42;
10 starpu_insert_task(work_cl, STARPU_RW, handle,
11   STARPU_VALUE, &value, sizeof(int), 0);
```

inserted tasks reading or writing over `array` have been executed. When a task becomes ready, it has to be scheduled over one of the available processing units. Users choose a scheduling policy to make this very complex decision. Several scheduling policies are implemented in StarPU but users can develop their own if they see fit. To take advantage of the heterogeneous cores found on a typical modern machine, some scheduling policies rely on performance models that are trained through simple machine-learning techniques such as least-square fitting.

StarPU has been extended to provide a distributed memory support named StarPU-MPI. The VSM of StarPU has been adapted to this extension: it is possible to register a piece of data without knowing its location in memory *i.e.* through a `NULL` pointer. Each handle has an owner MPI rank. If we assume that rank 0 is the owner of `array`, Algorithm 4 can simply be adapted into Algorithm 6 to take advantage of the distributed memory. Instead of using `starpu_insert_task`, tasks should be inserted through the MPI layer with `starpu_mpi_insert_task`. By using this layer, the runtime is able to check for necessary data transfers and schedule them in advance. StarPU-MPI can be compiled to use different communications engine backends: currently, it supports a generic MPI backend as well as a NewMadeleine backend [51]. Some backends are able to leverage specific features: through NewMadeleine, dynamic broadcasts are available [52].

Algorithm 6: Registering data with StarPU executed by rank `me`.

```
1 owner ← 0
2 array ← if me = owner then  malloc(10*sizeof(float)) else NULL
3 starpu_data_handle_t handle;
4 starpu_data_register(&handle, array, 10*sizeof(float);)
5 starpu_mpi_register(&handle, owner);
```

2.2.5 Summary

This section has presented several programming models that are being used in scientific computing applications. Their portability, productivity and the performance they deliver make them suitable candidates to guarantee they can run applications efficiently and effortlessly on a single computing node across supercomputers. To make sure these programming models are actually useful to target modern and future supercomputers, we have to present what algorithms they should be able to express. Indeed the mechanisms that are in place when dealing with large-scale distributed memory may be poorly delivered by runtime systems. Thus the next section is interested in scalable algorithms that are designed to perform efficiently on large-scale machines.

2.3 Dense numerical linear algebra software and algorithms

In this work we will mostly focus on dense numerical linear algebra algorithms and software which deal with the computation of operations that use dense matrices, i.e., matrices whose coefficients are all assumed to be mostly nonzero.

Numerical linear algebra libraries have been at the core of scientific computing applications because they provide all the necessary building blocks for large-scale simulations, solution of systems, etc.. They have been present from the birth of computer science and strive to keep up with novel hardware. The job of these libraries' developers is not only to enhance the capabilities of the software, it also includes code maintenance and bug fixing. Therefore the decision to undergo new programming paradigms for linear algebra libraries may be hazardous as it is difficult to ensure that a chosen programming model is sufficiently not error-prone, sufficiently easy to port and sufficiently resilient to new mechanisms that will be uncovered in the future.

Most numerical linear algebra libraries provide algorithms and methods that rely on efficient sequential building blocks to achieve high performance. Two prominent examples are the BLAS [84] and LAPACK [17] libraries which are, most often, optimized and provided by hardware vendors. These foundational libraries standardize sets of subprograms that are meant to run efficiently over a single computing core although most modern implementations often provide shared-memory parallel versions. The specifications of BLAS and LAPACK are regularly adapted in other libraries – cuBlas is a BLAS implementation for Nvidia GPUs. These subprograms are very important to design applications because they offer tools that are readily usable and relatively easy to compose for the programmer. Modern libraries dealing with shared and distributed memory rely as much as possible on these efficient kernels to either implement different algorithms or adapt them into routines that target novel architectures. Some of these state-of-the-art dense numerical

linear algebra libraries are presented in Section 2.3.1. Advanced scalable algorithms not yet implemented in these libraries are introduced in Section 2.3.2.

2.3.1 Distributed memory dense linear algebra libraries

When implementing software and algorithms for distributed-memory computers, design choices related to the (intimately related) distributions of data and operations on the participating nodes are of paramount importance. Because of the relatively regular nature of dense linear algebra algorithms, reference software libraries often (if not always) employ static distribution approaches; nevertheless the distribution has to be carefully designed in order to maximize the workload balance and reduce the volume and number of communications in order to achieve high-performance and scalability. In most of the widely-used dense linear algebra software libraries, data and workload distributions are designed similarly and only differ in minor details. Major differences, instead, lie in how communications are done among computing nodes (for example, whether they are overlapped with computations or not, whether and how collective communications are used), how intra-node parallelism is used (i.e., natively or through multithreaded BLAS/LAPACK) and whether they support accelerators.

Certainly the most widely-known dense linear algebra library for distributed-memory computers is ScaLAPACK [29], which was developed in the 90s but is still in great use. ScaLAPACK is written in C and Fortran 77 and relies on the message-passing parallel programming paradigm; it includes a communication library called BLACS (for Basic Linear Algebra Communication Subroutines) which provides communication primitives specifically designed for linear algebra algorithms and developed on top of MPI or PVM (Parallel Virtual Machine, a message-passing standard that has now become obsolete). All communications within ScaLAPACK are blocking and thus not overlapped with computations. In ScaLAPACK matrices are distributed according to a 2D Block Cyclic (2DBC) distribution exemplified in Figure 2.8. The P participating ranks are arranged in a logical grid of size $p \times q$; a $m \times n$ matrix A is then split in blocks of size $m_b \times n_b$ and each block A_{ij} is stored on the local memory of rank $q(i\%p)+j\%q$. This distribution has two main favorable properties. First, it simplifies the task of balancing the workload for a wide range of dense linear algebra algorithms. Second, with this distribution, in many dense linear algebra algorithms, most communication happen (potentially in parallel) within rows and columns of the ranks' grid; this property can be very conveniently exploited in the message-passing paradigm by defining one subcommunicator for each row and column of the grid of ranks. ScaLAPACK does not have native support for intra-node shared-memory parallelism but, rather, uses it indirectly by using multithreaded BLAS and LAPACK libraries for local computations. This is possible because all the blocks assigned to a rank are stored contiguously in memory, which makes it possible to call BLAS

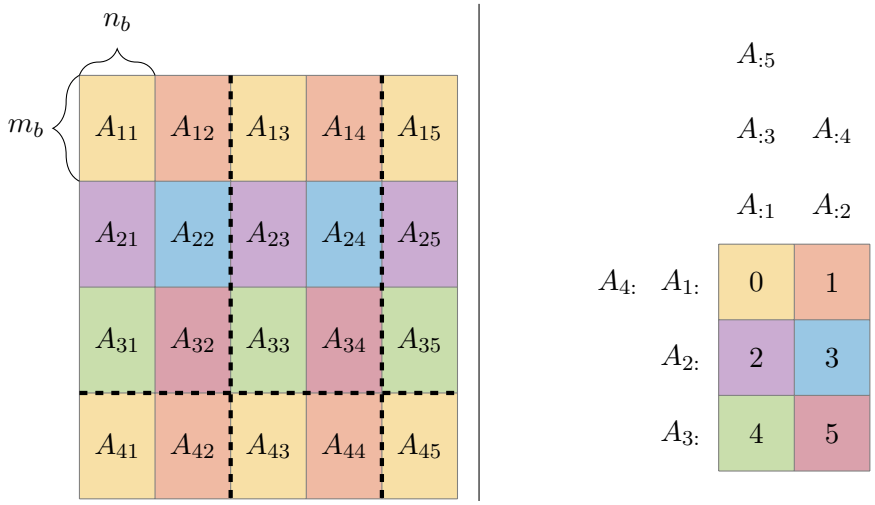


Figure 2.8: The 2D Block Cyclic layout with $p = 3$, $q = 2$ for a 4×5 matrix. Each rank is colored differently and assigned a number between 0 and $pq - 1$. **Left:** each block of the matrix A of size $m_b \times n_b$ is owned by a different rank. The rank is determined by cyclically stamping the matrix with the pattern of size $p \times q$. **Right:** the blocks owned by a given rank are the combination of rows and columns of A that intersect with the location of the rank in the logical grid. For instance, 1 owns the combination of $A_{4:}$ and $A_{1:}$ with $A_{:4}$ and $A_{:2}$ *i.e.* rank 1 owns $A_{44}, A_{42}, A_{14}, A_{12}$.

and LAPACK routines on large submatrices at once. ScaLAPACK does not support accelerators natively although some attempts have been made to port it on GPUs which essentially amount to synchronously offloading computations to GPUs through specialized BLAS or LAPACK libraries. ScaLAPACK has hardly evolved through the years; this is partly due its very monolithic design where the 2DBC data distribution is tightly hardwired in the algorithms and communication library. For this reason several attempts have been made at replacing this library (some of which we mention below) rather than porting it on modern architectures.

Elemental [95] was an attempt at providing a reference library for domain scientists to cover dense linear algebra routines. Instead of a 2DBC distribution, the Elemental library relies on a 2D Element Cyclic distribution, *i.e.*, elements owned by a given rank are contiguous in local memory – this is essentially a 2DBC layout with $m_b = n_b = 1$. This choice of distribution still allows for efficient use of level 3 BLAS. The cornerstone design of Elemental is to heavily rely on its `flame` runtime. The library leverages the C++ object-oriented capabilities to express the commonly-encountered matrix structures such that pointing toward a given (range of) element(s) is achieved transparently in computational kernels. All operations between ranks are amenable

to row or column communication patterns that are standardized in MPI. The efficient execution of BLAS is handled transparently as well as the portability to accelerated platforms equipped with GPUs.

Slate [57] has been coined the successor to ScaLAPACK by its designers. Its goal is to rewrite the entire library from the ground up using modern approaches – exploiting C++ features such as templating – and abstracting data transfers over GPUs through the OpenMP directive programming. Slate uses the 2DBC approach of ScaLAPACK however it departs from storing the locally-owned matrix inside a single contiguous array. Slate instead stores each block of the matrix inside its own contiguous array such that BLAS can be efficiently called over a single block. This makes it possible for Slate to handle natively intra-node shared-memory parallelism using OpenMP tasking rather than relying on multithreaded BLAS/LAPACK. Additionally, this approach allows storing distributed-memory matrices of different classes – such as triangular, symmetric or band matrices – in a more efficient way. Slate can leverage other layouts than the canonical 2DBC layout. Matrix classes are easily expressed in the objected-oriented programming offered through C++.

Chameleon [4] is another attempt at providing a replacement for ScaLAPACK. Written in C, it relies on task-based parallelism through the STF programming model. One peculiarity of Chameleon is that it can use different runtime systems, namely OpenMP, PaRSEC, Quark or StarPU. It supports intra-node shared-memory parallelism natively and, depending on the chosen runtime, distributed-memory parallelism and GPUs (from Nvidia and AMD in the latest release). By default, Chameleon uses the same storage format and distribution as Slate, i.e., 2DBC with local storage by blocks. However, thanks to high abstraction of the STF programming model, algorithms are expressed independently of the matrix distribution and therefore it is relatively easy to implement custom distributions.

DPlasma [39] also uses general-purpose runtime systems to its advantage. Implementations of linear algebra routines in DPlasma however are not written in the usual C/C++/Fortran programming language but rather the JDF DSL compiled by PaRSEC. While the approach differs from Chameleon, the use of a general runtime system allows both of the libraries to port to a wide range of architectures and outperform ScaLAPACK.

Modern libraries considerably improve the readability of their routines when compared with ScaLAPACK thanks to the use of higher abstraction made possible by modern programming languages and models. While a ubiquitous operation like the General Matrix-Matrix (GEMM) multiplication is written over hundreds of lines to describe intricate memory transfers, the descendants of ScaLAPACK often manage to express GEMM in about a hundred lines of source code – as low as a few dozens for Elemental. However, the scope of operations modern libraries are able to target is not as vast as ScaLAPACK. These libraries are expanding on a continuous-flow basis by integrating routines when needs arise. For the single pGEMM routine – the parallel im-

plementation of GEMM which has 3 implemented variants in ScaLAPACK – the discontinued Elemental was able to implement the 3 of them in dedicated routines but DPlasma, Chameleon and Slate fail to achieve the same coverage.

2.3.2 Scalable algorithms

The next sections deal with General Matrix Multiplication (GEMM– Section 2.3.2.1), Symmetric Matrix Multiplication (SYMM– Section 2.3.2.2) and matrix decompositions (POTRF– Section 2.3.2.3). Algorithms over 2D logical grids *e.g.* using matrices stored in 2DBC layouts are introduced first so that the reader can get acquainted with how the STF programming model can be used to express these routines. The presented 2D algorithms have been implemented in the linear algebra software libraries presented in Section 2.3.1. This section also presents variants of GEMM and POTRF that improve their scalability over the 2D variants. These improvements come from using 3D logical grids to map block-wise computations. Using 3D logical grids leads to a reduction in communication-volume: in fact, the 3D algorithms are often referred to as "communication-avoiding".

While 3D algorithms have been the objects of several studies they are not delivered by the state-of-the-art linear algebra packages we have presented in this chapter. Rather, these implementations are confined to open-source `git` repositories where they often remain standalone. The provided repositories regularly rely on home-brewed layers of abstractions that are tailored for numerical linear algebra. As such, they generally fail to separate concerns in a way that would foster sharing the diverse expertise of the computer science community. Oftentimes, these implementations rely on "MPI+OpenMP+CUDA" and have probably demanded great effort by their programmers to write while not necessarily being easy to port or simple to maintain.

2.3.2.1 Distributed memory scalable matrix multiplication algorithms

The GEMM operation, as defined in the BLAS standard, consists in computing

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

with C, $op(A)$ and $op(B)$ being, respectively $M \times N$, $M \times K$ and $K \times N$ real or complex matrices , $op(.)$ being either the identity, the transpose or conjugate transpose (only for complex matrices) operator and α and β real or complex scalars. Without loss of generality, in the remainder of this document 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 assume 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 be simply written as the triply nested loop in Algorithm 7, where the instruction in the innermost loop computes $C_{ij} = C_{ij} + A_{il} \cdot B_{lj}$. 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.

Algorithm 7: Sequential, blocked GEMM.

```
1 for i = 1...m do
2 |   for j = 1...n do
3 |   |   for l = 1...k do
4 |   |   |   call gemm (A_il, B_lj, C_ij)
```

When targeting distributed-memory parallel computers, the A, B and C matrices must be distributed among the ranks that participate in the computation. For the sake of simplicity, we will 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 [41], for example, has been proved to minimize both the communication bandwidth and latency [76, 21]. Nevertheless, this algorithm only works on square ranks' grids and is, therefore, unpractical. SUMMA [102, 58, 2] overcomes the limitations of the Cannon's algorithm and has become the most widely adopted algorithm in reference parallel dense linear algebra libraries such as ScaLAPACK [29] or PLAPACK [59]. In the SUMMA algorithm, shown in Algorithm 8, 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_{ij} blocks it owns and, therefore must receive the corresponding A_{il} and B_{lj} blocks; the outer product formulation allows transferring these blocks using efficient collective communications: the A_{il} block is broadcasted to all the ranks in the r-th grid row and the B_{lj} block is broadcasted to all the ranks in the c-th grid column. A pipelined version of this algorithm was also proposed [58] which further reduces the length of the critical path of the parallel matrix product; however, if non-blocking collective communications are available, the interest of this variant is limited with respect to the basic one.

We highly encourage the reader to take a look at the existing implementations in ScaLAPACK [1], Elemental [2], Slate [3], Chameleon [4] and Dplasma [5]. The algorithm that is implemented by these libraries is detailed in the follow-

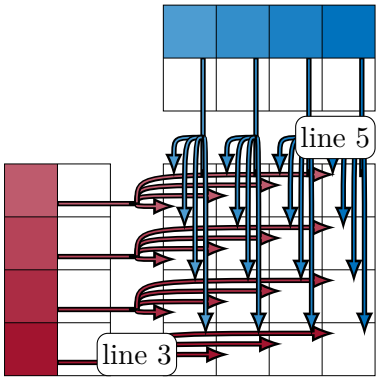


Figure 2.9: Pattern of communications of Stationary-C SUMMA for $l = 1$ on an example with a 4×4 ranks' grid, $m = n = 4$ and $k = 2$.

ing.

Algorithm 8: Stationary-C SUMMA algorithm as executed by the (r, c) rank.

```
1 for l = 1 ... k do
2 |   forall i such that i%p = r do
3 |   |   bcast(A_il, to:(r,:))
4 |   forall j such that j%q = c do
5 |   |   bcast(B_lj, to:(:,c))
6 |   forall i such that i%p = r do
7 |   |   forall j such that j%q = c do
8 |   |   |   call gemm (A_il, B_lj, C_ij)
```

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 put forward by Schatz, Geijn, and Poulson. 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 Algorithm 9, 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-

[1] https://github.com/Reference-ScaLAPACK/scalapack/blob/master/PBLAS/SRC/PTOOLS/PB_CpgemmAB.c

[2] https://github.com/LLNL/Elemental/blob/hydrogen/src/blas_like/level3/Gemm/NN.hpp

[3] https://github.com/icl-utk-edu/slate/blob/master/src/gemmC.cc

[4] https://gitlab.inria.fr/solverstack/chameleon/-/blob/master/compute/pzgemm.c

[5] https://github.com/ICLDisco/dplasma/blob/master/src/zgemm_NN_summa.jdf

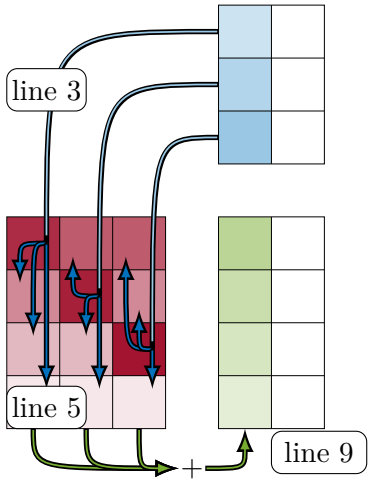


Figure 2.10: Pattern of communications of Stationary-A SUMMA for $j = 1$ on an example with a 4×3 ranks' grid, $m = 4$,$n = 2$,$k = 3$.

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_{ij}) 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 [102].

Algorithm 9: Stationary-A SUMMA algorithm as executed by the (r, c) rank.

```
1 for j = 1...n do
2 |   forall l such that l%p = r and l%q = c do
3 |   |   recv(B_lj, from:(r, j%q))
4 |   forall l such that l%p = c do
5 |   |   bcast(B_lj, to:(:, c))
6 |   forall i such that i%p = r do
7 |   |   forall l such that l%q = c do
8 |   |   |   call gemm (A_il, B_lj, iC^h_ij)
9 |   |   reduce(C^h_ij, to:(i%r, j%q))
```

The scalability of the SUMMA algorithm can be further improved using so-called 2.5D or 3D algorithms [60, 102]. 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 pseudocode for the 3D stat-C case executed by the (r, c, h) rank is reported in Algorithm 10. 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

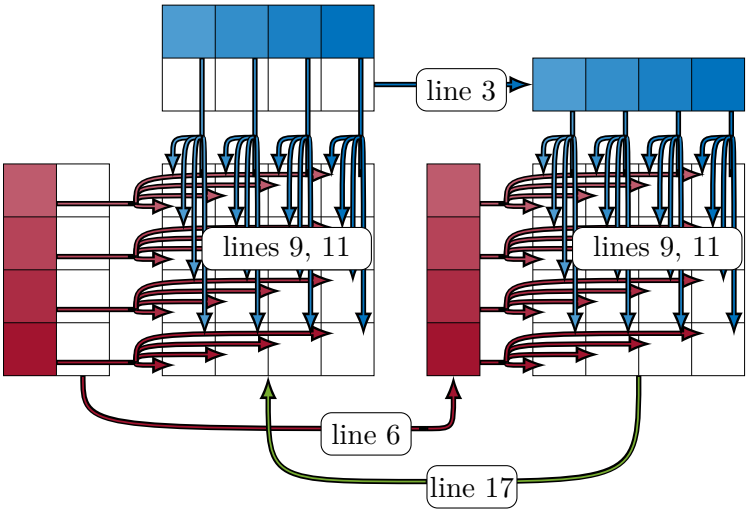


Figure 2.11: Pattern of communications are shown for the 3D stationary-C SUMMA algorithm on a $4 \times 4 \times 2$ ranks' grid with $m = n = 4$ and $k = 2$.

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, Geijn, and Poulson [102] for the related details.

Algorithm 10: 3D stationary-C SUMMA algorithm as executed by the (r, c, h) rank.

```
1  forall j such that h*k/s ≤ j < (h+1)*k/s, j%q = c do
2  |   forall i such that i%p = r do
3  |   |   recv(A_ij, from:(r,c,0))
4  forall i such that h*k/s ≤ i < (h+1)*k/s, i%p = r do
5  |   forall j such that j%q = c do
6  |   |   recv(B_ij, from:(r,c,0))
7  for l = h*k/s, (h+1)*k/s − 1 do
8  |   forall i such that i%p = r do
9  |   |   bcast(A_il, to:(r,:,h))
10 |   forall j such that j%q = c do
11 |   |   bcast(B_lj, to:(:,c,h))
12 |   forall i such that i%p = r do
13 |   |   forall j such that j%q = c do
14 |   |   |   call gemm (A_il, B_lj, C^h_ij)
15 forall i such that i%p = r do
16 |   forall j such that j%q = c do
17 |   |   reduce(C^h_ij, to:(r,c,0))
```

2.3.2.2 The case of A a symmetric matrix

The general matrix-matrix multiplication – *i.e.* not assuming A is symmetric (nor even square) – has been the focal point of many meticulous studies [1, 58, 102, 83]. On the other hand, relatively little attention has been devoted to handling the specific features of SYMM in a distributed-memory context. As a consequence, its implementation in reference codes such as ScaLAPACK [29] or Elemental [95] follows the same parallel design as GEMM, relying on a 2DBC data distribution.

2.3.2.3 Distributed memory scalable factorization algorithms

LAPACK standardizes multiple routines to factorize matrices [17]. These routines implement algorithms that take the properties of the matrices into consideration. In the case of factorization resulting in triangular matrices, two principal implementations are GETRF which outputs the LU factorization of an input matrix A of any size and POTRF which outputs the LL^T factorization of a symmetric positive definite matrix A. If not for numerical stability considerations that lead to pivoting in GETRF, both algorithms execute similarly: as they iterate over the columns of A, they factor the diagonal element a_{ii}, scale the elements below the diagonal using u_{ii} or l_{ii} and update the trailing submatrix through a rank-one update using the vector $l_{i,i+1:}$ – as well as $u_{i+1:,i}$ if A is not symmetric. A partially factorized matrix is presented in the left of Figure 2.12.

The same three steps described above are also found in blocked variants of the factorization algorithms. At each step, instead of factorizing the element a_{ii}, an entire diagonal block of A is factorized using GETRF or POTRF. All the off-diagonal blocks are then solved with respect to U_{ii} – or L_{ii}^T – through the `trsm` subprogram. Given square blocks of size b, updating the remainder of the matrix is equivalent to a rank-b update using `gemm` – or `syrk` if A is symmetric – with off-diagonal blocks (see Algorithm 1 from Buttari et al.[40]). A partially factorized, blocked matrix is presented in the right of Figure 2.12.

As explained above, in a distributed-memory setting an important design choice concerns the workload distribution. In the case of factorization algorithms, this assignment can be visualized as a 3D lattice where the element (i, j, l) corresponds to the processing of the block A_{ij} at the l^{th} step of the factorization. If $l > max(i, j)$, the A_{ij} block has already been factorized. Figure 2.13 illustrates the case of the LU factorization where blocks of color correspond to an operation executed over a block of the input matrix. While this representation can guide programmers to extract parallelism when designing software, it fails to account for existing dependencies in the operation: a block can be factorized (or solved) if and only if it has been updated with respect to all previous iterations. A pseudocode for the right-looking distributed-memory Cholesky factorization is provided in Algorithm 11. In

Figure 2.12: `GETRF` algorithm at a given step where the orange panel is the factorized L matrix, the purple panel is the factorized U matrix, the red element (resp. blocks) corresponds to the factorization step, the green elements (resp. blocks) correspond to the solve step and the blue elements (resp. blocks) correspond to the update step. **Left:** Element-wise factorization. **Right:** Block-wise factorization.

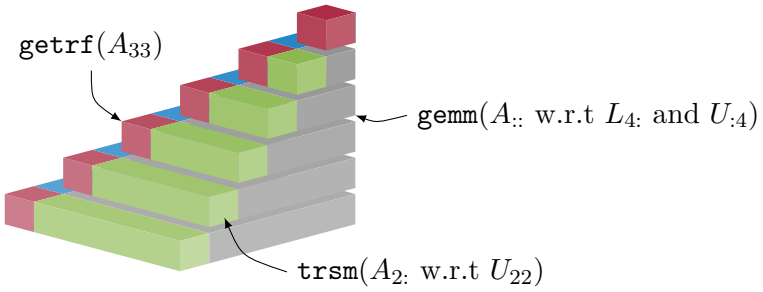


Figure 2.13: Visualization of the elementary block operations involved in the LU factorization. Red blocks correspond to `getrf` tasks applied on diagonal blocks, blue and green tasks are `trsm` tasks applied off the diagonal, and grey tasks correspond to update tasks.

this algorithm, updates are executed as early as possible to benefit from high parallelism. Updates can also be deferred as late as possible which lead to the left-looking variant of the LL^T factorization. The Cholesky factorization has been central, both because of its relative simplicity and relevance in scientific computing, to showcase the advancements of runtime systems in numerical linear algebra [6, 52].

The presented Cholesky decomposition relies on the 2DBC layout. Because there are plenty of independent statements through `trsm` and `gemm` calls, it is possible to obtain relatively good performance with this algorithm: each rank is involved in most steps to execute multiple block-wise operations. Nonetheless this distribution is not the most efficient to tackle the operation. First, it is important to use the symmetric property of the matrix in an advantageous way [25]. Then, as for Matrix-Matrix multiplications presented in Section 2.3.2.1, 3D logical grids of ranks can be operated to further reduce the communication volume [75].

3D Cholesky factorization algorithms have often been implemented through

Algorithm 11: Right-looking 2DBC Cholesky factorization of a matrix A by blocks for a rank (r, c).

```
1  for l = 1...m do
2  |  if l%q = c then
3  |  |  if l%p = r then
4  |  |  |  call potrf (A_ll)
5  |  |  bcast(A_ll,to:(:, c))
6  |  forall i = l + 1...m such that i%p = r do
7  |  |  if l%q = c then
8  |  |  |  call trsm (A_ll,A_il)
9  |  |  bcast(A_il,to:(r, :))
10 |  |  bcast(A_il,to:(:, c))
11 |  forall i = l + 1...m such that i%p = r do
12 |  |  if i%q = c then
13 |  |  |  call syrk (A_il,A_ij)
14 |  |  forall j = i + 1...m such that j%q = c do
15 |  |  |  if i%q = c then
16 |  |  |  |  call gemm (A_il,A_jl,A_ij)
```

the standard MPI+X programming model: it is possible to ease the design of factorization algorithms through the use of runtime systems. This approach enables the use of specific layouts to further enhance scalability – this has been shown with symmetric layouts for POTRF by Beaumont et al. [25]. The symmetric layouts they consider can be adapted to 2D logical grids as well as 3D ones. In their 3D algorithm, Beaumont et al. assigned each step of the Cholesky factorization to a single layer and they distributed the matrix accordingly. This round-robin storage and assignment of tasks is depicted in the right of Figure 2.14.

In their algorithms, Solomonik and Demmel assign the updates related to the l^{th} iteration to the $(l\%h)^{th}$ layer – assuming there are h layers in the 3D grid [105]. The first benefit of using a 3D logical grid comes with a reduction in communication volume. Similarly to matrix-matrix multiplication when broadcasts are executed by rows or columns of processes, the 3D logical grid of dimension $\sqrt{P/h} \times \sqrt{P/h} \times h$ ranks incurs fewer data movements than the 2D logical grid of $\sqrt{P} \times \sqrt{P}$ ranks. The second benefit comes with an increased parallelism by assigning more ranks to the computation of `trsm`. This increase in parallelism shortens the critical path. In their algorithm, Solomonik and Demmel chose to replicate the computation of diagonal block factorization over layers *i.e.* over an aisle of h ranks to be able to distribute their `trsm`s over columns of ranks across layers. By doing so, they are also able to bypass some steps required to broadcast the factorized, assembled diagonal block: each h rank is responsible to broadcast the replicated block inside its layer. The authors therefore propose a hierarchical cyclic distribution of the matrix A into "big blocks" owned by ranks in the same layer and "small blocks" owned

Algorithm 12: Right-looking 3D Cholesky factorization of a matrix A by blocks for a rank (r, c, h).

```
1  for l = 1...m do
2      if l%q = c then
3          if l%p = r then
4              all_reduce(A_ll,across:(r,c,:))
5              call potrf (A_ll)
6          bcast(A_ll,to:(:,c,h))
7      forall i = l+1...m such that i%p = r do
8          if l%q = c then
9              reduce(A_il,across:(r,c,:))
10             H ← layer of big block storing A_il
11             if h=H then call trsm (A_ll,A_il)
12             bcast(A_il,to:(r,:,h))
13             bcast(A_il,to:(:,c,h))
14     H ← l % h
15     if h=H then
16         forall i = l+1...m such that i%p = r do
17             if i%q = c then
18                 call syrk (A_il,A_ij)
19             forall j = i+1...m such that j%q = c do
20                 if i%q = c then
21                     call gemm (A_il,A_jl,A_ij)
```

by a single rank. Their mapping of tasks is depicted in the left of Figure 2.14. The initial interest of the authors was LU factorization: their work has been extended to Cholesky Factorization through UPC [60]. The case of sparse Cholesky factorization has also been considered [77].

A pseudocode for the 3D POTRF of Solomonik and Demmel is proposed in Algorithm 12. Note that this pseudocode leaves room to choose the distribution of the matrix A. It assumes that the diagonal block is factorized over multiple ranks. The symmetry is not strictly taken into account in this pseudocode as results of `trsm` are systematically broadcasts to $(r, :, h)$ and $(:, c, h)$. For the sake of simplicity the edge cases occurring when a block-row of block-column is stored over a single communicator are discarded and two broadcasts are submitted.

In SUMMA algorithms, all the outer products can be readily mapped to any layer available in the 3D grid. This property comes from the embarrassingly parallel nature of the matrix-matrix multiplication. For factorization algorithms the dependencies are much stronger and hinder parallelism: the updates in the trailing submatrix of a given step depend on the terminal tasks (factor,solve) of this step. Therefore update tasks are accumulated iteratively across layers. To increase parallelism, Kwasniewski et al. propose to break down the rank-b update operation in h updates of rank b/h [82]. The resulting

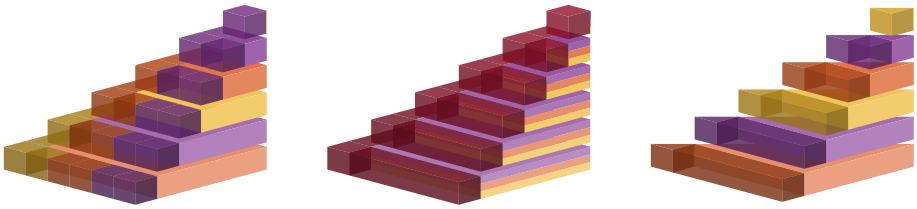


Figure 2.14: State-of-the-art factorization algorithms map tasks onto processes differently. Three layers are represented with three colors (purple, orange, yellow). **Left:** Solomonik and Demmel assign updates in a round-robin fashion over layers and distribute the matrix hierarchically. **Center:** Kwasniewski et al. partition updates over layers so all layers are involved at each iteration. **Right:** Beaumont et al. assign each iteration, both factorization and update, to a layer.

mapping of tasks from the algorithm they propose is depicted in the center of Figure 2.14. They essentially ensure that all layers are taking part in the bulk of computation – the update phase – as early as possible in the operation. By doing so, assuming a strictly synchronous setting, fewer ranks are left idle. Note that in the material of their experiments, Kwasniewski et al. rely on 2 layers at most.

2.3.3 Summary

This section has presented some dense linear algebra routines that are pervasive across scientific computing. Software libraries implementing such routines target large-scale machines where network communications are a major bottleneck: researchers have detailed "communication-avoiding" variants of these operations. Such research has led to a plethora of algorithms that are not systematically ported to modern software packages due to their relative complexity and tough maintenance. In the following section we detail some of the efforts of the scientific community to provide a scalable, portable and productive software stack over large-scale machines.

2.4 Context of the thesis and related efforts

The use of task-based programming models and runtime systems for high-performance mathematical libraries has received considerable interest in recent times; numerous research projects have been developed around this subject and the related literature is very rich and broad. An extensive review of all the related efforts is out of the scope of this document but we will briefly mention some of them.

This PhD is part of (and funded by) the ANR (the French national research agency) SOLvers for Heterogeneous Architectures over Runtime systems, In-

vestigating Scalability (SOLHARIS) project whose objective is to investigate the use of task-based parallelism and runtime systems to develop efficient and scalable mathematical libraries and, more precisely, dense and sparse linear solvers for distributed memory, heterogeneous parallel supercomputers. This project tackles this objective by gathering experts of linear algebra algorithms, of runtime systems and of scheduling. In the context of this project, ongoing efforts around Maxime Gonthier's PhD, aim at developing scheduling policies that incorporate memory constraints to better organize the data transfers over GPUs [62, 63]. Gwénolé Lucas' PhD, instead, focuses on developing an extension of the STF programming model to achieve a better management of tasks granularity through a refinement of the partitioning mechanism that provides hierarchical tasks [56]. Philippe Swartvagher's PhD [106] was devoted to enhancing the interfacing of the newMadeleine communication library with StarPU to increase the reach of newMadeleine features StarPU can take advantage of [52], notably, dynamic collectives which we will use in our work (see Chapter 3).

Related efforts include the NLAFET – Numerical Linear Algebra For Extreme scale systems – project funded by the European Union and ECP – Exascale Computing Project – funded by the American Department of Energy. These endeavors bring very large communities of experts from entirely different fields: separation of concerns is of great interest to fulfill their objectives. We refer the reader to reports from the two projects for more details [2][3].

Many dense and sparse linear algebra software packages have been produced that rely on the use of task-based parallelism and runtime systems. One of the earliest efforts is represented by the PLASMA [14] software package that provided parallel dense linear algebra routines developed using OpenMP tasking (formerly QUARK); as such, this package was designed to run solely on shared-memory parallel computers. The Chameleon library, mentioned above, also provides dense linear algebra methods and achieves parallelism through the use of different task-based runtime systems such as QUARK, OpenMP, StarPU and PaRSEC; consequently it can use a much broader range of architectures including distributed memory, heterogeneous computers. The PaStiX [68] software package implements a sparse direct solver based on the LU and Cholesky factorizations on top of the StarPU and PaRSEC runtime systems; it supports shared-memory systems with GPUs and support for distributed-memory computers has been achieved for full-rank – a low-rank feature being under development. The SPRAL package provides solvers and algorithms that are able to target GPU architecture through OpenMP offloading. Notably it includes SSIDS that target symmetric indefinite problems for shared-memory accelerated platforms; it delivers a multifrontal solver compatible with Nvidia GPUs that ensures numerical stability through threshold

[2]ECP reports `https://www.exascaleproject.org/reports/`
[3]NLAFET working notes `https://www.nlafet.eu/working-notes/`

partial pivoting [71]. The `qr_mumps` package implements a sparse direct solver based on the QR and Cholesky factorizations on top of the StarPU runtime; it currently supports shared-memory systems equipped with GPUs and support for distributed-memory computers is ongoing. Both these solvers were considerably improved and extended in the context of the SOLHARIS project and its predecessor, the SOLHAR project. Furthermore, all the solutions proposed in this PhD have been implemented and experimentally validated within the `qr_mumps` software package. While `qr_mumps` focuses on sparse linear algebra it also delivers some dense linear algebra routines that can be extended for a distributed-memory environment.

2.5 Concluding remarks

The scientific computing ecosystem has been getting visibly more complex at each technological breakthrough to achieve ever-increasing computing power. This escalation in complexity at a myriad of levels in the computer architecture has called for abstractions in programming models that have taken different forms – what is generally referred to as "runtime systems". The goal of an algorithm designer is then to write their algorithms using the programming model's interface offered by these runtime systems, adapting to the level of abstraction they provide. General task-based runtime systems have gained a lot of traction in the numerical linear algebra community as their relatively high level of abstraction allowed algorithm designers to focus on numerical aspects rather than hardware features or programming productivity. The generality of runtime systems is key to leverage off-centered features such as scheduling, memory management – including optimized transfers or checkpointing.

While the interest of task-based runtime systems and their associated programming models has been ever-clearer in the shared-memory setting, its use for petascale and exascale supercomputers remains questioned as novel scalable algorithms, exhibiting clever communication patterns and data management, are mostly confined to standalone libraries with home-brewed, domain-specific abstraction layers. A risk for the algorithm designer is that their clever techniques are only available when they have total control over their application and they could not delegate these techniques to a runtime.

Part I

Scalability of the STF programming model

Chapter 3

Scalable STF matrix multiply

State-of-the-art scalable algorithms sometimes favor MPI+X instead of general-purpose runtime systems such as PaRSEC or StarPU. This makes the implementation provided by their authors not (easily) portable to most architectures. As we will detail in this chapter, this lack of adoption comes from a lack of features in productive programming models such as STF. The following sections aim at ensuring the STF programming model is better equipped to face the requirement of distributed-memory systems. To this end, STF is extended by adding general-purpose features that are found in matrix-matrix multiplications.

We first detail in Section 3.1 how the STF model can be leveraged as it exists to express pGEMM, an omnipresent operation in dense linear algebra. This STF programming model is labeled "baseline". Section 3.2 details the key features that are required to improve the STF model such that scalable variants of pGEMM can be introduced directly. This improved STF model extends from the baseline one. The actual specification is detailed in Section 3.3 and implementation details that are addressed both in the runtime and in our software package are presented in Section 3.4. We validate our approach in Section 3.5 by presenting strong scalability results including state-of-the-art libraries that implement some variants of pGEMM.

3.1 Baseline STF model

The GEMM operation presented in the previous chapter in Algorithm 7 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

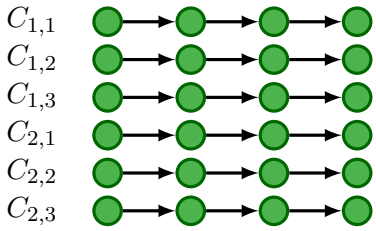


Figure 3.1: DAG with $m = 2, n = 3, k = 4$; green circles representing `gemm` tasks; each chain of tasks in the DAG corresponds to contributions to a single block of C.

runtime system. The expression of the algorithm is straightforward: for each block in the matrix C, k tasks are inserted with each task reading blocks of A and B and reading and writing the block of C. Figure 3.1 shows, as an example, the DAG corresponding to the case $m = 2,\ n = 3,\ k = 4$.

Algorithm 13: Parallel GEMM using the baseline STF model.

```
1 for i = 1...m do
2 |   for j = 1...n do
3 |   |   for l = 1...k do
4 |   |   |   insert_task (gemm,A_il:R,B_lj :R, C_ij:RW)
```

Thanks to the very high arithmetic intensity of the GEMM operation, the code of Algorithm 13 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. 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 [6], will be perfectly functional, its performance and scalability can be poor compared with what can be achieved with the algorithms described in section 2.3.2.1 for a number of reasons.

First, 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_{ij}, in the case of Figure 13). This can

lead to a very large volume of communications, for example, in the case where A and/or 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 $C_{ij} = A_{il} \cdot B_{lj} + C_{ij}$ 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_{ij} block, the task that computes $C_{ij} = A_{i,l+1} \cdot B_{l+1,j} + C_{ij}$ depends on the one computing $C_{ij} = A_{il} \cdot B_{lj} + C_{ij}$.

3.2 Proposed extensions to the STF model

The limitations discussed in the previous section make the baseline STF model [6] unsuitable for implementing state-of-the-art algorithms for distributed-memory systems such as those presented in section 2.3.2.1. 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 Chameleon dense linear algebra library uses this approach by submitting explicit copy tasks. In order to achieve collective communication patterns, the implementation provided by this library dedicates references to copies of blocks of A and B. Most notably, the implementation covers only the stationary-C variant and no stationary-A or -B variants are implemented following the same approach which indicates that this is not productive.

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 an 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 3.4.1 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 all the tasks that compute a $C_{ij} = A_{il} \cdot B_{lj} + C_{ij}$ contribution are inserted with `RW` (read-write) access mode on the C_{ij} 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 $D_{ij}^f,\ f = 1, \ldots, z$; the runtime system takes care of creating all the z temporary blocks and combining their content through dedicated tasks transparently. This reduction phase is carried asynchronously, the only constraint being that it must be completed prior to any other access to the C_{ij} 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_{ij} should be made commutable. It must be noted that a crucial design choice concerns the number z of temporary copies D_{ij}^f for each C_{ij} block: this is discussed in section 3.4.1.

Dynamic collective communications In order to take advantage of efficient and scalable collective communications, whose role is essential in the algorithms presented in section 2.3.2.1, we rely on the so-called *dynamic collective communications* feature proposed by Denis et al. [52]. 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 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/B 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 2.3.2.1 *i.e.* all of the 2D/3D stationary-$A/B/C$ variants can be conveniently implemented as in the pseudocode of Algorithm 14. Depending on the stationary variant defined by the `stat` variable and the number of grid levels defined by the `s` variable, the `task_map` function return the rank where a (i, j, l) task must be executed and the access mode on the C_{ij} block for this task. This function is described in more details in the next sections and in Table 3.1. It must be noted that the function provides the same result 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 pseudocode of Algorithm 14 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 chapter is that the proposed extended STF model allows one to express the three advanced GEMM algorithms (and communication patterns) described in section 2.3.2.1 with this extremely compact and simple code (Algorithm 14), together with an appropriate choice for the mapping and access modes. In Table 3.1, 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. The remaining variants are essentially similar and are not discussed further. Remarkably, other mapping policies could be proposed by algorithm designers: the STF code we propose is fundamentally unaffected by such newer designs. The underlying GEMM operation will be correctly computed while benefitting from newer tasks mappings.

Algorithm 14: Parallel GEMM using the improved STF model. The outputs of function `map` and `am` can be found in Table 3.1.

```
1 for i = 1...m do
2     for j = 1...n do
3         for j = 1...k do
4             rank, am = task_map(i,j,l,stat,s)
5             insert_task (gemm, A_il:R, B_lj:R, C_ij:am, rank:ON_RANK)
```

Stationary-C SUMMA In the stationary-C SUMMA algorithm the rank owning the C_{ij} block is in charge of all the $C_{ij} = A_{il} \cdot B_{lj} + C_{ij}$ tasks for $l = 1, \ldots, k$. Therefore, assuming a 2D block-cyclic distribution on a $p \times q$ grid, the `rank` output of the `task_map` function will be $(i\%p, j\%q)$. As for the

Table 3.1: 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 `task_map` method in Algorithm 14.

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

access mode `am` of the C_{ij} block returned by the function, 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_{ij} 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. This access mode has been available in StarPU 1.3 alongside the `REDUX` access mode. The dynamic collective communications feature will transparently group together all the transfers of a A_{il} (B_{lj}) 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 pseudocode in Algorithm 8 (p.35).

Stationary-A SUMMA In the stationary-A SUMMA variant, the rank in charge of computing $C_{ij} = A_{il} \cdot B_{lj} + C_{ij}$ is the one that owns the A_{il} block; therefore, the `rank` returned by the `task_map` function is $(i\%p, l\%q)$. As for the access mode for the C_{ij} block, this has to be `RANK_REDUX` to achieve the reduction on line 9 of the pseudocode in Algorithm 9 (p.36). The dynamic collective communications feature will detect that the B_{lj} 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 pseudocode in Algorithm 9. It must be noted that in MPI-based implementations, the broadcasting of B_{lj} 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 Algorithm 9 is not necessary.

3D Stationary-C SUMMA In the 3D stationary-C SUMMA variant, the rank selected for computing the contribution $A_{il} \cdot B_{lj}$ to the C_{ij} block is $(i\%p, j\%q, l/(k/s))$. The access mode to the C_{ij} block is `RANK_REDUX` to operate the reductions in line 17 of the pseudocode in Algorithm 10 (p.37). The

broadcasts in lines 9 and 11 of Algorithm 10, 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 6. Similarly to what is 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.

3.4 Implementation

In this section we discuss the practical implementation of the pseudocode of Algorithm 14 which we achieved using the StarPU runtime system and its STF programming API within the qr_mumps [12] library.

3.4.1 STF advanced features

The proposed implementation of Algorithm 14 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. for a thorough discussion of this feature [52]. This mechanism has been released in StarPU 1.4.0 and is implemented 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 [58, 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 at the shared memory level. 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 (resp. 3D stationary-C) variant, for each C_{ij} block, there can be as many copies as q (resp. s) times the number of workers per rank. At the

distributed memory level the user also has to call a method `redux_submit` to let the runtime system insert the subgraph related to the reduction. Another shortcoming of the available StarPU implementation of the `REDUX` access mode is that the reduction step is performed sequentially, that is, all the copies are assembled into the original one sequentially, one after the other. The overall 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; 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; we also incorporated a mechanism to automatically submit the subgraph related to the reduction part so that the user does not need to explicitly call `redux_submit` – they can still do it to bypass the default tree shape configuration. In all our experiments (see section 3.5) a binary tree was used.

3.4.2 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 2.3.2.1. 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 work deals with implementing all the discussed SUMMA variants and is not concerned with how 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 pseudocode of Algorithm 14.

First the scaling of the output matrix C by β is presented; second, the eventual transpositions and re-alignement of input matrices A and B are presented.

3.4.2.1 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 better efficiency and parallelism. Let's assume $k = 2$ in the pseudocode of

Algorithm 14; this implies that each C_{ij} block is concerned by the two tasks

$$\begin{aligned} \texttt{task1}: \ & C_{ij} = \alpha A_{i1} B_{1j} + \beta C_{ij} \\ \texttt{task2}: \ & C_{ij} = \alpha A_{i2} B_{12} + C_{ij} \end{aligned}$$

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:

$$\begin{aligned} \texttt{task1}: \ & C_{ij} = \beta C_{ij} \\ \texttt{task2}: \ & C_{ij} = \alpha A_{i1} B_{1j} + C_{ij} \\ \texttt{task3}: \ & C_{ij} = \alpha A_{i2} B_{12} + C_{ij} \end{aligned}$$

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.

3.4.2.2 Transposition and alignment

In MPI the scope of a collective communication is defined by a subcommunicator. 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.

3.4.3 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 [12]), 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 sys-

tems. Nevertheless, with some care in the programming and some appropriate techniques that we present in this section, it is possible to overcome this limitation even for very large workloads and computers. The experimental results shown in the next section have benefitted from such techniques.

3.4.3.1 DAG pruning

Our implementation employs a DAG pruning technique [6] to prevent each rank from creating all the tasks in the DAG as in the basic concurrent unrolling [114] (see the discussion in section 2.2.3). 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 that 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. As for the other variants, the same rule can be applied but, additionally, a node involved in a reduction must insert all the tasks that participate in it; this only implies a moderate increase in the local DAG generated on each node. It must be noted that the above pruning rules are generic and can be systematically applied to any algorithm regardless of its complexity. The effectiveness of the pruning obviously depends on how well-balanced is the distribution of data and workload in the implemented algorithm.

3.4.3.2 Efficient submission of tasks

It must be noted that all the possible nesting orders for the loops of Algorithm 14 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 3.5.2.3). 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.

3.5 Experiments

In this section we report experimental results that aim at assessing the effectiveness of the proposed approach. pGEMM 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. 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 [69]. An exhaustive comparison with other software packages is out of the scope of this manuscript. 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 [1]: this library has been developed using the MPI+X approach but relies on a carefully engineered abstraction layer that uses modern features of the C++ language. Codes for all the 2D stationary variants are implemented.

- Slate [2]: 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 [3]: 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

[1] commit 4abe4ef0 (June 24th 2022) from `https://github.com/LLNL/Elemental`

[2] commit bb597ae4 (June 2022) from `https://bitbucket.org/icl/slate`

[3] commit 9825fbf1 (June 2022) from `https://gitlab.inria.fr/solverstack/chameleon`

into temporary matrices. It uses a lookahead mechanism to overlap computations and communications. The runtime system chosen for our experiments is StarPU.

3.5.1 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, Platforms B and A.

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 [4] with the NewMadeleine communication backend [5] which, in the case of qr_mumps, provides support for the dynamic collective communications.

In order to ensure the fairness of the experimental comparison, we have tuned several parameters as explained below. Due to the very high number of experiments we have conducted, it was not possible to fully optimize all of these parameters simultaneously but our choices ensure that none of the tested packages was severely advantaged or penalized with respect to others and that, for all of them, performance was close to the optimum; this allows us to draw conclusions from the experiments below with good confidence.

We have tuned the number of ranks per node and of threads per rank making sure that all the available cores were used and that all the ranks and threads were correctly placed on the resources. For Slate, Chameleon and qr_mumps experiments were run with one rank per node using as many threads/workers as the available cores. In the case of ScaLAPACK multiple ranks (MPI processes) per node were used (24 and 64 for Skylake and Rome, 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).

We have chosen to experiment with multiple block sizes in all the packages; this parameter can be tuned 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.

[4] commit 0afdaeb09 (February 2021) from `https://gitlab.inria.fr/starpu/starpu`
[5] commit fdec689ab (December 2021) from `https://gitlab.inria.fr/pm2/pm2`

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. Because the operation timings are measured between `MPI_Barrier`-like functions, preceding tasks such as tasks writing the initial values in the matrices have already been processed and cannot be accounted for by the dynamic collective detection mechanism.

All the experiments are run using double precision real data; in the case of complex matrices we expect to achieve better scalability because of the higher arithmetic intensity.

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.

3.5.2 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 range of sizes was chosen so as to evaluate both strong and weak scalability and to evaluate performance on small as well as large matrices which might be of interest for different classes of applications. 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.

3.5.2.1 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 pGEMM 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 both problem types, we can observe in Figures 3.2 and 3.3 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.

For square matrices, we compared our implementations of the 2D and 3D stat-C variants. 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 [102, 60, 49] 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. 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.

3.5.2.2 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 pGEMM 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 3.4 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.

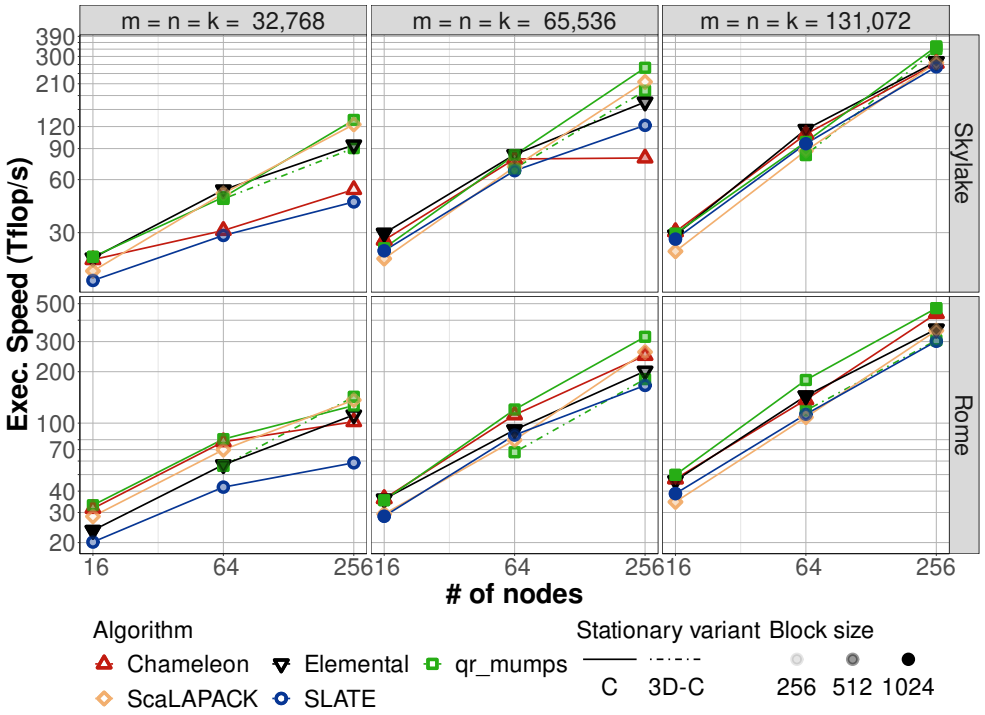


Figure 3.2: Comparisons of ScaLAPACK, Elemental, Slate, Chameleon and qr_mumps pGEMM on square matrices ($m = n = k$) and an increasing number of nodes. Markers correspond to the best median across runs using a given blocking for an algorithm.

3.5.2.3 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 communications 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

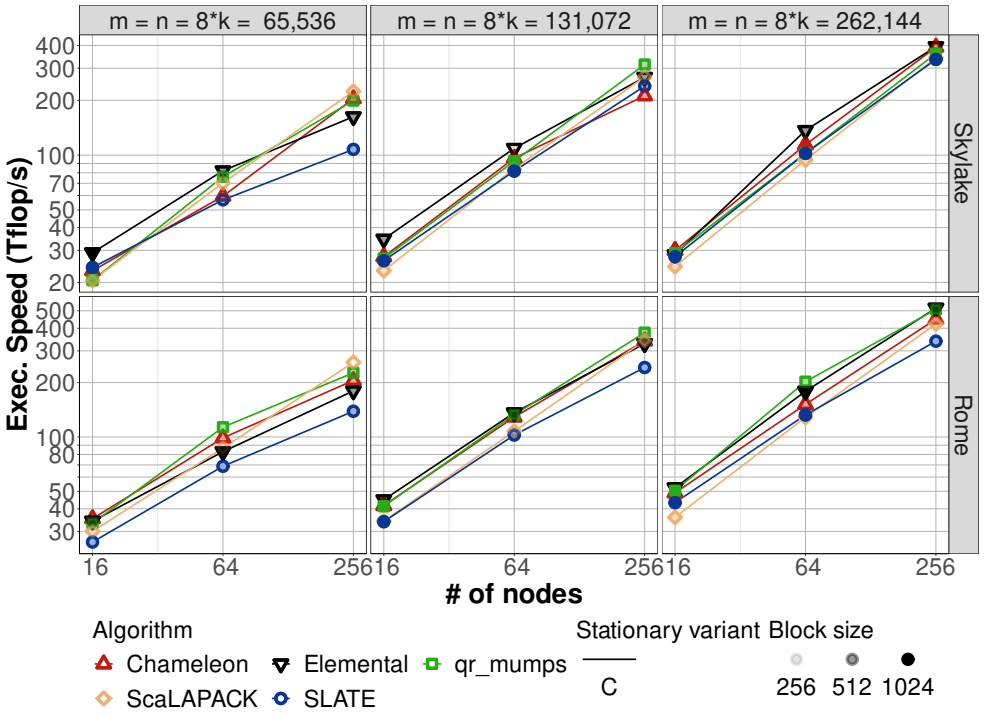


Figure 3.3: Comparisons of ScaLAPACK, Elemental, Slate, Chameleon and qr_mumps pGEMM involving a large C matrix ($m = n = 8k$) on an increasing number of nodes. Markers correspond to the best median across runs using a given blocking for an algorithm.

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. [6]). 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 3.5a and Figure 3.5b. 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

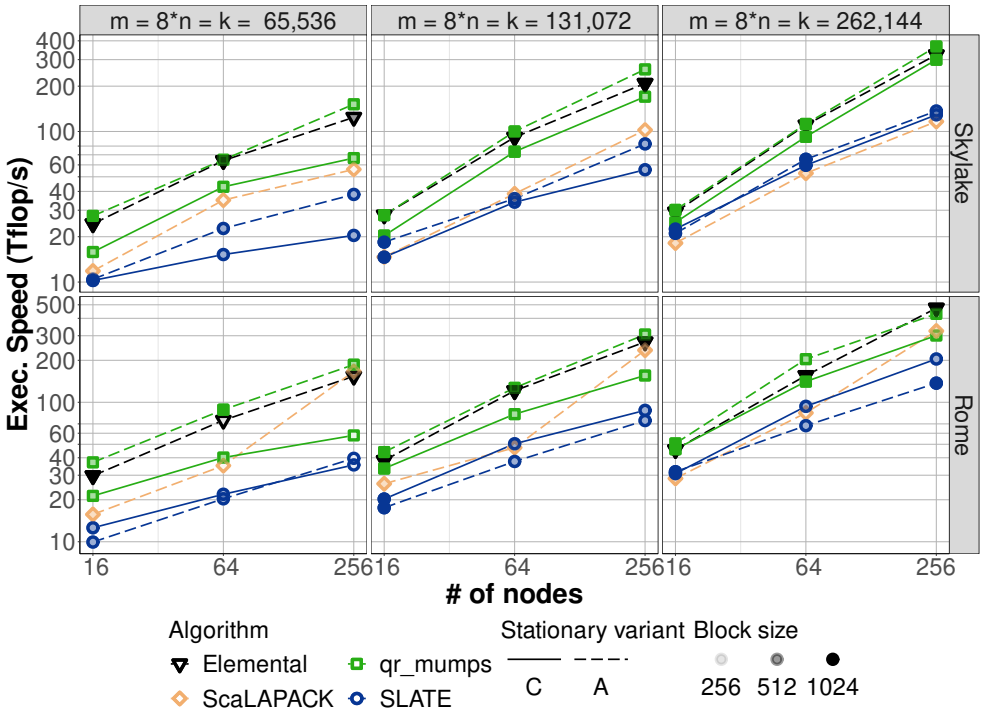
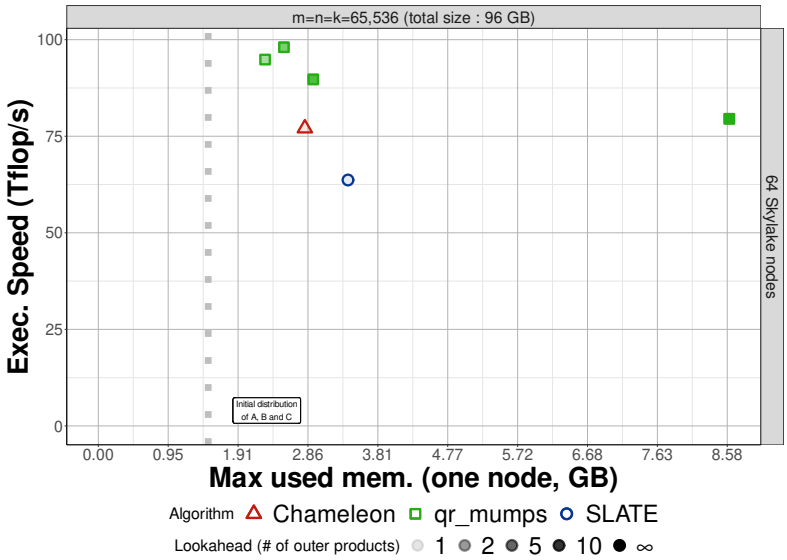


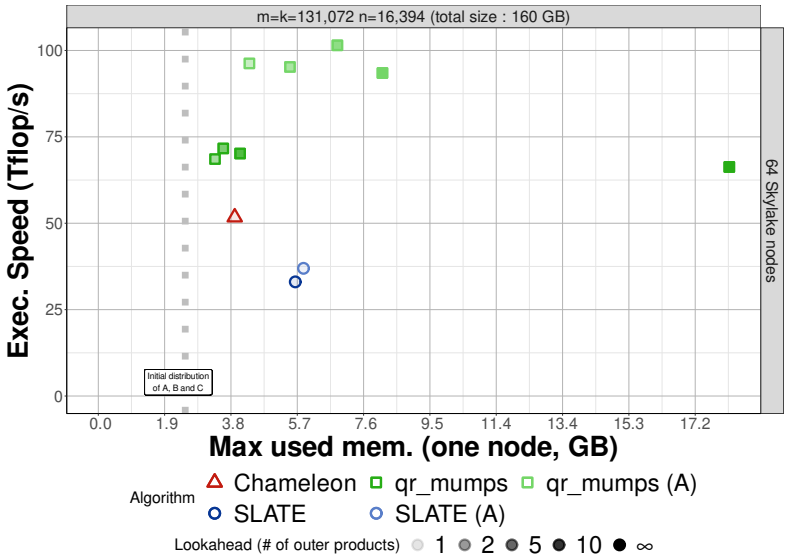
Figure 3.4: Comparisons of ScaLAPACK, Elemental, Slate and qr_mumps pGEMM involving a large A matrix ($m = 8n = k$) on an increasing number of nodes. Markers correspond to the best median across runs using a given blocking for an algorithm.

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 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 [12]. We reserve the implementation and study of such a feature for future work.



(a) Matrices are square of size 65,536 and a blocking of size 512.



(b) A is square of size 131,072 and C has 16,384 columns with a blocking of size 512.

Figure 3.5: Memory consumptions of Slate, Chameleon and qr_mumps pGEMM on different matrices using 64 nodes. qr_mumps uses different tasks window mechanisms.

3.6 Concluding remarks

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 scalability 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 the software ecosystem is ripe enough to ensure that the resulting code may be competitive against state-of-the-art, finely hand-tuned libraries.

Chapter 4

Replicating data write

In a task-based programming model such as STF, modifications on a piece of data are serialized. This means that two distinct workers managed by a runtime system will not execute concurrently two tasks modifying the same memory addresses. While this principle is important, we have seen how it can be beneficial to relax it in the case of reduction operations. In such a case, multiple copies of the piece of data coexist and while sequential consistency is ensured for each individual copy, out-of-order execution will produce the same final result. Reductions are only one such operation where a strict sequential consistency could be relaxed momentarily. This chapter focuses on an approach that relaxes sequential consistency to achieve **data write replication**; this corresponds to the case where multiple tasks, potentially on different computing nodes, redundantly compute over the same data. As we will explain in the following section, this feature can be used to reduce the communications at the cost of a slight increase (often negligible) in the operational complexity.

This chapter is outlined as follows: first, we motivate the interest of providing an interface to enable data replication in Section 4.1. Section 4.2 deals with the actual specification of the related novel interface – an additional access mode `SAME`– as well as a demonstration of its use in a stencil application. The following section, Section 4.3, is concerned with implementing an allreduce collective communication; this operation leads to the situation where multiple ranks own copies of the same data which can be concurrently updated through the data write replication feature. This mechanism, implemented through the extensions presented in this chapter, is actually leveraged in factorization algorithms.

4.1 Motivation

In a reduction, one rank starts with a piece of data it owns and other ranks start with a copy initialized according to the program's specifications. Each rank partially contributes to its copy of the data that it will assemble. The assembled result is stored by the owner of the data. Such a pattern, rooted *toward* the owner of the piece of data, is essential in several algorithms including the ones found in linear algebra computations. It is a core component of the extended STF programming model in Chapter 3.

Another feature of interest is the replication of data output or data modification. By replicating data modification, the same handle is modified on different distributed memories. We refer to this behavior as **data write replication**. The key component of the mechanism we present is the concurrent modification of data. Because this mechanism requires the initial value of the modified data to be exchanged, the corresponding DAG is rooted *from* the owner of the data. As the tasks modifying the data use the same input value, replication has to break the consistency of the serial accesses over the data. This feature can be used to trade off communication with computation as presented in Figure 4.1. If the DAG remains sequentially consistent, the modification of a data A may only be executed in one place to produce A_0,A_1,*etc.*. If another rank requires these values of A, multiple transfers between ranks have to be executed, as depicted in the left part of the figure. By replicating modifications of A on the two ranks, those transfers can be discarded: it is enough to transfer the initial value of A and let the ranks proceed with the computation as depicted in the right part of the figure. If this alteration of the DAG is beneficial to network communication, it is also clear that the computational workload increases because some work has been duplicated. We will see in Section 4.2.3 how this trade-off can improve scalability in practice.

To the extent of our knowledge, one element that is not studied in the literature is data write replication over multiple ranks in a runtime system. Nonetheless related mechanisms exist especially considering resiliency mechanisms. Better guarantees on the resiliency can be obtained through checkpointing, where the result of tasks is reliably stored, but also replication, where multiple identical tasks are scheduled. Designing and implementing such behaviors within runtime systems is possible without a dramatic overhead and with little to no intervention from the user [42, 85, 65]. Our contribution diverges from resiliency concerns as data write replication can be used to shorten the critical path and increase performance.

In the current programming model, including the extension proposed in Chapter 3, the sequential consistency remains central. Any relaxation of the sequential consistency is clearly framed. For instance when dealing with reduction patterns, access modes are defined to produce partial contributions which are not supposed to be read nor used by an application in their partial state. When considering replicating, a similar temporary relaxation should

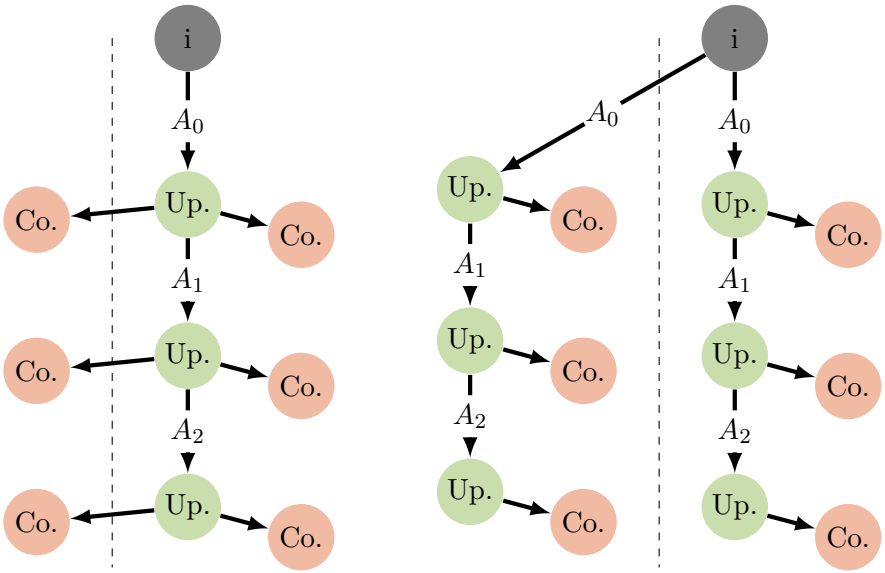


Figure 4.1: Replicating modifications of a piece of data may reduce the communication volume if sequential consistency is relaxed. Orange tasks reads the value of A. Green tasks modifies the value of A. **Left:** Strict sequential consistency is maintained. Transfers are issued after each update task. **Right:** Sequential consistency is relaxed. One transfer is issued.

be provided by the runtime system because the same piece of data cannot be modified in multiple places without risking invalidating previously-modified copies. Thus to enable the design of a replicating pattern (see Figure 4.1), a contract should be made between the user and the runtime system: the sequential consistency could be momentarily ignored in some clearly defined context.

4.2 Enabling redundant computation within STF

Data write replication as introduced in this chapter is found in algorithms presented in Section 2.3.2.3 and 4.2.3.2. In its extended state, presented in Chapter 3, the STF programming model lacks such replication that is a strong requirement in the expression of these algorithms. This section is dedicated to the specification of the related features.

We have detailed YarKhan's approach to replicate the traversal of any DAG submitted to an STF runtime system in Section 2.2.3. One can assume that such a DAG contains a task T that modifies a single piece of data D owned by a given rank P_{own}. We can further assume that the rank executing T is P_{exe} which is different from P_{own}. All the ranks have registered D as owned by P_{own}: a robust and simple way to maintain coherency for subsequent tasks is to transfer the new value of D back to its owner once T is executed. StarPU

plans these data transfers during task submission. In any case, because the value of D has been modified, possible copies that are stored by other ranks should be invalidated to make sure these ranks retrieve an up-to-date copy when executing tasks in the remaining portion of the DAG – especially if a distributed-memory cache mechanism has been implemented (as proposed by Agullo et al. [6]).

4.2.1 Obtaining local data over multiple ranks

The robust and systematic fallback to the owner contradicts the data write replication: two ranks cannot execute a task modifying the same data without redundant transfers between one another, possibly through the owner of the data. A user can bypass this behavior and implement replication by explicitly duplicating memory managed by the runtime. This fundamentally leads to tricking the runtime into thinking about the replicates and the original data as completely separate references: instead of modifying a data D owned by P_{own}, the executing rank P_{exe} can modify its own handle $D^{(P_{exe})}$ that appears entirely different to the runtime. While feasible, this approach may hurt readability as it provides an inelegant expression that makes it uncomfortable to write code productively. This approach might also lead to the creation of an overhead for the runtime related to the management of the additional handles.

Rather than this approach, we propose the following extension to the STF programming model: instead of inserting a task T using a piece of data D with the usual `WRITE` access mode, an additional access mode `SAME` is introduced. This access mode actually reduces to a flag over D that indicates that copies are produced across multiple ranks. The runtime is therefore not expected to transfer these copies back to the original data owner: the transfer is fundamentally bypassed. Such an access mode gives more manoeuvrability to the user as they get to adjust the behavior of the runtime system to their needs. Nonetheless, this approach puts more burden on the user regarding the correctness of the DAG as several copies now exist in the distributed memory whose consistency must be ensured. Note that, semantically, the `SAME` access mode allows inserting multiple tasks that produce, each, a copy of the same data. In the case where all these tasks are the same, i.e., use the same codelet, same data and same arguments, we can actually speak of *task replication*. This however does not necessarily have to be the case because in our approach nothing prevents these tasks to use a different codelet, different data (other than the one flagged with the `SAME` access mode) or different arguments. Obviously, in this case, it is the duty of the programmer to ensure that these tasks produce the same data, or, at least, copies that are equivalent for the purpose of the algorithm.

To facilitate the usage of this feature, a wrapper function can be implemented in the runtime system: this additional function, `insert_tasks`, takes

a list of replicating ranks `repl_ranks` along with the usual arguments (a codelet T, its own arguments, etc.). Note that `insert_tasks`, with an "s", differs from `insert_task`. This wrapper can be used to hide the usage of the novel access mode: all `Write` accesses are appended with `SAME` and the task T is submitted once for each rank in `repl_ranks` plus the owner if it was not included, transparently for the user. Its use is exemplified in Figure 4.2 for a piece of data A. It is important that the task executed by the owner of the original data is inserted last because each rank in `repl_ranks` should receive the initial, unmodified value of A. Consequently the owner of A should schedule sending the unmodified data before scheduling its own modification. Likewise the owner rank should insert all the tasks to ensure that every rank in `repl_ranks` get the unmodified value, but other ranks can prune the DAG.

```
insert_tasks(repl_ranks,T,A:RW)
```

```
repl_ranks ← unique({ repl_ranks, A.owner })
for rank in repl_ranks do
    insert_task (T, A:RW +SAME, rank:ON_RANK)
```

Figure 4.2: `insert_tasks` simplifies the use of the `SAME` access mode.

With the proposed additions to the programming model, the DAG introduced in the beginning of this chapter can be expressed in an STF code displayed in Algorithm 4.3. This algorithm shows how two very similar function calls in the STF programming model generate very different behaviors: while `insert_task` and `insert_tasks` have very similar arguments they lead to extremely dissimilar tradeoffs between computation and communication workloads. Given a completely submitted DAG the tradeoff of replicating tasks on several ranks may be evaluated by a runtime system; this evaluation could lead to an automatic data write replication. We have not evaluated the (engineering) cost of setting this detection mechanism up.

```
for it=... do
    insert_task (update,A:RW) or insert_tasks ({0,1},update,A:RW)
    for rank in {0,1} do
        insert_task (compute,A:R,rank:ON_RANK)
```

Figure 4.3: Either the blue or the red statement can be called. They result in different DAGs. The blue one corresponds to the left part of Figure 4.1 whereas the red one corresponds to the right part of the figure that leverages the proposed extensions of the STF model.

4.2.2 Transferring from alternative ranks

One straightforward advantage of using the `SAME` access mode on a data D is that any subsequent local task needing D will actually use the redundantly computed local copy rather than fetching D from its owner. However, in the case where a task needing D is executed on a rank which does not possess a local copy, any of the existing copies of D can be transferred to that rank prior to the execution of the task; a choice has to be made. A conservative policy is to source D from its owner. Although this approach is simple and very robust, it might not be the most efficient. First it makes the data output replication only useful in a limited number of cases where replicated data is only used locally. Second it may result in a loss of efficiency due to contention issues on the owner rank which is involved in numerous communications. As an example, consider the factorization algorithms detailed in Section 2.3.2.3. Here all the `trsm` tasks at a given iteration of the algorithm are spread over the existing layers of the 3D ranks grid; each of these tasks will fetch a copy of the diagonal block from a rank lying on the same layer. To address this situation, we developed the following method, which makes it possible to declare that a rank `rcv_rk` must retrieve a data `handle` from a designated rank `new_src` in subsequent tasks, rather than from its owner:

- `set_alternative_source(handle, rcv_rk, new_src)`

The default behavior of the runtime to fetch data from the owner rank is superseded: when `new_src` schedules data transfers for `handle` it is the defined alternative rank that is used. Because the source rank is set in-between task submission it is possible to benefit from other features already proposed by a runtime system such as dynamic multicast communications [52].

The `set_alternative_source` function may be called outside the replication mechanism. In such a case no valid copy of the data described by `handle` exists in `new_src`: a preliminary transfer has to be executed with the owner. When the alternative source `new_src` has been set, the cache mechanism can be updated to account for the valid copy it possesses. When replicating computation through `SAME`, such a cache entry is already set in the required state. Because `set_alternative_source` mostly consists in manipulating the records of the distributed-memory cache it has no practical overhead when used conjointly with `SAME`.

4.2.3 Stencil iterations over a 2D structured domain

In this section, we will be interested computing stencil iterations that will be detailed in the following paragraphs. The related algorithms provide an interesting framework to data write replication.

Some areas of scientific computing are interested in computing fields of *e.g.* temperature, velocity,*etc.* along a time dimension. It is then common

practice to discretize these fields into geometric domains to compute their values iteratively through time. The interactions between cells of these fields can often be described locally *i.e.* neighboring cells interact. If the interaction can be described through linear functions, then iterations are commonly described as a suite of sparse matrix-vector multiplications. The interaction of cell i over cell j can be stored in the (i, j) coefficient of this sparse matrix however some computations over meshes do not need to explicitly store all coefficients. In the case of partial differential equations, the next iteration is typically computed by means of an explicit stencil that is applied at each cell. Section 4.2.3.1 introduces some notation for the specific case of 2D stencils in distributed memory.

With the past and current trends in computer architectures, the scalability of stencil algorithms has been extensively studied both in a shared-memory and a distributed-memory setting. On parallel machines, it is necessary to distribute the input domain as well as the initial state of the system: because of this distribution, inputs need to be shared between parallel elements *e.g.* threads on a computing node or ranks on a cluster of computers. Then, when computing the next iteration over the domain, these parallel elements have to exchange their inputs to proceed further. When this exchange is performed at each iteration, this creates synchronization that hinders parallelism and degrades execution time. Section 4.2.3.2 presents some strategies and designs proposed in the literature to enhance the performance of such computation on modern machines.

The following sections detail STF implementations of a 2D stencil. Section 4.2.3.3 presents the implementation of a classical approach. A communication-avoiding approach leveraging data write replication is presented in Section 4.2.3.4. Both approaches are benchmarked in Section 4.2.3.5.

4.2.3.1 Problem notations

The geometric domain Ω storing the field of values can take different forms based on the underlying phenomenon that is simulated: in this work, only 2D structured meshes are considered with each cell having four neighbors that can be designated with cardinal orientations (north, south, west, east). The domain Ω of size $X \times Y$ includes boundary values that are constant across iterations *i.e.* $\Omega(1, :)$, $\Omega(X, :)$, $\Omega(:, 1)$ and $\Omega(:, Y)$ are not updated. We are interested in computing a simple 5-points stencil over the domain Ω: each cell is updated by computing a function of itself and its direct neighbors. In the remainder of this document, to address the distributed-memory setting, a subdomain Ω_r is considered as the aggregate of cells owned by rank r. We introduce the notation $\delta\Omega_r^{(d)}$ which is the union of cells at minimum distance 1 and maximum distance d of Ω_r. The distribution of subdomains is often defined according to a 2D Block layout as to minimize the length-surface ratio

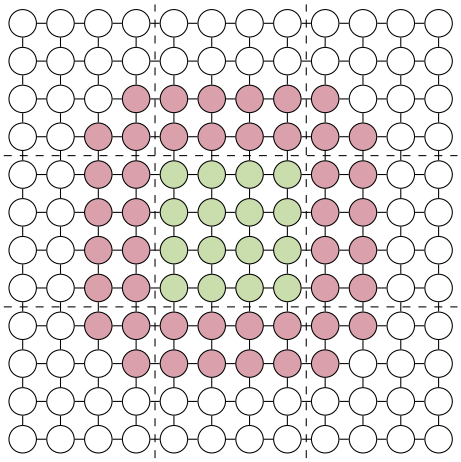


Figure 4.4: Cell 2D block layout. Green cells are owned by rank (1,1). Red cells are part of $\delta\Omega_{(1,1)}$, assuming $s = 2$. (1,1) retrieves red cells from neighboring ranks.

(similar to surface-volume ratio in the 3D case) of the cells owned by different ranks. A rank p may be designated by its coordinate (r, c).

4.2.3.2 Related works

Demmel et al.[50] provide two communication-avoiding parallel approaches to improve the scalability of the conventional procedure. These approaches reduce the number of messages exchanged by taking into account the size of the subdomain that is locally computable. In their Parallel Approach 1 (PA1) this amounts to exchanging cells owned by neighboring ranks that are up to distance s every s steps. We recall their PA1 algorithm for the 2D mesh case in Algorithm 15. With i iterations, the communication latency cost of the PA1 approach is in the range of $\mathcal{O}(i/s)$ [50]. The conventional approach fundamentally uses $s = 1$ and may be prone to synchronization issues. This decrease in latency by a factor of s has to be weighted against an increase in computational load as the communication-avoiding approaches replicate the execution of the stencil. $\delta\Omega_r$, without superscripts, refers to $\delta\Omega_r^{(s)}$. Assuming a rank r owns $x \times y$ cells, $\Omega \cup \delta\Omega_r$ contains $x + s \times y + s$ cells and $\delta\Omega_r$ alone contains $(x+2s)(y+2s) - xy - 4\frac{s(s-1)}{2} = 2s(x+y+1) + 2s^2$ cells. Figure 4.4 presents a layout of the cells where $(1, 1)$ owns 4×4 green cells and has to compute an extra 42 red cells, assuming $s = 2$.

Algorithm 15: PA1 executed from the process of rank $r = (r, c)$.

```
1 for it= 1, s + 1, ... do
2 |   recv δΩ_r from at most 8 ranks {(r+/–1, c+/–), (r+/–1, c), (r, c+/–1)}
3 |   for s_it= s − 1, s − 2, s − 2, , ... 0 do
4 |   |   update Ω_r ∪ δΩ_r^(s_it)
```

The PA1 approach has been studied repeatedly over the 2010s to better understand how it lifts the cost of synchronization, especially in the case of iterative solvers [70, 44]. Other techniques such as overlapping communication and computation have been studied [61]. The communication-avoiding algorithms have been ported to single-node accelerated platforms with success inside the Magma library to avoid data transfers to GPU devices [112]. A large portion of the literature has been focused on the productivity of the domain experts. In a shared-memory setting, compiler optimizations can be enough to abstract the reordering of vertical memory movement transparently to the programmer [22].

In a distributed-memory setting, the use of runtime systems has been extensively explored for the case of simpler algorithms that do not avoid communications [15, 67, 36, 5, 55]. Recent works have been geared toward enabling the communication-avoiding approaches in widely-available software stack such as Trilinos [113]. The use of PaRSEC has been explored to enable communication-avoiding techniques through its PTG interface: the work from Pei et al.[92] actually tackles similar concerns as the ones addressed in this chapter. The main difference resides in the approaches that are set up: the PTG interface has been used to implement the communication-avoiding algorithm while the work we present seek to modify the STF programming model with respect to this algorithm.

4.2.3.3 STF implementations of stencils

This section aims to present how the STF programming model can be leveraged to implement the conventional approach. The concerns (granularity, layout) raised in this section also applies to the communication-avoiding approach and have to be adapted.

One of the first concerns to address is the granularity of the tasks: it would be inefficient to submit a task per cell as it results in poor scaling over a large amount of computing resources. Indeed, updating a cell typically requires computing only a few flops and the flow of tasks from a sizeable domain Ω would create a large scheduling overhead. It is, instead, appropriate to aggregate cells into blocks, *e.g.* of $b \times b$ cells, such that each task updates all the cells in a block.

In order to simplify the implementation we will consider a static domain partitioning with blocks of variable size, illustrated in Figure 4.5. Blocks that are internal to a subdomain, i.e., blocks that can be updated without data owned by another subdomain, are of size $b \times b$. Instead, blocks that lie at the interface with another subdomain are smaller and have either two or four sides of size s. This partitioning makes it simple to retrieve all the ranks that replicate computation over a given block (it can be stored in a variable `neigh` for each block) and the retrieval of adjacent blocks as well as the computation of $\Omega_r \cup \delta\Omega_r$ both become trivial. Note that more sophisticated

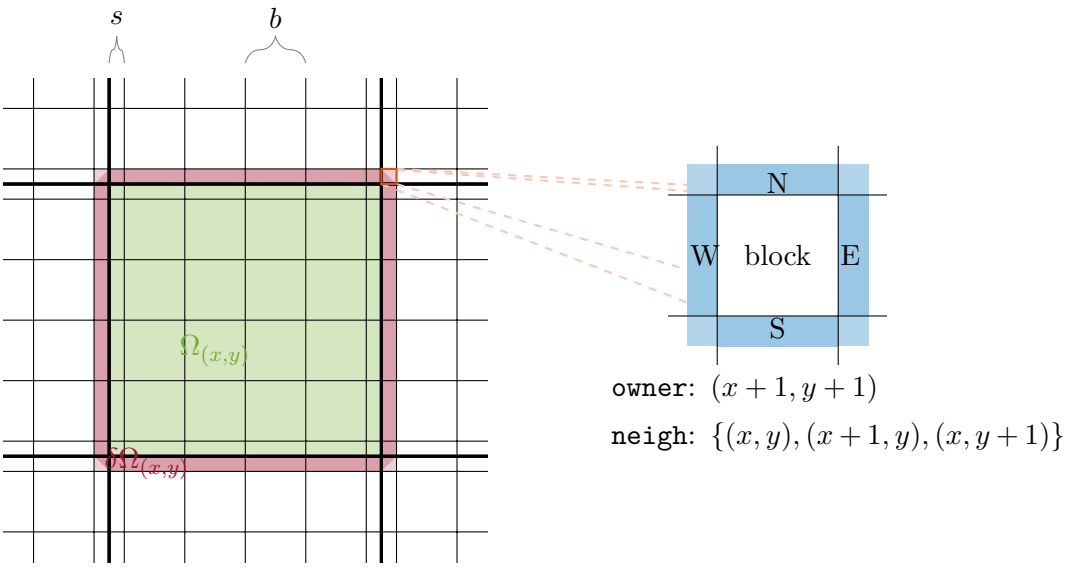


Figure 4.5: Layout for communication-avoiding 2D structured stencil algorithm. The subdomain owned by a rank (x, y) is highlighted in green. This rank will compute updates over its domain as well as the blocks highlighted in red. A block is zoomed-in to clarify what are its neighbors.

partitioning approaches may be employed using dynamic partitioning features provided by some runtime systems such as StarPU; we do not investigate such approaches because the sole objective of this experimental analysis is to validate the effectiveness of the data write replication feature.

In order to compute the next iterate over the mesh, storing Ω only once in a limited memory budget context makes the code more complex. Indeed, each block is updated with the values of its direct neighbors in the previous iteration. To concurrently compute a next iterate, these values should be available in-between steps. To simplify the implementation, we will consider that two grids exist and at each step one is read and the other is written to then they are swapped.

An STF implementation of the conventional approach, *i.e.* using $s = 1$ and not requiring replication of updates over cells, is presented in Algorithm 16. At each iteration the neighbor blocks at the interface of different ranks have to be exchanged. Every block b is updated using its neighbors indicated as $b.\{N, E, W, S\}$. This variant of the 2D 5-points stencil does not hold clever mechanism and the resulting code is relatively plain. In the next sections, the possibility to leverage replication, *i.e.* different values of s, is incorporated inside this expression.

Algorithm 16: Conventional STF implementation of a 5-points stencil.

```
1 for iter = 1 ... do
2     for block in Ω do
3         insert_task(update, block:RW,
4             block.N:R,
5             block.E:R,
6             block.W:R,
7             block.S:R)
```

4.2.3.4 Communication-Avoiding algorithm

The *communication-avoiding* variant of the classical stencil algorithm can be expressed using the STF programming model. This section focuses on the `PA1` algorithm (see Algorithm 15): we will describe how the baseline and the extended STF programming model can be used to implement it.

Using the baseline STF model results in Algorithm 17. This algorithm presents additional loops to set up the replication mechanism explicitly. These additional loops are used to manage copies of existing handles for each replicating ranks. Whether a computation over a block in Ω should be replicated by a rank r is verified by checking whether the block is also in $\Omega_r \cup \delta\Omega_r$ *i.e.* whether it is at distance up to s from a cell owned by r. This condition is easy to assess in the case of a 2D block layout. The handle that will be used in the `update` task by rank r can be denoted $B^{(r)}$, a replicate of B. In order to get the classical algorithm one can set $s = 1$ – in this case the explicit copies are bypassed.

Algorithm 17: 5-points stencil STF implementation without replication executed by rank r.

```
1  for it = 1, 2, ... do
2      if it % s = 1 then
3          for block in Ω_r ∪ δΩ_r do
4              for rank in block.neigh do
5                  insert_task (copy,block:R,block^(rank):RW)
6      for block in Ω_r ∪ δΩ_r do
7          local_block ← if r=block.owner then block else block^r
8          insert_task(update, local_block:RW,
9              block.N:R,
10             block.E:R,
11             block.W:R,
12             block.S:R)
```

It is possible to simplify the expression of Algorithm 17 by using the features presented in Section 4.2 that form an extended STF programming model. The resulting expression is provided in Algorithm 18. In this algorithm we do not use the `insert_tasks` wrapper function but rather submit each `update`

tasks individually using the `SAME` access mode because each replicating rank does not modify a given block similarly. For instance, assume a block B in Ω at the interface of two ranks r and t: if r owns the block then r naturally needs all of the neighboring blocks of B to update all the cells in it; however t needs only up to three of them. For instance t may not require $B.N$ because B is in the northern region of $\delta\Omega_t$. t would not need to receive $B.N$ because it does not need to update the top rows of B. The situation of the rank t is exemplified in Figure 4.6. Assuming $s = 3$ there are three cases to consider: first, the furthest row/column in $\delta\Omega_t$ is not updated; second, only the closest row/column in $\delta\Omega_t$ is updated; third, no cells in $\delta\Omega_t$ are updated. Thus, tasks modifying a given block are different across ranks: some should nullify handles that are unnecessary. Every s steps, each rank r should get the updated values in $\delta\Omega_r$: in order to retrieve these values in, it is enough to modify the access mode on blocks. Indeed, if the owner of a block uses a simple `RW` access mode every s steps – instead of `RW+Same` – then the runtime will invalidate the extra copies stored by the neighboring ranks. This will allow the scheduling of the transfers upon submitting the next iteration.

While an additional loop is added to iterate over ranks around any block, the expression allowed by replicating data write accesses is simplified compared to the one provided in Algorithm 17. The proposed implementation that uses the extension of the STF programming model for replication is more concise and does not require the user to manage data as precisely: they only have to be careful of the correctness of their algorithms by telling the runtime system when replication does make sense in their application.

Algorithm 18: 5-points stencil STF implementation **with replication** executed by rank r.

```
1  for it = 1, 2, ... do
2      for block in Ω ∪ δΩ_r do
3          for rank in {block.neigh,block.owner} do
4              if it % s = 0 & rank = block.owner then
5                  mode ← RW
6              else
7                  mode ← RW +SAME
8              insert_task( update, block:mode,
9                  block.N:R,
10                 block.E:R,
11                 block.W:R,
12                 block.S:R)
```

4.2.3.5 Experiments

In this section, we compare implementation of the 2D 5-points stencil communication-avoiding approach. This implementation was achieved using the StarPU

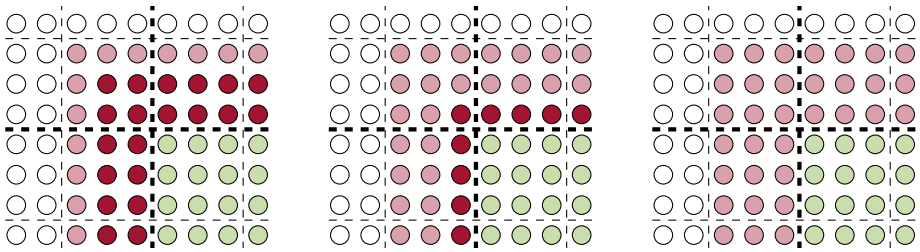


Figure 4.6: The green cells in Ω_t are owned by t and updated every step. When t updates blocks in $\delta\Omega_t$, not every cells in these blocks are updated ; only glowing red cells are. At each step, one row/column less is updated until, every s steps, $\delta\Omega_t$ is fetched from its owners with updated values. Assuming $s = 3$ and every s steps ; **Left:** the first iteration $s - 1 = 2$ rows/columns are updated. **Center:** the second iteration $s - 2 = 1$ rows/columns are updated. **Right:** the s-th iteration $s - s = 0$ no cells are updated.

runtime system. The proposed extension to enable data write replication has been added to the runtime in an experimental branch of its `git` repository[1]: the `SAME` access mode has been added and the cache mechanism adapted to support alternative sources. The experimental setup we use invokes one rank per computing node and one worker thread per computing core.

In our application, no actual computation is executed. In order to reflect the behavior of a real application, tasks sleep for a given time which is passed as an argument to the task. The value of this time is modified such that the interaction between the arithmetic intensity of the stencil operation and the scalability of the various DAG submission approaches can be assessed. The duration of a task is a linear function of the number of cells in the block that needs to be updated. Given a rank r and a block B at the i-th step using the communication-avoiding algorithm with parameter s, the computation is as follows

- B is owned by r then r updates all the cells in B.
- B is not owned by r and out of the adjacent blocks to B:
  - one is owned by r so r updates $b * i\%s$ cells.
  - none is owned by r so r updates $i\%s * i\%s$ cells.

To compute the speed of the operation, we assume that a single cell takes 9 Flop to be updated similarly to Pei et al. [92]. Thus, updating a block of size $m \times n$ takes $9mn$ Flop. In our benchmark, the time to process a cell is

[1]the code is available online at `https://gitlab.inria.fr/starpu/starpu/-/merge_requests/68`

denoted t_c and it is a tunable parameter of our experiments. The attainable peak performance by one core is denoted $p_c = {}^{9}/_{t_c}$. If i iterations over a mesh of size $X \times Y$ take t to complete, then the computation speed will be $p = i\frac{9XY}{t}$ and the utilization percentage of the machine, equipped with c cores, is $\frac{p}{cp_c}$. We use both Platforms A and B because their characteristics (number of cores, memory bandwidth, interconnect generation) offer a realistic comparison.

Figure 4.7 presents the volume of communication as the number of iteration increases for a mesh of size 1152×1152 using 8×8 processes with $b = 6$. This shows how the communication-avoiding variant behave at runtime compared to the conventional approach. From this figure, it is clear that the latency is more impactful on the conventional approach because it sends messages at each iteration. For the communication-avoiding variant, communication volume plateaus for s steps and between each plateau the gap is wider as s increases. We can observe a small discrepancy between all the variants as s increase. This comes from the fact that the exchanged subdomains $\delta\Omega$ are not trimmed at the edges: some useless cells are transferred because the blocks are rectangular despite the required regions being triangular. This overhead corresponds to $s^2/2$ cells sent up to 4 times each s iteration for each rank. We consider this gap to be negligible in our experiments but real applications might find it worth the effort to implement the packing and unpacking of arrays that describe the triangular blocks. This gap is further increased by the initial conditions that are aggregated into blocks as well: although they remain unchanged across iterations, they still have to be sent once at the first iteration.

Figure 4.8 presents performance results for 38 iterations in strong scalability. The problem size is fixed for all number of cores. The conventional approach is compared with the communication-avoiding approaches taking 3 values for the s parameter: 4, 16 and 38. Each value is only a small percentage of a block of size 1024. We have selected several t_c to test the robustness of approach with regard to this parameter. The selected values make it possible to go from a regime where tasks are fast and the machine is hardly utilized to a more efficient regime where tasks are relatively slow. These regimes can match actual computation in the literature as some interactions between cells can be very arithmetically intensive. We observe that the conventional approach is less scalable than the communication-avoiding one. With s sufficiently high (a small percentage of the block size) fewer messages are exchanged and the scalability is increased. We also observe that when a kernel is taking longer *i.e.* when its arithmetic intensity increases, as communications are fixed, then the achievable scalability is improved which is expected.

Figure 4.9 shows different values of the parameter s in a specific scenario using 36 Skylake computing nodes. In this scenario where the number of iterations is far larger than the size of the blocks, we can assess the importance of tuning s. We observe that up to $s = 32 = b/32$ it is beneficial to increment s. Past this threshold increasing s yields diminishing returns. This behavior

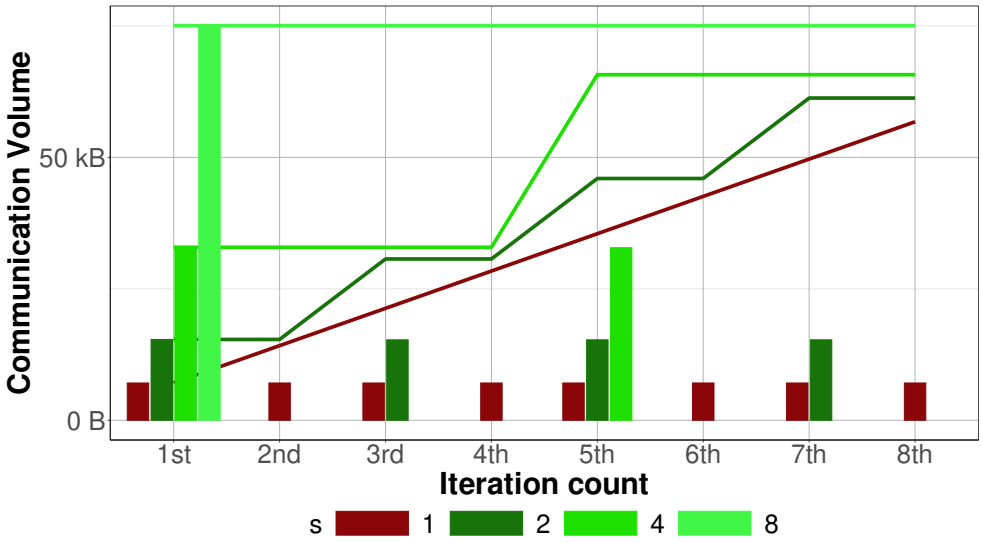


Figure 4.7: Communication volume along iterations evaluating conventional ($s = 1$) and communication-avoiding ($s > 1$) approaches for a 2D five-points stencil. Lines represent total communication volume. Bars represent communication volume at the i^{th} iteration.

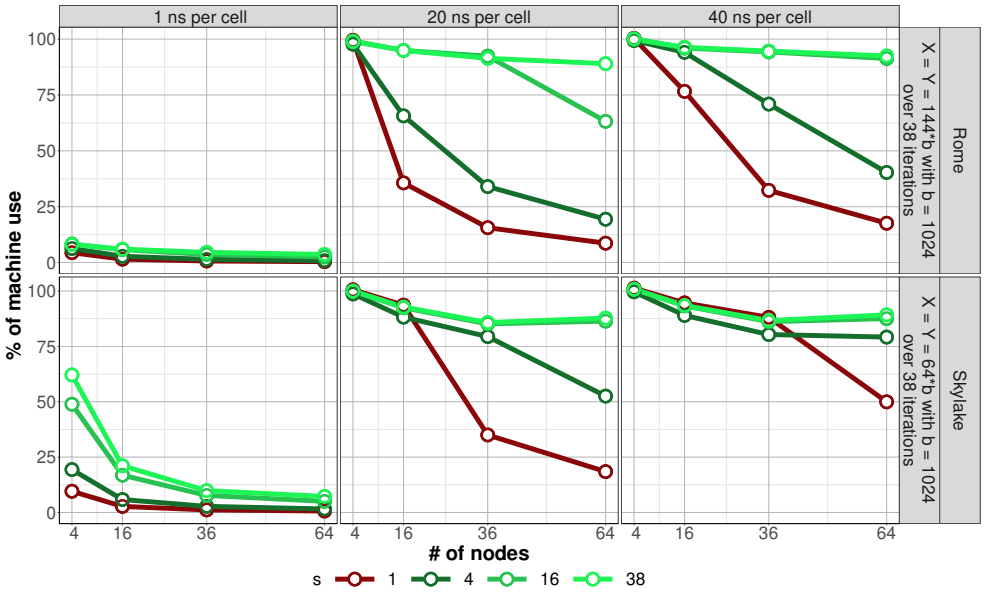


Figure 4.8: Strong scalability experiment evaluating classical and communication-avoiding approaches on a 2D five-points stencilfor stencil.

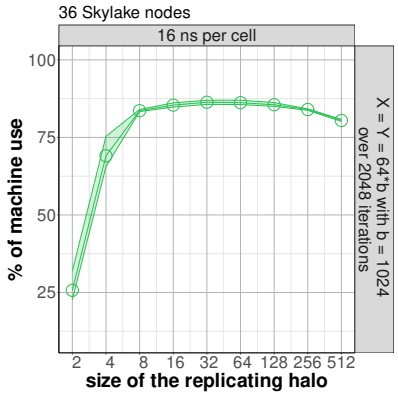


Figure 4.9: Different values of s for the communication-avoiding five-points stencil.

is expected since the amount of additional work that is replicated for a given rank is, not considering rank close to the borders of the domain, $(\frac{Y}{p}+2s)(\frac{X}{q}+2s)-\frac{XY}{pq} \sim 2s^2+2s(X/q+Y/p)$. With the latency improved by $\mathcal{O}(s)$ and the overall bandwidth unchanged the increased computational workload has to be responsible for some overhead when s is too large.

4.2.4 Summary

In this section, we have enabled data write replication in the STF programming model. The extension we provide amounts to an additional access mode `SAME` as well as a function to declare alternative sources to retrieve data. To assess the interest of this extension we have implemented a communication-avoiding stencil algorithm in which some cells are updated redundantly across ranks. Our extension makes it possible to avoid the pitfalls of the baseline STF model when implementing this algorithm: it makes task insertion less tedious by letting the user focus on algorithmic consideration rather than data management. We obtained compelling experimental results.

In the next section, the STF programming model is further extended. Our objective is to be able to address numerical linear algebra operations including factorization algorithms detailed in Section 2.3.2.3. Replication is one of the necessary tools to express these algorithms. The next section considers how to implement an allreduce reduction pattern and finally its use in implementing scalable Cholesky factorization.

4.3 Implementing an allreduce pattern

In factorization algorithms presented in Section 2.3.2.3, data is replicated as the result of an allreduce collective operation. In this operation, several contributing ranks share their partial contributions with one another until all

the ranks hold an assembled result. These assembled results are needed by several other ranks to proceed with their computation.

Instead of relying on an allreduce, the same result could be obtained by a simpler reduction followed by a broadcast. These features have been presented in Chapter 3 and are readily usable to implement scalable factorization algorithms using the STF mode. However the cost of a reduce followed by a broadcast is higher than that of an allreduce because this collective operation essentially executes several reductions in parallel. Once again, users can manually implement an allreduce operation by explicitly inserting the relative tasks. Rather than putting the burden of implementing their own allreduce on every single user, the runtime should provide such an operation. The runtime only needs to know how to reduce two contributions to submit the appropriate DAG: this addition to the interface of the STF can remain generic.

In the next section, we discuss the implementation of the allreduce operation inside a runtime providing the STF programming model. The following section leverages this feature as well as the extended STF programming model to implement scalable factorization algorithms.

4.3.1 Task-based allreduce STF DAG

The approach we propose to implement the allreduce collective operation is based on an extension of the reduce operation presented in Chapter 3. Let us assume that N ranks have, each, submitted a task updating some data D which is passed with the `RANK_REDUX` access mode. As explained in Chapter 3, the submission of tasks that operate the reduction can either be triggered implicitly, by submitting another task that requires D, or explicitly by calling a `redux_submit` function. This is illustrated in Figure 4.10 (*left*). Equivalently, we can define an `allredux_submit` method that, instead, inserts all the necessary tasks to operate an allreduce operation, in such a way that all participating ranks finally possess a copy of the assembled result.

A wide range of algorithms have been explored in the literature to address allreduce communication patterns [46, 94]: exploring the variety of patterns that can be implemented is not the topic of this work. These algorithms can be tailored to specific message sizes, network topology and number of contributing ranks. A naive yet reasonable approach is to implement a recursive doubling algorithm [81]; without loss of generality, we focus on this approach and we reserve the analysis of other patterns for future work. At the k^{th} step of the recursive doubling, there exists $\mathcal{O}(\frac{N}{2^k})$ different partial contributions, each of them held by $\mathcal{O}(2^k)$ reducing ranks over a total of N reducing ranks. After $\mathcal{O}(\log(N))$ steps all involved ranks hold the same result. Therefore, the `allredux_submit` method, basically amounts to the pseudocode in Algorithm 19 which presents a simplified version of the implemented algorithm where preliminary and postliminary steps required to handle the case where N is not a power of two are discarded. This algorithm has to update the

```
1 set_redux(handle,redux,init)
2 for rank in contributing ranks do
3 |   insert_task(contrib,
4 |   handle:RANK_REDUX)
5 redux_submit(handle)
```

```
1 set_redux(handle,redux,init)
2 for rank in contributing ranks do
3 |   insert_task(contrib,
4 |   handle:RANK_REDUX)
5 allredux_submit(handle)
```

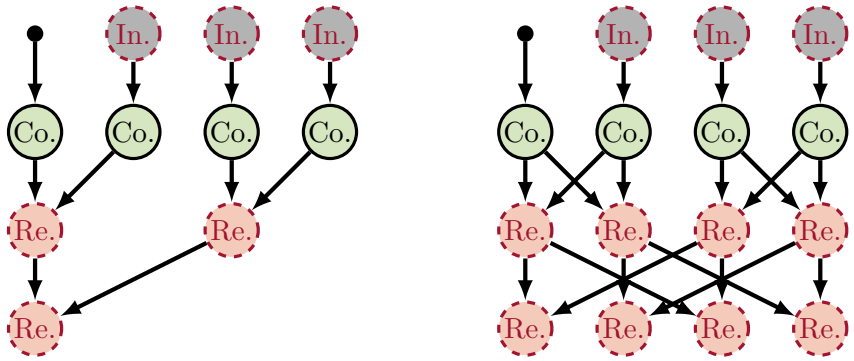


Figure 4.10: The reduce or the allreduce operation can be submitted by the runtime system when the user orovides a reduction codelet (orange tasks) and an initilization codelet (grey tasks). The submission can be triggered by the user once all the contributions (green tasks) have been submitted.

cache registry associated with D: after the DAG is submitted, the runtime should consider that all N contributing ranks hold a coherent value of D. Figure 4.10 (*right*) shows how the `allredux_submit` can be used in place of the `redux_submit` to achieve an allreduce operation.

Algorithm 19: Program to submit the sub-graph describing allreduce operation over a piece of data D using N contributing ranks. This sub-graph is submitted by rank `me`.

```
1  contributors ← N contributing ranks
2  my_id ← index of me in contributors
3  arity ← 1
4  while arity < 2^log2(N) do
5  |   if (my_id/arity)%2 == 0 then
6  |   |   partner ← contributors[my_id-arity];
7  |   else
8  |   |   partner ← contributors[my_id+arity];
9  |   insert_task(isend, D:R, partner)
10 |   T ← duplicate of handle describing D
11 |   insert_task(irecv, T:R, partner)
12 |   insert_task(redux, D:RW, T:R)
13 |   insert_task(invalidate, T:W)
14 |   arity ← arity * 2
15 update_cache()
```

The question arises of whether it is possible to implicitly trigger the insertion of the allreduce tasks as it is the case for the reduce operation. Obviously, if both operations rely on the same `RANK_REDUX` access mode, this is not possible because the runtime has no way of deciding whether to insert tasks to achieve a reduce or an allreduce. This problem can be overcome by introducing a new access mode, say, `RANK_ALLREDUX`; then, as soon as another task which uses the data in `R` or `W` mode is inserted, the runtime can automatically insert tasks to operate an allreduce among all the ranks that have previously submitted a task with this access mode. This clearly requires much more intrusive modifications of the runtime and we reserve it for future work. It must also be noted that the use of the `allredux_submit` makes it possible to implement the allreduce operation as an all-to-many collective communication. In this case the set of ranks owning a copy of the final assembled result can be different (a subset or a completely different set) than the set owning the data to be reduced; this can be achieved by passing a list of destination ranks to the `allredux_submit` method. This feature has been also considered for future work as a more appropriate algorithm than recursive doubling is required.

4.3.2 3D Cholesky factorization algorithmic variants

The use of 3D logical grids to compute distributed-memory matrix factorization remains the subject of recent studies. Such grids indeed defy the design of this operation vastly more than GEMM. When factorizing, different tasks are involved. First each block goes through one or multiple chains of (commutative) updates that are possibly assembled – this is a similar behavior to GEMM. However then the block is either solved or factorized before being broadcast: such a behavior creates more dependencies in the DAG of the operation. In order to account for the dependencies created by assigning updates across layers – a common design choice in the literature – the input matrix is not stored through the standard 2DBC layout but rather on more complex layouts that involve all layers of the 3D ranks grid. Additional design choices have been proposed in the last decade. These choices have however often been restricted to their respective designers' implementations and have hardly if never been adopted in common packages. In this section, we aim at leveraging the extended STF programming model in order to bring the complementary designs of the 3D Cholesky factorization under a single portable code.

We first adapt the algorithm from Beaumont et al. [25] to our extended set of features in Section 4.3.2.1. Through the features designed in this chapter, the LU factorization from Solomonik and Demmel [105] is adapted to the symmetric case in Section 4.3.2.2. Section 4.3.2.3 then consider the design proposed by Kwasniewski et al. [82]. The final STF implementation is parameterized to set up the complementary designs: Section 4.3.2.4 evaluates the performance of four variants of the Cholesky factorization we implemented as

well as several state-of-the-art packages.

4.3.2.1 3D update approach

Throughout an LU or Cholesky factorization, the trailing submatrix receives multiple contributions in the form of rank-k updates which are commutative and associative. As in the case of the GEMM operation described in Chapter 3, these properties can be easily exploited to design 3D algorithms: the computation of successive updates can be assigned to different layers in a 3D ranks grid and assembled using reduction operations prior to an elimination operation. This idea, which reduces the communication volume of the factorization, has already been used by Beaumont et al. in a recent work whose main objective is the design of a data distribution specifically suited for symmetric matrices and algorithms [25]. In the experimental results they assess the effectiveness of this distribution in a Cholesky factorization that relies on a cyclic distribution of the rank-k updates over the layers of a 3D ranks grid. Their algorithm was implemented in the Chameleon library using the basic STF model described in Section 3.1, which means that reductions are manually implemented through explicit data copies and tasks, that the mapping of tasks is bound to the data distribution and that no collective communications (i.e., broadcast) are used.

In this work we take advantage of the features available in the extended STF model that we introduced in Section 3.4.1 to obtain a more flexible, productive STF implementation of their algorithm. We have detailed how the purpose of the access mode `RANK_REDUX` is to spread the assembly of partial contributions over different layers. This access mode allows us to easily derive an STF implementation of the 3D Cholesky factorization from the 2D classical STF implementation. The resulting code is presented in Algorithm 20. The rank executing the update of the block A_{ij} at the l^{th} iteration of the factorization can be determined through a mapping function `map`. The `map` function takes the current iteration `i,j,l` as arguments. It could also be used to state the executing rank of the other tasks *i.e.* `trsm` and `potrf` tasks. In practice, when P nodes are involved in the computation, using `map(i,j,l) = q*(i%p) + j%q +` $P/h*(l\%h)$ evenly balances the computational load related to updates, where h is the number of layers.

4.3.2.2 Replicating diagonal block factorization

A more efficient but complex 3D Cholesky algorithm is proposed by Solomonik and Demmel [105]; a brief presentation of this algorithm is provided in Section 2.3.2.3. This algorithm, not only distributes rank-k updates of the trailing submatrix across the layers of the 3D ranks grid, but also the `trsm` operations. Note that these operations use the diagonal block after it has been factorized through a `potrf` operation; in order to reduce the cost of retrieving this diago-

Algorithm 20: STF-3D POTRF.

```
1 for l = 1...m do
2     insert_task (potrf, A_ll:RW, map(l,l,0):ON_RANK)
3     for i = l+1...m do
4         insert_task (trsm, A_ll:R; A_il:RW, map(i,l,0):ON_RANK)
5     for i = l+1...m do
6         for j = i...m do
7             rank ← map(i,j,l)
8             op ← if i = j then syrk else gemm
9             insert_task (op, A_il:R, A_jl^⊤:R, A_ij:RANK_REDUX, rank:ON_RANK)
```

nal block, this algorithm replicates `potrf` on every layer of the ranks grid. Finally, prior to being factorized, the diagonal block must be assembled because its contributions are spread over multiple layers; this can either be achieved through a reduce followed by a broadcast or, more efficiently, through an allreduce. Therefore, this algorithm can be efficiently implemented using the novel features of the STF model that we introduced in this chapter, namely, replicated data output and allreduce.

This implementation is presented in Algorithm 21. The actual replication is presented in Line 4 using the `insert_tasks` wrapper function, which can be used because the factorization of the diagonal block is replicated through identical `potrf` tasks. Once a diagonal block is factorized, it should be broadcast to proceed with the solve of the off-diagonal blocks: ranks that own such a block should retrieve the factorized diagonal block from a rank that lie on the same layer. This can be declared through the `set_alternative_source` method (see Line 7). The rest of the code is unchanged. The mapping that was proposed for the 3D update algorithm is still relevant and both algorithms are similarly compact.

4.3.2.3 Partitioning the updates

In Algorithm 22, the update of a block A_{ij} related to the l^{th} step of the factorization is sliced into H updates; we use superscripts over handles to denote the vertical slices – A^c_{il}. Setting $H = h$ results in the algorithm designed by Kwasniewski et al. [82]. To implement this slicing of updates, it is necessary to partition the blocks that result from `trsm` tasks. This partitioning may create a slight overhead to correctly pack the message of size $b \times {}^{b}/_{H}$ that is sent. StarPU provides a feature that allows us to do this partitioning dynamically in the course of the factorization. The data handle A^c_{il} describes the c-th vertical slice of A_{il} out of H slices. Such a handle could be declared while submitting the other tasks or, as is possible with StarPU, planned beforehand to be transparently handled during task submission. In practice, a dynamic partitioning can be expressed as tasks inserted into the DAG that enable or disable the use of the original *parent* data or its *children*; this fea-

Algorithm 21: STF-3D with diagonal computation replication POTRF.

```
1  for l = 1...m do
2      allredux_submit(A_ll)
3      ranks_on_aisle ← map(l,l,:)
4      insert_tasks(ranks_on_aisle,potrf,A_ll:RW)
5      for i = 1...h do
6          ranks_on_level ← map(:,:,i)
7          set_alternative_source(A_ll, map(l,l,i),ranks_on_level)
8      for i = l+1...m do
9          insert_task (trsm, A_ll :R; A_il :RW, map(i,l,i/max(p,q)):ON_RANK);
10     for i = l+1...m do
11         for j = i...m do
12             rank ← map(i,j,l)
13             op ← if i = j then syrk else gemm
14             insert_task (op, A_il:R, A_jl^⊤:R, A_ij:RANK_REDUX, rank:ON_RANK);
```

ture was recently used and improved to implement hierarchical tasks [56]. In order to implement this algorithm a loop must be added in the trailing submatrix update to iterate over the slices of a panel. This algorithmic variant also requires adding a dimension to the `map` function. Setting `map(i,j,l,c) = rank(`A_{ij}`) +` $^{P}/_{h} * ((l * H + c)\%h)$ yields a satisfactory load balancing as updates are issued in a round-robin fashion over layers. Moreover the presented algorithm extends from Algorithm 21: indeed, if $H = 1$, the partitioning operation would simply present an additional, redundant variable to handle A_{il}; additionally if $map(i, j, l, c = 1) = map(i, j, l)$ then Algorithm 22 clearly reduces to Algorithm 21.

The expression proposed in Algorithm 22 ends up offering a very versatile and compact way to approach matrix factorizations when pivoting is not needed. This expression may lead to the consideration of new algorithms agnostic of the matrix distributions and able to carry out combinations of state-of-the-art design choices. The goal of this work is not to extensively explore the possibilities allowed by this code but rather to provide a mixture of features that may help revisit state-of-the-art algorithms. The specified extended programming model is one way to achieve a compact, versatile implementation of a 3D pPOTRF.

4.3.2.4 Comparing the algorithmic variants

Several algorithmic variants are now available for benchmarking. For all the 3D variants, the matrix A is stored such that it is conforming to the mapping of tasks. One way to design such a conforming distribution is to start with a generic 2DBC layout of size $p \times q$ where $p * q = {}^{P}/_{h}$. This layout matches with the logical grid of the bottommost layer. Based on this initial distribution, matrix A can be redistributed to the existing h layers to better match the

Algorithm 22: STF-3D with diagonal computation replication POTRF and partition of the results of the TRSM.

```
1  for l = 1...m do
2      allredux_submit(A_ll)
3      ranks_on_aisle ← map(l,l,:,0)
4      insert_tasks(ranks_on_aisle,potrf,A_ll :RW)
5      for i = 1...h do
6          ranks_on_level ← map(:,:,i,0)
7          set_alternative_source(A_ll,map(l,l,i,0),ranks_on_level)
8      for i = l+1...m do
9          insert_task (trsm, A_ll :R, A_il :RW, map(i,l,i/max(p,q),0):ON_RANK)
10     for i = l+1...m do
11         for j = i...m do
12             for c = 1...H do
13                 rank ← map(i,j,l,c)
14                 op ← if i = j then syrk else gemm
15                 insert_task (op, A_il^c :R, A_jl^c ^T :R, A_ij:RANK_REDUX,
                    rank:ON_RANK)
```

computational pattern of 3D algorithms. Indeed, it is possible to assign a layer to subsets of $p \times q$ blocks of the matrix A. This assignment can be used as an offset from the initial layout. A round-robin procedure, depicted in Figure 4.11 for $p = q = 2$ and $h = 3$, can be implemented to make this assignment. This procedure can adapt to 2DBC layouts but also symmetric layouts described in Part II. For the Cholesky factorization, this means $h * q$ different ranks are involved in the computation of `trsm` tasks which is suitable to benefit from advanced scalable designs.

Four distinct algorithms can be benchmarked using the extended STF programming model:

- **2D** This is the classical 2D Cholesky factorization implementation in STF.
- **Baseline** This corresponds to the 3D Algorithm 20. For this algorithm, the input matrix A is distributed such that the l^{th} column (if A's upper half is allocated) is stored over the $l\%h^{th}$ layer. With an underlying 2DBC distribution, we set $\texttt{rank}(A_{ij}) = q * (i\%p + j\%q) + (j\%h) * pq$.
- **Repl** This corresponds to the 3D Algorithm 21. For this algorithm, the input matrix A is stored as presented in Figure 4.11. With an underlying 2DBC distribution, we set $\texttt{rank}(A_{ij}) = q*(i\%p+j\%q)+((\frac{m}{L}\frac{i}{L}+\frac{j}{L})\%h)*pq$ where $L = max(p, q)$.
- **Repl-Part** This corresponds to the 3D Algorithm 22 with $H = h$. For this algorithm, the input matrix A is stored the same way as **Repl** algorithm.

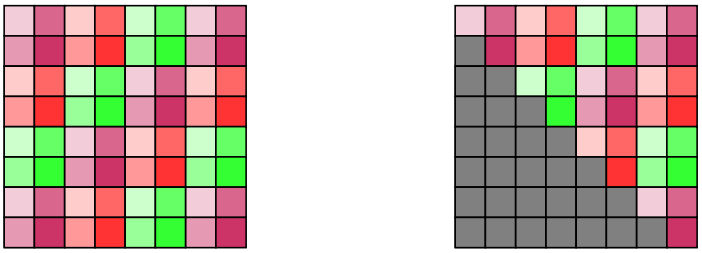


Figure 4.11: 3D distributions with "large" blocks distributed in a round-robin way. One color corresponds to one rank. The three color shades (purple, red, green) each correspond to a layer.

The first evaluation that can be made is the improvement in communication volume brought by 3D algorithms. From Figure 4.12, it is clear that the 3D algorithms reduce the communication volume with respect to the 2D standard algorithm. In the present experiment, 64 Skylake computing nodes are used. For the 2D algorithm, they are arranged in an 8×8 logical grid whereas for the 3D algorithm they are arranged in a $4 \times 4 \times 4$ logical grid. In a 3D grid of size $p \times q \times h$, the communication volume V for a Baseline 3D Cholesky factorization of a $m \times m$ matrix arranged in a block-cyclic $p \times q \times h$ layout can be computed as

$$\begin{aligned} V &= \sum_{i=1}^{m} \Big(\underbrace{(p-1)}_{\substack{\text{send the result} \\ \text{of } \texttt{potrf}}} + \underbrace{(m-i)(p-1+q-1)}_{\substack{\text{send the result} \\ \text{of } \texttt{trsm}}} \Big) + \underbrace{\frac{m(m+1)}{2}(h-1)}_{\substack{\text{assemble the result} \\ \text{of updates}}} \Big) \\ &\sim (p+q+h-3)\frac{m^2}{2} \end{aligned}$$

The increase from 3D to 2D algorithms in the presented figure is in the range of $\frac{8+8+1-3}{4+4+4-3} - 1 = 55\%$ which is close to what we measured in practice. We do not report the difference in communication volume across all the 3D variants because it is barely measurable. First, parallelizing the `trsm` tasks only affects the communication volume corresponding to sending the result of `potrf` tasks which is of the first order in m and thus negligible. Second, partitioning does not alter the communication volume at all: H slices of $b \times b/H$ elements weigh as much as one slice of $b \times b$ elements.

Figure 4.13 compares the scalability of the different variants implemented through the STF programming model. For these results, one MPI thread is created per computing node. Block sizes 256, 512 and 1024 are used to explore the tradeoff between arithmetic efficiency and task granularity. Our STF implementation relies on efficient MKL BLAS sequential kernels provided by the supercomputing facility. The 3D variants use $h = 2$ or $h = 4$: the best of the two configuration is reported. The 3D variants however obtain results that are occasionally significantly improving the performance of the 2D variant.

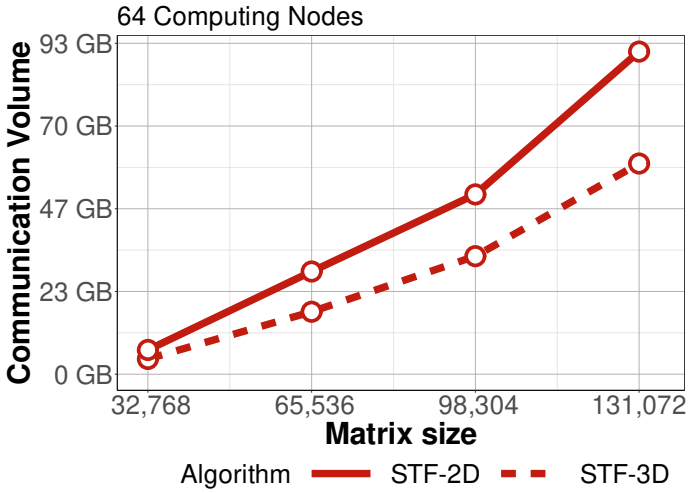


Figure 4.12: Communication volume of Cholesky factorization with 64 Skylake computing nodes. Matrix of increasing sizes are used.

This difference is rather unexpected because Figure 4.12 shows 3D algorithms consistently perform fewer communications. The most sensible interpretation of these results is that communications and computation are well overlapped most of the time and minimizing communication does not systematically yield more performance. We can observe that the **Repl** variant obtains the best results on 128 nodes for $m = 98,304$ and the **Baseline** one on 256 nodes for $m = 163,840$. Such cases seem to fall in a "sweet spot" where communications are costly enough while proving difficult to parallelize for 2D algorithms. For the smallest problem, strong scalability is difficult to maintain even for 3D algorithms as the amount of work is relatively short. In larger problems, the overhead that may come with advanced mechanisms – such as the overhead of replicated computation – is larger than what the overlap of computation and communication can absorb.

Figure 4.14 compares the best of the 2D and 3D STF approaches with several state-of-the-art libraries: ScaLAPACK, its successor SLATE and Elemental have already been presented in Section 3.5. CONFCHOX is included in the comparison: it is the library from Kwasniewski et al.[82] available online which provides an algorithm similar to **Repl+Part** [2]. This library relies on OpenMP to orchestrate multiple threads across cores as well as MPI to exchange messages. In this sense, CONFCHOX is implemented with a programming model that is close to SLATE. For CONFCHOX we have tested block sizes 256, 512 and 1024 and relied on the previously mentioned sequential BLAS kernels. Two MPI ranks are spawned per computing node as it proved efficient. One thread is spawned per core. Elemental and Slate follow the same mapping of threads. Elemental, CONFCHOX and Slate rely

[2]CONFCHOX `git` repository: `https://github.com/eth-cscs/conflux`

on OpenMP for thread orchestration while ScaLAPACK relies on MKL multithreaded BLAS to busy all cores. We have found that larger block sizes were beneficial in this multithreaded case (1024, 2048 and 4096). To simplify the reading of the graph, block sizes are not denoted by their dimension but qualified by "Small", "Medium", or "Large". For each variant in each library, each block size and each problem is run four times. We report the median performance of the best block size. Elemental failed to run for $m = 215,040$ with $P = 16$ when allocating the matrix. The dimensions of the logical grid of ranks have been chosen to be as square as possible as to minimize communications *i.e.* $p \sim q$. For CONFCHOX, $h = 2$ and $h = 4$ have been tested and we report the best median for any configuration. From this figure, we can observe that modern libraries are improving over ScaLAPACK which is expected because they rely on asynchronous MPI calls to transfer data. The observations that 3D algorithms perform relatively poorly remains. Notably the CONFCHOX library performs badly on 128 and 256 nodes which is unexpected as it gets remarkable performance on 16, 32 and 64 nodes. We have not explored this behavior further.

We are confident in the capacity of our own implementation to perform correctly: our 3D algorithms communicate less as expected. However, 3D algorithms have been designed on entirely different machines with entirely different assumptions from the one that match with modern implementations. It can be noted that the computing nodes on Platform B contain more cores and deliver more computing power than the ones used by Solomonik and Demmel or Kwasniewski et al. Moreover 3D algorithms often assume that communications are fully synchronous and each computation and communication steps of the routine are clearly defined. In all the library except ScaLAPACK this hypothesis is not verified: they all leverage asynchronous communications. Our implementation moreover benefits from the fine-grain parallelism allowed by the StarPU runtime system. Consequently the performance benefit of decreasing the communication volume may not be as stark because asynchronous communications overlap computations very well on modern machines.

Figure 4.15 details the performance of a single Platform B core. The figure evaluates the execution speed of multiple `gemm` executions with only the median reported. Here, the resulting matrix C is always square ($m = n$) but the update size k is increased from 64 up to the size of C. The graph shows that peak performance is attained when C is at least of dimension 512. It also shows that the update size should be large to maintain peak performance: with C of dimension 512, if $k = {}^{512}/_{4} = 128$ then the execution speed of the kernel is decreased by 10%. While this decrease in performance can seem negligible, it should be pointed out that because `gemm` tasks make up the largest portion of tasks a diminution in execution speed may become noticeable. Such a diminution of the kernel speed may explain why the **Repl+Part** variant in Figure 4.13 is underperforming all the other variants implemented in the STF model in the majority of the configurations. With the largest block size of 1024

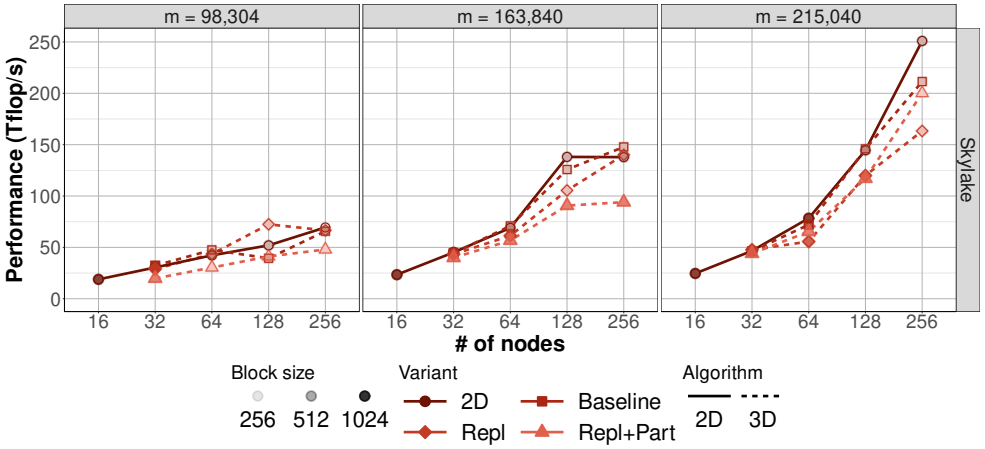


Figure 4.13: Benchmarking all 3D variants using the STF model in a strong scalability settings for some problem sizes of increasing dimension.

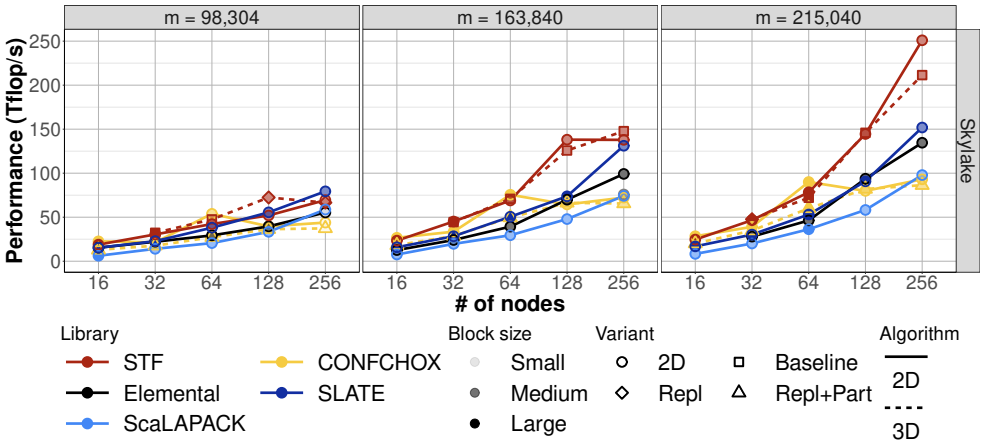


Figure 4.14: Benchmarking reference libraries and STF model in a strong scalability settings for some problem sizes of increasing dimension.

however, the **Repl+Part** variant obtained the best results for $m = 215,040$ using $P = 128$ and $P = 256$ nodes with 117 TFlop/s (+42% over 2D) and 143.3 TFlop/s (+19% over next best 3D variant **Repl**) respectively. These results, not reported in the figure because a smaller block size is more beneficial, hint at the potential interest of partitioning the updates across layers that could be explored in future work.

4.4 Concluding remarks

In this chapter, two features have been introduced to extend the STF programming model further.

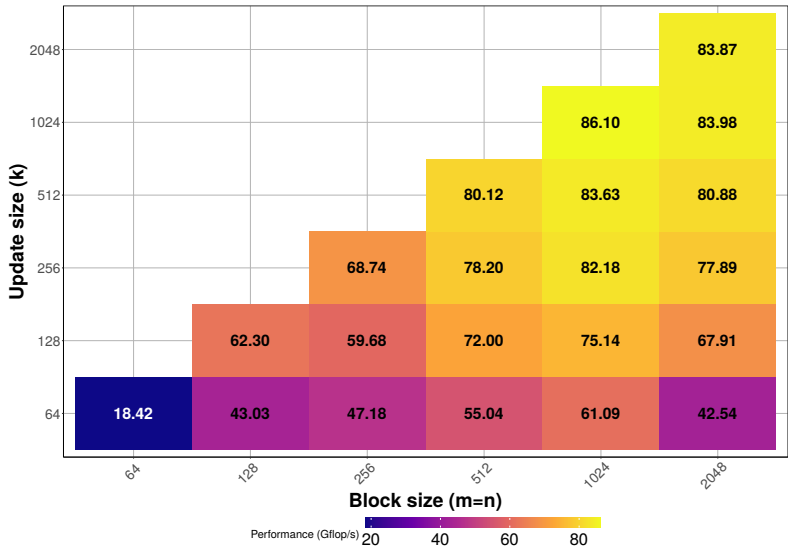


Figure 4.15: Double-precision GEMM single-core performance

The first one introduces an additional data access mode `SAME` that allows replicating data output. This access mode is provided to the programmer to declare replicated data modifications across ranges of nodes. The use of this access mode has been validated on a 2D 5-points stencil application by implementing a communication-avoiding algorithm. The resulting code is simplified by using the access mode – the programmer does not need to consider additional tasks to explicitly manage replicated copies, rather she/he just tell the runtime what tasks should be multiplied over which ranks. The experimental results assess the effectiveness of this new feature as the communication-avoiding variant proves to be more scalable than its conventional counterpart, as expected from the literature.

The second extension to the programming model is the addition of an allreduce operation. We introduced the `allredux_submit` method that, when called, generates appropriate tasks to compute the allreduce of a data that was previously updated by tasks submitted with the `RANK_REDUX` access mode. More work might be required to make this feature completely transparent, for example, through automatic tasks generation as in the case of the reduce operation; this approach, however, is very little intrusive and preserves the readability of the code.

The combination of the two extensions with those of Chapter 3 makes it possible to express scalable Cholesky factorization algorithms. Several variants of such algorithms exist and four have been considered. Because these variants incrementally expand the design of one another, only two routines have actually been implemented with parameters guiding their behaviors to target specific variants. The first routine covers the **Baseline** and **2D** vari-

ants. The second one covers the remaining variants. While it is technically possible to cover all variants within one code, the engineering effort albeit small was not deemed necessary. The final expression we have provided demonstrates that the STF programming model can provide compact and versatile code assembling key designs and concepts from the literature on scalable algorithms. Experiments have shown the STF implementations can be scalable with respect to reference implementations, sometimes outperforming them.

Part II

Symmetric operations

Chapter 5

Scalable Symmetric Matrix-Matrix multiplication

Level 3 BLAS standardizes operations that take the structural properties of the input matrices. This is the case for Matrix-Matrix multiplication that has a general routine (GEMM) but also a triangular variant (TRMM) as well as a symmetric one (SYMM). By taking these structural properties into account it is possible to reduce the number of transfers required by leveraging the symmetry or even to reduce the number of operations by taking zeroed out values in consideration. In this chapter we will consider the case of SYMM as it proves an opportunity to showcase the flexibility of the STF programming model. For SYMM only the matrix A is considered symmetric with B and C being full dense matrices: the operation computes $C \leftarrow \alpha AB + \beta C$ where $A^T = A$ and α and β are scalars.

Out of the matrix multiplication context, Beaumont et al. recently proposed to exploit the symmetry of matrices to enhance the distributed-memory Cholesky factorization of dense symmetric positive definite matrices [25]. The main idea is to rely on an alternative data distribution referred to as symmetric block cyclic (SBC). Still in the context of the Cholesky factorization, but in a sequential out-of-core setting, Beaumont et al. proposed in another study a variant referred to as Triangular Block Syrk (TBS) [26], and provided sharp bounds showing that TBS achieves the lowest possible I/O volume for this operation. However, since it does not readily apply to a distributed-memory context in the case of a Cholesky factorization, TBS was only considered in a sequential setting.

Section 5.1 is dedicated to the design of the SYMM operation using the extended STF programming model presented in Part I. This design is agnostic

of the layout of the matrices involved in the operation and thus symmetric layouts are adapted to this operation in Section 5.2 to benefit from a reduction in communication volume. Experimental evaluation is provided in Section 5.3. Matrix-matrix multiplication can be used to project a set of vectors B into a given euclidean space described by the symmetric matrix A. In the MultiDimensional Scaling (MDS) algorithm presented in Section 5.4 such projections make the bulk of the computational workload when randomized algorithms provide a satisfactory precision. An application using the MDS to analyze datasets is presented in Section 5.5. Through this application the practical interest of the symmetric layouts combined with the extended STF programming model are demonstrated.

5.1 SYMM task-based design

Fully-featured distributed-memory dense linear algebra libraries such as ScaLAPACK [29] or Elemental [95] implement SYMM with a 2DBC data distribution. Such a regular data distribution makes it possible to easily set up MPI communicators along rows and columns and ensure collective communications. On the contrary, if we want to consider irregular data distributions, like those discussed in Section 5.2, it may be challenging to implement the corresponding code directly through the MPI interface. We therefore aim at designing a SYMM routine completely independent of the proposed mappings so that we can then effortlessly implement any non-trivial mapping. Task-based programming allows for such a separation of concerns as discussed in Section 2.2.1. Without loss of generality, we restrict the discussion to the $C \leftarrow AB + C$ case, where A is symmetric and only its lower part explicitly stored. As in the previous chapters we also assume blocks of size b-by-b, so that A is a M-by-M block matrix ($m = M * b$) and B and C are M-by-N block matrices ($n = N * b$). The sequential algorithm of the SYMM consists of three nested loops where the two innermost loops perform a matrix - block-column product ($C_{*,j} \leftarrow C_{*,j} + AB_{*,j}$), while the outer loop goes through all N block-columns.

Algorithm 23: STF blocked SYMM.

```
1 for j = 1...N do
2 |   for i = 1...M do
3 |   |   for l = 1...M do
4 |   |   |   op ← if i = l then symm else gemm
5 |   |   |   blk_A ← if i ≤ l then A_il else A_li^⊤
6 |   |   |   insert_task (op, blk_A:R,B_lj:R,C_ij :RW)
```

Algorithm 23 shows how to implement this algorithm for distributed-memory machines with the baseline STF model. We may observe that the

STF pseudocode is very similar to a sequential one. Section 3.1 already discusses how the access mode are chosen in the GEMM case. The main difference between GEMM and SYMM is that the block of A multiplied by B_{lj} should be selected in the correct half of the matrix A. Moreover for diagonal blocks in A the `symm` kernel should be submitted rather than `gemm`.

Algorithm 24: Scalable STF block stationary-A SYMM.

```
1 for j = 1...N do
2     for i = 1...M do
3         for l = 1...M do
4             op ← if i = l then symm else gemm
5             block_A ← if i ≤ l then A_il else A_li^⊤
6             rank ← if i ≤ l then rk(A_il) else rk(A_li)
7             insert_task (op, block_A:R,B_lj:R,
                C_ij:RANK_REDUX,rank:ON_RANK)
```

The baseline STF algorithm would however prevent us from implementing stationary-A schemes. Indeed, first, performing a task on the MPI ranks owning data accessed in `RW` access mode implies that the C matrix stays in place and A and B are transferred through the network. The analyses from Section 5.2 assume that, because the matrix A is the largest one, it is instead preferable to keep A in place and only move around blocks of B and C. To implement such a *stationary-A* scheme, we can leverage the extended programming model detailed in Part I. Altogether the provided extensions allow us to reach our main goal of designing a SYMM routine completely independent of the mappings so that we can then effortlessly implement any non-trivial mapping and achieve the expected associated arithmetic intensity (AI) which is the workload size over data transfers volume.

5.2 Data distributions for SYMM

In this section, we present different data distributions for the $C \leftarrow AB$ matrix product as adding the matrix C to this result has no impact on data transfers. Distributions of increasing complexity and AI are presented. The results are summarized in Table 5.1. In this chapter we assume $m \gg n$, i.e., the matrix A is much larger than both B and C, in which case stationary-A schemes – introduced in Section 2.3.2.1 – are the best approaches to minimize communication volume.

We start by the easiest case: if the complete matrix A is stored, the best solution is the 2DBC distribution (line 1 in Table 5.1, and Section 5.2.2), and we analyze its communication volume. We then specialize to the case where only half of matrix A is stored (because of symmetry). We use the same analysis to show that the communication volume of the 2DBC distribution (line 2, and Section 5.2.3.1) is twice larger than in the previous case. We then de-

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

Table 5.1: Discussed pGEMM (denoted $\mathcal{G}$ in the table) and pSYMM (denoted $\mathcal{S}$) stationary-A schemes together with their communication volume Q and AI. P denotes the number of nodes, S the storage per node. The communication volume Q is expressed as a factor of mn (which is the common size of matrices B and C).

scribe two symmetric distributions: first SBC [25] (line 3, and Section 5.2.3.2), whose communication volume is lower by a factor of $\sqrt{2}$, then TBC (line 4, and Section 5.2.3.3), which we adapt from [26], whose communication volume is lower by another factor of $\sqrt{2}$. In total, SYMM with the TBC distribution achieves the same communication volume as 2DBC when the whole matrix is stored, thus saving a factor of 2 on storage. Section 5.2.4 summarizes these results and also proposes another interpretation when the memory is bounded. Section 5.2.5 extends the analysis to the 3D case [1, 102].

5.2.1 Generalities

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

With this total communication volume Q, we can also compute the AI, defined as

$$\text{AI} = \text{flop}/Q, \tag{5.1}$$

where flop is the total number of floating point operations. The number of floating point operations does not depend on the allocation, it is $2m^2n$ in all cases: one multiplication and one addition for each product computed. Hence, the AI is inversely proportional to the communication volume Q, and varies like $\frac{m}{\sqrt{P}}$ for all 2D distributions (see left part of the AI column in Table 5.1). However, we can also express AI as a function of the memory size of one node, denoted as S; this allows one to measure how efficient an algorithm is

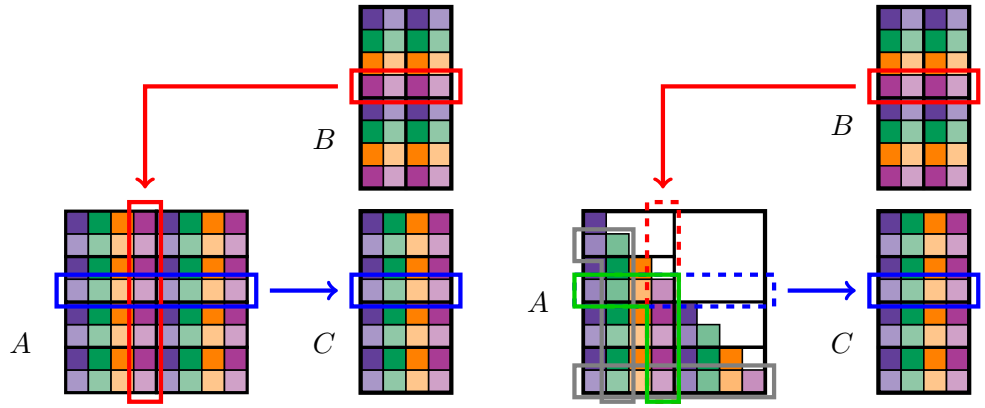


Figure 5.1: Communications incurred with an stationary-A matrix multiplication, with a 2DBC $(2, 4)$ distribution. **Left:** storing the whole matrix A. One block of B follows the red path and is sent to $p - 1 = 1$ nodes. A result computed on a row of A is involved in a reduction operation on q nodes (blue path), resulting in $q - 1 = 3$ messages sent. **Right:** storing the lower half of A. Both types of communication now involve $p + q - 1 = 5$ nodes, resulting in 4 messages sent. Parts of the matrix where fewer nodes are involved are highlighted in gray.

at using the values stored in memory. For all 2D distributions studied here, $\text{AI} = \Theta\left(\sqrt{S}\right)$; the efficiency of an algorithm is measured by how large the constant is, shown on the right part of the AI column in Table 5.1.

5.2.2 Stationary-A, general matrix multiplication

We first consider the situation where the whole matrix A is stored, and distributed among the nodes in a 2DBC (p, q) fashion. This situation is depicted on the left of Figure 5.1.

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

Similarly, consider a given row of matrix A. Since the nodes that own this row need to perform one reduction per column of C to send the total to the owner of the corresponding block in C, the total communication volume for the blocks of C is $Q^C = mn(q - 1)$ in the best case, and $Q^C = mnq$ in the

worst case (when the owner of a block in C never belongs to the corresponding set of nodes in the row of A).

The best case can be achieved if C is distributed with the same $p \times q$ 2DBC distribution, and B is distributed with the transpose distribution of dimension $q \times p$.

In total, the communication volume is $Q^{p,q}_{\text{GEMM}} = mn(p + q - 2)$. In practice, we often choose $p \simeq q \simeq \sqrt{P}$, so that $Q^{\text{2DBC}}_{\text{GEMM}} \simeq 2mn(\sqrt{P} - 1)$. Asymptotically, the AI is $\text{AI}^{\text{2DBC}}_{\text{GEMM}} \simeq \frac{2m^2n}{2mn\sqrt{P}} = \frac{m}{\sqrt{P}}$, with a memory usage $S = \frac{m^2}{P}$, which yields $\text{AI}^{\text{2DBC}}_{\text{GEMM}} \simeq \sqrt{S}$. This result is summarized in Table 5.1, line 1.

5.2.3 Stationary-A, symmetric case

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

5.2.3.1 Standard 2D block-cyclic distribution

In this case, the 2DBC distribution is only applied to the lower half of the matrix, the upper tiles are not being stored at all. The result is depicted on the right of Figure 5.1. We can apply the same kind of reasoning as for the previous case. However, now the set of nodes that own a given column of A can be of size [1] up to $p+q-1$: the p nodes that own the (truncated) column, plus the q nodes that own the (truncated) row that completes the column (the total is $p + q - 1$ because one node belongs to both the row and the column). Again, the best case is when the set of nodes that own the blocks of B is included in these $p + q$ nodes, and this yields a communication volume $Q^B = mn(p + q - 2)$. Similarly, we get $Q^C = mn(p + q - 2)$.

In total, $Q^{p,q}_{\text{SYMM}} = 2mn(p + q - 2)$. As we can see, the communication volume is twice as large as in the previous case, and can be written as $Q^{\text{2DBC}}_{\text{SYMM}} \simeq 4mn(\sqrt{P}-1)$. Asymptotically, the AI is $\text{AI}^{\text{2DBC}}_{\text{SYMM}} \simeq \frac{2m^2n}{4mn\sqrt{P}} = \frac{m}{2\sqrt{P}}$, with a memory usage $S = \frac{m^2}{2P}$, which yields $\text{AI}^{\text{2DBC}}_{\text{SYMM}} \simeq \sqrt{S/2}$. This result is summarized in Table 5.1, line 2. As we can see, the AI is twice smaller compared to the previous case, but since the memory usage is also smaller by a factor of 2, the AI expressed as a function of S is only lower by a factor of $\sqrt{2}$. As discussed in more details in Section 5.2.4, this means that if the memory of the nodes is the limiting factor, storing half the matrix allows to use half as many nodes, which partially offsets the overhead in terms of communication volume.

[1] The first q (respectively the last p) columns of A involve a slightly smaller number of nodes, because not all nodes appear in the truncated row (respectively column). The corresponding zones are highlighted in gray on the right of Figure 5.1. However, since we are interested in large matrices A where $m \gg p, q$, we decide to neglect this effect.

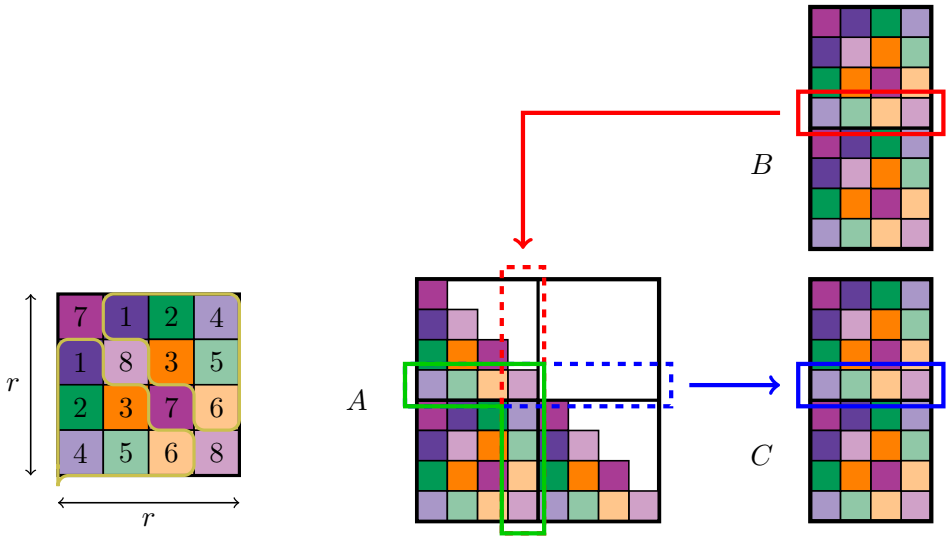


Figure 5.2: Symmetric Block Cyclic distribution. **Left:** the pattern with $r = 4$, using $P = 8$ nodes. The symmetrical lower and upper parts are highlighted. **Right:** communications induced when using SBC. Communications related to a row of matrices B and C both involve r nodes, resulting in $r - 1 = 3$ messages sent.

5.2.3.2 Symmetric Block Cyclic distribution

In order to reduce the amount of communication, we need to make sure that the nodes that own a truncated column of A are the same as the nodes that own the corresponding truncated row. The Symmetric Block Cyclic distribution (SBC) has been proposed in the context of the Symmetric Rank-k update (SYRK) and the Cholesky factorization [25], where a similar issue appeared. We describe here the basic version of SBC, defined for an even integer $r > 2$ (see Figure 5.2). It consists of a symmetric $r \times r$ pattern with $P = r^2/2$ nodes: $\frac{r(r-1)}{2}$ nodes are organized arbitrarily in one half of the pattern, and symmetrically on the other half. The remaining $\frac{r}{2}$ nodes are each placed on two locations in the diagonal.

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

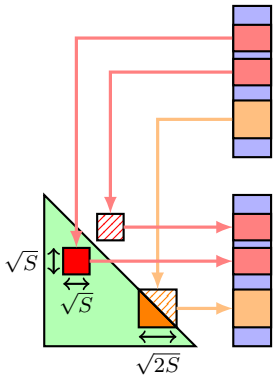


Figure 5.3: Triangles along the diagonal use fewer communications: both the red square and the orange triangle contain S elements. However, the corresponding operations for the square require $2\sqrt{S}$ rows of B, and only $\sqrt{2S}$ rows for the triangle.

the same AI as the 2DBC distribution when storing the whole matrix. This result is summarized in Table 5.1, line 3.

5.2.3.3 Triangular Block Cyclic distribution

Another recent work proposed a Triangular Block approach for the SYRK operation [26], which achieves provably the lowest possible quantity of data transferred. This work was presented in the context of sequential out-of-core computations, but we propose here a way to transform it into an allocation for distributed nodes.

We remind that with the SBC distribution, each node is assigned $S = \frac{m^2}{2P} = \frac{m^2}{r^2}$ blocks, and needs to receive 2 rows of matrix B for each repetition of the pattern. The number of required rows of matrix B is thus $h = \frac{2m}{r} = 2\sqrt{S}$. This is similar to assigning a square part of the matrix to a node, as shown on Figure 5.3: if a node is responsible for the red square of side $\sqrt{S}$, it needs to receive $\sqrt{S}$ rows of B to perform the operations in the lower half, and another $\sqrt{S}$ rows to perform the operations in the upper half.

The idea of the TBS algorithm [26] stems from the observation that, thanks to the symmetry of matrix A, triangular parts along the diagonal of A are involved in operations that require even fewer communications than square parts. Indeed, as shown with the orange triangle on Figure 5.3, if a node owns a triangle containing S elements along the diagonal of A (its side length is thus $\sqrt{2S}$), it only needs to receive $\sqrt{2S}$ rows from matrix B since the operations on the lower and upper half require the same rows of B. Similarly, this node only needs to participate in reduction operations on $\sqrt{2S}$ rows for matrix C. The TBS algorithm defines a solution where these favorable properties

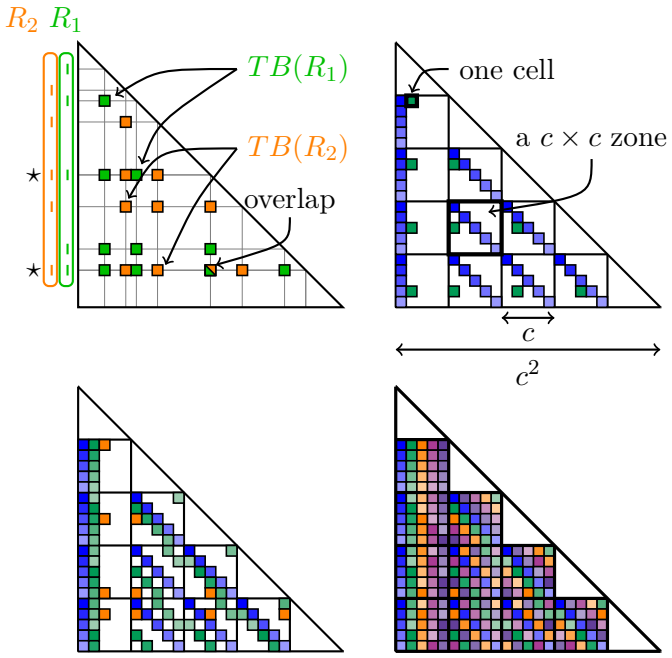


Figure 5.4: **Top Left:** two triangle blocks. An overlap happens when two triangle blocks have two rows in common ($\star$). **Top Right:** first column of the triangle-blocks in the TBC pattern. Gaps must be introduced for the triangle-block in the next column to avoid overlapping. **Bottom Left:** the gaps in the next column need to be larger, with a "wrap around" when reaching the bottom of the zone. **Bottom Right:** complete pattern for all the zones.

are extended to blocks away from the diagonal by ensuring that each node is assigned a set of blocks which can be gathered into a diagonal triangle using a symmetric permutation.

This leads to the notion of *triangle-blocks*, defined, for a given set R of row indices, as the set of blocks of the matrix A that a node can own while only requiring rows of matrix B indexed by R. Formally, the triangle-block associated with R is $TB(R) = \{(i,j) \in R^2 | i > j\}$. Figure 5.4 (left) shows examples of two triangle blocks. This notion generalizes the "triangle along the diagonal", since a triangle-block with $|R| = h$ contains $\sim \frac{h^2}{2}$ blocks of A, and the corresponding operations involve only h rows of matrices B and C. A triangle-block can indeed be seen as a triangle along the diagonal, up to reordering of the rows and columns of the matrix.

The key contribution of [26] is a method that makes it possible to partition almost all the matrix in disjoint triangle-blocks. This requires to assign a set of rows R_p to each node, so that any two sets R_p and $R_{p'}$ have at most one row in common. Indeed, as can be seen on the top-left of Figure 5.4, two triangle blocks overlap if they share two row indices. This implies that

the two corresponding nodes will receive the data necessary to perform an operation, however only one of them will actually perform it; communicating that data to the other node was not useful. Finding a distribution with no overlap ensures that we minimize the communication volume.

To apply these ideas in a distributed-memory setting, we propose to build a pattern where each triangle-block is assigned to a different node. Since this pattern is symmetric, for simplicity we only describe its lower half. We fix a prime integer $c > 2$, and build a symmetric pattern of size $c^2 \times c^2$, divided in $c \times c$ square *zones*, each containing c^2 cells. We partition the square zones among triangle-blocks, and the main idea is that each triangle-block has one cell in each zone. The top right of Figure 5.4 shows how the first c triangle-blocks are organized. The next triangle-block is also shown, and we can see that a gap must be introduced at each new row to ensure that it does not overlap with the previous blocks. The bottom left of Figure 5.4 shows the next step: the other blocks of the second column can be assigned with the same gaps. However, the next block in the third column needs to have larger gaps at each row to ensure that no overlap happens. With such large gaps, the last row would be outside the zone, so the actual row is chosen modulo c: this effectively "wraps around" at the boundary of the zone. The final partition with all the triangle-blocks is shown at the bottom right of this figure.

With this partitioning, each node receives $\frac{c(c-1)}{2}$ cells from the lower half of the pattern, but the cells in the triangular zones along the diagonal remain unassigned. The choice of the pattern size ensures that if we exclude the diagonal cells, each of these triangular zones also contain $\frac{c(c-1)}{2}$ cells. We can thus assign them to c additional nodes, which describes the entire lower half of the pattern. By replicating this symmetrically, we get a square pattern where only the non-diagonal cells remain unassigned. A more precise description of the pattern is described in Algorithm 25, and the resulting pattern for $c = 3$ is provided in the left of Figure 5.5. The iteration (i, j) of the loop in line 4 assigns the triangle-block which contains the cell of coordinates (i, j) of the top-most zone (which is the cell $(i+c, j)$ of the pattern). The idea behind line 5 is that each node should access one row in each zone: the value uc indicates the index of the first row on the u-th zone, and the value $i + (u - 1)j \mod c$ is the index of the row within this zone. We can see in this formula that there is a gap of size j between one row and the next (thus for two successive values of u), and that there is a modulo operation to perform the "wrap around", as described in the successive diagrams of Figure 5.4. The results from [26] (in particular Lemma 5.5), together with the condition that c is prime, ensure that the sets of rows assigned to different nodes overlap exactly once, so the sets of cells assigned to the nodes are disjoint, and each row contains exactly $c + 1$ nodes.

This procedure results in a symmetric pattern of size $c^2 \times c^2$, in which the diagonal cells are not allocated. However, there are $c(c + 1)$ nodes in total, and only c^2 diagonal cells. Each of these diagonal cells can be allocated to

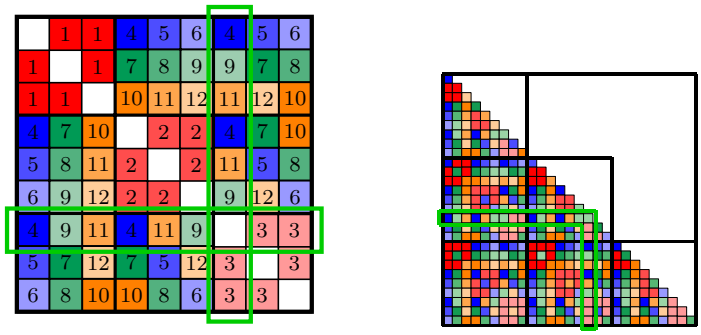


Figure 5.5: Triangular Block Cyclic distribution. **Left:** the pattern with $c = 3$, using $P = 12$ nodes, with no allocation on the diagonal blocks. **Right:** allocation of this pattern on a 27×27 matrix, where the diagonal blocks of the pattern are filled with the greedy algorithm. As shown in the highlighted part, each communication (related to matrix B or C) involves 4 different nodes (nodes 3, 4, 9, 11 in this example). For comparison, in the $(3, 4)$ 2DBC distribution, each communication involves 6 nodes.

Algorithm 25: TBC(c) pattern, on $P = c(c+1)$ nodes.

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

any node already present on the row without increasing the communication volume.

To obtain the final allocation, we replicate this incomplete pattern over the matrix A, and apply a greedy algorithm to allocate the remaining blocks : for each unassigned block, we pick the node with the lowest number of assigned blocks among all the nodes present in the row (and thus in the column, by symmetry of the pattern). The resulting allocation is thus not exactly a *cyclic* allocation, but it can nonetheless be computed very quickly. An example is provided on the right of Figure 5.5.

This pattern uses a total number of nodes $P = c(c+1)$, and each row and column of matrix A is allocated to a set of $c+1$ nodes. The communication volume can be evaluated just like previously: each communication related to a row of matrix B or C involves $c+1$ nodes, and we obtain $Q^B = Q^C = mnc$ in the best case. Thus, $Q = 2mnc \simeq 2mn\sqrt{P}$. This corresponds to

another improvement by a factor of $\sqrt{2}$ over SBC, and asymptotically the same communication volume as with the 2DBC distribution when storing the whole matrix. The AI is also improved by a factor of $\sqrt{2}$: $\text{AI}^{\text{TBC}}_{\text{SYMM}} \simeq \sqrt{2S}$. This is the best of both worlds: the reduced memory storage gained by storing only half the matrix, and the reduced communication volume. This result is summarized in Table 5.1, line 4.

5.2.4 Summary of AI analysis

Table 5.1 summarizes all the results. We propose two interpretations, depending on whether we read the left-hand side or the right-hand side, respectively, of the last column (AI) of the table. A first interpretation, with the left-hand side $\text{AI}(m, P)$, is as follows. Assuming an infinite storage ($S = \infty$), for a given size m of matrix A and a given number of nodes P, 2DBC SYMM has a lower AI by a factor of 2 than 2DBC GEMM. SBC and TBC improve the AI of SYMM by a factor of $\sqrt{2}$ and 2, respectively, thus in particular equaling that of 2DBC GEMM for the latter one while consuming twice less memory.

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

5.2.5 3D variants

For a fixed number of nodes P, the above discussion hints at a possible strategy for increasing the AI: increase the storage per node S. This is actually the idea behind the 3D extensions to 2DBC [1, 102], and it can also be applied to any of the above (stationary-A) distributions. The idea is to split the nodes into s slices (each with $\frac{P}{s}$ nodes) so that, on each slice, the whole matrix A is distributed with one of the above distributions. The matrix A is thus replicated s times. Matrices B and C are accordingly split into s column matrices $B_1, \ldots, B_s$ and $C_1, \ldots, C_s$, so that B_k and C_k are distributed among the nodes of slice k. Then the computation of all the $C_k \leftarrow \alpha AB_k + \beta C_k$ can be performed independently, with no additional communication since matrix A is replicated on each slice. For a fair comparison with the GEMM case where matrix A is assumed to be fully stored, we do not consider the communications involved in replicating A.

Using a 3D variant thus multiplies the storage cost S by a factor of s, for a benefit on the communication volume by a factor of $\sqrt{s}$, since the AI grows linearly with $\sqrt{S}$. In particular, using $s = 2$ slices with the TBC distribution

yields the same storage cost as the GEMM solution, with a communication volume lower by a factor of $\sqrt{2}$.

5.3 Experiments

Equipped with an adequate programming model and proper distributions that can be plugged in transparently, we can consider measuring their combined impact on communication volume as well as performance. In this section, we consider two operations that have a (positive definite) symmetric analogue: Section 5.3.1 assesses the pSYMM operation, 5.3.2 is interested in the Cholesky factorization pPOTRF.

5.3.1 Assessing the arithmetic intensity of SYMM

The code designed in Section 5.1 allows us to assess all the mappings discussed in Section 5.2. We study their impact on the AI and performance. We have implemented Algorithm 24 on top of the StarPU [20] task-based runtime system and the NewMadeleine communication back-end, which, combined, support the dynamic detection of collective communications [52]. In an applicative setting, where no synchronization is required between filling and computing the matrices, blocks of B are transferred through broadcasts transparently: the runtime detects them through dependencies in the DAG. In our benchmarking setting, artificial tasks are added to mimic the dependencies that allow a similar detection. We conducted our study in double precision on Platform B. All codes have been assessed with a block size b equal to 256, 512 and 1024, preliminary experiments having shown that these values allow for a good efficiency. Each configuration has been executed five times and we retrieve the median performance. pGEMM and pSYMM algorithms are executed with $(p = 8, q = 7)$ on 56 nodes for 2DBC distributions. 2DSBC $(r = 11)$ and 2DTBC $(c = 7)$ SYMM are executed on 55 and 56 nodes, respectively. 3DSBC $(s = 2, r = 8)$ and 3DTBC $(s = 2, c = 5)$ SYMM are executed on 56 and 60 nodes. The matrix size (m) of A varies while the number of columns of B and C is constant $(n = 8,192)$.

The top plot of Figure 5.6 presents the AI of the STF algorithms discussed in Section 5.2 as defined in Equation (5.1): AI = flop/Q. The total volume of communication Q is retrieved by StarPU. The results show that the expected theoretical ratios of AI from Section 5.2 are successfully achieved in practice.

The bottom of Figure 5.6 presents the resulting per-node performance. The first observation is that the AI gains of SBC and TBC do yield compelling performance benefits on lower size matrices where the AI is not sufficient to ensure a good overlapping between communications and computations. For instance with $m = 110k$, the performance improvement of using TBC (resp. SBC) over the 2DBC layout is approximately 23% (resp. 13%). The proposed

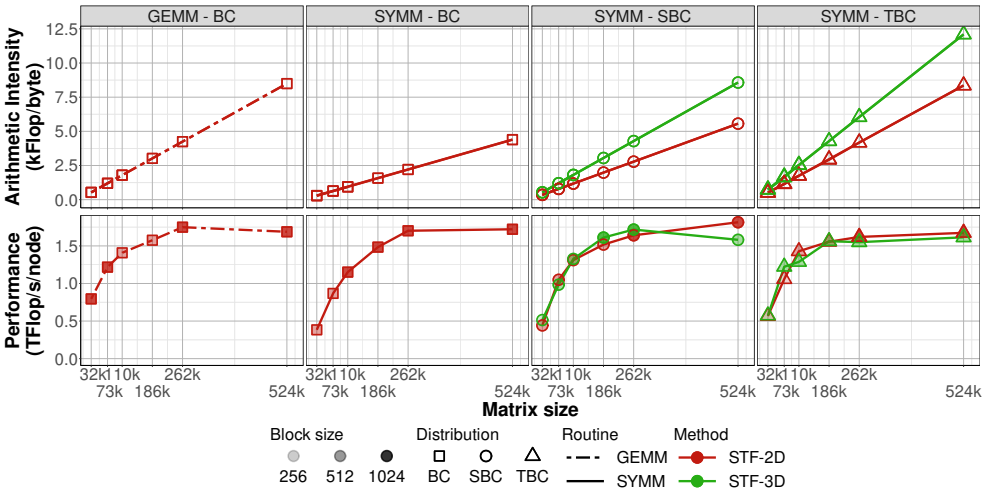


Figure 5.6: AI (top) and per-node performance (bottom) of the STF algorithm of section 5.1 with the various distributions of section 5.2

STF design with SBC and TBC SYMM achieves a performance roughly comparable with 2DBC GEMM, while requiring to store only half of the dominant matrix. The second main observation is that the AI advantage of 3DSBC and TBC does not consistently translate into performance improvement. While 3D symmetric distributions perform well on small problems, they do not outperform the 2D case on larger ones. This is consistent with what is presented in Part I. TBC is more impacted by this performance discrepancy despite having an higher AI than GEMM. A preliminary analysis suggests that this is due to contention in the network that happens because, unlike in BC, in TBC any rank participates in multiple broadcast communications.

Figure 5.7 presents the comparison of the GEMM and SYMM performance of our STF approach with state-of-the-art distributed-memory dense linear algebra libraries proposing a stationary-A implementation of SYMM, namely ScaLAPACK [29] (yellow) and Elemental [95] (black). We also report the GEMM performance of SLATE [57], a potential successor to ScaLAPACK for which a GPU portable stationary-A SYMM was not available. The first observation is the important gap between SYMM and GEMM performance of both ScaLAPACK and Elemental libraries. These results confirm the empirical observation that SYMM state-of-the-art codes achieve a lower performance than their GEMM counterpart. We recall that both these libraries implement SYMM with a 2DBC data distribution. The second main observation is that the STF algorithms proposed in Section 5.1 significantly improve over the stationary-A ScaLAPACK and Elemental SYMM reference implementations. This illustrates the strength of the programming model for designing efficient communication schemes with a high-level expression. Note also that a special care is required to handle rectangular matrices B and C.

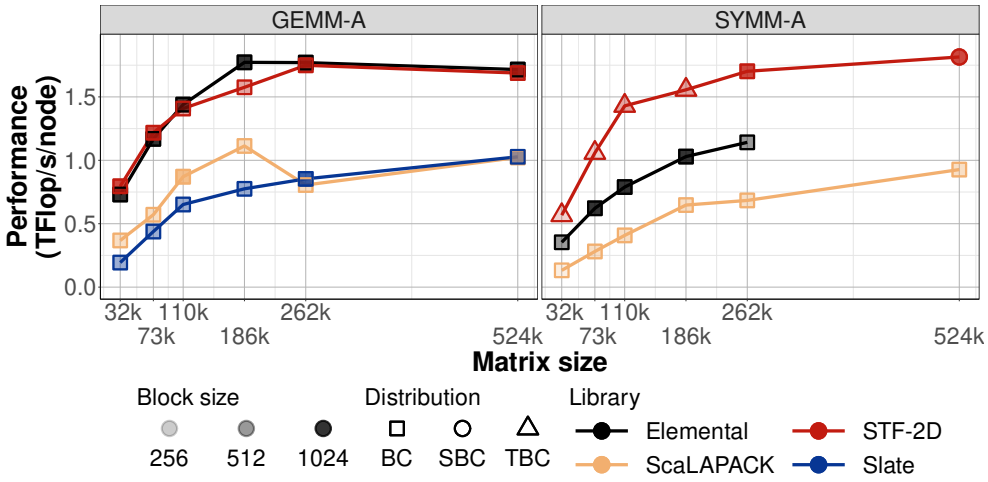


Figure 5.7: Per-node GEMM (left) and SYMM (right) performance of the proposed STF design compared with state-of-the-art libraries.

5.3.2 Improving the Cholesky factorization

The symmetric layouts can readily be used in the Cholesky factorization. The current section makes a detour from symmetric matrix multiplication and aims at presenting an evaluation of the SBC layout for the 3D Cholesky Factorization.

In Figure 5.8 the BC and SBC strong scalability capabilities are compared. The number of layers is fixed to at most two across all configurations. The SBC layout uses $P = hr^2/2$ nodes and as such it rarely matches the nodes count of the 2DBC layout. We have ensured that close-enough configurations were selected to provide a rigorous comparison.

We can observe that the symmetric layout improves overall performance. In the case of the 2D algorithm, the improvement in performance is most visible on the smallest problem which hardly allow computation and communication overlap. Because the 2D algorithm does not require additional, complex mechanisms it may suffer from a reduced overhead compared to the 3D algorithms for this problem. In the case of the 3D algorithms, the Baseline variant performance is remarkably better on both larger problems. A major difference between the BC and SBC layouts is the distribution of diagonal blocks: in the case of SBC, this distribution is evenly spread among the P ranks across the entire block-diagonal of the input matrix. Such an even distribution may mitigate the overhead of broadcasting multiple diagonal blocks without the need for complex replication mechanism.

Further study is required to refine the understanding of the observed improvements.

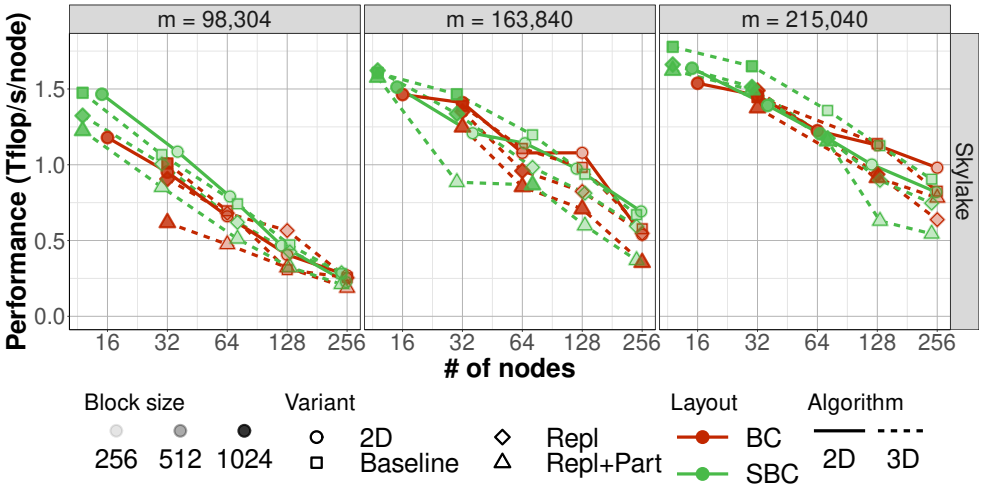


Figure 5.8: Cholesky BC vs. SBC

5.4 Multidimensional scaling and randomized singular value decomposition

Dimension reduction algorithms aim at transforming data from a high-dimensional space into a low-dimensional space while keeping the most meaningful properties of the original data [86, 111]. While the most well-known of these algorithms is certainly the principal component analysis (PCA) [91, 73], MDS [97, 115, 109] may be viewed as its analogue when data items are only known through their respective dissimilarities. As stated by Cox and Cox [48], in a narrow definition, MDS searches for a low dimensional space, usually Euclidean, in which points in the space represent the items, one point representing one item, and such that the distances between the points in the space match, as well as possible, the original dissimilarities.

From a numerical point of view, MDS resorts to processing a singular value decomposition (SVD) [27, 78], as PCA does. However, contrary to PCA, MDS uses an input matrix G built from representing dissimilarities between pairs of items and often referred to as the *Gram matrix*. The dissimilarity between pairs of items being a symmetric relation, the input matrix G is itself symmetric. As a consequence, the SVD of G is also its eigenvalue decomposition (EVD) up to the sign of the eigenvalues. We pursue the presentation with the SVD terminology, following [31, 32, 30].

When dealing with large data sets, performing an SVD may be out of reach due to memory or time to solution constraints. A major step forward has been the design of randomized SVD (RSVD) algorithms [98, 66], a fast and probabilistic approach which ensures the quality of the solution via random projections. Its usage within the MDS (RSVD-MDS) [31, 32, 30, 90] has allowed for processing large data sets while preserving the numerical robustness

of the standard SVD-MDS [90]. The main idea of the RSVD (here discussed when applied to G) is to approximate the column space of the m-by-m matrix G by only a small number n (such that $m \gg n$) of vectors through a linear combination of the columns. From a computational point of view, this step consists in forming an m-by-n random matrix Ω and perform the $Y \leftarrow G\Omega$ matrix product. After computing an orthonormal basis Q of Y, we compute the $Z \leftarrow GQ$ matrix product and, then, a deterministic SVD only needs to be performed on the tall and skinny m-by-n Z matrix. A complete description is provided in Algorithm 2 in [13].

As discussed in details in [13], the $Y \leftarrow G\Omega$ and $Z \leftarrow GQ$ matrix products are the dominant steps of both the RSVD and the whole RSVD-MDS algorithms. In the remainder of this manuscript, we will refer to them as matrix multiplication 1 (MM1) and matrix multiplication 2 (MM2), respectively. They have the exact same dimensions and can both be viewed in terms of the more common BLAS notations as a $C \leftarrow AB$ matrix product, where A is a symmetric m-by-m dense matrix, and B and C are both m-by-n dense matrices, with $m \gg n$. In order to maximize performance, both matrix products may be performed with a GEMM; however this implies that the dissimilarity matrix, initially stored in a symmetric format, has to be converted to full format, thus doubling the initial memory footprint [13].

The central question we aim at addressing in this chapter is whether, in a distributed-memory context, we can store such a symmetric matrix A in symmetric format to use SYMM while achieving comparable performance to GEMM.

5.5 Application to RSVD-MDS

The initial point of the present study was that, in the context of an RSVD-MDS dimension reduction algorithm [13], it was necessary to trade off performance, with 2DBC GEMM, with memory, with 2DBC SYMM, during the dominant steps (denoted MM1 and MM2 steps in Section 5.4) of the RSVD algorithm. The central question raised in Section 5.4 was whether we could store only half of the symmetric matrix through SYMM while achieving a performance on par with that of GEMM.

As discussed in Section 5.4, MDS computes a so-called Gram matrix G from an input matrix representing dissimilarities between pairs of items. In the context of our metabarcoding target application, items are diatoms collected in Geneva and dissimilarities between them are their genetic distances. The dataset [2] used as input for the MDS, fully described in [13], is a $10^6 \times 10^6$ matrix of genetic distances between sequences. We may consider either part of the data (*S1*, *S2*, *S3*, *S4*), leading to a matrix of reduced dimension, of the whole data set (*S5*). Table 5.2 presents the matrix size (m) and parallel

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

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

Table 5.2: Samples names, matrix size m, number of nodes P (and of MPI processes), parameters (p, q) for 2DBC mappings, and parameter c for TBC mapping.

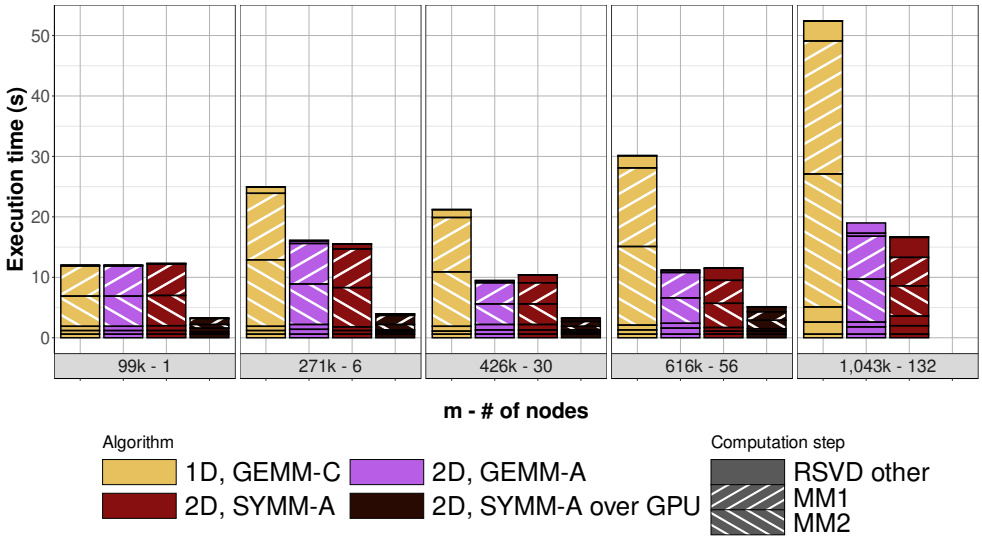


Figure 5.9: Execution time (s) of the RSVD with $n = 1,000$. MM1 and MM2 are assessed with TBC layouts for SYMM. Yellow bars denote 1DBC GEMM with a stationary-C scheme. Black bars denote TBC SYMM with GPUs turned on. GPU accelerated executions on 132 nodes was not possible because of a *Quality of Service* (QoS) limitation on Jean Zay supercomputer (no more than 512 GPUs per job). Five test cases are assessed, ranging from *S1* on 1 node (denoted 99k-1) to *S5* on 132 nodes (denoted 1,034k-132).

setup associated with each sample. The whole MDS algorithm consists of the computation of the Gram matrix G followed by an RSVD (which includes MM1 and MM2 steps). All the tests of this section are performed in single precision and a number $n = 1,000$ of columns for B and C, consistently with [13]. We conducted the study on Platform C. Intel MKL v. 19.0.4 provides the implementation of single-core kernels.

Figure 5.9 illustrates the impact on performance of the present study. The original code from [13] had been designed following the programming model of [114, 6] in which task mapping was inferred from the data mapping of the `RW-`

accessed blocks. This means that it relied on a stationary-C scheme, in which case the optimum GEMM mapping is 1DBC (yellow bars in Figure 5.9) as it is both a stationary-A and -C variant. The application of the new programming model presented in Part I allows us to employ a 2DBC stationary-A variant (l.1 of Table 5.1 p.102 and purple bars in Figure 5.9 after this line number). The significant improvement shows the interest of the new programming model when dealing with a task-based approach. Figure 5.9 furthermore shows that it is possible to store only half of the symmetric matrix through a TBC SYMM (l. 4 in Table 5.1 and red bars in Figure 5.9) while achieving a performance competitive with (2DBC) GEMM, which positively answers the question that originally motivated this work. As this observation applies to both MM1 and MM2 matrix multiplication steps and since they altogether dominate the RSVD algorithm, this directly translates into a significant improvement for the entire RSVD.

As recalled above, MDS requires computing the Gram matrix G before applying the RSVD itself. We also redesigned this step, which is mainly a reduction, using the `RANK_REDUX` access mode introduced in the extended STF programming model. The previous implementation of this step relied on six intermediary passes where elements are successively reduced on a per-block, per-rank-column then per-rank-row basis to obtain a vector d^2_+ and a scalar d^2_{++} [13]. These passes relied on multiple temporary buffers explicitly provided by the programmer. With our extended programming model, we can leverage the `lassq` routine provided by LAPACK and simply submit a single pass over the data to obtain d^2_+. d^2_{++} is then obtained by submitting a second last pass over d^2_+. The accumulation in temporary buffers is handled by the runtime system which makes it possible to maintain a compact expression on the library's side. This also lets the runtime system opportunistically process reductions which can assist the improvement in performance we have observed.

We have not reported detailed figures on the matter, however, the execution time of the entire RSVD-MDS algorithm (Gram computation and RSVD altogether) on the whole data set (*S5*) using 132 nodes (5,280 CPU cores) has been reduced from 70 seconds, with the original code of [13], to 25 seconds, with TBC SYMM together with the new Gram step design, while using about half the memory.

We complete the study with the illustration of the capability of task-based codes to exploit heterogeneous architectures. Without any change in the code (other than providing the Cuda cuBLAS kernels of single-GPU kernels), the runtime system may execute tasks on CPU or GPU [20]. A subset of Jean Zay nodes have the exact same characteristics as described above in CPU-only case but are furthermore enhanced with four NVIDIA Tesla V100 SXM2 GPUs (32 GB). Cuda v. 10.1.2 is used. Black bars in Figure 5.9 correspond to the execution of the RSVD with GPUs enabled, relying on TBC SYMM for the matrix multiplication. The results show a considerable improvement over the CPU-only case in spite of the relatively low number of columns ($n = 1,000$

only) of B and C, a typical set up for the application.

5.6 Concluding remarks

We experimentally confirmed that reference distributed-memory libraries achieve a lower performance with SYMM than with GEMM. We showed that an efficient design of the communication schemes can significantly alleviate this gap. Moreover we showed that part of the gap is explained by a lower AI of 2DBC SYMM compared to GEMM (by a factor of 2). We considered two alternative data distributions SBC and TBC: SBC is a direct adaptation to the matrix multiplication case of a study of the Cholesky factorization [25] and TBC is a distributed-memory (and even a parallel) adaptation of the ideas behind TBS [26], a sequential out-of-core algorithm. We proved that SBC and TBC improve the AI of SYMM by a factor of $\sqrt{2}$ and 2, respectively, thus in particular equaling that of 2DBC GEMM for the latter one. In the case where we allow SYMM to store an amount of memory equivalent to a full matrix as 2DBC GEMM does, we furthermore showed that 3D TBC with $s = 2$ slices achieves a higher AI than 2DBC GEMM by a factor of $\sqrt{2}$. Our experimental study showed that the improvement of the AI translates into a compelling performance enhancement, up to the point of roughly matching GEMM performance. However, the highest AI does not always translate into the best performance.

The resulting code has been integrated in a metabarcoding application. It consists in a MDS dimension reduction algorithm based on RSVD whose main computational steps are two dense matrix multiplications involving a symmetric input matrix. While one had to trade-off between performance, with GEMM, or memory, with SYMM, we showed that, altogether, the proposed STF design and the new TBC distribution now achieve a performance competitive with GEMM. This study also showed that algorithms involving very irregular data and task distributions can now be implemented with a code easy to write, read and maintain thanks to the latest developments on the scalability of the STF model, while ensuring a competitive performance.

Chapter 6

Conclusion

This thesis is focused on features in the STF programming model that are helpful to productively write linear algebra routines designed for large-scale architectures. In this chapter, the main contributions are recalled and future lines of research are presented.

Contributions

Being able to target every architecture with a single code is a peak achievement of any programmer. This goal does not simply entail the execution of the code but rather the realization of certain performance metrics. As the HPC community is faced with a plethora of heterogeneous architectures, such an achievement would result in less time being devoted to adapting or rewriting code and more time would be allocated researching cutting-edge algorithms tailored to real-world numerical properties. In this thesis we have contributed toward the long-lasting horizon of offering universally portable performance and effective production of algorithms.

Our principal contribution is the extension of the STF programming model described in Part I to better accommodate the mechanisms that are commonly found in scalable (linear algebra) algorithms. The STF programming model is already appreciated for its productivity yet its use in writing scalable algorithms has not permeated linear algebra packages. In this work, we have identified four features that help users diminish the amount of mechanisms they need to exhaustively set up by delegating them to the runtime system. One of them is the ability to map tasks to any rank available. Consequently a runtime system should integrate a data management component that manages data ownership and the coherency of data trans-

fers over distributed memories. Another cornerstone feature is the ability to dynamically detect collective communication at runtime: aggregating point-to-point data transfers into collective operations enforces the design of scalable algorithms. The presented work stands on these two features that were achieved in previous works. We have completed the programming model with two lacking mechanisms: **distributed memory reduction patterns** presented in Chapter 3 and **data write replication** introduced in Chapter 4. These elements are present in linear algebra algorithms because the manipulation of matrices often involve 1) commutative and associative operations 2) multiple transfers of submatrices on which an operation can be applied pre- or postliminary. We have specified these two mechanisms with adequate details into access modes (`RANK_REDUX`,`SAME`) and utility functions (`(all)redux_submit`,`set_alternative_source`). This specification has led to the integration of the proposed enhancements within the StarPU runtime system - distributed memory reduction patterns are already released as part of StarPU 1.4.0 however data write replication remains under (consolidating) development as of writing. These improvements on the side of the runtime system allowed us to develop dense linear algebra routines as part of the `qr_mumps` and the Chameleon packages. The expression we have provided maintains the separation of concerns that is much appreciated to better incorporate progress from scheduling and runtime experts. For both pGEMM and pPOTRF we have been able to provide an expression that describes the behaviors of scalable algorithms in a single versatile code; we are not limited by the default "owner-computes" strategy when describing the reduction patterns. We plan on releasing these routines as they have proved to deliver performances on par or even outperfoming state-of-the-art libraries such as Slate or Elemental. Performance has been assessed for pGEMM, pSYMM and pPOTRF and further progress can be considered that we detail in the Perspectives section. We have benchmarked libraries on up to 256 modern computing nodes which amount to 1.3 and 1.1 PFlop/s (`Rmax`) on platforms A and B, a noteworthy amount of computing power.

Our work did not only explore the capabilities of sequential-like code: in Part II we have leveraged the productive expression permitted by the extended STF programming model into the Fast Methods for Randomized numerical linear algebra (FMR) package. This linear algebra package is leveraged by the Diodon [1] library. This library is used by domain scientists to perform data analysis. It relies on the omnipresent matrix-matrix multiplication to obtain random projections that form the foundation of methods such as MDS. For the Diodon library Chapter 5 has demonstrated the importance of the extended programming model in simplifying the use of scalable mechanisms while taking the symmetry of the input matrix into consideration. By doing so the memory consumption of the operation has been halved and the inte-

[1] diodon git repository `https://gitlab.inria.fr/diodon/cppdiodon`

gration of the proper symmetric matrix layouts reduced the communication volume while maintaining scalable performance. We can easily leverage the stationary-A schemes to better deal with the dimensions of the random projections. The Gram matrix computation also exploits reduction operations that we have efficiently and compactly delegated to the runtime system. With the overall improvement of the MDS algorithm we have been able to maintain a satisfactory scalability on homogeneous and heterogeneous platforms processing actual scientific datasets. The advantageous symmetric layouts have been transposed to our symmetric (positive definite) matrix decompositions – for which they were originally designed: without modifying our 3D POTRF code, we were able to compare the SBC layout with its 2DBC counterpart and observe its relative superiority.

We have demonstrated the applicability of the STF programming model in expressing state-of-the-art scalable linear algebra algorithms. While our work was only interested in two prominent routines, the mechanisms we have put forth are pervasive in linear algebra. Thus our focus on simple sequential-like code can be adapted to take other structural or numerical matrix properties into account while still being portable across architectures.

Perspectives

Applicability of the STF, dense linear algebra and beyond By incorporating tools in a runtime system to design algorithms we have made it easier to consider adapting other linear algebra operations to "communication-avoiding" techniques. The most immediate ones are TRSM routines that can be adapted for 3D logical grids as well as dense factorization algorithms that require pivoting. Tournament pivoting can notably be viewed as a reduction pattern over pivots and a column of blocks. Other pivoting strategies may be explored. Providing a single code that can readily switch between pivoting strategies would be an interesting endeavor – whether through the STF or another task-based programming model. Providing TRSM, POTRF and GETRF routines through the STF programming model would be a first step toward state-of-the-art scalable dense direct solvers. The usual techniques found in sparse linear algebra could also benefit from the programming model we have set up. Algorithms such as fan-both sparse factorizations [18] or supernodal sparse factorization [101] might be expressed in a single versatile routine that makes it possible to span different processes configuration through a restricted set of parameters. While our main interest has been dense numerical linear algebra, the scope of our work encompasses the algo-

rithms the extended programming model is capable to express. It is entirely possible to use the STF programming model to consider other applicative domains. Text manipulation is more omnipresent than floating-point operations in some computer science domains and programming models like `MapReduce` often address this need.

Scalable algorithms design The 3D algorithms we have evaluated in this work have obtained relatively limited results: their improvement over 2D variants was either marginal or happening in specific configurations of problem sizes and number of cores. We believe that further investigation should be pursued to evaluate these algorithms. Specifically, the platforms used to perform the evaluation should be extended to exascale and/or heterogeneous machines to better challenge the hypotheses that support their designs. To this end, we plan on making our implementations of 3D algorithms readily available for further studies. For the GPU case, our approach may require the use of expert schedulers and/or a fine-grained submission sequence which necessitates evaluation. It is apparent that scheduling communications (in what order should data transfers be processed and what pattern should a collective operation takes) is essential and should be explored further. Consequently the expressions we have provided might be further modified to test and compare communications policies. These policies should be refined and designed from theoretical considerations that need to be consolidated in the case of asynchronous communications.

Delegating the reduction operations We have implemented the reduction operations through the insertion of user-provided tasks on the side of the runtime system. This is profitable to the programmer because they do not need to implement the submission of this subgraph themselves; they can also benefit from the automatic submission for the reduce pattern. This design could be modified by delegating the totality of the reduction operations to a communication backend. This will have an impact on the scheduling of multiple reduction patterns because the worker handling the communication engine will have to execute all of the (multiple) reduction tasks sequentially instead of them being scheduled concurrently over multiple cores. Delegating the reduction patterns to a communication backend may however be a sustainable way to strengthen separation of concerns between shared-memory data management and distributed-memory data transfers.

Runtime systems and submission loops On the side of the runtime system, further perspectives are worthy of consideration. First, when submitting distributed memory DAGS, some of the mechanisms we have proposed may be delivered transparently by the runtime system based on distributed memory protocols. These protocols could be implemented by the user – with

some default ones provided by the runtime system – to toggle the use of data write replication or allreduce reduction. Additional protocols could integrate memory constraints to alleviate the need for a window mechanism we have showcased in the case of pGEMM. Second, the efficient submission of tasks could be explored further by giving more directions to the compiler when it comes to loops involved in the submission phase. The pruning of tasks is typically controlled by verifying conditions and it is naturally prone to create unnecessary branches when a CPU evaluates it. Branchless techniques combined with suitable iterators that can be dynamically instantiated based on problem dimensions and hardware resources may help to squeeze more computing power out of supercomputers. This would mostly apply to regular workload found in dense linear algebra.

Software releases Some engineering work remains to be done to ensure our contributions get properly released as parts of the StarPU and `qr_mumps` packages. This work should mainly focus on removing the more experimental aspects of the routines that have been developed to make their use as straightforward and documented as possible.

Overall we believe that being able to consider sequential-like code to obtain satisfactory performance on petascale machines opens doors for plenty of theoretical, exploratory and applicative lines of research.

Scientific communications

Journal articles

- Emmanuel Agullo et al. “Task-Based Parallel Programming for Scalable Matrix Product Algorithms”. In: *ACM Transactions on Mathematical Software* 49.2 (June 2023). ISSN: 0098-3500. DOI: 10.1145/3583560. eprint: https://hal.science/hal-03936659. URL: https://doi.org/10.1145/3583560

Conference proceedings

- Emmanuel Agullo et al. “On the Arithmetic Intensity of Distributed-Memory Dense Matrix Multiplication Involving a Symmetric Input Matrix (SYMM)”. in: *IPDPS 2023 - 37th International Parallel and Distributed Processing Symposium*. Ed. by IEEE. IEEE. St. Petersburg, FL, United States, May 2023, pp. 357–367. DOI: 10.1109/IPDPS54959.2023.00044. URL: https://inria.hal.science/hal-04093162

Talks in conferences

- Emmanuel Agullo et al. “Task-Based Parallel Programming for Scalable Algorithms : Application to Matrix Multiplication”. SIAM Parallel Processing. 2022. URL: https://meetings.siam.org/sess/dsp_talk.cfm?p=117562
- Emmanuel Agullo et al. “Task-Based Parallel Programming for Scalable Algorithms : Application to Matrix Multiplication”. Sparse Days. 2022. URL: https://sparsedays.cerfacs.fr/wp-content/uploads/sites/72/2022/10/Jego.pdf

Bibliography

[1] Ramesh C Agarwal, Susanne M Balle, Fred G Gustavson, Mahesh Joshi, and Prasad Palkar. "A three- dimensional approach to parallel matrix multiplication". In: *IBM Journal of Research and Development* 39.5 (1995), pp. 575–582. URL: http://citeseerx.ist.psu.edu/viewdoc/download?%20doi=10.1.1.120.4575&rep=rep1&type=pdf.

[2] Ramesh C. Agarwal, Fred G. Gustavson, and Mohammad Zubair. "A High-Performance Matrix-Multiplication Algorithm on a Distributed-Memory Parallel Computer, Using Overlapped Communication". In: *IBM J. Res. Dev.* 38.6 (Nov. 1994), pp. 673–681. ISSN: 0018-8646. DOI: 10.1147/rd.386.0673. URL: https://doi.org/10.1147/rd.386.0673.

[3] Emmanuel Agullo, Cedric Augonnet, Jack Dongarra, Mathieu Faverge, Hatem Ltaief, Samuel Thibault, and Stanimire Tomov. "QR Factorization on a Multicore Node Enhanced with Multiple GPU Accelerators". In: *Proceedings of 25th International Parallel and Distributed Processing Symposium (IPDPS'11)*. 2011, pp. 932–943. DOI: http://doi.ieeecomputersociety.org/10.1109/IPDPS.2011.90.

[4] Emmanuel Agullo, Cedric Augonnet, Jack Dongarra, Hatem Ltaief, R. Namyst, Samuel Thibault, and Stanimire Tomov. "A Hybridization Methodology for High-Performance Linear Algebra Software for GPUs". In: *in GPU Computing Gems, Jade Edition* 2 (2011), pp. 473–484.

[5] Emmanuel Agullo, Olivier Aumage, Berenger Bramas, Olivier Coulaud, and Samuel Pitoiset. "Bridging the Gap Between OpenMP and Task-Based Runtime Systems for the Fast Multipole Method". In: *IEEE Transactions on Parallel and Distributed Systems* 28.10 (2017), pp. 2794–2807. DOI: 10.1109/TPDS.2017.2697857.

[6] Emmanuel Agullo, Olivier Aumage, Mathieu Faverge, Nathalie Furmento, Florent Pruvost, Marc Sergent, and Samuel Thibault. "Achieving High Performance on Supercomputers with a Sequential Task-based Programming Model". In: *IEEE Transactions on Parallel and Distributed Systems* (2017). DOI: 10.1109/TPDS.2017.2766064. URL: https://hal.inria.fr/hal-01618526.

[7] Emmanuel Agullo, George Bosilca, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. "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*. Ed. by 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. Cham: Springer International Publishing, 2017, pp. 175–186. ISBN: 978-3-319-58943-5. DOI: 10.1007/978-3-319-58943-5_14. eprint: https://hal.archives-ouvertes.fr/hal-01337748. URL: http://dx.doi.org/10.1007/978-3-319-58943-5_14.

[8] Emmanuel Agullo, Alfredo Buttari, Olivier Coulaud, Lionel Eyraud-Dubois, Mathieu Faverge, Alain Franc, Abdou Guermouche, Antoine Jego, Romain Peressoni, and Florent Pruvost. "On the Arithmetic Intensity of Distributed-Memory Dense Matrix Multiplication Involving a Symmetric Input Matrix (SYMM)". In: *IPDPS 2023 - 37th International Parallel and Distributed Processing Symposium*. Ed. by IEEE. IEEE. St. Petersburg, FL, United States, May 2023, pp. 357–367. DOI: 10.1109/IPDPS54959.2023.00044. URL: https://inria.hal.science/hal-04093162.

[9] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Julien Herrmann, and Antoine Jego. "Task-Based Parallel Programming for Scalable Algorithms : Application to Matrix Multiplication". SIAM Parallel Processing. 2022. URL: https://meetings.siam.org/sess/dsp_talk.cfm?p=117562.

[10] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Julien Herrmann, and Antoine Jego. "Task-Based Parallel Programming for Scalable Algorithms : Application to Matrix Multiplication". Sparse Days. 2022. URL: https://sparsedays.cerfacs.fr/wp-content/uploads/sites/72/2022/10/Jego.pdf.

[11] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, Julien Herrmann, and Antoine Jego. "Task-Based Parallel Programming for Scalable Matrix Product Algorithms". In: *ACM Transactions on Mathematical Software* 49.2 (June 2023). ISSN: 0098-3500. DOI: 10.1145/3583560.

eprint: https://hal.science/hal-03936659. URL: https://doi.org/10.1145/3583560.

[12] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. "Implementing Multifrontal Sparse Solvers for Multicore Architectures with Sequential Task Flow Runtime Systems". In: *ACM Trans. Math. Softw.* 43.2 (Aug. 2016), 13:1–13:22. ISSN: 0098-3500. DOI: 10.1145/2898348. eprint: https://hal.inria.fr/hal-01333645. URL: http://doi.acm.org/10.1145/2898348.

[13] Emmanuel Agullo, Olivier Coulaud, Alexandre Denis, Mathieu Faverge, Alain Franc, Jean-Marc Frigerio, Nathalie Furmento, Adrien Guilbaud, Emmanuel Jeannot, Romain Peressoni, Florent Pruvost, and Samuel Thibault. *Task-based randomized singular value decomposition and multidimensional scaling*. Research Report RR-9482. Inria Bordeaux - Sud Ouest ; Inrae - BioGeCo, Sept. 2022, p. 37. URL: https://hal.inria.fr/hal-03773985.

[14] Emmanuel Agullo, Jim Demmel, Jack Dongarra, Bilel Hadri, Jakub Kurzak, Julien Langou, Hatem Ltaief, Piotr Luszczek, and Stanimire Tomov. "Numerical linear algebra on emerging architectures: The PLASMA and MAGMA projects". In: *Journal of Physics: Conference Series* 180.1 (2009), p. 012037. URL: http://stacks.iop.org/1742-6596/180/i=1/a=012037.

[15] Emmanuel Agullo, Luc Giraud, Abdou Guermouche, Stojce Nakov, and Jean Roman. "Pipelining the CG Solver Over a Runtime System". In: *GPU Technology Conference*. NVIIDA. San Jose, United States, Mar. 2013. URL: https://inria.hal.science/hal-00934948.

[16] AMD. *AMD ROCm documentation*. 2023. URL: https://rocm.docs.amd.com/en/latest/.

[17] Edward Anderson, Zhaojun Bai, Christian Bischof, Susan Blackford, James Demmel, Jack Dongarra, Jeremy Du Croz, Anne Greenbaum, Sven Hammarling, Alan McKenney, and Danny Sorensen. *LAPACK Users' Guide*. Third. Philadelphia, PA: Society for Industrial and Applied Mathematics, 1999. ISBN: 0-89871-447-8 (paperback).

[18] Cleve Ashcraft. "The Fan-Both Family of Column-Based Distributed Cholesky Factorization Algorithms". In: *Graph Theory and Sparse Matrix Computation*. Ed. by Alan George, John R. Gilbert, and Joseph W. H. Liu. New York, NY: Springer New York, 1993, pp. 159–190. ISBN: 978-1-4613-8369-7.

[19] "Cédric" Augonnet. "Scheduling Tasks over Multicore machines enhanced with Accelerators: a Runtime System's Perspective". PhD thesis. Université de Bordeaux, Sept. 2011.

[20] Cedric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Concurrency and Computation: Practice and Experience, Special Issue: Euro-Par 2009* 23 (2 Feb. 2011), pp. 187–198. DOI: 10.1002/cpe.1631. URL: http://hal.inria.fr/inria-00550877.

[21] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. "Minimizing Communication in Numerical Linear Algebra". In: *SIAM Journal on Matrix Analysis and Applications* 32.3 (2011), pp. 866–901. DOI: 10.1137/090769156. eprint: http://dx.doi.org/10.1137/090769156. URL: http://dx.doi.org/10.1137/090769156.

[22] Protonu Basu, Anand Venkat, Mary Hall, Samuel Williams, Brian Van Straalen, and Leonid Oliker. "Compiler generation and autotuning of communication-avoiding operators for geometric multigrid". In: *20th Annual International Conference on High Performance Computing*. 2013, pp. 452–461. DOI: 10.1109/HiPC.2013.6799131.

[23] Michael Bauer, Wonchan Lee, Elliott Slaughter, Zhihao Jia, Mario Di Renzo, Manolis Papadakis, Galen Shipman, Patrick McCormick, Michael Garland, and Alex Aiken. "Scaling Implicit Parallelism via Dynamic Control Replication". In: *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*. PPoPP '21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 105–118. ISBN: 9781450382946. DOI: 10.1145/3437801.3441587. URL: https://doi.org/10.1145/3437801.3441587.

[24] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. "Legion: expressing locality and independence with logical regions". In: *SC Conference on High Performance Computing Networking, Storage and Analysis, SC '12, Salt Lake City, UT, USA - November 11 - 15, 2012*. 2012, p. 66. URL: http://dl.acm.org/citation.cfm?id=2389086.

[25] Olivier Beaumont, Philippe Duchon, Lionel Eyraud-Dubois, Julien Langou, and Mathieu Vérité. "Symmetric Block-Cyclic Distribution: Fewer Communications Leads to Faster Dense Cholesky Factorization". In: *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. SC '22. Dallas, Texas: IEEE Press, 2022. ISBN: 9784665454445.

[26] Olivier Beaumont, Lionel Eyraud-Dubois, Mathieu Vérité, and Julien Langou. "I/O-Optimal Algorithms for Symmetric Linear Algebra Kernels". In: *ACM Symposium on Parallelism in Algorithms and Architectures*. Association for Computing Machinery : SIGACT, SIGARCH. Philadelphie, United States, July 2022. URL: https://hal.inria.fr/hal-03580531.

[27] E Beltrami. In: *Giornale di Matematiche ad Uso degli Studenti Delle Universita* (1873).

[28] A. J. Bernstein. "Analysis of Programs for Parallel Processing". In: *IEEE Transactions on Electronic Computers* EC-15.5 (1966), pp. 757–763. DOI: 10.1109/PGEC.1966.264565.

[29] L. Susan Blackford, Jaeyoung Choi, Andrew J. Cleary, Eduardo F. D'Azevedo, James Demmel, Inderjit S. Dhillon, Jack J. Dongarra, Sven Hammarling, Greg Henry, Antoine Petitet, Ken Stanley, David W. Walker, and R. Clinton Whaley. "ScaLAPACK: A Linear Algebra Library for Message-Passing Computers". In: *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing, PPSC 1997, Hyatt Regency Minneapolis on Nicollel Mall Hotel, Minneapolis, Minnesota, USA, March 14-17, 1997*. SIAM, 1997.

[30] Pierre Blanchard. "Fast hierarchical algorithms for the low-rank approximation of matrices with applications to materials physics, geostatistics and data analysis". Theses. Université de Bordeaux, Feb. 2017. URL: https://tel.archives-ouvertes.fr/tel-01534930.

[31] Pierre Blanchard, Philippe Chaumeil, Jean-Marc Frigerio, Frédéric Rimet, Franck Salin, Sylvie Thérond, Olivier Coulaud, and Alain Franc. "A geometric view of Biodiversity: scaling to metagenomics". In: *arXiv preprint arXiv:1803.02272* (2018).

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

[33] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. "Cilk: An Efficient Multithreaded Runtime System". In: *J. Parallel Distrib. Comput.* 37.1 (1996), pp. 55–69.

[34] The OpenMP architecture review board. *OpenMP 4.0 Complete specifications*. 2013.

[35] The OpenMP architecture review board. *OpenMP OpenMP Fortran Application Program Interface 1.0*. 1997.

[36] Lionel Boillot, George Bosilca, Emmanuel Agullo, and Henri Calandra. "Task-Based Programming for Seismic Imaging: Preliminary Results". In: *2014 IEEE Intl Conf on High Performance Computing and Communications, 2014 IEEE 6th Intl Symp on Cyberspace Safety and Security, 2014 IEEE 11th Intl Conf on Embedded Software and Syst (HPCC,CSS,ICESS)*. 2014, pp. 1259–1266. DOI: 10.1109/HPCC.2014.205.

[37] G. Bosilca, R.J. Harrison, T. Herault, M.M. Javanmard, P. Nookala, and E.F. Valeev. "The Template Task Graph (TTG) - an emerging practical dataflow programming paradigm for scientific simulation at extreme scale". In: *2020 IEEE/ACM Fifth International Workshop on Extreme Scale Programming Models and Middleware (ESPM2)*. 2020, pp. 1–7. DOI: 10.1109/ESPM251964.2020.00011.

[38] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. "PaRSEC: Exploiting Heterogeneity to Enhance Scalability". In: *Computing in Science and Engineering* 15.6 (2013), pp. 36–45. DOI: 10.1109/MCSE.2013.98. URL: http://dx.doi.org/10.1109/MCSE.2013.98.

[39] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Azzam Haidar, Thomas Hérault, Jakub Kurzak, Julien Langou, Pierre Lemarinier, Hatem Ltaief, Piotr Luszczek, Asym Yarkhan, and Jack J. Dongarra. "Distibuted Dense Numerical Linear Algebra Algorithms on massively parallel architectures: DPLASMA". In: *Proceedings of the 25th IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011*. Anchorage, United States, May 2011, pp. 1432–1441. URL: https://hal.inria.fr/hal-00809680.

[40] Alfredo Buttari, Julien Langou, Jakub Kurzak, and Jack Dongarra. "A class of parallel tiled linear algebra algorithms for multicore architectures". In: *Parallel Comput.* 35 (1 Jan. 2009), pp. 38–53. ISSN: 0167-8191. DOI: 10.1016/j.parco.2008.10.002. eprint: https://hal.archives-ouvertes.fr/hal-02420965. URL: http://dl.acm.org/citation.cfm?id=1486274.1486415.

[41] Lynn Elliot Cannon. "A Cellular Computer to Implement the Kalman Filter Algorithm". AAI7010025. PhD thesis. USA: Montana State University, 1969.

[42] Chongxiao Cao, Thomas Herault, George Bosilca, and Jack Dongarra. "Design for a Soft Error Resilient Dynamic Task-Based Runtime". In: *2015 IEEE International Parallel and Distributed Processing Symposium*. 2015, pp. 765–774. DOI: 10.1109/IPDPS.2015.81.

[43] William W. Carlson, Jesse M. Draper, David E. Culler, Katherine A. Yelick, Eugene D. Brooks, and Karen H. Warren. "Introduction to UPC and Language Specification". In: 2000.

[44] Erin Carson, Nicholas Knight, and James Demmel. "Avoiding Communication in Nonsymmetric Lanczos-Based Krylov Subspace Methods". In: *SIAM Journal on Scientific Computing* 35.5 (2013), S42–S61. DOI: 10.1137/120881191. eprint: https://doi.org/10.1137/120881191. URL: https://doi.org/10.1137/120881191.

[45] Erin C. Carson. "The Adaptive s-Step Conjugate Gradient Method". In: *SIAM Journal on Matrix Analysis and Applications* 39.3 (2018), pp. 1318–1338. DOI: 10.1137/16M1107942. eprint: https://doi.org/10.1137/16M1107942. URL: https://doi.org/10.1137/16M1107942.

[46] Ernie Chan, Marcel Heimlich, Avi Purkayastha, and Robert van de Geijn. "Collective communication: theory, practice, and experience". In: *Concurrency and Computation: Practice and Experience* 19.13 (2007), pp. 1749–1783. DOI: https://doi.org/10.1002/cpe.1206. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.1206. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.1206.

[47] NVIDIA Corporation. *NVIDIA CUDA Best Practices Guide Version 4.0*. 2011.

[48] T.F. Cox and M. A. A. Cox. *Multidimensional Scaling - Second edition*. Vol. 88. Monographs on Statistics and Applied Probability. Chapman & al., 2001.

[49] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. "Communication-optimal Parallel and Sequential QR and LU Factorizations". In: *SIAM J. Sci. Comput.* 34.1 (Feb. 2012), pp. 206–239. ISSN: 1064-8275. URL: http://dx.doi.org/10.1137/080731992.

[50] James Demmel, Mark Hoemmen, Marghoob Mohiyuddin, and Katherine Yelick. "Avoiding communication in sparse matrix computations". In: *2008 IEEE International Symposium on Parallel and Distributed Processing*. 2008, pp. 1–12. DOI: 10.1109/IPDPS.2008.4536305.

[51] Alexandre Denis. "Scalability of the NewMadeleine Communication Library for Large Numbers of MPI Point-to-Point Requests". In: *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 2019, pp. 371–380. DOI: 10.1109/CCGRID.2019.00051.

[52] Alexandre Denis, Emmanuel Jeannot, Philippe Swartvagher, and Samuel Thibault. "Using Dynamic Broadcasts to Improve Task-Based Runtime Performances". In: *Euro-Par 2020: Parallel Processing*. Ed. by Maciej Malawski and Krzysztof Rzadca. Cham: Springer International Publishing, 2020, pp. 443–457. ISBN: 978-3-030-57675-2.

[53] R.H. Dennard, F.H. Gaensslen, Hwa-Nien Yu, V.L. Rideout, E. Bassous, and A.R. LeBlanc. "Design of ion-implanted MOSFET's with very small physical dimensions". In: *IEEE Journal of Solid-State Circuits* 9.5 (1974), pp. 256–268. DOI: 10.1109/JSSC.1974.1050511.

[54] ALEJANDRO DURAN, EDUARD AYGUADÉ, ROSA M. BADIA, JESÚS LABARTA, LUIS MARTINELL, XAVIER MARTORELL, and JUDIT PLANAS. "OmpSs: A PROPOSAL FOR PROGRAMMING HETEROGENEOUS MULTI-CORE ARCHITECTURES". In: *Parallel Processing Letters* 21.02 (2011), pp. 173–193. DOI: 10.1142/S0129626411000151. eprint: https://doi.org/10.1142/S0129626411000151. URL: https://doi.org/10.1142/S0129626411000151.

[55] Essadki, Mohamed, Jung, Jonathan, Larat, Adam, Pelletier, Milan, and Perrier, Vincent. "A Task-Driven Implementation of a Simple Numerical Solver for Hyperbolic Conservation Laws". In: *ESAIM: ProcS* 63 (2018), pp. 228–247. DOI: 10.1051/proc/201863228. URL: https://doi.org/10.1051/proc/201863228.

[56] Mathieu Faverge, Nathalie Furmento, Abdou Guermouche, Gwenolé Lucas, Raymond Namyst, Samuel Thibault, and Pierre-André Wacrenier. "Programming heterogeneous architectures using hierarchical tasks". In: *Concurrency and Computation: Practice and Experience* n/a.n/a (2023), e7811. DOI: https://doi.org/10.1002/cpe.7811. eprint: https://onlinelibrary.wiley.com/doi/pdf/10.1002/cpe.7811. URL: https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.7811.

[57] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. "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. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356223. URL: https://doi.org/10.1145/3295500.3356223.

[58] Robert van de Geijn and Jerrell Watts. "SUMMA: scalable universal matrix multiplication algorithm". In: *CONCURRENCY: PRACTICE AND EXPERIENCE* 9.4 (1997), pp. 255–274. URL: http://www.netlib.org/lapack/lawnspdf/lawn96.pdf.

[59] Robert A. van de Geijn. *Using PLAPACK - parallel linear algebra package*. MIT Press, 1997. ISBN: 978-0-262-72026-7.

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

[61] P. Ghysels, T. J. Ashby, K. Meerbergen, and W. Vanroose. "Hiding Global Communication Latency in the GMRES Algorithm on Massively Parallel Machines". In: *SIAM Journal on Scientific Computing* 35.1 (2013), pp. C48–C71. DOI: 10.1137/12086563X. eprint: https://doi.org/10.1137/12086563X. URL: https://doi.org/10.1137/12086563X.

[62] Maxime Gonthier, Loris Marchal, and Samuel Thibault. "Memory-Aware Scheduling of Tasks Sharing Data on Multiple GPUs with Dynamic Runtime Systems". In: *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 2022, pp. 694–704. DOI: 10.1109/IPDPS53621.2022.00073.

[63] Maxime Gonthier, Loris Marchal, and Samuel Thibault. "Taming data locality for task scheduling under memory constraint in runtime systems". In: *Future Generation Computer Systems* 143 (2023), pp. 305–321. ISSN: 0167-739X. DOI: https://doi.org/10.1016/j.future.2023.01.024. URL: https://www.sciencedirect.com/science/article/pii/S0167739X23000328.

[64] Khronos group. *SYCL 1.2.1 specification*. 2020. URL: https://registry.khronos.org/SYCL/specs/sycl-1.2.1.pdf.

[65] Nikunj Gupta, Jackson R. Mayo, Adrian S. Lemoine, and Hartmut Kaiser. *Implementing Software Resiliency in HPX for Extreme Scale Computing*. 2020. arXiv: 2004.07203 [cs.DC].

[66] Nathan Halko, Per-Gunnar Martinsson, and Joel A Tropp. "Finding structure with randomness: Probabilistic algorithms for constructing approximate matrix decompositions". In: *SIAM review* 53.2 (2011), pp. 217–288.

[67] T. Heller, H. Kaiser, and K. Iglberger. "Application of the ParalleX execution model to stencil-based problems". In: *Computer Science - Research and Development* 28.2 (May 2013), pp. 253–261. ISSN: 1865-2042. DOI: 10.1007/s00450-012-0217-1. URL: https://doi.org/10.1007/s00450-012-0217-1.

[68] Pascal Hénon, Pierre Ramet, and Jean Roman. "PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems". In: *Parallel Computing* 28.2 (Jan. 2002), pp. 301–321.

[69] Thomas Herault, Yves Robert, George Bosilca, and Jack Dongarra. "Generic Matrix Multiplication for Multi-GPU Accelerated Distributed-Memory Platforms over PaRSEC". In: *ScalA 2019 - IEEE/ACM 10th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. Denver, United States: IEEE, Nov. 2019, pp. 33–41. DOI: 10.1109/ScalA49573.2019.00010. URL: https://hal.inria.fr/hal-02436180.

[70] Mark Hoemmen. "Communication-avoiding Krylov subspace methods". PhD thesis. University of California at Berkeley, 2010.

[71] Jonathan D Hogg, Evgueni Ovtchinnikov, and Jennifer A Scott. "A sparse symmetric indefinite direct solver for GPU architectures". In: *ACM Transactions on Mathematical Software (TOMS)* 42.1 (2016), pp. 1–25.

[72] Reazul Hoque, Thomas Herault, George Bosilca, and Jack Dongarra. "Dynamic Task Discovery in PaRSEC: A Data-Flow Task-Based Runtime". In: *Proceedings of the 8th Workshop on Latest Advances in Scalable Algorithms for Large-Scale Systems*. ScalA '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351256. DOI: 10.1145/3148226.3148233. URL: https://doi.org/10.1145/3148226.3148233.

[73] Harold Hotelling. "Analysis of a complex of statistical variables into principal components." In: *Journal of educational psychology* 24.6 (1933), p. 417.

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

[75] Dror Irony and Sivan Toledo. "Trading Replication for Communication in Parallel Distributed-Memory Dense Solvers". In: *Parallel Processing Letters* 12.01 (2002), pp. 79–94. DOI: 10.1142/S0129626402000847. eprint: https://doi.org/10.1142/S0129626402000847. URL: https://doi.org/10.1142/S0129626402000847.

[76] Dror Irony, Sivan Toledo, and Alexander Tiskin. "Communication lower bounds for distributed-memory matrix multiplication". In: *Journal of Parallel and Distributed Computing* 64.9 (2004), pp. 1017–1026. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2004.03.021. URL: https://www.sciencedirect.com/science/article/pii/S0743731504000437.

[77] Mathias Jacquelin, Yili Zheng, Esmond G. Ng, and Katherine A. Yelick. "An Asynchronous Task-based Fan-Both Sparse Cholesky Solver". In: *ArXiv* abs/1608.00044 (2016).

[78] Camille Jordan. "Mémoire sur les formes bilinéaires." In: *Journal de mathématiques pures et appliquées* 19 (1874), pp. 35–54.

[79] Laxmikant Kale, Anshu Arya, Abhinav Bhatele, Abhishek Gupta, Nikhil Jain, Pritish Jetley, Jonathan Lifflander, Phil Miller, Yanhua Sun, Ramprasad Venkataraman, Lukasz Wesolowski, and Gengbin Zheng. *Charm++ for Productivity and Performance: A Submission to the 2011*

HPC Class II Challenge. Tech. rep. 11-49. Parallel Programming Laboratory, Nov. 2011.

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

[81] Peter M. Kogge and Harold S. Stone. "A Parallel Algorithm for the Efficient Solution of a General Class of Recurrence Equations". In: *IEEE Transactions on Computers* C-22.8 (1973), pp. 786–793. DOI: 10.1109/TC.1973.5009159.

[82] Grzegorz Kwasniewski, Marko Kabic, Tal Ben-Nun, Alexandros Nikolaos Ziogas, Jens Eirik Saethre, André Gaillard, Timo Schneider, Maciej Besta, Anton Kozhevnikov, Joost VandeVondele, and Torsten Hoefler. "On the Parallel I/O Optimality of Linear Algebra Kernels: Near-Optimal Matrix Factorizations". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '21. St. Louis, Missouri: Association for Computing Machinery, 2021. ISBN: 9781450384421. DOI: 10.1145/3458817.3476167. URL: https://doi.org/10.1145/3458817.3476167.

[83] Grzegorz Kwasniewski, Marko Kabić, Maciej Besta, Joost VandeVondele, Raffaele Solcà, and Torsten Hoefler. "Red-Blue Pebbling Revisited: Near Optimal Parallel Matrix-Matrix Multiplication". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '19. Denver, Colorado: Association for Computing Machinery, 2019. ISBN: 9781450362290. DOI: 10.1145/3295500.3356181. URL: https://doi.org/10.1145/3295500.3356181.

[84] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. "Basic Linear Algebra Subprograms for Fortran Usage". In: *ACM Trans. Math. Softw.* 5.3 (Sept. 1979), pp. 308–323. ISSN: 0098-3500. DOI: 10.1145/355841.355847. URL: https://doi.org/10.1145/355841.355847.

[85] Romain Lion and Samuel Thibault. "From tasks graphs to asynchronous distributed checkpointing with local restart". In: *2020 IEEE/ACM 10th Workshop on Fault Tolerance for HPC at eXtreme Scale (FTXS)*. 2020, pp. 31–40. DOI: 10.1109/FTXS51974.2020.00009.

[86] K. V. Mardia, J.T. Kent, and J. M. Bibby. *Multivariate Analysis*. Probability and Mathematical Statistics. Academic Press, 1979.

[87] Gordon E. Moore. "Cramming more components onto integrated circuits". In: *Electronics* 38.8 (1965), pp. 114–117.

[88] A. Munshi. *The OpenCL Specification, Khronos OpenCL Working Group, Version 1.1, Revision 44*. 2011.

[89] OpenACC. *OpenACC Programming and Best Practices Guide*. 2022. URL: https://www.openacc.org/sites/default/files/inline-files/openacc-guide.pdf.

[90] Emmanuel Paradis. "Multidimensional scaling with very large datasets". In: *Journal of Computational and Graphical Statistics* 27.4 (2018), pp. 935–939.

[91] Karl Pearson. "LIII. On lines and planes of closest fit to systems of points in space". In: *The London, Edinburgh, and Dublin philosophical magazine and journal of science* 2.11 (1901), pp. 559–572.

[92] Y. Pei, Q. Cao, G. Bosilca, P. Luszczek, V. Eijkhout, and J. Dongarra. "Communication Avoiding 2D Stencil Implementations over PaRSEC Task-Based Runtime". In: *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*. Los Alamitos, CA, USA: IEEE Computer Society, May 2020, pp. 721–729. DOI: 10.1109/IPDPSW50202.2020.00127. URL: https://doi.ieeecomputersociety.org/10.1109/IPDPSW50202.2020.00127.

[93] Josep M. Perez, Vicenç Beltran, Jesus Labarta, and Eduard Ayguadé. "Improving the Integration of Task Nesting and Dependencies in OpenMP". In: *Procceeding of IPDPS'17*. 2017, pp. 809–818.

[94] Jelena Pješivac-Grbović, Thara Angskun, George Bosilca, Graham E. Fagg, Edgar Gabriel, and Jack J. Dongarra. "Performance analysis of MPI collective operations". In: *Cluster Computing* 10.2 (June 2007), pp. 127–143. ISSN: 1573-7543. DOI: 10.1007/s10586-007-0012-0. URL: https://doi.org/10.1007/s10586-007-0012-0.

[95] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. "Elemental: A New Framework for Distributed Memory Dense Matrix Computations". In: *ACM Trans. Math. Softw.* 39.2 (Feb. 2013). ISSN: 0098-3500. DOI: 10.1145/2427023.2427030. URL: https://doi.org/10.1145/2427023.2427030.

[96] James Reinders. *Intel Threading Building Blocks: Outfitting C++ for Multi-Core Processor Parallelism*. O'Reilly, 2007.

[97] Marion Webster Richardson. "Multidimensional psychophysics". In: *Psychological Bulletin* 35 (1938), pp. 659–660.

[98] Vladimir Rokhlin, Arthur Szlam, and Mark Tygert. "A Randomized Algorithm for Principal Component Analysis". In: *SIAM Journal on Matrix Analysis and Applications* 31.3 (Jan. 2010), pp. 1100–1124. ISSN: 0895-4798. DOI: 10.1137/080736417.

[99] Emanuel H. Rubensson and Elias Rudberg. "Chunks and Tasks: A programming model for parallelization of dynamic algorithms". In: *Parallel Computing* 40.7 (2014). 7th Workshop on Parallel Matrix Algorithms and Applications, pp. 328–343. ISSN: 0167-8191. DOI: https://doi.org/10.1016/j.parco.2013.09.006. URL: https://www.sciencedirect.com/science/article/pii/S016781911300118X.

[100] Emanuel H. Rubensson, Elias Rudberg, Anastasia Kruchinina, and Anton G. Artemov. "The Chunks and Tasks Matrix Library". In: *SoftwareX* 19 (July 2022). ISSN: 2352-7110. DOI: 10.1016/j.softx.2022.101159. URL: https://doi.org/10.1016/j.softx.2022.101159.

[101] Piyush Sao, Xiaoye S. Li, and Richard Vuduc. "A communication-avoiding 3D algorithm for sparse LU factorization on heterogeneous systems". In: *Journal of Parallel and Distributed Computing* 131 (2019), pp. 218–234. ISSN: 0743-7315. DOI: https://doi.org/10.1016/j.jpdc.2019.03.004. URL: https://www.sciencedirect.com/science/article/pii/S0743731518305197.

[102] Martin D. Schatz, Robert A. van de Geijn, and Jack Poulson. "Parallel matrix multiplication: a systematic journey". In: *SIAM Journal on Scientific Computing* 38.6 (2016), pp. 748–781. URL: http://www.cs.utexas.edu/users/flame/pubs/2D3DFinal.pdf.

[103] Elliott Slaughter, Wonchan Lee, Sean Treichler, Michael Bauer, and Alex Aiken. "Regent: a high-productivity programming language for HPC with logical regions". In: *SC '15: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. 2015, pp. 1–12. DOI: 10.1145/2807591.2807629.

[104] Elliott Slaughter, Wonchan Lee, Sean Treichler, Wen Zhang, Michael Bauer, Galen Shipman, Patrick McCormick, and Alex Aiken. "Control Replication: Compiling Implicit Parallelism to Efficient SPMD with Logical Regions". In: *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*. SC '17. Denver, Colorado: Association for Computing Machinery, 2017. ISBN: 9781450351140. DOI: 10.1145/3126908.3126949. URL: https://doi.org/10.1145/3126908.3126949.

[105] Edgar Solomonik and James Demmel. "Communication-optimal Parallel 2.5D Matrix Multiplication and LU Factorization Algorithms". In: *Proceedings of the 17th International Conference on Parallel Processing - Volume Part II*. Euro-Par'11. Bordeaux, France: Springer-Verlag, 2011, pp. 90–109. ISBN: 978-3-642-23396-8. URL: http://dl.acm.org/citation.cfm?id=2033408.2033420.

[106] Philippe Swartvagher. "On the Interactions between HPC Task-based Runtime Systems and Communication Libraries". Theses. Université de Bordeaux, Nov. 2022. URL: https://theses.hal.science/tel-03989856.

[107] CORPORATE The MPI Forum. "MPI: A Message Passing Interface". In: *Proceedings of the 1993 ACM/IEEE Conference on Supercomputing*. Supercomputing '93. Portland, Oregon, USA: Association for Computing Machinery, 1993, pp. 878–883. ISBN: 0818643404. DOI: 10.1145/169627.169855. URL: https://doi.org/10.1145/169627.169855.

[108] Samuel Thibault. "On Runtime Systems for Task-based Programming on Heterogeneous Platforms". Habilitation à diriger des recherches. Université de Bordeaux, Dec. 2018. URL: https://inria.hal.science/tel-01959127.

[109] W. S. Torgerson. "Multidimensional Scaling: I. Theory and Method". In: *Psychometrika* 17.4 (1952), pp. 401–419.

[110] Christian R. Trott, Damien Lebrun-Grandié, Daniel Arndt, Jan Ciesko, Vinh Dang, Nathan Ellingwood, Rahulkumar Gayatri, Evan Harvey, Daisy S. Hollman, Dan Ibanez, Nevin Liber, Jonathan Madsen, Jeff Miles, David Poliakoff, Amy Powell, Sivasankaran Rajamanickam, Mikael Simberg, Dan Sunderland, Bruno Turcksin, and Jeremiah Wilke. "Kokkos 3: Programming Model Extensions for the Exascale Era". In: *IEEE Transactions on Parallel and Distributed Systems* 33.4 (2022), pp. 805–817. DOI: 10.1109/TPDS.2021.3097283.

[111] Laurens Van Der Maaten, Eric Postma, Jaap Van den Herik, et al. "Dimensionality reduction: a comparative". In: *J Mach Learn Res* 10.66-71 (2009), p. 13.

[112] Ichitaro Yamazaki, Hartwig Anzt, Stanimire Tomov, Mark Hoemmen, and Jack Dongarra. "Improving the Performance of CA-GMRES on Multicores with Multiple GPUs". In: *2014 IEEE 28th International Parallel and Distributed Processing Symposium*. 2014, pp. 382–391. DOI: 10.1109/IPDPS.2014.48.

[113] Ichitaro Yamazaki, Stephen Thomas, Mark Hoemmen, Erik G. Boman, Katarzyna Świrydowicz, and James J. Elliott. "Low-synchronization orthogonalization schemes for s-step and pipelined Krylov solvers in Trilinos". In: *Proceedings of the 2020 SIAM Conference on Parallel Processing for Scientific Computing (PP)*, pp. 118–128. DOI: 10.1137/1.9781611976137.11. eprint: https://epubs.siam.org/doi/pdf/10.1137/1.9781611976137.11. URL: https://epubs.siam.org/doi/abs/10.1137/1.9781611976137.11.

[114] Asim YarKhan. "Dynamic task execution on shared and distributed memory architectures". PhD thesis. 2012.

[115] Gale Young and Aiston S Householder. "Discussion of a set of points in terms of their mutual distances". In: *Psychometrika* 3.1 (1938), pp. 19–22.