Algorithmes d'allocation statique pour la planification d'applications haute performance

Mathieu Vérité

▶ To cite this version:

Mathieu Vérité. Algorithmes d'allocation statique pour la planification d'applications haute performance. Calcul parallèle, distribué et partagé [cs.DC]. Université de Bordeaux, 2022. Français. NNT : 2022BORD0349 . tel-03956040

HAL Id: tel-03956040
https://theses.hal.science/tel-03956040v1

Submitted on 25 Jan 2023

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

THÈSE PRÉSENTÉE
POUR OBTENIR LE GRADE DE

**DOCTEUR DE
L'UNIVERSITÉ DE BORDEAUX**

ÉCOLE DOCTORALE DE MATHÉMATIQUES ET D'INFORMATIQUE

Par **Mathieu Vérité**

Algorithmes d'allocation statique pour la planification d'applications haute performance

Sous la direction de : **Olivier Beaumont**
Co-encadrant : **Lionel Eyraud-Dubois**

Soutenue le 7 décembre 2022

Membres du jury:

Alfredo Buttari	Directeur de recherche	CNRS / Université Toulouse III (Toulouse)	Rapporteur
Laura Grigori	Directrice de recherche	Inria / Sorbonne Université (Paris)	Rapporteure
Natacha Béreux	Docteure	Ministère de l'Éducation Nationale (Paris)	Examinatrice
Amina Guermouche	Maîtresse de conférence	Inria / Université de Bordeaux (Talence)	Examinatrice
Arnaud Legrand	Directeur de recherche	CNRS / Université Grenoble Alpes (Grenoble)	Examinateur
Raymond Namyst	Professeur des universités	Inria / Université de Bordeaux (Talence)	Président du jury
Olivier Beaumont	Directeur de recherche	Inria / Université de Bordeaux (Talence)	Directeur de thèse
Lionel Eyraud-Dubois	Chargé de recherche	Inria / Université de Bordeaux (Talence)	Co-encadrant

Static Data Allocation Algorithms for Scheduling High Performance Applications

Abstract: Linear algebra applications are commonly used nowadays to solve large scale problems whose size requires the use of distributed and parallel execution on dedicated computation platforms. Many modern linear algebra libraries rely on runtime systems that implement task-based model for their parallel execution. Such tools allow to achieve high performance by performing dynamic scheduling and automatic handling of communications on a set of distributed resources. At the same time, they simplify the implementation of linear algebra operations by decoupling the data distribution from the computations and relieve the programmer from explicit management of communications. Although runtime systems enable the use of virtually any data distribution, most linear algebra applications still rely on the traditional 2D Block Cyclic distribution inherited from the early years of High Performance Computing, when parallel applications were essentially described using basic MPI primitives. In this work we explore the possibilities offered by runtime systems and we design data distributions adapted to the parallel distributed execution of specific linear algebra operations, namely matrix multiplication, symmetric rank-k update, LU and Cholesky factorization. We show that it is possible to design original data distribution schemes that are best fitted to the characteristics of each operation. By taking into account communications reduction and load balancing, the newly developed solutions manage to outperform classic distributions in many configurations, including dense homogeneous cases, both in terms of theoretical and actual parallel performance. This work illustrates that significant improvements over state-of-the-art solutions are achievable by a more careful design of sophisticated data distributions which can in turn be easily implemented using modern task-based linear algebra libraries.
Keywords: load balancing, scheduling, runtime schedulers

Inria, HiePACS
UMR 5800 LaBRi, 33000 Bordeaux, France.

Algorithmes d'allocation statique pour la planification d'applications haute performance

Résumé : De nos jours, les applications d'algèbre linéraire sont couramment utilisées pour traiter des problèmes dont la grande taille requiert une exécution parallèle distribuée par des plate-formes de calcul dédiées. De nombreuses librairies d'algèbre linéaire reposent sur l'utilisation d'ordonnanceurs dynamiques utilisant un modèle d'exécution à base de tâches. De tels outils permettent d'atteindre de hauts niveaux de performance en appliquant un ordonnancement dynamique des tâches et une gestion automatique des communications pour un ensemble de ressources de calcul distribuées. Dans le même temps, ils simplifient la mise en œuvre des opérations d'algèbre linéaire en découplant la distribution de données et les calculs et exemptent le programmeur de la gestion explicite des communications. Bien que les ordonnanceurs dynamiques à base de tâches permettent l'utilisation de virtuellement n'importe quelle distribution de données, une grande partie des librairies d'algèbre liéaire reposent toujours sur la distribution classique 2D Bloc Cyclique héritée des premiers temps du domaine du Calcul Hautes Performances durant lequel la description des applications parallèles reposait essentiellement sur des primitives MPI rigides. Dans cette thèse, nous explorons les possibilités qu'offrent les ordonnanceurs dynamiques à base de tâches et cherchons à concevoir des distributions de données adaptées à l'exécution parallèle distribuée d'opérations d'algèbre linéaire particulières, plus précisément la multiplication de matrices, l'opération *symmetric rank-k update*, la factorisation LU et la factorisation de Cholesky. Nous montrons qu'il est possible de concevoir des distributions de données originales mieux adaptées aux caractéristiques de chaque opération. Prenant en compte la réduction des communications et l'équilibrage de la charge de travail, les solutions développées parviennent à surpasser les distributions classiques dans de nombreuses configurations, en particulier dans les cas denses et homogènes, tant sur les performances théoriques qu'expérimentales. Ce travail illustre que d'importants gains par rapport aux solutions de l'état de l'art actuel sont atteignables grâce à une conception plus fine des distributions de données qui peuvent être facilement mises en œuvre dans des librairies d'algèbre linéaire modernes utilisant un modèle d'exécution à base de tâches.

Mots-clés : équilibrage de charge, ordonnancement, ordonnanceurs dynamiques

---

Inria, HiePACS
UMR 5800 LaBRi, 33000 Bordeaux, France.

Acknowledgments

First and foremost, I want to thank my two supervisors Olivier Beaumont and Lionel Eyraud-Dubois for being so nice and helpful throughout those three years: they were always patient, supportive and available for discussion. It has been a privilege to spend this time working together, always in a friendly and positive atmosphere. I learned so much from them during my time at Inria and it is hard to express in a few words how much I feel lucky for this experience, so simply thank you!

I also want to thank all the jury members for participating to the evaluation this work. I particularly thank Alfredo Buttari and Laura Grigori for reviewing my thesis so carefully and providing thorough feedback about the manuscript. Thanks to Raymond Namyst for accepting the role of president of the jury and handling smoothly the discussions and Q&A session. I also thank Amina Guermouche and Arnaud Legrand for their expert remarks regarding this work and the interesting questions and discussions during the defense. A special thank to Natacha Béreux whose work inspired many developments presented in this thesis, her presence as jury member was greatly appreciated.

I thank all the members of the HiePACS team at Inria Bordeaux; it was a real pleasure to meet such nice and talented people and have the chance to work with them. Huge thanks to Emmanuel Agullo for his help regarding numerical linear algebra and all the mind opening discussions about reproducibility in research, science in general, but also politics and other fancier topics. Many thanks to Pierre Ramet and Mathieu Faverge for their invaluable help regarding the `Chameleon` library; I wouldn't have been able to carry out the experiments for this thesis without them. Thanks to Abdou Guermouche for his help and ability to put in simple words complex problems and the advice regarding the preparation of the defense and writing of the manuscript. Thanks to Luc Giraud for being always helpful with all the administrative tasks and for his support during the lockdown period. I also want to thank Samuel Thibault from the Storm team for his guidance during my first teaching experience at the University of Bordeaux.

Of course, many thanks to all the current and former "dwellers" of the HiePACS open space: Esra, Martina, Yanfei, Pierre, Jean-François, Florent, Tony, Romain, Gilles, Xunyi, Alena, Marek, Nick, Alycia. The cheerful and friendly atmosphere they created, the funny moments and nice discussions we've had, transformed those three years into unforgettable memories. A special thank to my cubicle neighbor Romain for patiently bearing my noisy working style, providing good jokes on a daily basis and helping me with cunning tips to brute force many little computer problems.

Finally, thanks to Adeline for her unconditional support and for reminding me that there is more in life than sitting in front of a computer.

Résumé étendu

Contexte scientifique : le calcul haute performance pour les applications d'algèbre linéaire

De nos jours, le calcul numérique est un outil incontournable dans de nombreux domaines scientifiques et industriels. Que ce soit pour la modélisation, la simulation ou l'optimisation, le calcul numérique est une méthode efficace, et souvent la seule alternative disponible, pour traiter des problèmes complexes et de grande taille. Depuis plusieurs décennies, l'augmentation constante de la puissance de calcul a conduit à l'émergence du domaine du calcul intensif et haute performance (HPC). Les techniques développées dans ce cadre ont pour objectif d'exploiter au mieux les technologies émergentes et tirer parti des capacités de calcul toujours croissantes pour les applications. L'accroissement de la puissance de calcul des plates-formes s'accompagne néanmoins d'une plus importante complexité de fonctionnement.

Alors que l'approche numérique est un outil polyvalent pour traiter des problèmes scientifiques et d'ingénierie de nature variée, de nombreuses applications scientifiques reposent essentiellement sur un nombre restreint d'opérations d'algèbre linéaire. Le fonctionnement complet d'une application se résume alors souvent à l'exécution d'une seule ou d'une combinaison d'opérations numériques d'algèbre linéaire effectuées sur les matrices associées aux entrées du problème et l'essentiel du temps d'exécution correspond à ces opérations.

Dans ce travail, on s'intéresse à certaines des opérations d'algèbre linéaires les plus courantes dans le contexte du calcul hautes performances, à savoir :

- la **multiplication de matrices** (GEMM) : étant données $\mathbf{A}$ et $\mathbf{B}$, calculer $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{B}$;
- la **mise à jour de rang-k** (SYRK) : étant donnée $\mathbf{A}$, calculer $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{A}^{\mathsf{T}}$;
- la **factorisation LU** (GETRF) : étant donnée $\mathbf{A}$, calculer deux matrices triangulaires inférieure $\mathbf{L}$ et supérieure $\mathbf{U}$ telles que $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$;
- la **factorisation de Cholesky** (POTRF) : étant donnée $\mathbf{A}$ symétrique définie positive, calculer une matrice triangulaire inférieure $\mathbf{L}$ telle que $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^{\mathsf{T}}$.

Les opérations d'algèbre linéaire étant centrales pour un grand nombre d'applications de calcul scientifique, leur formalisation sous forme d'algorithmes efficaces est essen-

tielle et a été entreprise dès les premiers temps du calcul hautes performances. La bibliothèque `BLAS` [73], développé depuis les années 1970, fournit une description formelle des principales opérations d'algèbre linéaire sous forme d'algorithmes génériques. Les implémentations séquentielles basées sur ces algorithmes génériques et optimisées pour chaque architecture matérielle sont appelées *noyaux*. Ces implémentations, telles que celles proposées dans les bibliothèques `OpenBLAS` [69, 74] ou `MKL` [72], exploitent les caractéristiques spécifiques des architectures matérielles pour atteindre des performances élevées sur une seule unité de calcul. Les descriptions formelles des algorithmes fourni par `BLAS` constituent un standard *de facto* et sont utilisés comme blocs élémentaires pour construire des opérations plus élaborées dans d'autres bibliothèques d'algèbre linéaire, comme `LAPACK` [73].

Contexte technique : calcul parallèle distribué et exécution à base de tâches

Malgré la tendance générale à l'augmentation de la puissance de calcul, l'exécution séquentielle d'opérations sur une seule ressource est un modèle intrinsèquement limité en termes de performances. Les exigences croissantes des applications scientifiques en termes d'échelle et de précision ont donc naturellement conduit à l'émergence du calcul parallèle, c'est-à-dire l'utilisation simultanée des ressources de calcul pour une même opération. Les plates-formes de calcul modernes regroupent plusieurs unités de calcul, appelées *nœuds* typiquement constitués : d'une mémoire (RAM), d'un processeur (CPU) comportant plusieurs cœurs et, de plus en plus souvent, d'un ou plusieurs *accélérateurs* (GPU), ces derniers disposant d'une mémoire dédiée. Les nœuds sont connectés via un réseau haute performance visant à fournir la plus grande bande passante et la plus petite latence possible. Cette architecture matérielle permet d'exploiter deux niveaux de parallélisme : (i) à l'échelle des unités de calcul (CPU, GPU), en mémoire partagée, grâce aux architectures multi-cœurs, (ii) à l'échelle de la plateforme de calcul, par l'exécution distribuée des applications utilisant simultanément plusieurs dizaines voire centaines de nœuds. L'échelle des problèmes considérés aujourd'hui nécessite d'utiliser ces deux niveaux de parallélisme afin d'assurer un temps d'exécution raisonnable. Il est donc primordial de comprendre et d'optimiser le comportement parallèle des applications, notamment les opérations d'algèbre linéaire sous-jacentes, afin d'obtenir les meilleures performances possibles.

Distribution de données et affectation des tâches

Dans ce travail, nous considérons l'exécution parallèle et distribuée d'opérations d'algèbre linéaire dans ce modèle générique de plate-forme de calcul utilisant P nœuds identiques. L'exécution d'une opération en parallèle nécessite que les données d'entrée et de sortie soient divisées et distribuées aux nœuds. En pratique, pour les opérations d'algèbre linéaire, les données correspondant à des matrices sont divisées en *blocs* ou *tuiles* de taille identiques. Les tâches constitutives d'une opération d'algèbre linéaire prennent place à l'échelle des tuiles et correspondent à l'exécution d'un noyau spécifique par une ressource de calcul, un cœur d'un CPU ou un GPU, sur un nœud. Cette division des

données permet d'exploiter efficacement le parallélisme de la plateforme puisque plusieurs tâches effectuées sur différentes tuiles peuvent être exécutées en parallèle par différentes ressources de calcul, que ce soit par le même nœud ou par des nœuds différents. On appelle *distribution de données* l'affectation d'un ensemble de tuiles à chaque nœud ; on qualifie celle-ci de 2D, ou *statique*, lorsque chaque tuile existe dans la mémoire d'un seul nœud. Une telle configuration est standard dans la littérature et dans les bibliothèques d'algèbre linéaire parallèle. Il existe des distributions plus complexes faisant appel à la réplication des données, appelées 2.5D ou 3D. On s'intéresse dans ce travail à des distributions des données 2D, sauf dans la section 2.2.3

Pour une opération d'algèbre linéaire donnée, l'ensemble des tâches qui la compose peut être abstrait sous la forme d'un *Graphe Acyclique Orienté* (cDAG) Dans un tel cDAG, les tâches correspondent aux sommets et les dépendances aux arêtes orientées : la présence d'une arête entre deux sommets indique que la tâche associée au sommet enfant ne peut pas démarrer avant que les tâches associées aux sommets parents soient terminées. Le cDAG associé à une opération explicite ainsi les dépendances de tâche inhérentes à l'opération. D'autre part, ces dépendances de tâche correspondent à des dépendances de donnée, les sorties de certaines tâches étant les entrées d'autres. Dans un contexte distribué, ces dépendances de donnée rendent nécessaires des communications entre les nœuds, car aucun ne dispose de la totalité des données. Le nombre et l'organisation de ces communications est directement fonction de la distribution de données utilisée. Pour les limiter, la répartition des tâches durant l'exécution suit souvent la règle *owner computes*, c'est-à-dire que chaque nœud est en charge de réaliser toutes les tâches qui modifient les tuiles dont il possède les données. Cette affectation des tâches s'articule naturellement avec une distribution de données 2D.

Jusqu'à récemment, pour décrire l'exécution d'un opération, les communications entre les nœuds devaient être explicitées et mises en œuvre à l'aide de technologies logicielles dédiées, généralement via le protocole **M**essage **P**assing **I**nterface (`MPI`). Cela rend particulièrement difficile le développement des applications et leur évolution. Par conséquent, la plupart des bibliothèques d'algèbre linéaire depuis les début du calcul parallèle et jusque récemment utilisent presque exclusivement la distribution 2D **B**lock **C**yclic (**BC**), qui est devenue un standard *de facto*. C'est le cas de la bibliothèque `ScaLAPACK` [22] mais aussi de solutions logicielles plus récentes, telles que `SLATE` [40], `Elemental` [64] ou `COnfLUX` [57]. C'est une distribution basée sur un motif répétée qui est très régulière, ce qui implique des dépendances des donnée assez simple et des schémas de communication faciles à décrire. Par ailleurs elle possède des qualités importantes dans un contexte parallèle distribué : un équilibrage de charge global parfait entre les nœuds, un bon équilibrage de charge au cours de l'exécution, un nombre limité de nœuds présent sur chaque ligne et colonne (de l'ordre de $\sqrt{P}$) limitant le volume de communication. Malgré ces propriétés intéressantes, la distribution BC n'est pas optimale dans toutes les situations. L'objectif principal de ce travail est de proposer des alternatives plus efficaces, notamment des distributions basées sur des motifs répétés, dans des configurations spécifiques.

Modèle d'exécution à base de tâches et ordonnanceurs dynamiques

Depuis une dizaine d'années, le modèle d'exécution *à base de tâches* suscite beaucoup d'intérêt dans le domaine du calcul hautes performances et est considéré comme l'une des directions de recherche les plus prometteuses pour le développement d'applications efficaces et flexibles dans un contexte parallèle distribué. Dans ce paradigme, la description et l'exécution du calcul sont découplées de la gestion des données et des dépendances de tâche : le premier aspect est laissé à la charge d'une bibliothèque spécifique tandis que le second est délégué à un outil dédié, l'*ordonnanceur dynamique*. Ce dernier gère l'ordonnancement des tâches pour assurer l'exactitude des calculs et optimiser le temps d'exécution. D'une part, à l'échelle de chaque nœud, les tâches sont réparties dynamiquement parmi les ressources en fonction de leurs caractéristiques, du noyau à exécuter, de l'état courant de chaque ressource et d'une politique d'ordonnancement. À l'échelle de la plate-forme, d'autre part, l'ordonnanceur assure le respect des dépendances de donnée et de tâche en déclenchant les communications inter nœuds requises au bon moment.

L'utilisation du modèle d'exécution à base de tâches présente des avantages significatifs comparativement à une construction rigide des applications basée sur une description explicite de l'ordonnancement des tâches et des communications. D'une part, les ordonnanceurs dynamiques prenant les décisions au cours de l'exécution, ils offrent la possibilité de déployer des stratégies de gestion des données et d'ordonnancement à grain fin. À l'échelle de chaque nœud, ils sont capables de gérer de manière automatique des ressources hétérogènes (des cœur de CPU et différents GPU) et de les utiliser au mieux. À l'échelle de la plate forme, ce paradigme permet une exécution asynchrone de toutes les tâches composant une opération. Ceci augmente le parallélisme durant l'exécution et offre la possibilité de recouvrir les communications et les calculs qui sont effectués en parallèle. D'autre part, le modèle d'exécution à base de tâches offre la possibilité d'utiliser de manière transparente des distributions de données complètement irrégulières qui peuvent être mieux adaptées à chaque opération considérée. Cette possibilité constitue d'ailleurs un pré-requis aux développements présentés dans ce travail.

Le modèle d'exécution à base de tâches offre des avantages importants par rapport à une conception rigides des applications. Ainsi, de nombreuses bibliothèques d'algèbre linéaire modernes implémentent désormais entièrement ce paradigme. C'est le cas par exemple des bibliothèques d'algèbre linéaire dense `Chameleon` [3–5], `DPLASMA` [23] ou `FLAME` [47] ou la bibliothèque d'algèbre linéaire creuse `PaStiX` [49]. Elles s'appuient, pour gérer les dépendances et les communications, sur des ordonnanceurs dynamiques qui sont développés parallèlement, tels que `StarPU` [11, 33], `PaRSEC` [24, 25], `SuperMatrix` [30, 66] ou `OmpSs` [35]. Au-delà des avantages liés au développement et à la conception, les bibliothèques qui s'appuient sur le modèle d'exécution à base de tâches atteignent des performances nettement supérieures à celles des logiciels basés sur `MPI`, comme cela est illustré par exemple dans [4].

Contributions

Dans ce travail, nous nous intéressons à l'exécution parallèle distribuée d'opérations d'algèbre linéaire dans le modèle d'exécution à base de tâches. Nous traitons principalement de questions liées à l'équilibrage de charge et aux communications qui se posent dans ce contexte, l'objectif étant de minimiser le temps d'exécution total de ces opérations. Nous nous intéressons aux méthodes de résolution directes sur des matrices denses ou compressées et développons des distributions de données statiques adaptées à des opérations et des configurations spécifiques. Bien que ce ne soit pas l'objet de ce travail, la plupart des résultats présentés sont valables indépendamment de l'hétérogénéité intra-nœuds, *i.e.* les différences entre ressources de calcul.

La première partie du document est consacrée à l'étude des opérations qui utilisent des matrices d'entrée symétriques : la mise à jour de rang-k et la factorisation de Cholesky. Nous considérons l'exécution parallèle distribuée de ces opérations sur des matrices denses.

Dans le chapitre 2 nous proposons une distribution de données originale dénommée **S**ymmetric **B**lock **C**yclic (**SBC**) qui utilise la symétrie de la matrice d'entrée pour réduire le volume total de communication pour la factorisation de Cholesky. Nous détaillons le schéma de communication de cette nouvelle distribution par rapport à la distribution BC classique et présentons une estimation formelle détaillée du volume total de communication : le volume de communication généré par SBC est réduit d'un facteur $\sqrt{2}$ par rapport à BC. Les résultats expérimentaux obtenus à l'aide de la bibliothèque `Chameleon` associée à l'ordonnanceur dynamique `StarPU` permettent clairement d'observer une amélioration significative des performances lors de l'utilisation de SBC par rapport à BC. Ces résultats illustrent également que le recouvrement des communications et des calculs autorisés par le modèle d'exécution à base de tâches est intrinsèquement limité en raison des dépendances entre les tâches et constitue donc un argument en faveur d'une stratégie complémentaire de réduction globale des communications. En plus de ces résultats, une extension 2.5D de la distribution SBC est décrite et testée sur la même configuration. L'extension de l'utilisation de la distribution SBC à des séquences d'opérations qui incluent une factorisation de Cholesky est également discutée ; des résultats préliminaires sont présentés pour les opérations de résolution (POSV) et d'inversion (POTRI).

Le chapitre 3 se concentre sur l'aspect théorique du problème de minimisation des communications pour la mise à jour de rang-k et la factorisation de Cholesky sur des matrices denses. Contrairement au chapitre précédent, afin d'obtenir des bornes inférieures fortes sur le volume de communication, nous nous plaçons dans un contexte *out-of-core* : on dispose d'un unique nœud avec une mémoire limitée de taille M qui effectue les calculs, seule une partie des données peut être présente en mémoire, et les communications interviennent entre la mémoire locale limitée et une mémoire distante illimitée. Dans cette configuration, il est possible d'établir des bornes pour les deux opérations sur le volume de communication minimum nécessaire. Nous prouvons que l'exécution de la mise à jour de rang-k sur une matrice de taille $m \times n$ nécessite au moins $Q_{\text{SYRK}}. \geqslant \frac{1}{\sqrt{2}} \frac{m^2 n}{M}$ communications ; d'autre part, la factorisation de Cholesky d'une matrice $m \times m$ symétrique définie positive nécessite $Q_{\text{Cholesky}}. \geqslant \frac{1}{3\sqrt{2}} \frac{m^3}{M}$ communications. Ces bornes correspondent à une amélioration d'un facteur $\sqrt{2}$ par rapport à la meilleure borne sur les communications de

l'état de l'art pour les mêmes opérations, proposée par Olivry *et al.* [62]. Ces deux bornes utilisent une expression mathématique que nous qualifions d'*explicite* car le coefficient du terme dominant est explicitement défini tandis que les termes d'ordre inférieur peuvent être exprimés à l'aide de notations asymptotiques. Nous utilisons ce terme par opposition à *asymptotique* qui désigne les expressions où le terme dominant n'apparaît que dans une notation asymptotique et où son coefficient reste indéfini. La différence est d'une importance majeure car de nombreux résultats historiques de l'état de l'art considérés comme optimaux atteignent en fait une optimalité uniquement asymptotique. Dans le même chapitre, nous présentons deux algorithmes dans le contexte out-of-core: **T**riangle **B**lock **S**YRK (**TBS**) pour la mise à jour de rang-k et **L**arge **B**lock **C**holesky (**LBC**) pour la factorisation de Cholesky. Nous montrons que ces algorithmes atteignent effectivement une optimalité explicite vis à vis du nombre de communications.

Ces résultats permettent de clore le problème de minimisation des communications dans un contexte out-of-core pour la mise à jour de rang-k et la factorisation de Cholesky d'un point de vue théorique, fournissant une borne inférieure caractérisée comme optimale puisque atteinte par les solutions des algorithmes proposés. En outre, les techniques développées pour élaborer ces algorithmes apportent un nouvel éclairage sur la manière d'adapter les distributions de données pour d'autres opérations ou ensembles d'opérations utilisant des données d'entrée symétriques. Les expériences menées avec la distribution SBC illustrent la facilité de mise en œuvre et d'utilisation de ces distributions de données sur mesure dans un contexte parallèle distribué et faisant appel au modèle d'exécution à base de tâches. Les résultats montrent également clairement le gain de performance significatif qui peut être attendu de la réduction du volume total de communication. Dans la deuxième partie de ce travail, nous essayons d'étendre les techniques élaborées dans la première partie à d'autres configurations plus complexes, toujours dans un contexte parallèle et distribué. Nous explorons deux directions de recherche différentes :

- la conception et l'application de distributions statiques sur mesure pour les opérations sur des matrices compressées, ce qui implique des tâches hétérogènes ;
- pour des matrices denses, l'extension et l'amélioration des distributions BC et SBC existantes à un nombre quelconque de nœuds.

Dans le chapitre 4, nous considérons les opérations non symétriques, la multiplication de matrices et la factorisation LU dans le cas de matrices compressées utilisant le format **B**lock **L**ow **R**ank (**BLR**). Dans ce contexte, nous proposons un modèle de performance qui suppose que des taux de compression des tuiles des matrices d'entrée impliquent des temps d'exécution différents pour chaque tâche. Contrairement au cas dense, l'ensemble des tâches considérées est donc hétérogène. Nous étudions alors le problème de l'équilibrage de charge entre les nœuds tout en imposant des contraintes additionnelles sur le nombre de nœuds différents par ligne et par colonne afin de limiter les communications. Ces contraintes sont basées sur la structure de la distribution classique Block Cyclic (BC). Après avoir formalisé le compromis entre les aspects d'équilibrage et de réduction des communications comme un problème d'optimisation discret, nous présentons deux distributions de données originales élaborées à l'aide d'heuristiques : (i) **B**lock **C**yclic **E**xtended (**BCE**) étend de manière naturelle la distribution BC pour s'adapter aux contraintes liées aux

communications tout en permettant un meilleur équilibrage de la charge ; (ii) **R**andom **S**u**b**sets (**RSB**) est une stratégie en deux étapes reposant sur des sous-ensembles de nœuds avec des propriétés souhaitables calculées préalablement ; elle fournit des distributions de données non régulières. Nous présentons des évaluations expérimentales de ces deux stratégies obtenus par simulations d'exécutions sur des cas de test synthétiques. Les résultats montrent que BCE et RSB assurent un meilleur équilibrage de la charge que la distribution BC dans toutes les configurations testées. Elles atteignent aussi systématiquement un temps d'exécution plus réduit que BC, bien que les deux stratégies présentent des comportements différents : alors que les résultats obtenus avec la distribution BCE sont très cohérents, la distribution RSB présente de grands écarts entre les valeurs d'équilibrage de charge et les temps d'exécution pour la factorisation LU. Cela illustre l'importance de la régularité de la distribution des données pour exploiter le parallélisme du calcul pour les opérations présentant de nombreuses dépendances entre les tâches.

Le chapitre 5 est consacré à l'extension à un nombre quelconque de nœuds des distributions à base de motifs répétés pour les factorisation LU et de Cholesky sur des matrices denses. Bien que présentant des propriétés souhaitables concernant l'équilibrage de la charge et la limitation des communications, les distributions Block Cyclic et Symmetric Block Cyclic ne sont efficaces, ou même disponibles, que pour des valeurs spécifiques du nombre de nœuds P. D'une part, BC limite efficacement les communications pour les opérations non symétriques lorsque le nombre de nœuds disponibles peut être écrit sous la forme $P = pq$ avec p et q proches de $\sqrt{P}$. D'autre part, SBC réduit le volume de communication par rapport à BC pour les opérations symétriques mais n'est disponible que pour des valeurs spécifiques de P : lorsque $P = \frac{r(r-1)}{2}$ ou $P = \frac{r^2}{2}$, avec $r \in \mathbb{N}^*$. Dans ce chapitre, nous élaborons des distributions présentant la même propriété de réduction des communications que ces deux méthodes mais pouvant utiliser efficacement tous les nœuds disponibles. Nous proposons d'abord un modèle formel pour évaluer toute distribution basée sur des motifs répétés vis à vis de l'équilibrage de la charge et de la limitation des communications et nous l'utilisons pour guider la conception de deux distributions de données originales. Pour la factorisation LU, la méthode **G**eneralized **B**lock **C**yclic (**G-BC**) fournit des distributions à motif quasi cyclique pour tout nombre de nœuds. Nous prouvons que ces solutions atteignent une optimalité explicite concernant la métrique de communication que nous avons défini. Pour la factorisation de Cholesky, nous présentons un algorithme glouton randomisé, **G**reedy **C**ol**R**ow & **M**atching (**GCR&M**) qui vise à générer des solutions avec des caractéristiques similaires. Pour tester la qualité des solutions fournies par ces deux algorithmes, nous avons réalisé des expériences utilisant la bibliothèque d'algèbre linéaire `Chameleon` associée à l'ordonnanceur dynamique `StarPU`. Les résultats montrent que les méthodes G-BC et GCR&M sont plus performantes que leurs homologues respectifs, BC et SBC. En particulier, pour les valeurs de P où les motifs BC ont des dimensions très déséquilibrées ou s'il n'existe même pas de solution SBC, les méthodes G-BC et GCR&M parviennent à obtenir des performances plus élevées par nœud tout en utilisant tous les nœuds disponibles. Un tel résultat est très prometteur car il démontre la possibilité de concevoir des distributions de données qui s'adaptent à des configuration très diverses.

Perspectives

Les techniques développées dans ce travail et les résultats qu'elles ont permis d'obtenir constituent un argument pratique en faveur d'un usage généralisé du modèle d'exécution à base de tâches pour les opérations d'algèbre linéaire dans un contexte parallèle distribué. En effet, les distributions de données présentées permettent clairement d'atteindre de meilleures performances en comparaison de la distribution classique Block Cyclic pour les opérations considérées, tandis que leur implémentation dans la bibliothèque `Chameleon` reste particulièrement aisée et leur utilisation automatiquement gérée par l'ordonnanceur dynamique `StarPU`.

Pour les matrices denses, les méthodes présentées dans les chapitres 2 et 5, cela ouvre de larges perspectives quant aux possibilités future d'utilisation de distributions adaptées aux spécificités de chaque opération d'algèbre linéaire et laisse espérer que d'important gains de performances sont restent à exploiter. En outre, la faciliter d'implémentation via l'utilisation d'ordonnanceurs dynamiques permet d'entrevoir la possibilité de combiner des distributions de données très spécifiques avec des stratégies de ré-allocation pour des applications complexes faisant appel à plusieurs opérations successives.

Les bornes inférieures sur le volume de communication obtenues pour la mise à jour de rang-k et la factorisation de Cholesky illustrent l'importance du développement d'outils efficaces permettant d'évaluer la complexité des opérations, dans ce cas vis à vis des communications, afin d'identifier les gains potentiels liés à l'amélioration des algorithmes de résolution. Un aboutissement particulièrement utile de ces travaux pourrait être l'intégration, dans l'outil `IOLB` développé par Olivry *et al.* [62], de la technique *ad hoc* utilisée pour détecter la réutilisation de données dans les opérations symétriques au cœur de l'élaboration des bornes théoriques et des algorithmes optimaux proposés.

Enfin, il semble naturel d'étendre les méthodes développées dans ce travail aux cas d'opérations sur des matrices compressées ou creuses, étant donnée leur importances tant dans le milieu académique que pour les applications industrielles. Les approches que nous avons élaboré pour l'exécution parallèle distribuée des opérations de multiplication de matrices et de factorisation LU avec des matrices compressées selon le format BLR illustrent la difficulté du problème d'équilibrage de charge couplé à la minimisation des communications dans le cas de tâches hétérogènes. Néanmoins il existe une large variété d'heuristiques, telles que celles proposées dans ce travail, pouvant être adaptées et combinées. Là encore, l'utilisation d'ordonnanceurs dynamiques dans le modèle d'exécution à base de tâches offre la possibilité d'explorer un large panel de stratégies de distribution de données et laisse espérer d'important gain de performances par rapport aux solutions classiques telles que la distribution Block Cyclic.

Contents

Contents xvi

List of Figures xviii

List of Tables xx

List of Algorithms xx

1 Introduction 1
1.1 Context . . . 1
1.1.1 Linear Algebra in Scientific Computing . . . 1
1.1.2 Parallel and Distributed Computing . . . 3
1.2 Task-Based Execution Model and Runtime Systems . . . 12
1.2.1 Dynamic Task and Communication Management . . . 13
1.2.2 Asynchronous Execution . . . 15
1.3 Contribution and Outline . . . 18

2 Overall Communication Reduction: Effect on Performance 23
2.1 State-of-the-art . . . 24
2.1.1 Communication Lower Bounds . . . 24
2.1.2 Algorithms . . . 26
2.2 Communication Analysis of BC and SBC Distributions . . . 27
2.2.1 Symmetric Block Cyclic Distribution . . . 28
2.2.2 Analysis of the Communication Volume . . . 30
2.2.3 2.5D Variants of the Data Distributions . . . 34
2.3 Experiments . . . 36
2.3.1 Experimental Setup . . . 36
2.3.2 Data Distributions Parameters . . . 38
2.3.3 Reduction of the Communication Volume . . . 39
2.3.4 Performance Results for Cholesky factorization . . . 40
2.3.5 Performance for Other Operations . . . 44
2.4 Conclusion . . . 48

3 Communication Optimal Algorithms for Symmetric Operations 51
3.1 State-of-the-art . . . 52
3.1.1 Summary of Communication Bounds . . . 52

3.1.2 Overview of Sequential Algorithms 53
3.2 Preliminaries 55
3.2.1 Motivation 55
3.2.2 Methodology, Assumptions and Notations 58
3.2.3 Triangle Blocks 59
3.3 Lower Bounds 60
3.3.1 Symmetric Multiplication (SYRK) 60
3.3.2 Cholesky Factorization 64
3.4 Communication-Optimal Algorithms 64
3.4.1 TBS: Triangle Block SYRK 65
3.4.2 LBC: Large Block Cholesky 71
3.5 Conclusion 74

4 Communication Aware Allocation Strategies in Block Low Rank Context 79
4.1 Introduction 79
4.2 Related Work 81
4.2.1 Data Compression 81
4.2.2 Heterogeneous Allocation Problems in Linear Algebra 82
4.2.3 Applied Perspective 82
4.3 Problem Description and Modeling 83
4.3.1 Communication Scheme of the Operations 83
4.3.2 Block Low Rank Compression 85
4.3.3 Performance Model and Evaluation Metrics 85
4.3.4 General Problem Modeling: Load Balancing versus Communication Trade-Off 87
4.3.5 Bin Packing Problem Variant 88
4.4 Data Distribution Schemes 89
4.4.1 Block Cyclic 89
4.4.2 Block Cyclic Extended 90
4.4.3 Random Subsets Algorithm 91
4.5 Experiments 96
4.5.1 Test Cases 96
4.5.2 Results 98
4.6 Analysis of Block Cyclic Extended Algorithm 100
4.7 Conclusion 103

5 Data Distribution Schemes for Dense Linear Algebra Factorizations on Any Number of Nodes 105
5.1 Introduction 105
5.2 Context 108
5.3 Model and Notations 109
5.3.1 Case of LU factorization 109
5.3.2 Case of Cholesky factorization 111
5.3.3 Communication cost metric 111

5.4 Generalized Block Cyclic 112
5.4.1 Pattern Construction 112
5.4.2 Pattern Properties 113
5.4.3 Evaluation of G-BC 114
5.5 Data Distributions for Symmetric Matrices 114
5.5.1 Greedy ColRow & Matching 116
5.5.2 Evaluation of GCR&M 118
5.6 Experimental Performance Evaluation 119
5.6.1 Experimental Setup 119
5.6.2 Results for Generalized Block Cyclic 120
5.6.3 Results for Greedy ColRow & Matching 121
5.7 Conclusion 121

Conclusion and Perspectives **133**

Bibliography **137**

List of Figures

1.1 Examples of 2D BC distribution over a 12×12 symmetric matrix (only lower triangular part referenced). Each color represents a node. 7
1.2 cDAG of the Cholesky factorization for a 5×5 matrix 9
1.3 Illustration of the owner computes rule (dashed horizontal lines) and inter-node communications (black arrows) (Cholesky factorization of a 5×5 matrix using BC distribution with $P = 6$ nodes) 11
1.4 Illustration of synchronous and asynchronous executions; the gray color indicates that the task on the corresponding tile has been completed (Cholesky factorization of a 5×5 matrix using BC distribution with $P = 6$ nodes) 16

2.1 SBC allocation: repetition of the 4×4 pattern (using $P = 6$ nodes; one color per node) over a 12×12 matrix. Diagonal positions in the pattern are omitted. 28
2.2 Basic version of SBC pattern, with additional nodes, for $r = 4$. 29
2.3 Extended SBC distribution for $r = 5$, $P = 10$. Left: generic pattern, right: patterns with diagonal nodes. 30
2.4 Example of the SBC distribution for $r = 6$, $P = 15$. Left: generic pattern, right: 5 sets of diagonal nodes, with four normal packs (in two shades of orange) and one bonus pack (in green). 31
2.5 Diagonal patterns: two normal packs (in orange) and one bonus pack (in green) generate 3 diagonal patterns used in a round-robin column-wise fashion. . . . 31

2.6 Communication scheme associated with tile $\mathbf{A}(5,2)$ in a 12×12 matrix distributed according to BC or SBC using 6 and 8 nodes. 33
2.7 Performance of the Cholesky factorization over a matrix of size $m = 50000$ using `Chameleon- StarPU` on a single node for different tile size. 38
2.8 Measured volume of inter-node communication during POTRF, for $P = 20$ and 21 (one tile of dimension $b \times b$ in double precision is 2 MB). 39
2.9 Performance per node for Cholesky factorization using 2D and 2.5D versions of BC and SBC for $P = 28$, 30 and 32. 40
2.10 Strong scaling of BC and SBC for $m_b = 200,000$. 41
2.11 Performance per node for Cholesky factorization (GFlop/s per node) using BC and SBC, for P ranging from 15 to 36. 42
2.12 Total running time of Cholesky factorization using BC and SBC allocations, for P ranging from 15 to 36. 43
2.13 Performance per node for POSV with $\mathbf{A}$ distributed using BC and SBC, $P = 28$ 45
2.14 Performance per node for POTRI using BC and SBC, $P = 28$. 48

3.1 Zones and blocks in the TBS algorithm. Each block has one element in each zone. 67
3.2 *Left:* $f^{i,j}(u)$ gives the position of the row of $B^{i,j}$ within the u-th row of zones. *Right:* which parts of the matrix $\mathbf{C}$ are computed by which method in the TBS algorithm. 67
3.3 Algorithm LBC: updating the three parts of $\mathbf{A}$ at iteration i 72

4.1 Example of BCE allocation: $P = 6$, $p_{\max} = 4$, $\mathbf{G}$ is 3×4 91
4.2 Example of "dead end" configuration ($m_b = 8$, $P = 6$, $p_{\max} = 3$) 92
4.3 Example of rank distribution and cumulated execution time for LU factorization on a 30×30 matrix using individual execution time for each type of task: $C_{GETRF} = 2$, $C_{TRSM} = 2$, $C_{GEMM} = 4$ 97
4.4 Results for matrix multiplication 99
4.5 Results for LU factorization 100
4.6 BCE load balancing for matrix multiplication 101
4.7 BCE load balancing for LU factorization 102

5.1 Sample results for LU factorization of matrix A of size $m \times m$, $m = 50,000$ to $200,000$, distributed using 2DBC with different pattern shapes. 106
5.2 Communication scheme for LU and Cholesky factorization using BC distribution over an $m_b \times m_b$ matrix, $m_b = 12$, using $P = 6$ nodes laid out as a 2×3 pattern. 110
5.3 Example of the G-BC pattern for $P = 10$, thus $a = 4$, $b = 3$ and $c = 2$. Left: incomplete pattern $\mathcal{IP}$, right: full G-BC pattern. 112
5.4 Total cost T of G-2DBC and the best 2DBC for varying values of P. 115
5.5 Illustration of the first phase of GCR&M algorithm on a 9×9 pattern using $P = 12$ nodes. Gray cells are uncovered, the striped zone is the additional colrow, thick purple highlighted cells are newly covered cells. 117

5.6 Total cost T of the symmetric and non-symmetric patterns for varying values of P. 118
5.7 Performance results for LU factorization using $P = 23$ nodes 124
5.8 Performance results for LU factorization using $P = 31$ nodes 125
5.9 Performance results for LU factorization using $P = 35$ nodes 126
5.10 Performance results for LU factorization using $P = 39$ nodes 127
5.11 Performance results for Cholesky factorization using $P = 23$ nodes 128
5.12 Performance results for Cholesky factorization using $P = 31$ nodes 129
5.13 Performance results for Cholesky factorization using $P = 35$ nodes 130
5.14 Performance results for Cholesky factorization using $P = 39$ nodes 131

List of Tables

2.1 absolute and relative execution time of kernels in the selected experimental setup (double precision, $b = 500$) 37
2.2 Sizes of the considered distributions 39

3.1 Summary of the total communication volume for the symmetric rank-k update and Cholesky factorization: theoretical lower bound and value achieved by the algorithms 77

4.1 Values of $p_{\max}$ (the maximum number of different nodes per row and column) according to the values of the parameters (P, α) and associated pattern sizes (r, c) used for BCE 98

5.1 Sizes and characteristics of the SBC and GCR&M patterns used for the experiments; the SBC pattern for $P = 32$ is the basic version of size 8×8 122

List of Algorithms

1	Element-wise Cholesky factorization algorithm	2
2	Tiled matrix multiplication algorithm (GEMM)	3
3	Tiled symmetric rank-k update algorithm (SYRK)	4
4	Tiled Cholesky factorization algorithm (POTRF)	4
5	Tiled LU factorization algorithm (GETRF)	5
6	Tiled triangular solve algorithm (TRSM)	44
7	Tiled symmetric matrix multiplication algorithm (LAUUM)	46
8	Tiled triangular inversion algorithm (TRTRI)	47
9	ELM_SYRK, element-wise symmetric rank-k update algorithm	54
10	ELM_CHOL, element-wise Cholesky factorization algorithm	54
11	NBK_GEMM, narrow-blocks single tile matrix multiplication	54
12	NBK_SYRK, narrow-blocks single tile symmetric rank-k update	55
13	NBK_TRSM, narrow-blocks single tile triangular solve	55
14	OOC_SYRK, out-of-core symmetric rank-k update	56
15	OOC_TRSM, out-of-core triangular solve	57
16	OOC_CHOL, out-of-core Cholesky factorization	58
17	Pseudo-code of generic out-of-core SYRK algorithm	65
18	TBS, Triangle Block SYRK algorithm	70
19	LBC, Large Block Cholesky algorithm	71
20	Tiled matrix multiplication algorithm (GEMM)	83
21	Tiled LU factorization algorithm (LU)	84
22	Largest Processing Time algorithm	92
23	Random Subsets greedy algorithm	94
24	Subsets Generation algorithm	95
25	Greedy ColRow & Matching Algorithm	116

Chapter 1

Introduction

1.1 Context

1.1.1 Linear Algebra in Scientific Computing

With the advent of modern computing, numerical applications are nowadays an unavoidable aspect of scientific research and engineering. Whether it is for modeling, simulation, optimization, numerical computation is a tool of choice, and often the only available alternative, to handle highly complex problems. Computationally intensive applications are central in several scientific and industrial fields: simulation of physical systems, combinatorial optimization, AI, etc. For several decades, the dramatic increase in computing power has enabled handling problems of unprecedented scale and complexity. The field of High Performance Computing (HPC) has followed the same trend in terms of scientific development, both from a formal and applied perspective, to make the best use of emerging technologies and leverage their ever growing computing capabilities for the applications.

While the numerical approach is a versatile tool for handling various scientific and engineering problems of diverse nature, many computationally intensive scientific applications essentially rely on a relatively small subset of the computing possibilities, namely numerical linear algebra operations. Indeed, linear algebra appears at the core of several applications, especially simulation and modeling of complex physical systems for which analytical solution is too challenging: fluid mechanics, electro-magnetic propagation, etc. In this type of applications, the modeling is performed via a discretization of the studied domain and linearization of the differential equations describing the phenomenon. The complete application execution then often comes down to a single or a combination of linear algebra operations performed on matrices associated with the problem inputs. For such applications, a large proportion of the time-consuming tasks logically comes from the execution of those linear algebra operations. In this work we focus on some of the most common operations, namely:

- **matrix multiplication**: for given matrices $\mathbf{A}$, $m \times k$, $\mathbf{B}$, $k \times n$, and $\mathbf{C}$, $m \times n$, compute $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{B}$;
- **symmetric rank-k update**: for given matrices $\mathbf{A}$, $m \times n$, and $\mathbf{C}$, $m \times m$, compute $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{A}^{\mathsf{T}}$;

- **LU factorization**: for a given matrix $\mathbf{A}$, $m \times m$, compute the lower triangular $\mathbf{L}$ and upper triangular $\mathbf{U}$ matrices, both $m \times m$, such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$;
- **Cholesky factorization**: for a given $m \times m$ symmetric positive definite matrix $\mathbf{A}$, compute the lower triangular matrix $\mathbf{L}$, $m \times m$, such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^{\mathsf{T}}$.

The description of the tiled execution of those operations can be found in Algorithm 2 for the matrix multiplication, Algorithm 3 for the symmetric rank-k update, Algorithm 5 for the LU factorization, and Algorithm 4 for the Cholesky factorization. An element-wise version of that last operation is also described in Algorithm 1.

Let us illustrate the use of linear algebra operations with an example application from [12]. In this work, the authors study the propagation of electro-magnetic waves on the surface of an object; the whole domain can be discretized using Boundary Elements Method (BEM). The interactions between each two elements according to Maxwell's laws are then linearized, *i.e.* the relationships between the unknown values of interest, namely electric and magnetic fields, are approximated using linear relationships. Coefficients describing all the relationships between the unknowns are gathered into a matrix $\mathbf{A}$ which entirely describe the considered problem. The solution of the problem, *i.e.* finding the values of all the unknowns, then comes down to solving the linear system $\mathbf{A} \cdot \mathbf{x} = \mathbf{b}$ where $\mathbf{b}$ corresponds to boundary conditions. This can be performed in various different ways according to the characteristics of $\mathbf{A}$. One classical direct solution method in the dense case, *i.e.* when $\mathbf{A}$ features a majority of non zero values, consists in performing the factorization of $\mathbf{A}$ as a product of two triangular matrices and then apply two triangular solutions successively to retrieve the final values of $\mathbf{x}$. We provide below an outline of the steps of such a method in the case of a symmetric positive definite matrix $\mathbf{A}$, largely used as case study in the following chapters.

Algorithm 1: Element-wise Cholesky factorization algorithm

Input: ($\mathbf{A}$): $\mathbf{A}$ is $m \times m$ symmetric positive definite
Output: ($\mathbf{L}$): $\mathbf{L}$ is $m \times m$ lower triangular such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^{\mathsf{T}}$

```
1 for k = 1 to m do
2 |   A_{k,k} = sqrt(A_{k,k})
3 |   for i = k + 1 to m do
4 |   |   A_{i,k} = A_{i,k} / A_{k,k}
5 |   |   for j = k + 1 to i do
6 |   |   |   A_{i,j} -= A_{i,k} · A_{j,k}          update operations
7 L ← lower triangular part of A
```

1. factorize $\mathbf{A}$ using the Cholesky factorization: compute $\mathbf{L}$ such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^{\mathsf{T}}$;
2. solve the triangular system: compute $\mathbf{y}$ such that $\mathbf{L}^{\mathsf{T}} \cdot \mathbf{y} = \mathbf{b}$;
3. solve the triangular system: compute $\mathbf{x}$ such that $\mathbf{L} \cdot \mathbf{x} = \mathbf{y}$.

This makes use of the Cholesky factorization, step (1), which is a central linear algebra operation detailed in Algorithm 1 in its element-wise form.

Linear algebra is such a central part of numerical scientific computing that the formalization of its essential operations as efficient algorithms dates back from the early times of computer science. The set of Basic Linear Algebra Subprograms (`BLAS` [73]), developed as a library since the 1970s, actually provides a formal description of the core linear algebra operations as generic algorithms. Starting from the most basics operations, such as scalar multiplication or dot product, it followed an incremental development to build up three subsets, referred as *levels*, of algorithms now considered as *de facto* standard specifications of the associated operations. Each describes the elementary computations required for a given operation as would be performed sequentially by a single computing unit. It is therefore not very different from a plain pseudo-code description such as Algorithm 1 for Cholesky factorization. The algorithms provided in the `BLAS` library are used as a common basis to build hardware specific sequential implementations of the linear algebra operations at a low level. Such implementations, including the widely used open source `OpenBLAS` [69, 74] and proprietary `MKL` [72] library for Intel x86, make use of architecture specific features to reach high performance on a single computing unit. The formal descriptions of the algorithms provided by `BLAS` are also used as building blocks for more elaborated operations in other linear algebra libraries such as `LAPACK` [73].

Although the term may have a broader meaning in the literature, in this work we denote *kernel* the algorithm specification for a given linear algebra operation at element level. We call *arithmetic operations* the computations performed directly on elements of the matrix in the execution of a kernel. `BLAS` and `LAPACK` nomenclature is used to designate those kernels, such as GEMM for matrix multiplication, SYRK for symmetric rank-k update, GETRF for LU factorization and POTRF for Cholesky factorization.

Algorithm 2: Tiled matrix multiplication algorithm (GEMM)

Input: ($\mathbf{C}$, $\mathbf{A}$, $\mathbf{B}$): $\mathbf{C}$ is $m_b \times n_b$, $\mathbf{A}$ is $m_b \times k_b$, $\mathbf{B}$ is $k_b \times n_b$
Output: ($\mathbf{C}$): such that $\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$

1 **for** $k = 1 \dots k_b$ **do**
2   **for** $i = 1 \dots m_b$ **do**
3   **for** $j = 1 \dots n_b$ **do**
4   GEMM(i, j, k): GEMM($\mathbf{C}(i, j), \mathbf{A}(i, k), \mathbf{B}(k, j)$)

1.1.2 Parallel and Distributed Computing

Scientific applications requirements in terms of scale and precision is always growing, making them ever more computationally intensive. At the same time, sequential computing on a single processing unit is an execution model inherently limited in terms of performance. To cope with the soaring demand of computing power, technological breakthroughs allowed to leverage sequential hardware performance by enabling two levels of parallelism. In the context of shared memory computing units, multi-core architecture

Algorithm 3: Tiled symmetric rank-k update algorithm (SYRK)

Input: $(\mathbf{C}, \mathbf{A})$: $\mathbf{C}$ is $m_b \times m_b$ symmetric, $\mathbf{A}$ is $m_b \times n_b$
Output: $(\mathbf{C})$: such that $\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{A}^\mathsf{T}$

```
1 for k = 1 ... n_b do
2     for i = 1 ... m_b do
3         SYRK(i,k): SYRK(C(i,i), A(i,k))
4         for j = i − 1 ... n_b do
5             GEMM(i,j,k): GEMM(C(i,j), A(i,k), A(j,k))
```

Algorithm 4: Tiled Cholesky factorization algorithm (POTRF)

Input: $(\mathbf{A})$: $\mathbf{A}$ is $m_b \times m_b$ symmetric positive definite
Output: $(\mathbf{L})$: $\mathbf{L}$ is $m_b \times m_b$ lower triangular such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\mathsf{T}$

```
1 for k = 1 ... m_b do
2     POTRF(k): POTRF(A(k,k))
3     for i = k + 1 ... m_b do
4         TRSM(i,k): TRSM(A(i,k), A(k,k))
5         SYRK(i,k): SYRK(A(i,i), A(i,k))
6         for j = k + 1 ... i − 1 do
7             GEMM(i,j,k): GEMM(A(i,j), A(i,k), A(j,k))
8 L ← lower triangular part of A
```

has emerged and is nowadays standard for all processors and more recently accelerators (GPU). In the meantime, computing platforms have become massively parallel, allowing distributed execution of applications simultaneously using several computing nodes. The scale of the problems considered nowadays requires to make use of both levels of parallelism in order to ensure a reasonable running time. Therefore, it is paramount to understand and optimize the parallel behavior of applications, especially the underlying linear algebra operations, to achieve the highest possible performance.

1.1.2.1 Parallel Computing Platforms

Modern computing platforms gather several computing units referred as *nodes*. They are generally dedicated pieces of hardware composed of a memory (RAM), a processor (CPU) featuring multiple cores and sometimes one or several accelerators that are actually GPU. We refer to the CPU cores and GPUs as *workers*. They are connected together and to the node memory, generally via a bus. While GPUs feature their own dedicated memory, CPUs generally only have cache memories of smaller size, organized in hierarchical levels. For homogeneous nodes, *i.e.* composed of a single type of CPU, the set of workers can be considered as a single shared memory computing unit, fetching from and writing data to the RAM. At the platform level, nodes are connected via a high performance network whose characteristics, such as the topology, aim at providing the largest bandwidth and

Algorithm 5: Tiled LU factorization algorithm (GETRF)

Input: ($\mathbf{A}$): $\mathbf{A}$ is $m_b \times m_b$
Output: ($\mathbf{L}$, $\mathbf{U}$): $\mathbf{L}$ is $m_b \times m_b$ lower triangular, $\mathbf{U}$ is $m_b \times m_b$ upper triangular, such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$

```
1 for k = 1 ... m_b do
2     GETRF(k): GETRF(A(k,k))
3     for i = k+1 ... m_b do
4         TRSM(i,k): TRSM(A(i,k), A(k,k))
5         TRSM(k,i): TRSM(A(k,i), A(k,k))
6         for j = k+1 ... m_b do
7             GEMM(i,j,k): GEMM(A(i,j), A(i,k), A(k,j))
8 L ← lower triangular part of A
9 U ← upper triangular part of A
```

smallest latency between any two nodes.

1.1.2.2 Data Distributions

In this work, we are considering the parallel and distributed execution of linear algebra operations under this quite generic computing platform model. To perform operation in parallel using several nodes, input and output data are divided into chunks and distributed to nodes. In practice, for linear algebra operations, data correspond to matrices whose values can be gathered in different manner to make chunks. Non-zero values of sparse matrices can be packed in order to reduce the memory footprint, for example. In the following, we consider that matrices are divided into square *blocks* or *tiles*. Each tile corresponds to a sub-matrix of dimensions $b \times b$ with contiguous indices. It is particularly suited to the dense case: as all tiles are identical, linear algebra operations of the same type can be efficiently performed at the scale of a tile using a single highly optimized kernel. Besides, it leverages the parallelism of the entire global operation as the execution of each operation at the scale of one tile is a task performed by a single worker of a node. Several operations performed on different tiles can therefore be executed in parallel by different workers, whether in the same or in different nodes.

Let us now introduce the notations to describe the division of matrices into square tiles using the Cholesky factorization as example. For such an operation, matrix $\mathbf{A}$ represents both the input and the output since matrix $\mathbf{L}$ is written over $\mathbf{A}$. Assume that $\mathbf{A}$ is of dimensions $m \times m$ elements and only the lower triangular part of $\mathbf{A}$ is referenced. Let $(i,j) \in \{1,\ldots,m\}^2$, $i \geqslant j$. We denote $\mathbf{A}_{i,j}$ the element of $\mathbf{A}$ in position (i,j). We extend the notation to sets of indices: let $I \subset \{1,\ldots,m\}$ and $J \subset \{1,\ldots,\max(I)\}$, $\mathbf{A}_{I,J}$ denotes the sub-matrix of $\mathbf{A}$ with positions $(i,j) \in I \times J$. To simplify some notations, we also define that, when a single dot is used instead of a set of indices, it means all valid indices. Now, for a given tile size b, matrix $\mathbf{A}$ can be divided into tiles in the sense that we are considering sub-matrices of $\mathbf{A}$ that are $b \times b$ squares. Let $m_b = \frac{m}{b}$, then for any

$(i, j) \in \{1, \ldots, m_b\}^2,\ i \geqslant j$, the tile in position (i, j) is denoted $\mathbf{A}(i, j)$ and corresponds to the sub-matrix:

$$\mathbf{A}(i, j) = \mathbf{A}_{\{(i-1)m_b+1,\ldots,im_b\},\{(j-1)m_b+1,\ldots,jm_b\}} \tag{1.1}$$

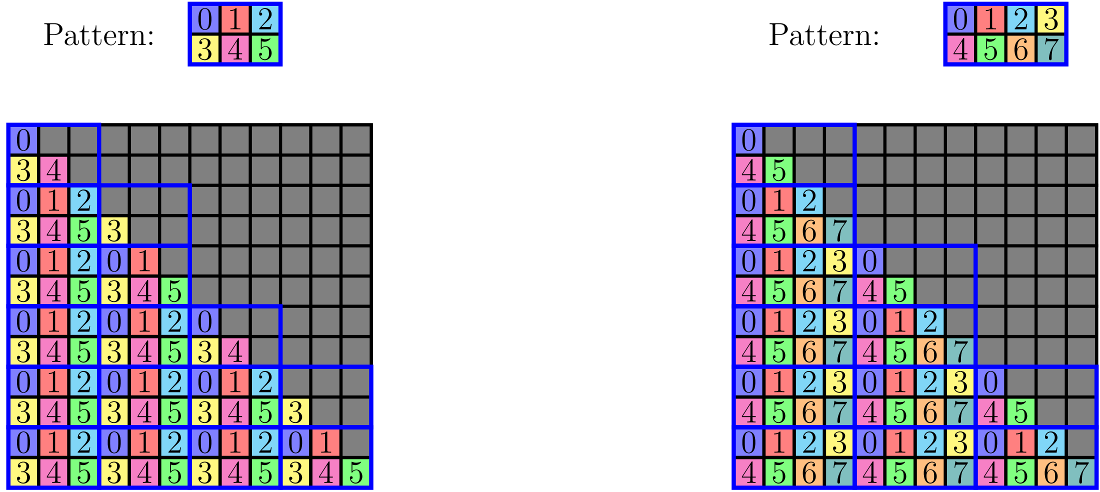
For simplicity and unless otherwise specified, we use elements and tile indices starting at 1 in the rest of the document.

The tile size b is generally selected such that $\frac{m}{b}$ is integer although the formal definition of tiles, Equation 1.1, can easily be extended to deal with non-square ones. Applying kernels to non-square tiles may however raise technical difficulties. Hence, unless otherwise specified, we assume in this work that $m_b = \frac{m}{b} \in \mathbb{N}^*$, which is not a restrictive assumption in practice.

Then, in a distributed setting, we denote *data distribution* the assignment of each tile to a node. In the following, the distribution of a matrix is represented by another matrix denoted $\mathbf{D}$ whose elements are integer indices of nodes, starting at 0: the element in $\mathbf{D}$ at a given position is the index of the node owning the tile of the matrix at the same position. Let us illustrate it again using the Cholesky factorization: assume $\mathbf{A}$ is a symmetric positive definite matrix of size $m \times m$, divided in m_b^2 tiles of size $b \times b$ where $m_b = \frac{m}{b} \in \mathbb{N}^*$. It is both the input and output of the Cholesky factorization operation. Since $\mathbf{A}$ is symmetric, the whole input data actually contains only $\frac{m_b(m_b+1)}{2}$ tiles, one half of the matrix. We assume that only the lower half of $\mathbf{A}$ is referenced. A data distribution of distribution $\mathbf{A}$ among P nodes is a assignment of the tiles of $\mathbf{A}$ to the nodes represented by a matrix $\mathbf{D} \in \mathcal{M}_{m_b \times m_b}(\{0, \ldots, P-1\})$. Let $\mathbf{D}$ be such a distribution, let $(i, j) \in \{1, \ldots, m_b\},\ i \geqslant j$ and $p_1 \in \{0, \ldots, P-1\}$. Then $\mathbf{D}_{i,j} = p_1$ means that all the elements of tile $\mathbf{A}(i, j)$ are present in the memory of the node with index p_1. In this work, we assimilate each node to its index and use the following terminology to describe the distribution: we say that the node p_1 *owns* the tile $\mathbf{A}(i, j)$ and conversely that tile the $\mathbf{A}(i, j)$ is *assigned* or *allocated* to the node p_1. When there is no possible ambiguity about the matrix considered, such as for the Cholesky factorization, we can simply assimilate the tile $\mathbf{A}(i, j)$ to its position (i, j) and say that p_1 is present at position (i, j). Thereby defined, data distributions are called 2D, which indicates that each tile is owned by a single node. Such a configuration is standard in the literature and libraries for parallel linear algebra operations. However more complex distributions making use of data replication, referred to as 2.5D or 3D, exist. They are more precisely defined and discussed to some extent in Section 2.2.3 of Chapter 2. In this work, unless otherwise specified, we consider 2D data distribution.

1.1.2.3 Block Cyclic Distribution

In a parallel and distributed setting, the **B**lock **C**yclic distribution (**BC**) is the most widely used data distribution scheme for linear algebra operations. In many linear algebra libraries, it is the default [40, 64], and sometimes the only available [57] data distribution, in particular in the `ScaLAPACK` library [22] which can be considered as reference since it is the extension of `LAPACK` to the parallel and distributed setting. It is very regular, which implies quite simple data dependency and communication schemes that are easy to



(a) $P = 6$ nodes; 2×3 pattern.

(b) $P = 8$ nodes; 2×4 pattern.

Figure 1.1: Examples of 2D BC distribution over a 12×12 symmetric matrix (only lower triangular part referenced). Each color represents a node.

describe. This contributed to its large popularity and widespread usage, particularly in the early times of parallel computing, because of the technical difficulty of implementation to handle task dependencies and communications, as detailed in the following Sections 1.1.2.4 and 1.1.2.5.

Assuming a parallel and distributed execution using P nodes, the standard BC distribution is based on a repeated $p \times q$ pattern that we denote $\mathbf{G}$, for "grid", such that $P = pq$. Node indices are associated with a position (x, y) in the pattern in a round-robin fashion. For instance, a row-wise distribution is defined by:

$$\forall (x, y) \in \{1, \ldots, p\} \times \{1, \ldots, q\} : \quad \mathbf{G}_{x,y} = (x-1)q + (y-1)$$

The BC distribution $\mathbf{D}$ of a matrix of dimensions $m_b \times n_b$ tiles is then defined as:

$$\forall (i, j) \in \{1, \ldots, m_b\} \times \{1, \ldots, n_b\} : \quad \mathbf{D}_{i,j} = \mathbf{G}_{i \bmod p, j \bmod q}$$

Illustrations of BC distributions can be seen Figures 1.1a and 1.1b.

Several properties of the BC distribution are common to all pattern-based distributions.

1. The load balancing associated with the entire distribution $\mathbf{D}$ depends on the balancing of positions in $\mathbf{G}$ among the nodes. Because the distribution is simply the replication of the pattern, all the nodes in $\mathbf{G}$ appear regularly in $\mathbf{D}$. Thus, assuming that the pattern is replicated many times, *i.e.* the matrix size is large compared to the pattern size, for each type of task, all nodes present in the pattern are roughly assigned the same number of tasks. Hence, in the case of homogeneous tasks, in particular when dealing with dense matrices, if all nodes appear the same number of times in $\mathbf{G}$, then the entire distribution is well balanced in terms of workload.

2. Pattern-based distributions imply a local load balancing which in turn translates into a good load balancing in the course of the execution of an operation. This is a particularly desirable feature for operations where the domain shrinks as the computation progresses, such as LU and Cholesky factorization because it maintains load balancing among all nodes even in the last phase of the operation.

3. The set of nodes that appear in each row and column of $\mathbf{D}$ is the same as the set of nodes in the corresponding row or column of $\mathbf{G}$.

In a parallel and distributed context, the characteristics (1) and (2) are all the more desirable that the pattern used for a complete distribution is small. The pattern used by the BC distribution is actually the smallest possible using P nodes since each node appears exactly once. When using such a pattern of size $p \times q$, there are p different nodes in each column of $\mathbf{D}$ and q different nodes on each row. As illustrated in the analysis in Section 2.2.2, for many linear algebra operations the number of different nodes per row and column in a distribution is often a characteristic that defines the communication volume it generates. For the BC distribution, since $P = pq$, minimizing p and q simultaneously, or $p + q$, implies $p = q = \sqrt{P}$, which corresponds to a square pattern. It is the minimum achievable number of nodes per row and column for a pattern-based distribution using all P nodes.

Despite featuring such interesting properties, the BC distribution is not optimal in all situations. Providing more efficient alternatives, especially pattern-based distributions, in specific configurations is the main objective of this work.

1.1.2.4 Task Assignment

The parallel execution of a linear algebra operation in a distributed setting occurs at the scale of tiles. At this level, we say that nodes performs *tasks* on elements gathered as tiles in order to complete the entire operation. Indeed, a *blocked* or *tiled* version of any operation can be defined that uses tiles as the elementary pieces of data handled when performing tasks. Algorithm 4 illustrates a tiled version of the Cholesky factorization algorithm. The generic definition of an operation, as shown in Algorithm 1 for example, and its tiled version are equivalent regarding the set of arithmetic operations actually performed on each element of the matrices. Tiled algorithms use a coarser granularity of data and tasks but describe the execution of operations in a form that better fits parallel and distributed settings. A task *assigned* or *allocated* to a node is actually performed by one of its worker executing the associated kernel on the required tiles. Therefore we use the `LAPACK` kernel nomenclature to refer to tasks in the description of tiled algorithms; though they are not formally kernels they are completed by executing the corresponding kernel on a worker. Going back to the Cholesky factorization example using its tiled algorithm description in Algorithm 4: let $k \in \{1, \ldots, m_b\}$, let $(i, j) \in \{1, \ldots, m_b\}$, $i \geqslant j$, and let us consider the task of updating tile $\mathbf{A}(i, j)$ at iteration k, *i.e.* computing $\mathbf{A}(i, j) \mathrel{+}= \mathbf{A}(i, k) \cdot \mathbf{A}(j, k)$. According to `LAPACK` nomenclature, we denote it $\text{GEMM}(\mathbf{A}(i, j), \mathbf{A}(i, k), \mathbf{A}(j, k))$ further simplified as $\text{GEMM}(i, j, k)$, as can be observed line 7. It corresponds to a matrix multiplication applied to tiles. To illustrate the concept in a generic way, let us assume for now that there is no connection between

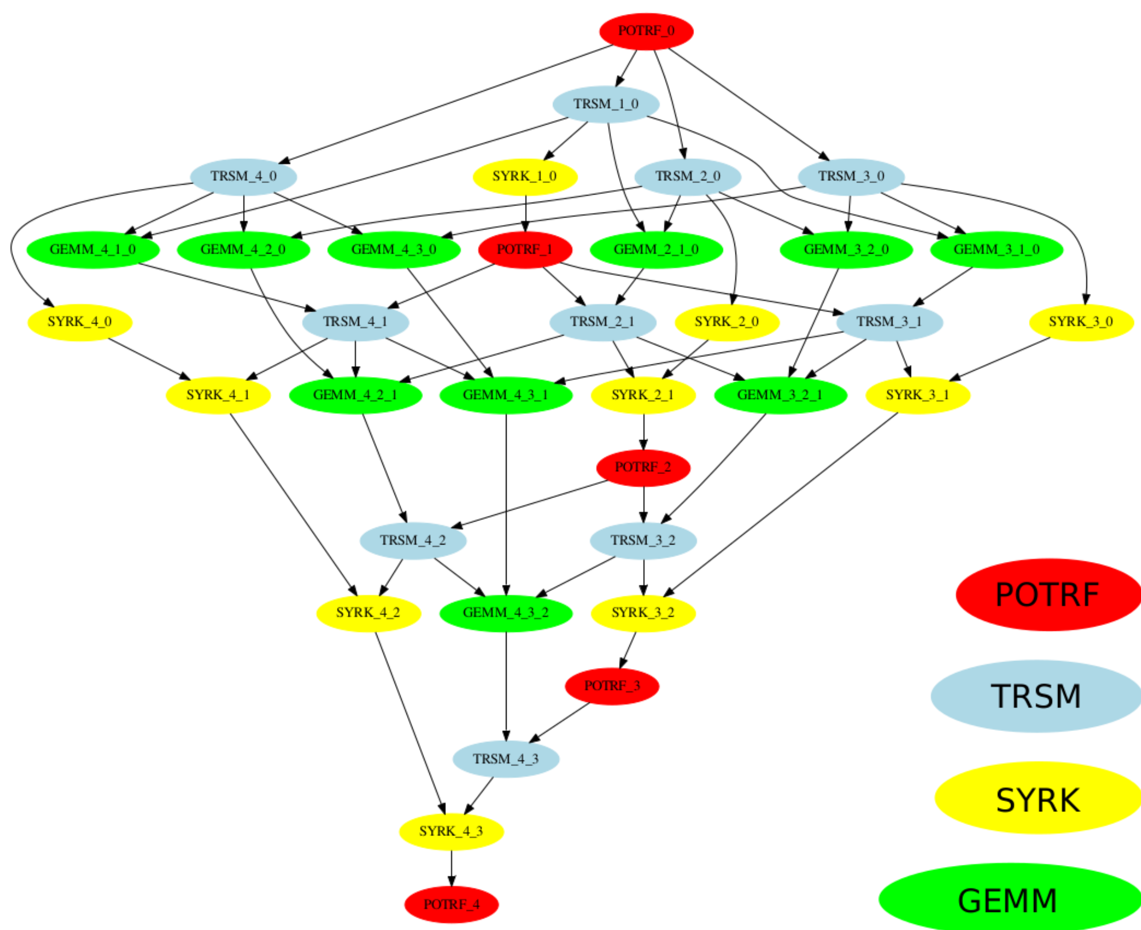


Figure 1.2: cDAG of the Cholesky factorization for a 5×5 matrix

data distribution and task assignment. Let us consider that this task is assigned to the node p_1. To complete it, one worker of p_1 executes the sequential kernel GEMM on tiles $\mathbf{A}(i,j)$, $\mathbf{A}(i,k)$, and $\mathbf{A}(j,k)$. Since kernel implementations are optimized for specific architectures and the workers of p_1 may be different from one another, the completion of the same task can correspond to different actual executions of the associated kernel. The *task-based* execution model, described in Section 1.2, however allows to study parallelism at node level without having to consider this aspect: inside a given node, a dedicated tool, the *runtime system*, dynamically allocates tasks to workers.

In tiled algorithms, dependencies occur between tasks instead of arithmetic operations. They can be abstracted in the form of a ***c****omputational* ***D****irected* ***A****cyclic* ***G****raph* (**cDAG**) as could be done at element level for arithmetic operations. In such a cDAG, tasks correspond to vertices and dependencies to directed edges: an edge between two vertices means that the task associated with the child vertex cannot be started before the task associated with the parent vertex is completed. As example, the cDAG of the Cholesky factorization of a 5×5 tiled matrix can be seen Figure 1.2. Vertices are labeled with the name of their associated tasks.

The set of scalar arithmetic operations necessary to complete a given linear algebra operation remains the same regardless of how the matrix is divided into tiles and distributed among nodes. One can note that, on the contrary, the computation time required to execute an operation may depend on the division of the matrix. The granularity of the division nevertheless defines the number of tasks that compose the operation: a larger tile size b implies that the input or output matrices are divided into fewer tiles and therefore the whole operation consists of fewer tasks. Moreover, each task is completed by executing a kernel on a tile, which is simply a $b \times b$ sub-matrix. Now, the efficiency of a kernel execution on a worker depends on the size of the matrix to which it is applied, in particular it cannot reach the best performance if the matrix is too small. There is therefore a trade-off in selecting the tile dimension b:

- it must be small enough so that the operation consists of sufficiently many tasks in order to ensure load balancing when assigning them to nodes;
- it must be large enough so that the corresponding kernel reaches the best performance when performed on the selected worker.

That second constraint is quite challenging to satisfy because nodes can be heterogeneous and are themselves composed of workers of different types (CPU cores, GPUs). Hence, the optimal granularity of data to execute a given kernel depends on the worker. Though it does not raise any problem from a formal point of view, the choice of the value of b is a difficult topic in practice as it directly affects the performance. It is therefore discussed on a case by case basis depending on the experiment carried out, notably in Section 2.3.1 of Chapter 2.

At the scale of a computing platform, assigning tasks to nodes dynamically is very challenging. Indeed, in order to ensure the correctness of the operation, *i.e.* that the set of dependencies is satisfied during the execution, a dynamic strategy would require to maintain a consistent distributed knowledge among nodes of the tasks assignment to be able to issue communications between the correct sender and receiver. Furthermore, since the input and output matrices are distributed between nodes, it would require being able to specify communications dynamically, which is difficult, as explained in the following Section 1.1.2.5. One can even imagine a dynamic reallocation of tiles during execution to better fit dependencies and communications constraints, however such a strategy is way too challenging for the current execution models and tools. Hence, the state-of-the-art and only practical method used in linear algebra libraries is to assign tasks to node statically prior to the execution. The most widely used method is called the *owner computes* rule: it states that a node which owns a tile is assigned all the tasks that modify this tile, *i.e.* overwrite its values. One can note that, in the case of a task modifying multiple tiles, the owner computes rule may be very difficult to apply because if several tasks modify a given subsets of tiles, then all those tiles must be allocated to the same node, which may strongly restrict the possibilities to design distributions. The operations we consider however do not involve such tasks and designing distributions adapted to those tasks is beyond the scope of this work. Using again as example the Cholesky factorization of matrix $\mathbf{A}$ of dimensions $m_b \times m_b$ tiles: let $(i, j) \in \{1, \ldots, m_b\}$, $i \geqslant j$, and assume that

node p_1 owns tile $\mathbf{A}(i, j)$. Then p_1 is assigned all the tasks which modify $\mathbf{A}(i, j)$: assuming $i > j$, these are TRSM(i, j) and GEMM(i, j, k) for all $1 \leqslant k < j$. An illustration of the assignment of tasks according to the owner computes rule can be seen on Figure 1.3; it uses a 3D representation to depict the set of all tasks, the third dimension corresponding to iterations of the algorithm. On this figure, we can see that the node 2 owns the tile $\mathbf{A}(5, 3)$ therefore performs all the tasks that modify it, namely GEMM$(5, 3, 1)$, GEMM$(5, 3, 2)$ and TRSM$(5, 3)$. We say that we use a *static* distribution for a given operation if a 2D data distribution is used and the execution follows the owner computes rule. In the following, unless otherwise specified, we assume that the owner computes rule always applies. Therefore, we qualify indifferently as *static* or 2D distributions where each tile is owned by a single node.

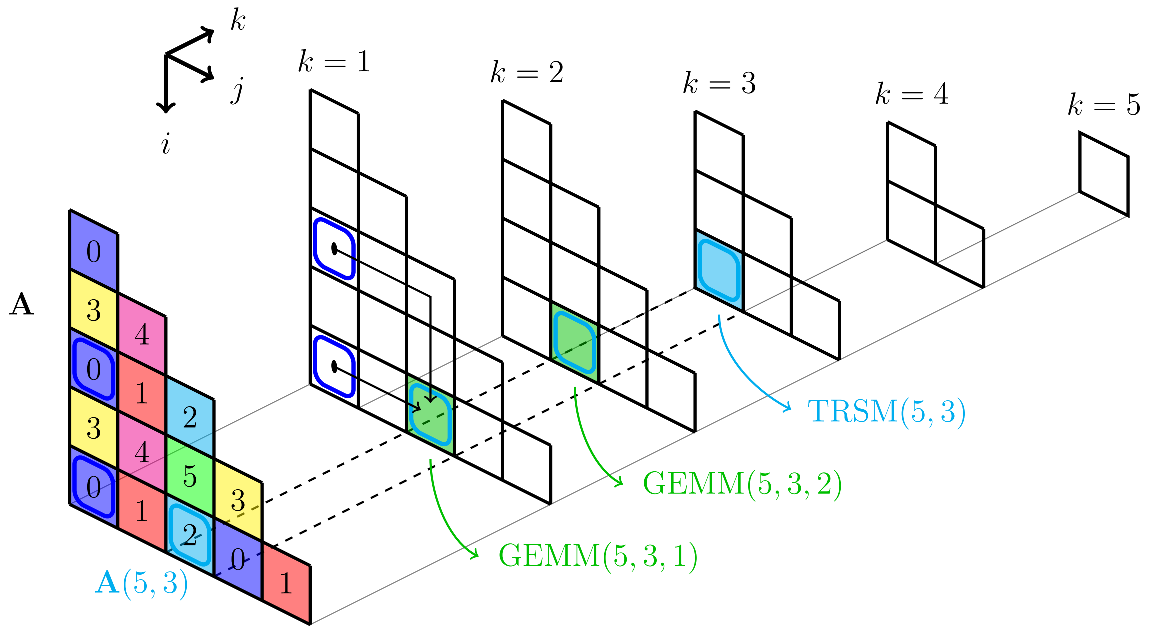


Figure 1.3: Illustration of the owner computes rule (dashed horizontal lines) and inter-node communications (black arrows) (Cholesky factorization of a 5×5 matrix using BC distribution with $P = 6$ nodes)

1.1.2.5 Communications

For a given linear algebra operation, tile-based tasks often involve several tiles as input and a single tile as output. The output tile is generally overwritten, *i.e.* its values are modified in place, it is thus also considered an input of the tasks. In the parallel and distributed setting considered, since the matrices are split and distributed among nodes on a tile basis, the completion of a task by a given node often requires some of the input tiles to be transferred to the node. More simply, if a node is assigned a task requiring input tiles that it does not own, it must fetch them from the owner node, inducing communication of data over the network. Let us illustrate it using the Cholesky factorization example: let

$\mathbf{A}$ of size $m_b \times m_b$ tiles, $(i, j) \in \{1, \ldots, m_b\}$, $i \geqslant j$, and $1 \leqslant k < j$. Assume that the node p_1 is assigned the task GEMM(i, j, k). Let us assume also that p_1 owns the tile $\mathbf{A}(i, j)$ but the tiles $\mathbf{A}(i, k)$ and $\mathbf{A}(j, k)$ are owned respectively by two other different nodes p_2 and p_3. Then to perform the task GEMM(i, j, k), *i.e.* compute $\mathbf{A}(i, j) \mathrel{+}= \mathbf{A}(i, k) \cdot \mathbf{A}(j, k)$, p_1 must fetch the input tiles $\mathbf{A}(i, k)$ and $\mathbf{A}(j, k)$ from their respective owner nodes: it implies the communication of $\mathbf{A}(i, k)$ from p_2 to p_1 and the communication of $\mathbf{A}(j, k)$ from p_3 to p_1. Figure 1.3 illustrates the inter-node communications: node 2 is assigned the task GEMM$(5, 3, 1)$ which requires tiles $\mathbf{A}(3, 1)$ and $\mathbf{A}(5, 1)$ as input which are both owned by node 0 (in this case p_2 and p_3 are the same node). Hence two communications must occur from 0 to 2 to transfer these tiles. In this work, we simply denote *communication* the transfer of data of between any two nodes. When considering tiled algorithms, such as in Chapter 2, one communication corresponds to the transfer of one tile, when we consider element-wise algorithms, it is the transfer of a single element.

As can be observed from the previous example, the set of communications required during the execution of an operation depends on both the data distribution and the task assignment. Besides, the communications must be scheduled in accordance with the task dependencies in order to ensure the correctness of the execution. Until recently, for a given operation, communications between nodes had to be explicitly described as part of the whole execution and implemented using dedicated software technologies, the most prominent one being the **M**essage **P**assing **I**nterface (`MPI`) protocol. The interdependence between the data distribution, the tasks assignment, and the scheduling of communications, plus the fact that communications directives must be explicitly described prior to the execution, make the design and implementation of distributed applications extremely challenging. This heavily influences the design of linear algebra libraries that rely on very regular distributions, especially static BC distribution, for both data and tasks, allowing a simpler expression of communication schemes using fixed `MPI` primitives. Furthermore, the majority of linear algebra libraries strictly applies owner computes rule as it simplifies the problem by merging data and tasks distributions. One of the main objectives in the development of runtime systems is to completely alleviate those issues, as discussed in details in Section 1.2.

1.2 Task-Based Execution Model and Runtime Systems

As mentioned in Section 1.1.2.5, developing parallel distributed linear algebra applications is challenging. Many libraries, in particular `ScaLAPACK`, are therefore designed using the same building blocks: usage of a very regular data distribution, generally Block Cyclic (BC), assignment of tasks according to the owner computes rule and explicit description of the communications. Such a development scheme allows for designing robust and efficient applications that are tailored for a given set of operations on a given hardware. The specific feature of such a design nevertheless comes with strong drawbacks: applications are generally challenging to develop, they lack flexibility and portability which means that they are not easily adapted to other operations or platforms, they cannot be easily

modified which strongly hinders the possibility to follow the evolution of both formal models and hardware architecture.

For about a decade, the task-based execution model has gained much interest in the field of HPC and is now regarded as one of the most promising research direction for the development of efficient and flexible applications in the parallel distributed setting. The core idea is to decouple the description of the actual computation from data and task dependency management. While the first aspect is application dependent and left to a specific library, the second one is delegated to a dedicated tool, the *runtime system* or *runtime scheduler*, which handles dependencies during the execution. On one hand this allows for the development of very specific and complex libraries that can achieve high performance without having to explicitly describe data and task distributions and communications. On the other hand, since runtime systems take decision during the execution, they enable the elaboration and deployment of fine grain data management and scheduling strategies. This design provides a much higher flexibility in the development of both aspects and leverages the large performance gains related to task and communication scheduling.

As the task-based model offers such important advantages over application specific designs, many modern linear algebra libraries now fully implement it and solely rely on the underlying runtime system to handle dependencies and communications. For instance, it is the case of the dense linear algebra libraries `Chameleon` [3–5], `DPLASMA` [23] or `FLAME` [47] or the sparse linear algebra library `PaStiX` [49], which can be interfaced with different runtime systems. At the same time, several runtime systems are developed, such as `StarPU` [11, 33], `PaRSEC` [24, 25], `SuperMatrix` [30, 66] or `OmpSs` [35], that are based on different task management models. Beyond the advantages related to development and design, libraries relying on the task-based execution model and runtime systems are shown to achieve high performance. As example, in [4], the authors show that `Chameleon` backed by `StarPU` and `DPLASMA` using `PaRSEC` reach significantly higher performance on classic linear algebra operations, namely matrix multiplication and Cholesky factorization, than the reference `ScaLAPACK` library.

In this entire work, we elaborated data distribution strategies assuming that all operations are executed under the task-based execution model. Indeed, it is the ability of runtime systems to automatically manage tasks and dependencies solely from the description of the distribution of data that enabled to actually implement the original distribution strategies proposed as our core contribution.

1.2.1 Dynamic Task and Communication Management

In the task-based execution model, runtime systems are dedicated to handling tasks and communications and ensuring the correctness of the execution according to the dependencies. Different programming paradigms exist to fulfill this role that are used in existing runtime systems. In this work, we only consider the widely used `StarPU` tool that is based on concepts of the **S**equential **T**ask **F**low (**STF**) paradigm. Though its internal functioning is not fully generic, its behavior still illustrates the advantages associated with the task-based execution model. In the context of this work, such a choice is not expected to

impact the results since the data allocation strategies we elaborated are formally agnostic of the implementation characteristics of the runtime system.

Let us consider the execution of a linear algebra operation in a parallel distributed setting using a dedicated library backed by the `StarPU` runtime system. Then each node involved in the execution runs an instance of the runtime system. At the scale of a given node, during the execution, several tasks assigned with the node may be available at a given time, *i.e.* their associated input data is in memory and their inward dependencies are satisfied. Then the runtime system is in charge of deciding how to execute those tasks, which includes:

- defining a schedule for the tasks;
- for a given task, selecting which worker of the node executes the associated kernel.

The decisions are taken dynamically according to the state of the node: waiting tasks (that are submitted but not ready yet), tasks currently executed, idle workers, and a predefined scheduling policy. To come up with relevant scheduling decisions, the runtime policy can use as input task priorities provided by the application and per-worker performance models that are estimated from previous executions. Priorities allow the user to provide, via the application, global information about a target schedule. They are often computed as the length of the critical path up to the final task, like in the `Chameleon` library. Performance models help selecting the best suited worker for a given task by estimating how fast the associated kernel is expected to execute on each available worker. Hence, the runtime system is able to deal with nodes featuring heterogeneous workers, especially GPUs, and make the best use of their specific computing capacities. For example, the default `StarPU` scheduling policy, called *dmdas*, uses a single ordered queue of tasks from where the highest priority task is pulled each time a worker becomes idle or a new task is assigned to the node. It then assigns the task to a worker in order to minimize its expected completion time.

At the scale of the whole platform, the difficulty is to ensure the correctness of the execution in a distributed way, *i.e.* ensure that tasks respect the dependencies of the operation. In the considered distributed setting, it implies in turn that communications must occur accordingly to the schedule of the tasks. Under the task-based execution model, the runtime system is in charge of enforcing the task dependencies and issuing the communications at the right time of the schedule. It does so using the informations provided by the application: (i) the complete data distribution among all nodes and (ii) the set of all tasks to perform along with their respective input and output. The application also determines the assignment of tasks to nodes, usually relying on the owner computes rule. Following the principles of the STF paradigm, each instance of the `StarPU` runtime system running on each node unrolls the complete cDAG of the operation as tasks are completed, translating data dependencies into task dependencies. In practice, each instance of `StarPU` also eliminates the tasks of the cDAG without any dependency relationship to its own tasks. Such a "pruning" of the cDAG enables to maintain the scalability of the runtime system as a whole and limits its overhead. Since each task is issued by the application with its associated input and output, and since each instance of

the runtime system knows the entire data distribution, it can thus infer which communications are required to copy all input tiles, respectively elements, from their owner to the node performing the task. More precisely, a dependency between two tasks implies that the output of the parent task is required as input for the child task. If those tasks are not assigned to the same node, `StarPU` simply requires a communication between the nodes and inserts it as a regular task inside the cDAG between the two tasks. On one hand, this mechanism ensures the correctness of the computations because a communication is considered as an additional inward dependency for the task of the receiving node that must be satisfied before the task starts. Furthermore, it completely alleviates the burden of explicitly describing communications since they are inferred from the data distribution and the dependencies.

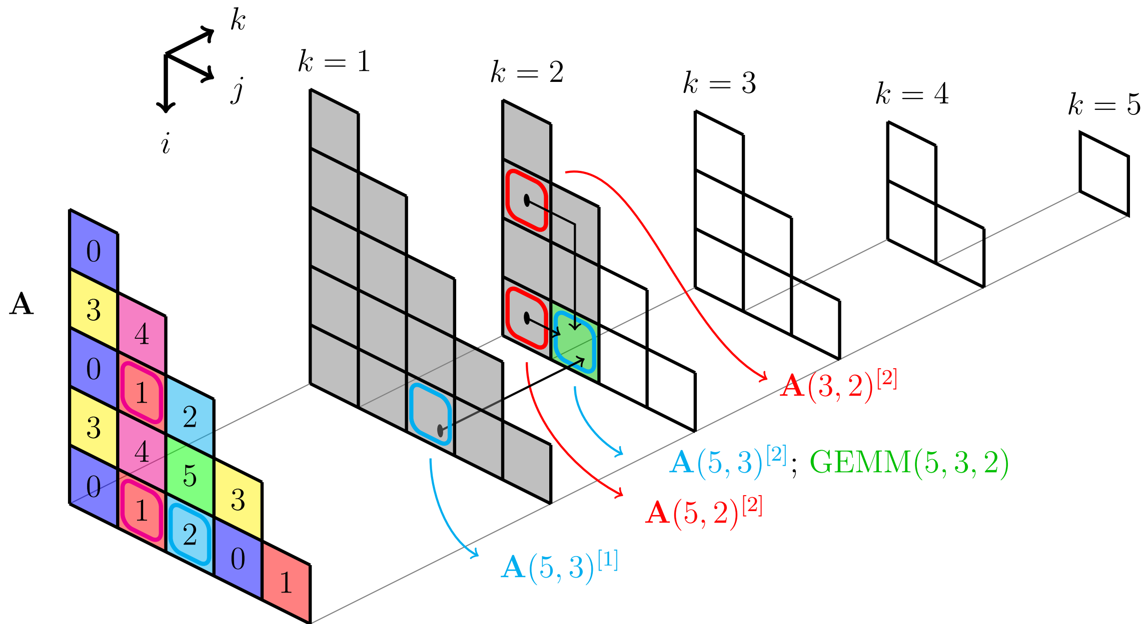
The task-based execution model and the runtime systems also enable the use of completely irregular data distributions that can be more adapted to the considered operation or platform configuration. The use of non standard data distributions that can be seamlessly handled by runtime systems is the central point of Chapter 2. An illustration of such distributions adapted to operations with symmetric input can be found in Section 2.3 of this chapter.

1.2.2 Asynchronous Execution

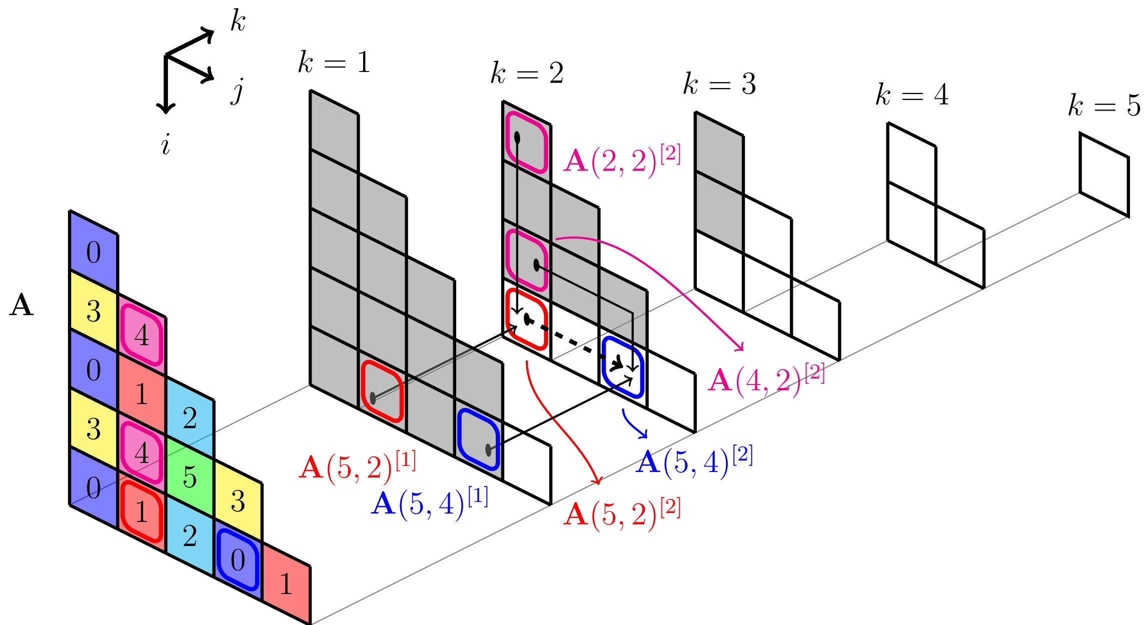
For most linear algebra operations, several elements of the output matrix are updated incrementally according to the sequence of arithmetic operations that compose the algorithm. In the case of LU and Cholesky factorization, this corresponds to the outer loop in the algorithm description. As an example, for the Cholesky factorization, described in Algorithm 1, all elements $\mathbf{A}_{i,j}$, for all $(i, j) \in \{1, \ldots, m\}^2$, $i > j$, are updated using the operation line 6 at each iteration of the outer loop line 1.

Let us now consider the execution of those operations in a parallel and distributed setting. As already mentioned, in such a configuration, classic `MPI`-based linear algebra libraries use explicit descriptions of communications between nodes. Because of the difficulty to elaborate such description while ensuring the correctness of the computations, those libraries often rely on synchronization. A synchronized execution is a sequence of non overlapping *periods* during which predefined subsets of tasks are performed in parallel by all the nodes. All tasks of a period are completed before the next period start. We denote *synchronization points* the instants between two successive periods. The reference `ScaLAPACK` library implements parallel distributed linear algebra operations according to this synchronized execution model. For LU and Cholesky factorization, synchronization is naturally performed according to the iterations of the outer loop: tasks that would be performed during the same iteration in a sequential execution belong to the same period. Using again the tiled Cholesky factorization algorithm as example: let us assume that the current index of iteration of the outer loop is $k \in \{1, \ldots, m_b\}$. Then tasks POTRF(k), TRSM(i, k) and SYRK(i, k) for $k < i \leqslant m_b$, and GEMM(i, j, k) for $k < j < i$ are performed during this period. Afterwards the execution reaches a synchronization point.

In the schedule of such a synchronized execution, all tasks of a given iteration are completed before any task of the next iteration starts. It is illustrated on Figure 1.4a using a 3D representation where the third dimension is used to depict the iterations: the



(a) Example of a synchronous execution: the current iteration is $k = 2$



(b) Example of the "uneven" progression of tasks in an asynchronous execution.

Figure 1.4: Illustration of synchronous and asynchronous executions; the gray color indicates that the task on the corresponding tile has been completed (Cholesky factorization of a 5×5 matrix using BC distribution with $P = 6$ nodes)

figure represents the state of the execution during the iteration $k = 2$ of the Cholesky factorization of a 5×5 matrix; all tasks of the previous iteration 1 are already completed while no task of the next iteration 3 has started yet. Such a schedule facilitates the description of the communications between nodes and ensures the correctness of the computations. The value of each tile, respectively each element for element-wise versions of the operations, are entirely defined by the index of the outer loop, *i.e.* this index is enough to describe which update have already been performed on the values of the tile, respectively the element. In the following, we denote this index the *update index* and use it to precisely identify the value of each tile, respectively an element, by adding as an exponent index between brackets. Still using the tiled Cholesky factorization as example: for $(i, j) \in \{1, \dots, m_b\}$, $i > j$, and $k \leqslant j$, $\mathbf{A}(i, j)^{[k]}$ is the value of tile $\mathbf{A}(i, j)$ computed at iteration k of the loop line 7 of Algorithm 4. Note that for $k = j$ it corresponds to the output of the algorithm: $\mathbf{A}(i, j)^{[j]} = \mathbf{L}(i, j)$. Additionally, we define values with update index index [0] as the input values.

On the other hand, in the task-based execution model, the runtime system does not take into account the sequential description of the algorithm of the operation but only the dependencies between tasks. As detailed previously in Section 1.2.1, using the `StarPU` runtime system that we consider, the entire cDAG is known by each node, as is the data distribution. The schedule of the communications between nodes is directly inferred from the data distribution and dependencies and they occur in accordance with the existing task dependencies of the operation. Now, beyond simplifying the management of communications and task dependencies, the task-based execution model and runtime systems allow an asynchronous execution of the operation, *i.e.* the schedule is no longer composed as a sequence of non overlapping periods separated by synchronization points. Hence, tasks associated with a given index of the outer loop can remain uncompleted while tasks associated with following iterations are performed. The update index remains however necessary to identify the values of each tile, respectively element. To illustrate how asynchronous execution can occur, let us again consider the parallel distributed Cholesky factorization of a 5×5 matrix using a static BC distribution over 6 nodes, as can be seen on Figure 1.4b. The 3D representation enables to show simultaneously the tasks and data dependencies and thus allows to observe the progression of the execution, each completed task being depicted in grey. First we can observe that the state of the execution illustrated on this figure corresponds to an asynchronous execution since the $\text{SYRK}(5, 1)$ has still not been performed while tasks corresponding to the iterations 2 and 3 have already been completed. Let us now illustrate how communications inferred by the runtime system from data dependencies are inserted at the correct position in the cDAG in order to ensure the correctness of the computation. At the state of execution depicted on the figure, the tile $\mathbf{A}(5, 4)^{[1]}$ exists in the memory of node 0, the tile $\mathbf{A}(5, 2)^{[1]}$ in the memory of node 1 and the tiles $\mathbf{A}(2, 2)^{[2]}$ and $\mathbf{A}(4, 2)^{[2]}$ in the memory of node 4. The $\text{TRSM}(5, 2)$ has not been performed yet. To update the tile $\mathbf{A}(5, 4)$, node 0 needs to perform task $\text{GEMM}(5, 4, 2)$ using $\mathbf{A}(4, 2)$ and $\mathbf{A}(5, 2)$ as input. As it knows the data distribution, `StarPU` infers that communications are needed to transfer the tile $\mathbf{A}(4, 2)$ from node 4 to node 0 and the tile $\mathbf{A}(5, 2)$ from node 1 to node 0. That last communication is depicted as a dashed arrow. Moreover, `StarPU` is aware of task dependencies as it unrolls the cDAG. Since $\text{TRSM}(5, 2)$ is a parent of $\text{GEMM}(5, 4, 2)$, the output of the first

one, *i.e.* $\mathbf{A}(5,2)^{[2]}$, must be used as input for the second one. Hence, StarPU can add the communication of tile $\mathbf{A}(5,2)$ from node 1 to node 0 in the cDAG: this communication is a child of task TRSM$(5,2)$ and needs therefore to occur after its completion and is a parent of task GEMM$(5,4,2)$ and must thus occur before it starts. This way, the correct value of the tile $\mathbf{A}(5,2)$ is transferred to node 0 to perform GEMM$(5,4,2)$.

Asynchronous execution features clear advantages over synchronized execution. First of all, it allows efficient load balancing among nodes in the course of the execution by removing any constraint on the general schedule while automatically ensuring that dependencies between tasks are satisfied. Secondly, it allows an "uneven" progression in the cDAG, *i.e.* tasks that would be performed early in a synchronized execution can be postponed to a very late stage of an asynchronous schedule, and conversely. Tasks closer to the critical path can thus be prioritized over others, enabling better general schedules of the operation. Though tasks prioritization is not dynamically handled and is left to the application, careful static priorities based on simple analysis of the critical path in the cDAG already allows to generate very efficient schedules. Finally, since communications are formally handled asynchronously like regular tasks in the cDAG, they can be performed in parallel to tasks. Hence, their adverse effect on the running time can be significantly reduced by overlapping them with computations.

As a consequence, linear algebra libraries implementing task-based execution model backed by runtime systems have shown significantly superior performance over MPI-based tools as is illustrated for example in [4].

1.3 Contribution and Outline

In this work we are interested in the task-based parallel and distributed execution of linear algebra operations. We mainly deal with load balancing and communications issues that arise in such a context, our objective being to minimize the total running time of those operations. We focus on direct resolution methods and dense or compressed matrices and develop static data distributions adapted to specific operations and configurations. More precisely, we use four classical linear algebra operations as study case: matrix multiplication, symmetric rank-k update, LU and Cholesky factorization, and consider task-based execution backed by the StarPU runtime system using identical nodes. Though it is not the focus of this work, most results are valid regardless of the intra nodes heterogeneity, *i.e.* of their workers.

The first part of the document is dedicated to the study of operations which use symmetric input matrices: the symmetric rank-k update and the Cholesky factorization. We consider the parallel distributed execution of those operations on dense matrices.

In Chapter 2 we propose an original data distribution called **S**ymmetric **B**lock **C**yclic (**SBC**) that makes use of the symmetry of the input matrix $\mathbf{A}$ to reduce the overall communication volume for the Cholesky factorization. We provide a description of the communication scheme for this new distribution compared to classic BC along with a detailed formal estimation of the total communication volume: indeed, the communication volume generated by SBC is smaller by a factor of $\sqrt{2}$ than the one generated by BC. Experimental results carried out using the Chameleon library backed by the StarPU runtime

system clearly show significant performance improvements when using SBC rather than BC. It illustrates that the overlap of communications with computations allowed by the task-based execution model is inherently limited because of tasks dependencies and thus supports the complementary strategy of overall communications reduction. Additionally to those results, a 2.5D extension of the SBC distribution is described and tested on one configuration. Extending the use of the SBC distribution to sequence of operations that include a Cholesky factorization is also discussed and preliminary results are shown for solve (POSV) and inversion (POTRI) workflows.

Chapter 3 focuses on the theoretical aspect of the communication minimization problem for dense symmetric rank-k update and Cholesky factorization. Contrary to the previous chapters, in order to obtain strong lower bounds on the communication volume, we consider the classic *out-of-core* setting: a single node with limited memory of size M performs the operation, only part of the data can fit into the memory, and communications occur between the local limited memory and a distant unlimited one. In this configuration we are able to derive bounds for both operations on the minimum necessary communication volume. We prove that the symmetric rank-k update of a $m \times n$ matrix $\mathbf{A}$ requires at least $Q_{\mathrm{SYRK}} \geqslant \frac{1}{\sqrt{2}} \frac{m^2 n}{\sqrt{M}}$ communications; the Cholesky factorization of $m \times m$ symmetric positive definite matrix $\mathbf{A}$ requires $Q_{\mathrm{Cholesky}} \geqslant \frac{1}{3\sqrt{2}} \frac{m^3}{\sqrt{M}}$. These bounds correspond to an improvement by a factor of $\sqrt{2}$ over the previously known best bound on communications for the same operations by Olivry *et al.* [62]. Both bounds use a mathematical expressions that we refer as *explicit*, because the coefficient of the dominant term is explicitly defined while lower order terms may be expressed using asymptotic notations. We use this term as opposed to *asymptotic* that refers to expressions where the dominant term only appears in an asymptotic notation and its coefficient remains undefined. Indeed the difference is of major importance because many historical state-of-the-art results viewed as optimal actually achieve *asymptotic* optimality. In the same chapter, we present two out-of-core algorithms: **T**riangle **B**lock **S**YRK (**TBS**) for the symmetric rank-k update and **L**arge **B**lock **C**holesky (**LBC**) for the Cholesky factorization. We show that those algorithms indeed achieve explicit optimality regarding communications, which means that we can express the respective minimal communication volume they generate as an explicit expression, whose coefficient of the dominant term is the same as in the expression of the bound.

Those original results definitely close the problem of communication minimization in the out-of-core setting for both the symmetric rank-k update and the Cholesky factorization from a theoretical perspective. Besides, the techniques developed to elaborate those algorithms bring new insight about how to tailor data distributions for other operations or sets of operations using symmetric input data. The experiments carried out with the SBC distribution illustrate the ease of implementation and usability of such tailored data distributions in a parallel context in the task-based execution model. The results also clearly show the significant performance gain that can be expected from reducing the total communication volume. In the second part of this work, we try to extend the techniques elaborated in the first part to other more complex configurations, still in a parallel and distributed context. We explore two different directions of research:

- the design and application of tailored static distributions for operations on com-

pressed matrices, which implies heterogeneous tasks;

- for dense matrices, the extension and improvement of the existing BC and SBC distributions to any number of nodes.

In Chapter 4, we consider non-symmetric operations, the matrix multiplication and the LU factorization in the case of compressed matrices using the **B**lock **L**ow **R**ank (**BLR**) format. We carry out the study using a performance model which assumes that uneven compression ratios between the tiles of the input matrices lead to different execution times of each task. Therefore, contrary to the dense case, the set of tasks considered are heterogeneous. Then we study the problem of balancing the heterogeneous workloads among the nodes while imposing additional constraints on the number of different nodes per row and column in order to limit communications. Those constraints are based on the structure of the classic Block Cyclic (BC) distribution Having formalized the trade-off between balancing and communications aspects as a discrete optimization problem, we present two original data distributions elaborated using heuristics: (i) **B**lock **C**yclic **E**xtended (**BCE**) naturally extends the BC distribution to fit the communication related constraints of our trade-off optimization problem while allowing better load balancing; (ii) **R**andom **Sub**sets (**RSB**) is a two steps strategy relying on subsets of nodes with desirable properties computed off-line; it provides non regular data distributions. We present experimental evaluations of those two strategies using simulated executions over synthetic test cases. Results show that BCE and RSB achieve better load balancing than the BC distribution in all tested configurations. They also systematically reach shorter running time than BC, although the two strategies show different behaviors: while results using BCE are very consistent, RSB shows large discrepancies between load balancing values and running times for the LU factorization. This give an insight of the importance of the regularity of the data distribution for operations featuring tight dependencies between tasks.

Chapter 5 is dedicated to extensions to any number of nodes of the cyclic pattern-based distributions for both LU and Cholesky factorization in the dense case. On one hand, the BC distribution efficiently limits communications for non-symmetric operations when the number of available nodes can be written as $P = pq$ with p and q close to $\sqrt{P}$. On the other hand, the SBC distribution reduce the communication volume compared to BC for symmetric operations but is available only for specific values of P: when $P = \frac{r(r-1)}{2}$ or $P = \frac{r^2}{2}$, with $r \in \mathbb{N}^*$. In this chapter we elaborate distributions featuring the same property of communication reduction than those distributions which can make efficient use of all available nodes. We first provide a formal model to evaluate any cyclic pattern-based distribution regarding its load balancing and communication limitations properties and use it to guide the design of two original data distribution schemes. For the LU factorization, the **G**eneralized **B**lock **C**yclic distribution (**G-BC**) provides an almost-cyclic pattern distribution for any number of nodes provided as input. We prove that its solutions achieve explicit optimality regarding the communication metric we have defined. For the Cholesky factorization, we present the randomized greedy algorithm, **G**reedy **C**ol**R**ow & **M**atching (**GCR&M**) that is aimed at generating solutions with similar characteristics, though there is no guarantee regarding the quality of the output

distribution. We carry out experiments using the `Chameleon` library backed by the `StarPU` runtime system to test the quality of the solutions provided by those two algorithms. The comparative results show that both G-BC and GCR&M methods outperform their respective counterparts, BC and SBC. In particular, for values of P where BC patterns have very unbalanced dimensions or there is even no SBC solution, G-BC and GCR&M manage to achieve higher per-node performance while making use of all available nodes. As they are based on patterns that are almost optimal regarding the number of generated communications, G-BC and GCR&M solutions actually reduce the total running time of operations compared respectively to BC and SBC solutions.

Finally, the last part provides a summary of the results obtained and techniques elaborated in this work along with concluding remarks. Perspectives about related problems and potential research directions to extend the results detailed in this document are also presented.

Chapter 2

Overall Communication Reduction: Effect on Performance

In a parallel and distributed setting, inter-node communications are an important factor affecting the overall performance of linear algebra applications. As mentioned in Section 1.1.2.5 their adverse effect on performance can be mitigated by either overlapping them with computations or, more simply, by reducing their overall volume. For operations which involve many dependencies, it is difficult to reach a high level of parallelism during the entire execution, as in the final phase of LU or Cholesky factorization for example. Thus the possibility of overlapping with computations is limited and therefore trying to reduce the volume of data transferred between nodes is expected to provide higher performance gain. The distribution of data is a direct lever to control the communication volume between nodes when complying with the owner computes rule. In this chapter, we consider the Cholesky factorization performed in parallel by a set of identical nodes. The initial matrix $\mathbf{A}$ is divided into tiles and the data associated with each tile are owned by a node according to a specific distribution. We focus on the dense version of the operation: all values of the input matrix $\mathbf{A}$ are considered and all tiles are handled without any compression. This implies that for a given kernel, the number of arithmetic operations remains the same regardless of the tile to which it is applied, hence execution times are similar. Using identical resource and considering a dense matrix makes the problem of balancing the workload among nodes quite simple: since GEMM tasks dominate the total number of arithmetic operations, other type of tasks can be neglected. Balancing load between nodes is thus equivalent to evenly distributing GEMM tasks. The objective is now to design a data distribution scheme that reduces the total communication volume, counted as the number of tiles transfered between nodes, when performing the Cholesky factorization. In the following, we develop such a distribution by taking advantage of the symmetry of the input matrix $\mathbf{A}$.

The chapter is divided as follows: the first section reviews the related work regarding communication avoiding data distributions for operations on dense matrices and details the communication volume generated when performing distributed Cholesky factorization using the classical Block Cyclic (BC) distribution. The second section presents an original data distribution, Symmetric Block Cyclic (SBC), designed to reduce the generated communication volume: indeed we show that using this distribution, performing

the Cholesky factorization requires a factor $\sqrt{2}$ less communications than with static BC. So called 2.5D variants of both distributions are discussed in the following section and their expected impact on performance analyzed. Finally, the last section is dedicated to experimental results.

2.1 State-of-the-art

2.1.1 Communication Lower Bounds

In a parallel and distributed setting, communication is a major topic as moving data between nodes over the network on a computing platform requires time and consumes energy. Data movement between computing resources is indeed a well known bottleneck for many distributed applications. Since the computation power progresses faster than the interconnection capacity, minimizing communications is acknowledged as a central challenge for the performance of parallel computing.

Regarding linear algebra applications, research works on communication-avoiding techniques has advanced through the development of theoretical lower bounds which express the minimal communication volume necessary to perform a given operation. Both theoretical and practical aspects are developed together: striving to reach these lower bounds has led to the design of more efficient algorithms in terms of communication reduction.

In order to obtain these lower bounds, two simplified machine models are generally considered in the literature:

1. *Two-levels memory* or *out-of-core* model: the machine features one fast and limited memory of size M and one "slow" and unlimited memory. Input required for any computation must reside in fast memory to be performed.

2. Parallel model: P nodes, all equipped with a memory of size M, can communicate through a network.

The first model is extensively used to prove communication lower bounds because of its simple setting. Results are generally expressed as function of the memory size M.

Deriving lower bounds in the parallel model is more difficult. It is often carried out by studying the communication volume of a single node using the out-of-core model and considering the set of all other nodes as a single "slow" memory with which data transfers occur. However this requires an assumption on the distribution of data among nodes. Indeed without additional constraint the trivial solution to minimize communications is to perform all computation using as few nodes as memory size allows, that can be a single one. Such solutions are of no interest in this context, hence it is generally assumed that the input data is evenly spread across all available nodes. This assumption may be referred to as *memory scalable* because it means that the total available memory capacity PM is proportional to the input data.

The first results on communications lower bound were obtained for classical matrix multiplication: $\mathbf{C} = \mathbf{AB}$. For simplicity they are expressed assuming that $\mathbf{A}$, $\mathbf{B}$ and $\mathbf{C}$ are of identical square dimension $m \times m$.

The paper from Hong and Kung [50] can be considered as the founding piece of subsequent development on the topic. In this work, the authors consider a machine following the out-of-core model. A set of rules referred as the *pebbling game* models the required data transfers between the two types of memory. Based on the analysis of the DAG, the authors derive asymptotic lower bounds for the number of data transfers required between the two levels of memory. Then $\Omega(\frac{m^3}{\sqrt{M}})$ data transfers are required to perform matrix multiplication.

Based on this work and using a geometric approach from [58], Irony *et al.* [53] extended the lower bound to the parallel case. It is formulated as a *memory-communication trade-off*: any node which has M words of local memory and performs W arithmetic operations (here multiplications) must send or receive at least $\frac{W}{2\sqrt{2}\sqrt{M}} - M$ words. Under the memory scalable assumption, *i.e.* $M = \mathcal{O}(\frac{m^2}{P})$, the result from [53] implies that the minimum communication volume per node is $\Omega(\frac{m^2}{\sqrt{P}})$.

This result for the parallel case was generalized to the Cholesky factorization in [13] by means of reduction: the multiplication of matrices $\mathbf{A}\cdot\mathbf{B}$ can be performed via the Cholesky factorization of a larger matrix derived from $\mathbf{A}$ and $\mathbf{B}$. The asymptotic lower bound on the total number of communications is therefore valid for the Cholesky factorization. The same authors later showed in [14] that those bounds are part of a more general framework applicable to almost all direct linear algebra applications in both the dense and sparse cases. For such operations the total communication volume is $\Omega(\frac{\#\text{ arithmetic operations}}{\sqrt{M}})$ where M is the memory capacity of a single node in the out-of-core model or the amount of data owned by each of P nodes in the parallel model. Thus similar bounds can be derived for a wide range of kernels including factorizations (Cholesky, LU, LDL^T), operations using orthogonal matrices (QR) eigenvalues and singular values decomposition and, to some extend, sequence of such operations.

Additional work by Solomonik *et al.* [67] presents an original way of modeling the dependencies of any operation as a *lattice-hypergraph* which enables the authors to extend the memory-communication trade-off to take into account synchronizations and express bounds about the communication on the critical path.

A more recent approach strives to get non asymptotic results, *i.e.* provide explicit coefficient for the dominant term of the lower bound formula. In [31] Christ *et al.* developed a methodology based on a discrete version of Hölder-Brascamp-Lieb inequality [26] which allows to deal with any algorithm that can be expressed as nested loops of elementary operations and array accesses. It can be viewed as a generalization of all previous arguments: by seeing the set of computations as a volume and using geometric projections, the minimum number of data transfers can be computed. Olivry *et al.* [62] used this methodology and other techniques to develop the `IOLB` tool that performs automatic analysis to derive lower bounds on communications. This tool provided improved bounds for several kernels in the out-of-core model. For Cholesky factorization the minimum number of data transfers is lower bounded by $\frac{m^3}{6\sqrt{M}} + \mathcal{O}(m^2)$.

2.1.2 Algorithms

Historically the implementations of linear algebra operations in a distributed context have been developed outside of the task-based execution model and using the owner computes rule thus requiring handcrafted description of inter-node communications. Hence the data distributions needed to be simple and regular. The 2D BC data distribution is therefore the *de facto* standard in many linear algebra libraries. The classical `ScaLAPACK` library [22] contains parallel implementations of many linear algebra operations using this distribution. It is a natural and relevant choice since BC ensures load balancing between nodes (each node owns the same number of tiles) and even load balancing over time, *i.e.* when the size of the computation domain shrinks during the execution, such as the trailing matrix in the Cholesky factorization. Regarding communications, Irony *et al.* proved in [53] that 2D BC is asymptotically optimal for matrix multiplication under the memory scalable assumption: with 2D BC distribution $M = \Theta(\frac{m^2}{P})$ then performing matrix multiplication requires $\mathcal{O}(\frac{m^2}{\sqrt{P}})$ communications per node. This result was extended to the Cholesky factorization in [13]: the authors show that the `ScaLAPACK` implementation of the Cholesky factorization, which uses the 2D BC distribution, is a factor of $\log(P)$ from the lower bounds on communications per node, *i.e.* $\mathcal{O}(\frac{m^2}{\sqrt{P}}\log(P))$, if used with a non constant block size $b = \Theta(\frac{m}{\sqrt{P}})$.

Though being the most natural and most easily implemented, the data distribution is not restricted to 2D layouts such as 2D BC. When extra memory is available, it is actually possible to replicate the input and/or output data on several nodes and thus to design 3D algorithms that associate tasks rather than tiles to nodes, no longer following the owner computes rule. In [53] the authors detail a 3D BC algorithm for matrix multiplication that is the natural extension of 2D BC. They show that it corresponds to the second limit cases of the memory scalable assumption where $M = \mathcal{O}(\frac{m^2}{P^{\frac{2}{3}}})$. Just like the 2D BC distribution, this 3D version algorithm is asymptotically optimal regarding communications since it matches the lower bound derived from the memory-communication trade-off. In [52] Irony *et al.* propose a fully 3D algorithm for the triangular solve operation and the LU factorization without pivoting. They show that their algorithms for the LU factorization generates a total of $2.5m^2P^{\frac{1}{3}} + o(m^2)$ communications.

There is actually a continuum between 2D and 3D algorithms where the trade-off between memory footprint and communication is controlled by the number of data replication denoted c. Intermediate versions, *i.e.* replicating some data without using all available memory, are referred as 2.5D algorithms. Data replication allows to expect better performance via a higher degree of parallelism at the cost of extra communications. Hence many variants based on the 2.5D principle have been designed for many kernels to increase performance compared to classical 2D BC.

In [68] Solomonik *et al.* bridge the gap between 2D and 3D distributions by presenting more generic 2.5D algorithms for the matrix multiplication and the LU factorization with pivoting. The algorithm for the matrix multiplication is asymptotically optimal: it generates $\Theta(\frac{m^2}{\sqrt{Pc}})$ communications per node, matching the bounds from [53] in the special 3D case $c = P^{\frac{1}{3}}$. The algorithm for LU factorization makes use of tournament pivoting, first introduced by Grigori *et al.* in [46], which decreases the necessary communications

to find a suitable pivot compared to classical partial pivoting method. The authors show that it also reaches asymptotic optimality for the total communication volume if used with a non constant block size $\frac{m}{r\sqrt{Pc}} \times \frac{m}{r\sqrt{Pc}}$ where $r = \Omega(\log(P))$.

In [41] Georganas *et al.* propose a quite similar implementation of 2.5D distribution for the parallel Cholesky factorization on a Cray system. Using a carefully designed pipelining of two consecutive iterations, the authors manage to increase the overlap of communication by computation thus improving the overall performance.

In 2021, Kwasniewski *et al.* [57] present a similar implementation for both LU and Cholesky factorization that further reduces the total communication volume by using *row masking* technique to propagate pivot rows among nodes. They show that their implementation for the parallel LU factorization, `COnfLUX`, and Cholesky factorization, `COnfCHOX`, generates no more than $\frac{m^2}{\sqrt{Pc}} + \mathcal{O}(\frac{m^2}{P})$ communications per node.

Experimental results from those studies show that 2.5D algorithms can achieve higher performance than conventional 2D algorithms.

2.2 Communication Analysis of BC and SBC Distributions

Most of the algorithms mentioned in the previous paragraph (2.1.2) are variants of a BC distribution, whether using classical 2D layout or more elaborated 3D or 2.5D ones. As explained, several implementations are asymptotically optimal regarding the communication volume generated for the matrix multiplication, LU and Cholesky factorization. However it is difficult to evaluate which performs better since the coefficient of the actual number of communications is not explicit.

In the following we provide a detailed study of the communication performance of two data distributions for the parallel Cholesky factorization using P identical nodes. We first detail an original 2D distribution, Symmetric Block Cyclic (SBC), designed to make use of the symmetry of the input matrix. Then we evaluate the communication volume generated by SBC compared to BC: first the communication scheme associated with the Cholesky factorization algorithm is analyzed, then we provide closed form formulas for the number of communications that are implied by each of the two distributions. We show that the Cholesky factorization using SBC generates fewer communications than using BC by a factor of $\sqrt{2}$. The communication volume is evaluated as the number of tiles that need to be transfered between nodes, counted individually, and assuming point-to-point transfers. This corresponds to what is often referred in the literature as the *bandwidth* cost of communications. We do not take into consideration the *latency* cost, *i.e.* the number of messages required to transfer data if a packing strategy is applied.

This section mainly deals with the 2D version of data distributions which implies that the owner computes rule applies. However, notice that the results regarding the communication volume generated are valid regardless of the scheduling of tasks, whether the execution is highly synchronized or performed under the task-based execution model. Extension of the analysis and results regarding SBC to 2.5D and 3D distributions are provided in the last part of this section.

Generic pattern:

	0	1	3
0		2	4
1	2		5
3	4	5	

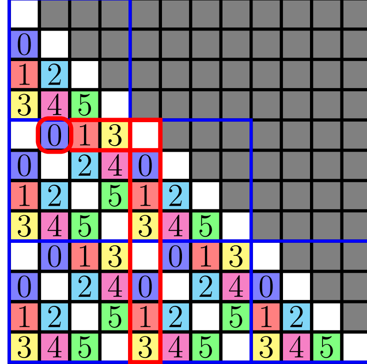
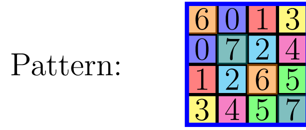


Figure 2.1: SBC allocation: repetition of the 4×4 pattern (using $P = 6$ nodes; one color per node) over a 12×12 matrix. Diagonal positions in the pattern are omitted.

2.2.1 Symmetric Block Cyclic Distribution

We now introduce a new distribution called Symmetric Block Cyclic (SBC) where the two sets of nodes for rows and columns are the same. This is a desirable feature to reduce the number of inter node communications, as detailed in the next Section 2.2.2. This is achieved via a pattern, larger than for the BC distribution, in which each node appears twice. Let us consider that the number of nodes is given by $P = \frac{r(r-1)}{2}$ for some integer $r \geqslant 2$. The SBC distribution is built from a $r \times r$ pattern, which is to be repeated over the whole matrix. In the pattern, each node is assigned a pair (x, y) with $x < y$, and is associated with two indices, at coordinates (x, y) and (y, x). This pattern is then repeated across the matrix just like for the BC distribution: tile (i, j) is assigned to the node associated with the index $(i \mod r, j \mod r)$. The resulting distribution for $r = 4$ and $m_b = 12$ can be seen on Figure 2.1.

In the subsequent procedure, we call such pattern a *generic* pattern. It is unfinished and thus cannot be used as is to produce a valid distribution for matrix $\mathbf{A}$ since its diagonal positions are not assigned. To produce a complete pattern from a generic one, diagonal indices, those with coordinates (x, x), require to be assigned in a specific way to ensure that nodes present on a row and column are the same while preserving load balancing between all nodes. To achieve a good load balance, it is important that each node appears the same number of times on the diagonal, the generic pattern being already balanced. Below we present two possible solutions to this issue: the *basic* version of SBC which is only valid for even values of r and uses $\frac{r}{2}$ additional nodes on the diagonal; the *extended* version which does not use any additional node and is valid for all r.



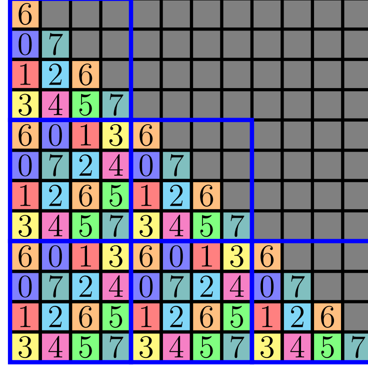


Figure 2.2: Basic version of SBC pattern, with additional nodes, for $r = 4$.

2.2.1.1 Basic Version of SBC

For even values of r, we can add $\frac{r}{2}$ nodes to the generic pattern, and assign each of them to two coordinates on the diagonal of the pattern. We assign them in a round-robin fashion, as indicated on Figure 2.2. It yields a single valid balanced pattern featuring an identical set nodes on any row and column. The final distribution using $P = \frac{r^2}{2}$ nodes is then produced by repeating this single pattern over the whole matrix $\mathbf{A}$.

2.2.1.2 Extended version of SBC

The second solution assign diagonal positions of the pattern to the same set of $P = \frac{r(r-1)}{2}$ nodes. For each diagonal position, we want to choose nodes that are already present in the same row or column. Since there are only r positions on the diagonal, achieving overall load balance requires to define a set of patterns, which only differ by their diagonal entries. They are then repeated alternatively over the matrix so as to balance the load between all nodes.

Let us first consider the case of odd r. Starting from a generic pattern, we build $\frac{r-1}{2}$ complete patterns in the following way: for l in $\{1, \ldots, \frac{r-1}{2}\}$, the diagonal positions of the l-th pattern are assigned to the nodes associated with index $(1, 1+l), (2, 2+l), ..., (r-l, r)$, which we call first group, and index $(1, r+1-l), ..., (l, r)$, the second group. Hence each node appears on the diagonal of exactly one pattern: those of the first group on the same row as their index in the right-upper part of the generic pattern, those of the second group on the same column. Both patterns are depicted on the right of Figure 2.3, where nodes of the first group are in blue, nodes of the second group are in red.

The case of even r is based on the same idea, but with an additional difficulty: using the same construction leads to both groups of the last pattern containing twice the same set

	0	1	3	6
0		2	4	7
1	2		5	8
3	4	5		9
6	7	8	9	

0	0	1	3	6
0	2	2	4	7
1	2	5	5	8
3	4	5	9	9
6	7	8	9	6

1	0	1	3	6
0	4	2	4	7
1	2	8	5	8
3	4	5	3	9
6	7	8	9	7

Figure 2.3: Extended SBC distribution for $r = 5$, $P = 10$. Left: generic pattern, right: patterns with diagonal nodes.

of $\frac{r}{2}$ nodes. Therefore using it to produce a complete pattern would result in imbalanced loads. Instead we create $r - 1$ patterns using the following procedure:

1. The first $\frac{r}{2} - 1$ patterns are obtained just like previously. Then for each pattern, the set of nodes assigned to diagonal positions are considered as two packs of $\frac{r}{2}$ nodes, the *left* and the *right* pack.

2. A last bonus pack is obtained with nodes at index $(r/2 + 1, 1), ..., (r, r/2)$. This pack can be used to assign the top left half (on the same row) or the bottom right half (on the same column) of the diagonal. At this point any combination of a left pack and a right pack can be used to assign the diagonal position and create a valid pattern; the bonus pack can be used either as a left or as a right pack.

3. We create $\frac{r}{2}$ additional patterns by assigning diagonal positions as follows: shifting the left packs of the diagonal of the first $\frac{r}{2} - 1$ patterns; adding the bonus pack at the top of the list of the left packs and at the bottom of the list of the right packs; combining them together.

With this construction, each node appears on the diagonal of exactly two patterns, and always on the same row or column. The result for $r = 6$ is shown on Figure 2.4, where packs are represented with different colors.

Such procedure yields a set of $\frac{r-1}{2}$ patterns if r is odd, respectively $r-1$ if r is even, each of those featuring an identical set of nodes on any row and column. The set of patterns is overall balanced. The final distribution using $P = \frac{r(r-1)}{2}$ nodes is then produced by repeating the set of patterns over matrix $\mathbf{A}$ in a round-robin column-wise fashion. An example of distribution for $r = 4$ and $m_b = 12$ is depicted in Figure 2.5.

2.2.2 Analysis of the Communication Volume

Let us first analyze the communication scheme associated with the distributed Cholesky factorization. The algorithm of Cholesky factorization takes as input a symmetric positive definite matrix $\mathbf{A}$ and overwrites it with a lower triangular matrix $\mathbf{L}$ such that: $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\intercal$. The tiled version of this operation, in the right looking variant, is described in Algorithm 4.

The algorithm of the operation shows the data dependencies associated with the tasks from a receiver perspective: for each task that updates a tile, it details which other tile must be used as input. Now, to ease the analysis, especially the evaluation of the

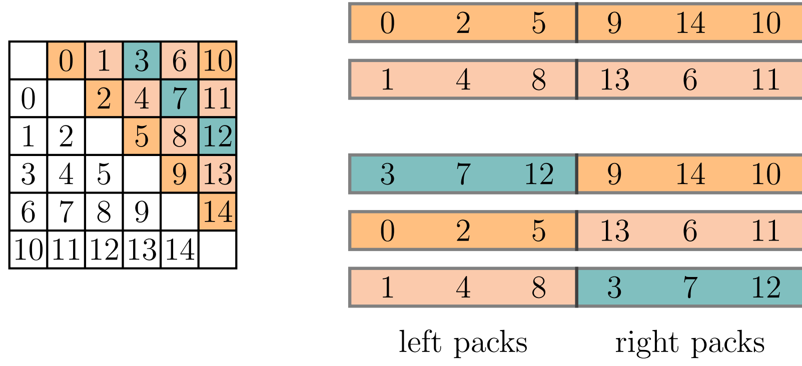


Figure 2.4: Example of the SBC distribution for $r = 6$, $P = 15$. Left: generic pattern, right: 5 sets of diagonal nodes, with four normal packs (in two shades of orange) and one bonus pack (in green).

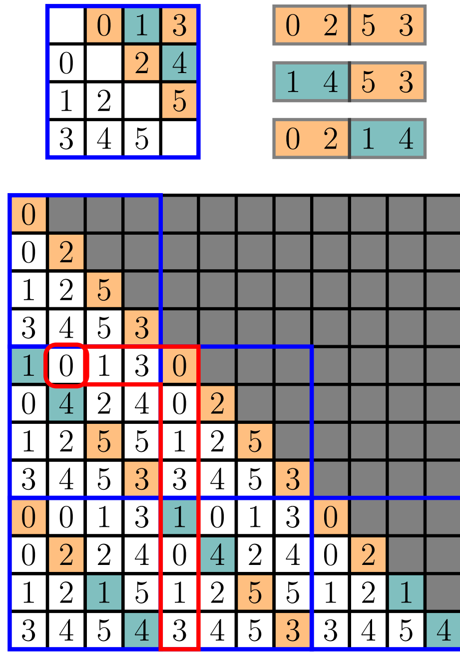


Figure 2.5: Diagonal patterns: two normal packs (in orange) and one bonus pack (in green) generate 3 diagonal patterns used in a round-robin column-wise fashion.

number of communications, we can simply reverse the description: for each tile, describe which tasks must use it as input. Note that for all tasks, the tile to which the task is applied is used as input. For example, tile $\mathbf{A}(i,j)$, for $k < j < i$, is both input and output of GEMM(i,j,k). When considering only 2D distributions, the operation is performed according to the owner computes rule. Hence, in this case, for all tasks, the data dependency involving the tile that is modified by the task does not generate any communication because input and output correspond to the same tile, which is assigned to a single node. On the contrary, those dependencies must be taken into account for the analysis of 2.5D or 3D distributions, as illustrated in Section 2.2.3.

For Cholesky factorization, according to Algorithm 4:

1. each tile $\mathbf{A}(k,k)$, for $k \in \{1,\dots,m_b-1\}$, output of POTRF(k) is used as input of TRSM(i,k), for $i \in \{k+1,\dots,m_b\}$, to update tiles $\mathbf{A}(i,k)$;

2. each tile $\mathbf{A}(i,k)$, for $k \in \{1,\dots,m_b-1\}$, $i \in \{k+1,\dots,m_b\}$, output of TRSM(i,k) is used as input of:

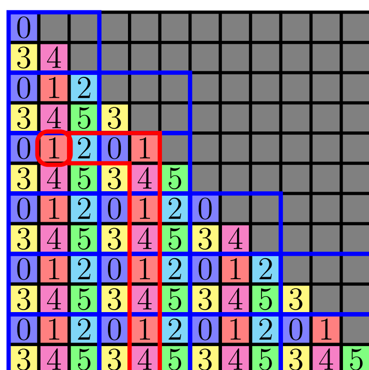
   - SYRK(i,k) to update tile $\mathbf{A}(i,i)$;
   - GEMM(i,j,k), for $j \in \{k+1,\dots,i-1\}$, to update tiles $\mathbf{A}(i,j)$;
   - GEMM(j,i,k), for $j \in \{i+1,\dots,m_b\}$, to update tiles $\mathbf{A}(j,i)$.

Dependencies (1) imply that each tile $\mathbf{A}(k,k)$ is sent downward to all nodes owning tiles along the corresponding column k. Dependencies (2) imply that each tile $\mathbf{A}(i,k)$ is sent to the right along row i up to the diagonal position, and from the diagonal position downward along column i.

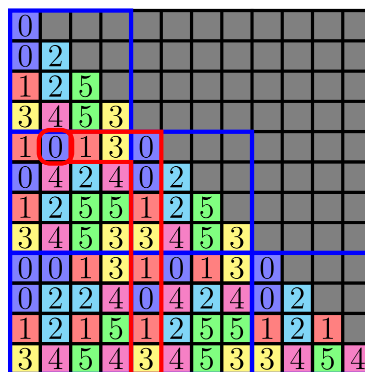
According to this communication scheme, dependencies (1) imply $\Omega(m_b)$ communications while dependencies (2) imply $\Omega(m_b^2)$. In the following, to evaluate the dominant factor of the number of communications generated when using different data distributions, we can thus only consider the second type of dependencies.

The two data distributions BC and SBC are detailed respectively in Section 1.1.2.3 and Section 2.2.1. Using the analysis of the communication scheme of the distributed Cholesky factorization, we can now assess how they perform in terms of total communication volume. Let $k \in \{1,\dots,m_b-1\}$ and $i \in \{k+1,\dots,m_b\}$.

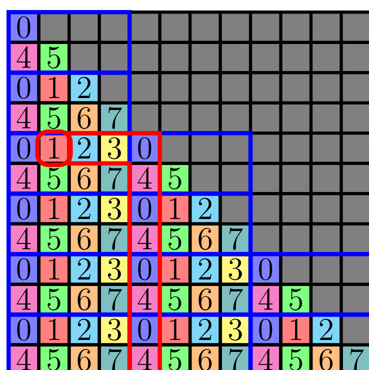
Block Cyclic Assume $\mathbf{A}$ is distributed among P nodes according to BC distribution using a $p \times q$ pattern. With such distribution, tile $\mathbf{A}(i,k)$ as output of the TRSM(i,k) task is required by the q nodes which are assigned tiles $\mathbf{A}(i,j)$ for $k+1 \leqslant j \leqslant i$ and the p nodes which are assigned tiles $\mathbf{A}(j,i)$ for $i \leqslant j \leqslant m_b$. This involves $p+q-1$ different nodes, and since one of them performed the TRSM(i,k) task, tile $\mathbf{A}(i,k)$ needs to be sent to $p+q-2$ nodes. An example of such communication pattern is highlighted in red on Figure 2.6a: node 1 performs TRSM$(5,2)$ updating tile $\mathbf{A}(5,2)$ to its final value; it is transfered to nodes 2 and 0 on the same row, and to node 4 on the corresponding column.



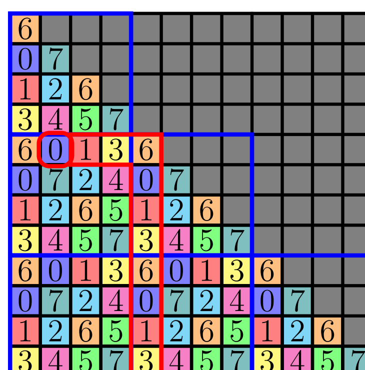
(a) BC; $P = 6$; $(p, q) = (2, 3)$



(b) SBC extended; $P = 6$; $r = 4$



(c) BC; $P = 8$; $(p, q) = (2, 4)$



(d) SBC basic; $P = 8$; $r = 4$

Figure 2.6: Communication scheme associated with tile $\mathbf{A}(5, 2)$ in a 12×12 matrix distributed according to BC or SBC using 6 and 8 nodes.

Symmetric Block Cyclic With the SBC distribution, tiles on row i and column i are assigned to the same set of r nodes for the basic version, respectively $r - 1$ nodes for the extended version. One of them performs the TRSM(i, k) task which implies that tile $\mathbf{A}(i, k)$ needs to be sent to $r - 1$ nodes for the basic version, respectively $r - 2$ for the extended one, as can be seen on Figure 2.6b: 0 performs TRSM$(5, 2)$ on tile $\mathbf{A}(5, 2)$ which is transfered to nodes 1 and 3 that are present on the same row and the corresponding column. We can observe that the SBC distribution takes advantage of data reuse to perform the GEMM tasks: indeed, because of the symmetry of $\mathbf{A}$, tile $\mathbf{A}(i, k)$ as output of TRSM(i, k) is used as input of GEMM tasks on row i but also on column i. This in not the case in LU factorization without pivoting, which shows a cDAG and communication scheme very similar to Cholesky factorization but with non-symmetric data. For this operation, $\mathbf{A}(i, k)$ output of TRSM(i, k) and $\mathbf{A}(k, i)$ output of TRSM(k, i) are not the same. Therefore, the same data reuse cannot be used to reduce the communication of the input of the GEMM tasks.

The previous analysis is valid for all TRSM(i, k) tasks. Then it yields the following

result:

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

- $Q_{basic} = S \cdot (r-1) + o(S)$ *for the basic version,*
- $Q_{extended} = S \cdot (r-2) + o(S)$ *for the extended version.*

We can compare this result asymptotically when P grows to infinity with the BC distribution. For a given value of P, the parameter r using SBC extended is such that $P = \frac{r(r-1)}{2}$. When P grows to infinity, $r \underset{P\to+\infty}{\sim} \sqrt{2P}$ for both basic and extended versions For a matrix with S elements, SBC induces a communication volume of:

$$S \cdot (r-2) \underset{P\to+\infty}{\sim} \sqrt{2}S\sqrt{P}$$

On the other hand, the communication volume generated using BC is: $S \cdot (p+q-2)$. Since $pq = P$, it is minimum for $p = q = \sqrt{P}$. For a fair comparison, we thus consider BC distributions using the most advantageous pattern layout regarding communications, which implies that $p \underset{P\to+\infty}{\sim} \sqrt{P}$. Then the communication volume induced is:

$$S \cdot (2p-2) \underset{P\to+\infty}{\sim} 2S\sqrt{P}$$

Hence Symmetric Block Cyclic induces a communication volume smaller by a factor of $\sqrt{2}$ than Block Cyclic for both weak scaling, where S grows at the same rate as P, and strong scaling, where S remains constant while P grows.

2.2.3 2.5D Variants of the Data Distributions

Similarly to the 2.5D BC distribution (used for example in [57]), it is possible to use SBC distribution in a 2.5D context. We describe it here with the basic version of SBC, but the extended version can be used as well. Assume that we have $P = c\frac{r^2}{2}$ nodes for some value or $r > 1$ and $c > 1$. In the 2.5D SBC distribution, these P nodes are partitioned into c slices of $\frac{r^2}{2}$ nodes. Each slice stores a copy of the matrix $\mathbf{A}$, distributed with the SBC distribution with parameter r.

Each iteration is performed by a different slice, in a round-robin fashion: tasks corresponding to iteration k are performed according to distribution of slice $k \mod c$. Because of the data replication and task distribution, c versions of each tile of matrix $\mathbf{A}$ exist simultaneously and are owned by nodes on the corresponding slice. We denote as $\mathbf{A}(i,j)^{[l]}$, $1 \leqslant j \leqslant i \leqslant m_b$, $1 \leqslant l \leqslant c$, those c versions. Updates of each tile are accumulated as modifications of the c different versions. More precisely, for tile $\mathbf{A}(i,j)$ with $1 < j < i < m_b$, respectively tile $\mathbf{A}(i,i)$, all GEMM(i,j,k) for $k \in \{1,\ldots,j-1\}$, respectively SYRK(i,k), are distributed among the c nodes that are assigned this tile in the corresponding slice. For example, the node assigned tile $\mathbf{A}(i,j)^{[l]}$ in slice $l \in \{1,\ldots,c\}$ performs all tasks

$\{\text{GEMM}(i,j,k),\ k \in \{1,\ldots,j-1\} \mid k \mod c = l\}$. Since the complete update of each tile consist in a sum of multiplications, namely the GEMM and SYRK tasks, those can be performed independently on different nodes.

Now for each tile, the aggregation of all its partially updated versions spread across c nodes is required before computing its final value using a TRSM or POTRF task. The aggregation step is simply a sum of all updates. In our implementation, the node in charge of the final TRSM or POTRF task is also in charge of the aggregation of all updates. This adds a new source of communication: each tile is part of $c-1$ communications, whether by a reduce over c nodes or by $c-1$ point-to-point operations, it does not change the total communication volume. For instance, the final value of tile $\mathbf{A}(i,j)$, respectively $\mathbf{A}(i,i)$, is computed by the node to which it is assigned on slice i.

- This nodes first aggregates all partially updated version of the tile: $\mathbf{A}(i,j) \leftarrow \sum_{l\in\{1,\ldots,i\}} \mathbf{A}(i,j)_{l \mod c}^{[l]}$.
- Then it computes the final value of the tile by performing a $\text{TRSM}(i,j)$, respectively a $\text{SYRK}(i,i)$: $\mathbf{A}(i,j) \leftarrow \text{TRSM}(\mathbf{A}(i,j), \mathbf{A}(i,i))$.

On the other hand, final values of tiles output of POTRF and TRSM tasks are always communicated among nodes *within* a single slice.

With a synchronized, pure `MPI` implementation, performing each iteration on a subset of the nodes would result in each slice working one after the other, thus preventing parallelism. But a task-based implementation allows an iteration to be started before all the GEMM updates are finished. Nodes on one slice can thus start working while tasks of the previous slice are still ongoing, and parallelism can be achieved. We show in Section 2.3 that such a 2.5D implementation can achieves higher performance compared to the simple 2D approach.

2.2.3.1 Communication Volume with Limited Memory

Let us consider a 2.5D SBC basic distribution as defined above: c slices, each containing a basic 2D SBC distribution with parameter r. This distribution involves $\frac{r^2}{2} \cdot c$ nodes. We denote again the total size of matrix $\mathbf{A}$ as $S = \frac{m_b(m_b+1)}{2}$. As mentioned in Theorem 1, each tile is sent to $r-1$ nodes, so the communication volume for the intermediate results is $Q_1 = S \cdot (r-1)$. In addition, each tile of the result is replicated c times, and needs to be aggregated on one node, resulting in $Q_2 = S \cdot (c-1)$ communications. The total is thus $Q = Q_1 + Q_2 = S \cdot (r+c-2)$.

Let us consider that we have P nodes with an amount of memory M per node, and that the dimension m_b of the matrix grows to infinity while $M \underset{m_b\to+\infty}{=} o(\frac{m_b^2}{P^{2/3}})$. We can use as many slices as memory size allows, in a way similar to [57], *i.e.* set c to $\frac{PM}{S}$. Since $S \underset{m_b\to+\infty}{\sim} \frac{m_b^2}{2}$, this gives $c \underset{m_b\to+\infty}{\sim} \frac{2PM}{m_b^2}$, and $r = \sqrt{2\frac{P}{c}} \underset{m_b\to+\infty}{\sim} \frac{m_b}{\sqrt{M}}$. We thus have

$Q_1 \underset{m_b \to +\infty}{\sim} \frac{m_b^3}{2\sqrt{M}}$ and $Q_2 \underset{m_b \to +\infty}{\sim} \frac{PM}{m_b^2}$. The assumption on M ensures that the overall data movement is dominated by Q_1 and asymptotically we obtain $Q = \frac{1}{2} \cdot \frac{m^3}{\sqrt{M}} + o(m_b^3)$.

This can be compared to the previous result by Kwasniewski *et al.* [57], who prove that their 2.5D Block Cyclic algorithm performs a communication volume of $\frac{m_b^3}{\sqrt{M}} + o(m_b^3)$. Our result is a factor of 2 improvement.

2.2.3.2 Number of Slices with Large Memory

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

$$\begin{cases} 1 - u \cdot 2rc = 0 \\ 1 - u \cdot r^2 = 0 \end{cases}$$

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

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

2.3 Experiments

2.3.1 Experimental Setup

To test the performance of the SBC distribution and compare it to BC, we carry out experiments considering three different operations: the main one is the Cholesky factorization (POTRF), but we also evaluate the distributions for two other operations which make use of the Cholesky factorization: the linear system solving (POSV) and the symmetric matrix inversion (POTRI).

For each operation and each distribution, a random symmetric positive definite matrix **A** is generated, along with a matrix **B** as right-hand-side for POSV, and distributed among the nodes. Once the matrix has been generated and distributed, the computing time to perform the operation is measured and the volume of data exchanged during the operation recorded. The process is repeated 5 times to ensure that the results are statistically meaningful. In all plots in the remaining of this chapter, each point represent

the average result over the 5 runs for each experiment while the shaded zone shows the actual range of minimum to maximum values.

The performance obtained when running one of the considered operation is measured in GFlop/s per node. More precisely, for an execution of duration t using P nodes, this value is given by $F = \frac{\#\text{flops}}{t \cdot P}$, where #flops is the number of flops for the operation that depends on the matrix size m_b. This allows to fairly compare results obtained by approaches using different numbers of nodes. For example, in Figure 2.9, some allocations use 28 nodes while others use 30 nodes. Directly comparing execution times is therefore not so relevant in such case.

All experiments are performed in double precision on the **bora** cluster of PlaFRIM [75], which contains 42 nodes each with 36 *Intel Xeon Skylake Gold 6240* cores, for a total of 1512 cores. The nodes are connected with a 100Gb/s OmniPath network. We use Intel MKL 2020, `Open MPI` version 4.0.3, `StarPU` version 1.3.8, and `Chameleon` version 1.1.0. The hardware peak for one core in this setup is estimated as follows: 2.6 GHz (core base frequency) $\times$ 8 (double-precision Flop per cycle) $\times$ 2 (FMA feature). It yields a maximum rate of 41.6 GFlop/s for each core, equivalent to 1497.6 GFlop/s for one node (36 cores) and 1414.4 GFlop/s for 34 cores. In following plots regarding performance, a dotted line indicates that latter value of the theoretical peak achievable by `StarPU`, using only 34 cores per node.

We ran 50000×50000 Cholesky factorization on a single node using different tile sizes from 100 to 1000 to determine the most appropriate for this kernel and this hardware. As shown on Figure 2.7, almost maximum value of GFlop/s is reached as soon as tile size is at least 500×500. Therefore we use tile size $b = 500$ for all subsequent experiments in order to maximize computation throughput and parallelism. Table 2.1 show the average performance achieved for the kernels considered in our experimental setup.

	POTRF	GETRF	SYRK	TRSM	GEMM
Exe. time (ms)	1.238	2.947	2.827	2.404	4.703
Relative exe. time (to POTRF)	1	2.380	2.2828	1.941	3.798

Table 2.1: absolute and relative execution time of kernels in the selected experimental setup (double precision, $b = 500$)

In the rest of this chapter and all experiments using the BC or SBC distributions, all individual kernels are executed sequentially with this block size by a single CPU core.

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

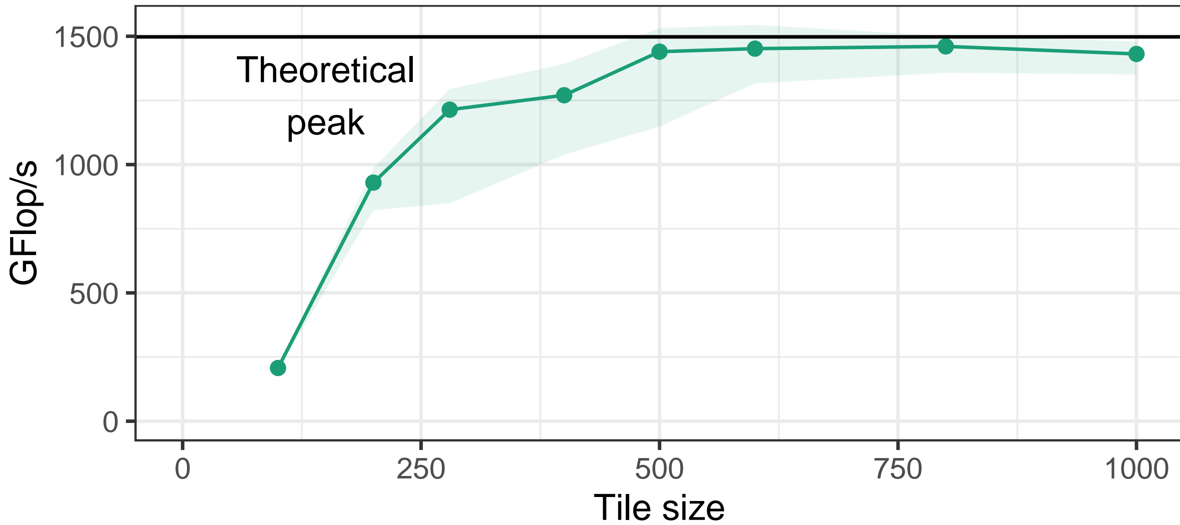


Figure 2.7: Performance of the Cholesky factorization over a matrix of size $m = 50000$ using `Chameleon- StarPU` on a single node for different tile size.

2.3.2 Data Distributions Parameters

Within the `Chameleon` framework, we compare the 2D versions of BC and SBC distributions in the following setups:

- The SBC extended distribution with values of r ranging from 6 to 9, which respectively correspond to a number of nodes $P = 15$, 21, 28 and 36. Since each node contains 36 cores, the number of cores ranges from 510 to 1224.

- The BC distribution implemented by default in the `Chameleon` library, featuring two options with a similar number of nodes, in order to cover the best possible parameters p and q. This ensures that we avoid any unfairness based on a choice of P that would be ill suited to BC, for example for $r = 7$, using $P = 21$ forces to use a 7×3 pattern, whereas a 5×4 pattern is closer to a square and thus generates fewer communications. The parameters used are summarized on Table 2.2.

Results for 2.5D versions of both BC and SBC are also provided for the case $P = 28$ or 30 in order to illustrate additional gains that can be achieved by such more complex data and tasks distribution scheme.

The `Chameleon` library was shown to be very competitive with other state-of-the-art implementations by Agullo *et al.* [4]. In the following experiments we thus focus on illustrating the benefits of the proposed SBC data distribution compared to classical BC. In addition, we also compare with the recently proposed `COnfCHOX` library. `COnfCHOX` is able to use a 2.5D version and is designed to automatically select the most appropriated distribution according to a provided value for P. In our experimental setups however, its automatic parametrization only selects a 2D distribution. The experiments on our platform confirmed that the 2.5D distributions resulted in worse performance than the

Symmetric Block Cyclic		Block Cyclic		
r	P	p	q	P
6	15	5	3	15
		4	4	16
7	21	5	4	20
		7	3	21
8	28	7	4	28
		6	5	30
		8	4	32
9	36	7	5	35
		6	6	36

Table 2.2: Sizes of the considered distributions

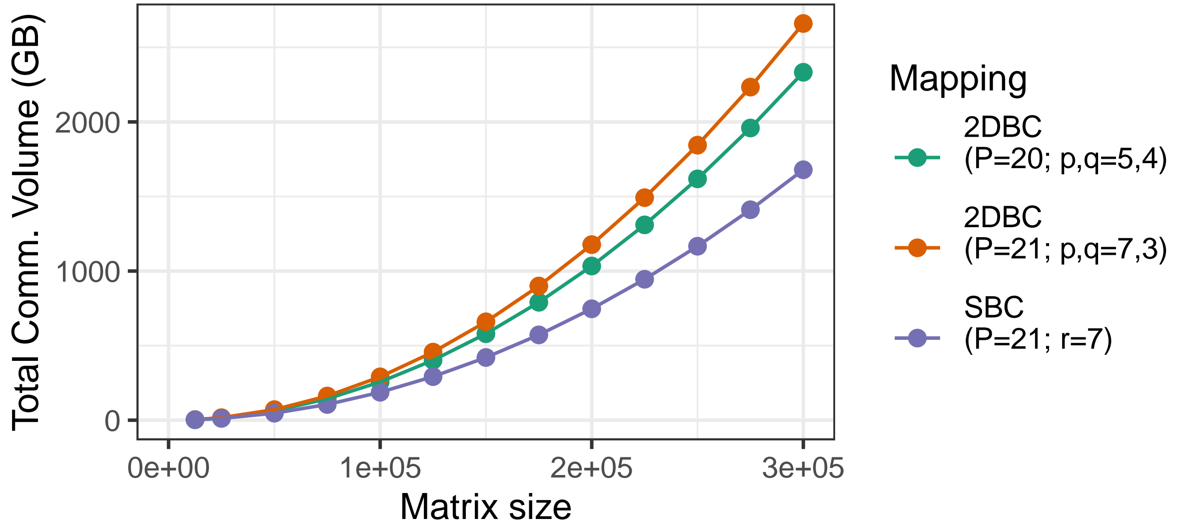


Figure 2.8: Measured volume of inter-node communication during POTRF, for $P = 20$ and 21 (one tile of dimension $b \times b$ in double precision is 2 MB).

plain 2D distribution. This is why we only present results for `COnfCHOX` with a 2D block-cyclic distribution.

2.3.3 Reduction of the Communication Volume

Figure 2.8 shows the evolution with input matrix size of the overall communication volume, in GBytes, when performing Cholesky factorization using either BC or SBC data distribution. The test case used as illustration is $r = 7$. As expected, the SBC distribution induces a significantly smaller communication volume, for all values of m_b, even for cases where BC uses fewer nodes.

The results shown on Figure 2.8 are experimental results, but it is easy to compute

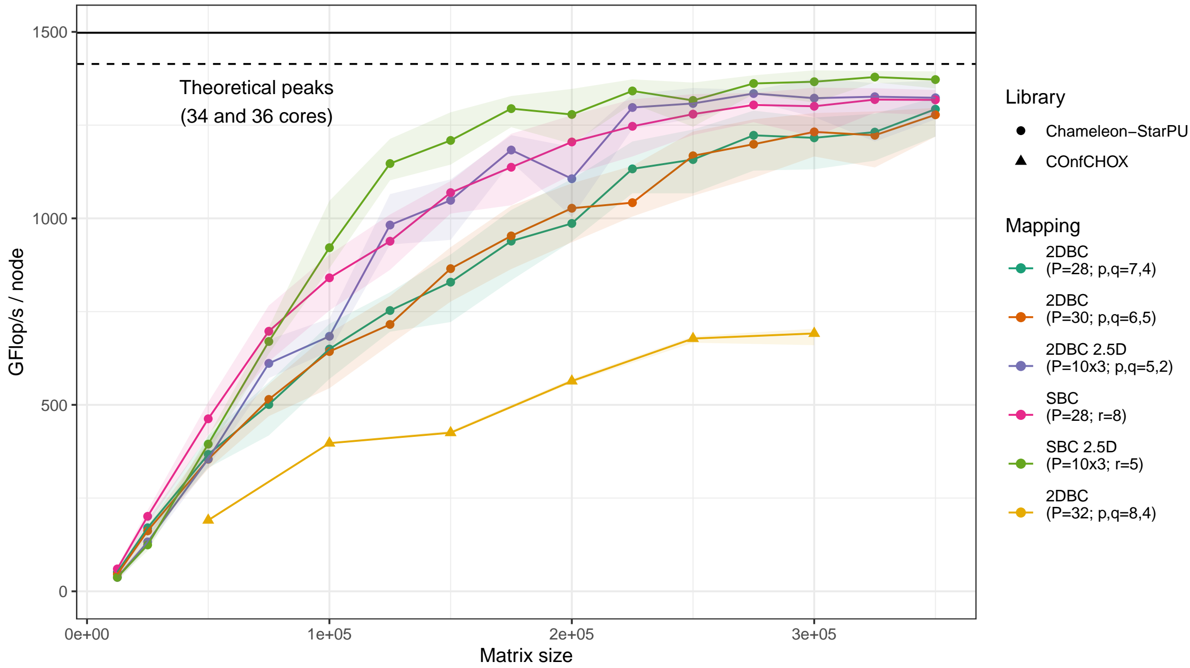


Figure 2.9: Performance per node for Cholesky factorization using 2D and 2.5D versions of BC and SBC for $P = 28$, 30 and 32.

analytically the total communication volume induced for any given distribution. Indeed, `Chameleon` and `StarPU` do not modify the initial placement of data and tasks, only their scheduling. Besides, all communications are done at the tile level using point to point transfers between nodes without any additional optimizations such as detection of collective communications or message aggregation. Hence, the knowledge of the entire data distribution is sufficient to know a priori which tiles need to be transfered between any two nodes, and thus derive the actual communication volume. Indeed, for all experiments carried out, the measured communication volume accurately matches the exepected value computed analytically using the data distribution.

2.3.4 Performance Results for Cholesky factorization

In this section we present experimental results for the distribution parameters detailed in Section 2.3.2. We first use the test case with $P = 28$ to 32 nodes to perform a thorough comparison of our 2D `Chameleon- StarPU` distributions with the alternative BC implementation in `COnfCHOX` library and with more elaborated 2.5D versions. Additional results using only `Chameleon- StarPU` 2D distributions with different number of nodes are shown on Figures 2.11 for performance and Figures 2.12 for the total running time.

Figure 2.9 shows the results obtained by all approaches for the $r = 8$ case which corresponds to $P = 28$, where the 2.5D variants use $c = 3$ slices. The `COnfCHOX` library obtains significantly better performance with power-of-two number of nodes, so we present the results with $P = 32$ for this implementation. The `Chameleon` library clearly outperforms the `COnfCHOX` implementation and manages to approach the peak performance for very

large matrix sizes. This is an expected result because the `COnfCHOX` library implements the Cholesky factorization using synchronized iterations. On the other hand, by using the task-based execution model, `Chameleon` can perform tasks asynchronously thus allowing to overlap communications with computations. This results in better performance.

The comparison between the allocation schemes in `Chameleon` shows that reducing the communication volume increases the performance further. This is particularly true for intermediate values of m_b where the communication has the most impact: indeed, when m_b is large, the $\mathcal{O}(m_b^3)$ computation cost overshadows the $\mathcal{O}(\frac{m_b^2}{\sqrt{P}})$ communication cost.

In the case of intermediate values of m_b, the benefit gained from the SBC distribution over the BC distribution is similar to the benefit gained by the 2.5D approach, which SBC achieves without increasing the memory requirements. Furthermore, these benefits are not exclusive and the 2.5D SBC distribution yields even better performance than all other schemes. In total, the 2D SBC distribution obtains up to 23% improvement over 2D BC and the 2.5D SBC distribution achieves up to 11% improvement over 2D SBC. 2.5D SBC also achieves improvement of up to 27% over the standard 2D BC distribution.

Figure 2.11 displays similar results for other values of r, focusing on the relative performance of 2D SBC and BC. We can see that the improvement observed above is valid over all tested values of P. The evolution of total running time against matrix size is provided Figure 2.12 for the same values of P. The plots illustrate the overall reduction of running time achieved by the SBC distribution compared to BC. Since the performance gain is limited for very large matrices, only results for $m_b \leqslant 200,000$ are shown to highlight the differences.

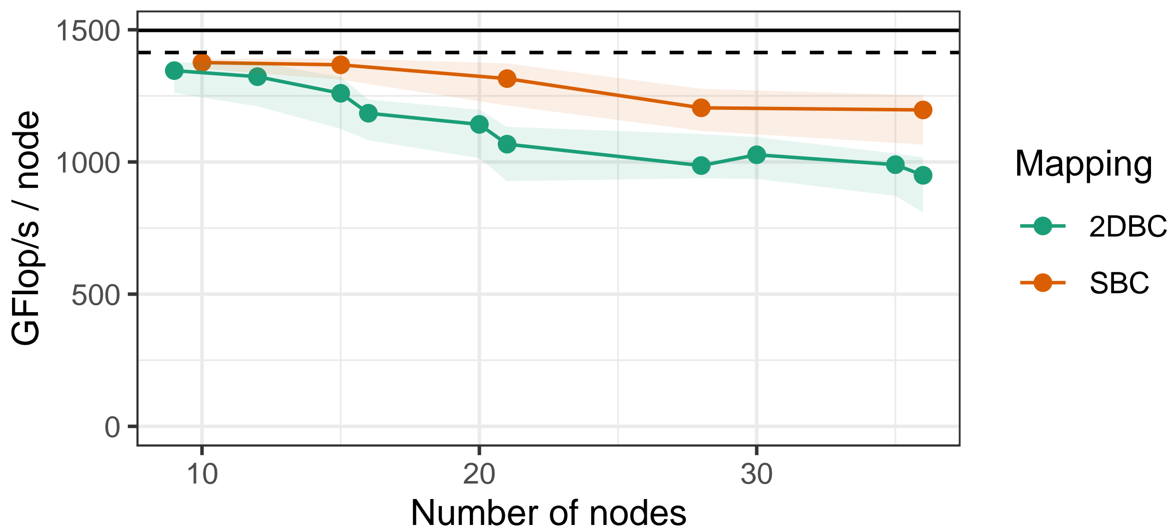
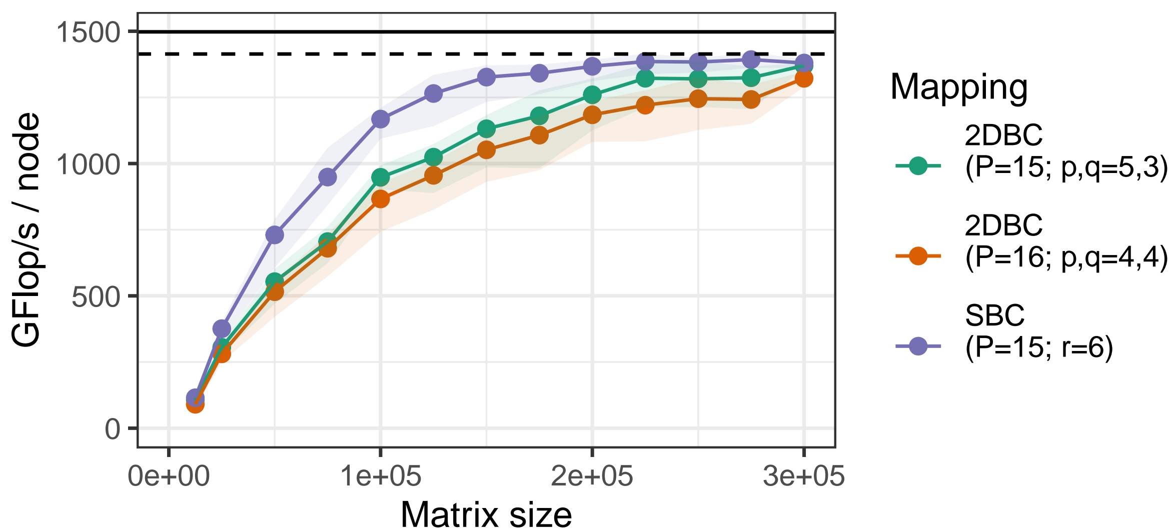
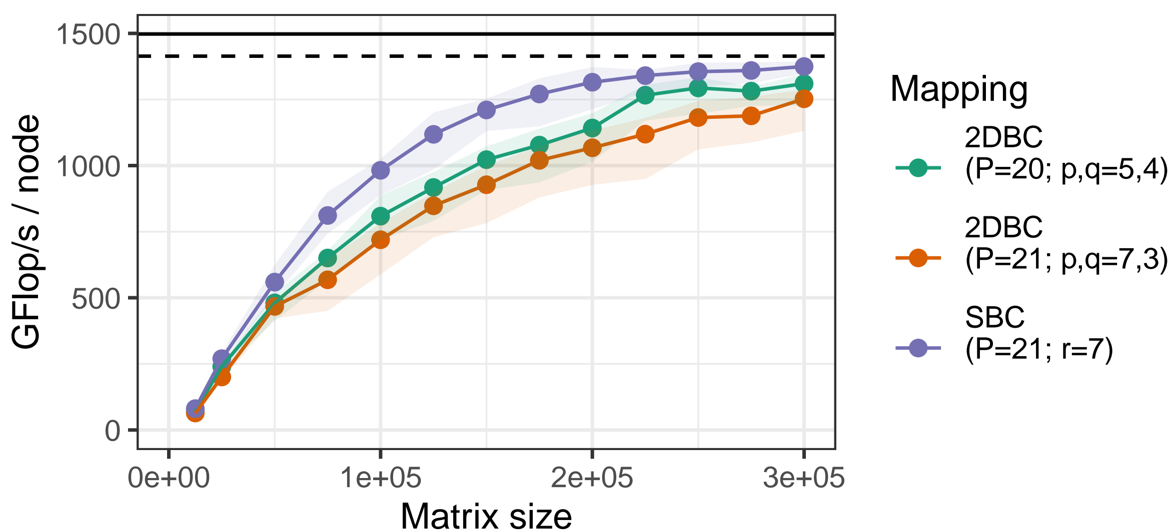


Figure 2.10: Strong scaling of BC and SBC for $m_b = 200,000$.

The same results are presented Figure 2.10 with focus on strong-scaling. It shows that SBC has a much better scalability than BC. For example, for a matrix size $m_b = 200,000$, SBC with $P = 36$ achieves roughly the same efficiency per node as BC with $P = 16$.





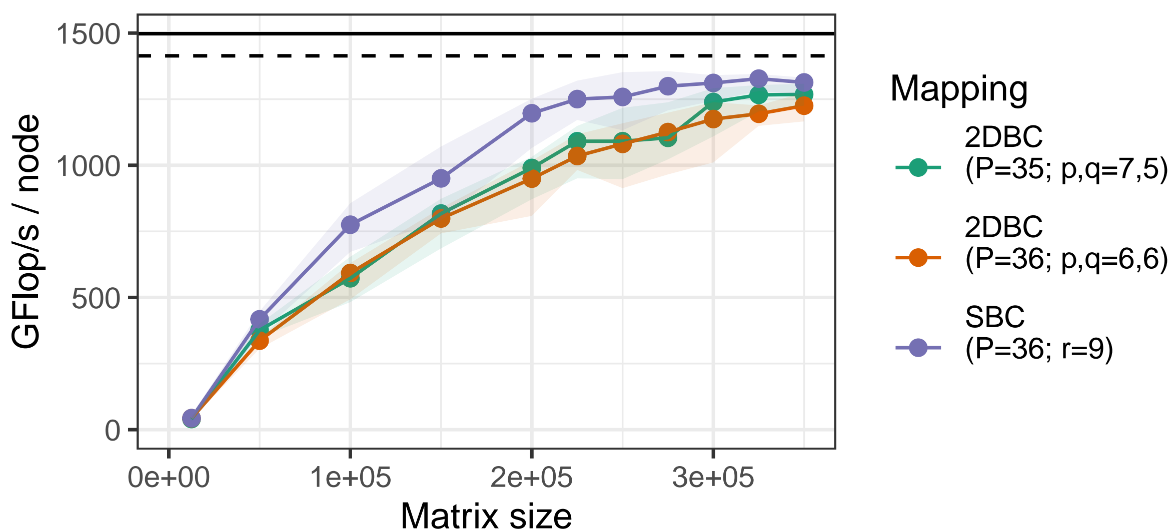
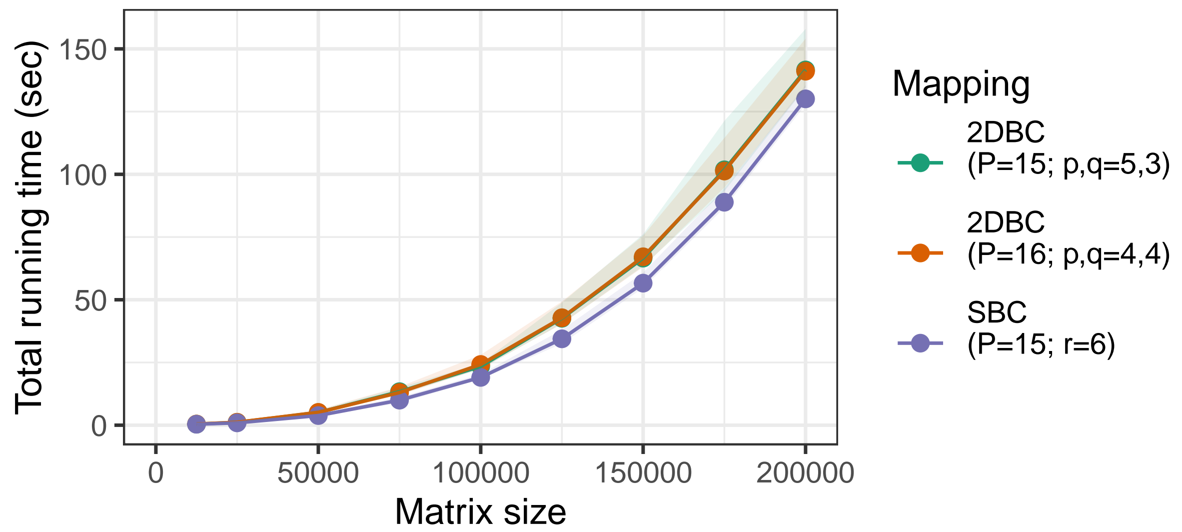
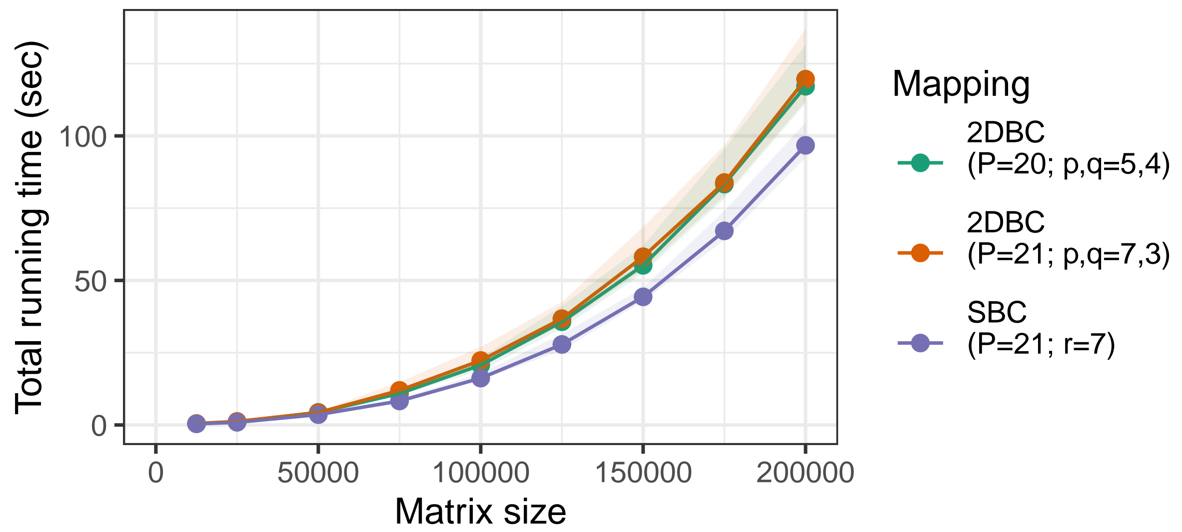


Figure 2.11: Performance per node for Cholesky factorization (GFlop/s per node) using BC and SBC, for P ranging from 15 to 36.





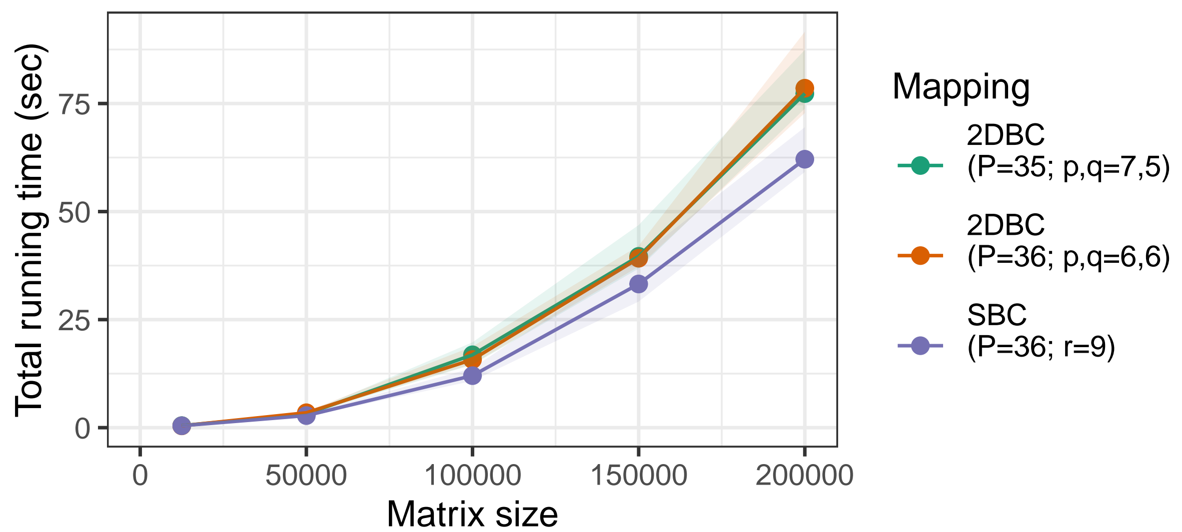


Figure 2.12: Total running time of Cholesky factorization using BC and SBC allocations, for P ranging from 15 to 36.

2.3.5 Performance for Other Operations

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

2.3.5.1 Solving linear systems

POSV aims at solving the linear system $\mathbf{A} \cdot \mathbf{X} = \mathbf{B}$ for the unknown $\mathbf{X}$ where $\mathbf{A}$ is symmetric positive definite and $\mathbf{B}$ is $m_b \times n_b$. Multiple right-hand sides can be gathered as columns of $\mathbf{B}$ to perform several solves simultaneously. Our test case uses a right-hand side of dimensions $m_b \times 1$ tiles which is customary. POSV involves three steps:

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

The tiled version of triangular solve operation (TRSM) is presented in Algorithm 6. Computations on $\mathbf{B}$ are column-wise independent and thus totally parallel, *i.e.* n_b can be selected as large as desired without sacrificing on parallelism. Those are dealt with by the external `for` loop, line 1 in the algorithm.

Algorithm 6: Tiled triangular solve algorithm (TRSM)

Input: ($\mathbf{B}$, $\mathbf{A}$): $\mathbf{B}$ is $m_b \times n_b$, $\mathbf{A}$ is $m_b \times m_b$ lower triangular
Output: ($\mathbf{X}$): $\mathbf{X}$ is $m_b \times n_b$ such that $\mathbf{X} = \mathbf{B} \cdot \mathbf{A}^{-1}$

1 **for** $k = 1 \dots n_b$ **do**
2   **for** $j = 1 \dots m_b$ **do**
3         $\mathbf{B}(j,k) \leftarrow \text{TRSM}(j,k)$: $\text{TRSM}(\mathbf{B}(j,k), \mathbf{A}(j,j))$
4   **for** $i = j+1 \dots m_b$ **do**
5             $\mathbf{B}(i,k) \leftarrow \text{GEMM}(i,k,j)$: $\text{GEMM}(\mathbf{B}(i,k), \mathbf{A}(i,j), \mathbf{B}(j,k))$

Because of the task dependencies, distributed TRSM operation requires two types of inter-node communications for each column $k \in \{1, \dots, n_b\}$:

a) each tile $\mathbf{A}(i,j)$ for $1 \leqslant j \leqslant i \leqslant m_b$ is used as input to update tile $\mathbf{B}(i,k)$, for $1 \leqslant k \leqslant n_b$, wether via $\text{TRSM}(i,k)$ or $\text{GEMM}(i,k,j)$;

b) each tile $\mathbf{B}(j,k)$ for $j \in \{1, \dots, m_b - 1\}$ and $k \in \{1, \dots, n_b - 1\}$, output of $\text{TRSM}(j,k)$, is used as input to update tile $\mathbf{B}(i,k)$ for $i > j$ via $\text{GEMM}(i,k,j)$.

In our case $\mathbf{B}$ is one tile wide. Thus the communication volume of type (a) depends on the number of different nodes that own tiles of $\mathbf{A}$ on a single row, denoted q_A, and whether the node owning the tile of $\mathbf{B}$ on the same row is among them. In this latter case

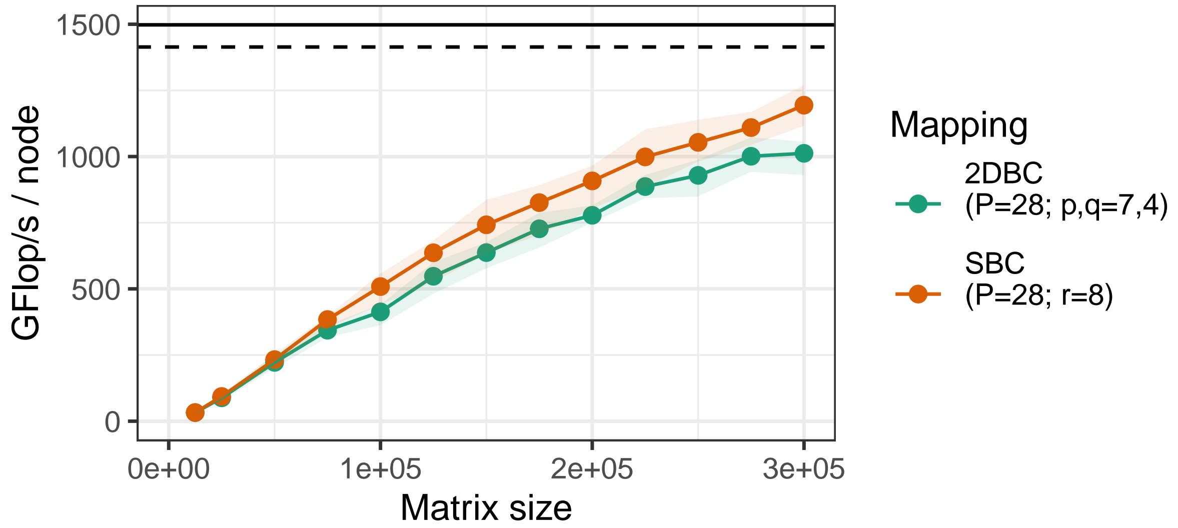


Figure 2.13: Performance per node for POSV with $\mathbf{A}$ distributed using BC and SBC, $P = 28$

the associated number of communication is: $\frac{m_b(m_b-1)}{2}(q_A - 1)$. Otherwise $q_A - 1$ becomes q_A in the expression of the number of communications. The communication volume of (b) only depends on the variety of nodes owning tiles in the column of $\mathbf{B}$, denoted p_B. The associated number of communications is: $m_b p_B$

Since the communication volume of type (a) dominates, we use a 1D row data allocation for matrix $\mathbf{B}$ such that for all $j \in \{1, \ldots, m_b\}$, $\mathbf{B}(j, 1)$ is owned by the same node as $\mathbf{A}(j, j)$.

The resulting performance for $r = 8$ is presented in Figure 2.13. We can observe that SBC achieves better performance than BC, however the improvement ratio is lower than for the Cholesky factorization alone, as can be seen on Figure 2.9. This can be explained easily: the TRSM operations, performed at stages (2) and (3) of POSV, induce additional computations and communications compared to POTRF alone, performed at stage (1). Now the additional communications are independent of the distribution of $\mathbf{A}$. Hence, the relative effect of using SBC distribution on the total running time of the operation is lower for POSV than for POTRF alone.

2.3.5.2 Inversion operation with data redistribution

POTRI is the operation used to compute the inverse of symmetric positive definite matrix $\mathbf{A}$. It is composed of three steps:

1. Cholesky factorization $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\mathsf{T}$: $\mathbf{L} \leftarrow \text{POTRF}(\mathbf{A})$

2. triangular inversion, compute $\mathbf{L}^{-1}$: $\text{TRTRI}(\mathbf{A})$

3. symmetric matrix multiplication $\mathbf{A}^{-1} = (\mathbf{L}^{-1})^\mathsf{T} \cdot \mathbf{L}^{-1}$: $\mathbf{A}^{-1} \leftarrow \text{LAUUM}(\mathbf{L}^{-1})$

Algorithm 7: Tiled symmetric matrix multiplication algorithm (LAUUM)

Input: $(\mathbf{W})$: $\mathbf{W}$ is $m_b \times m_b$ lower triangular
Output: $(\mathbf{A}^{-1})$: $\mathbf{A}^{-1}$ is $m_b \times m_b$ symmetric such that $\mathbf{A}^{-1} = \mathbf{W}^\mathsf{T} \cdot \mathbf{W}$

```
1 for i = 1 ... m_b do
2   for j = 1 ... i-1 do
3     W(j,j) ← SYRK(j,i): SYRK(W(j,j), W(i,i))
4     for k = j+1 ... i-1 do
5       W(k,j) ← GEMM(k,j,i): GEMM(W(k,j), W(k,i), W(i,j))
6     W(i,j) ← TRMM(i,j): TRMM(W(i,i), W(i,j))
7   W(i,i) ← LAUUM(i): LAUUM(W(i,i))
8 A^{-1} ← W
```

The steps of LAUUM operation are described in Algorithm 7. Notice that it makes use of TRMM as a sub-operation. TRMM is similar to a GEMM operation, which is a matrix multiplication, except that the first input matrix on the left of the product is triangular. It implies the following data dependencies:

a) each tile $\mathbf{W}(i,i)$, for $i \in \{1,\dots,m_b\}$, output of LAUUM(i) is used as input of:

- SYRK(j), for $j \in \{1,\dots,i-1\}$, to update tiles $\mathbf{W}(j,j)$;
- TRMM(i,j), for $j \in \{1,\dots,i-1\}$, to update tiles $\mathbf{W}(i,j)$;

b) each tile $\mathbf{W}(i,j)$, for $i \in \{1,\dots,m_b\}$, $j \in \{1,\dots,i-1\}$, output of TRMM(i,j) is used as input of GEMM(k,j,i), for $k \in \{j+1,\dots,i-1\}$, to update tiles $\mathbf{W}(k,j)$.

LAUUM actually features the same data dependencies as POTRF. In particular, just as for POTRF, the dominant part of the total number of communications generated when performing LAUUM comes from data dependency of type (b): output of TRMM $\rightarrow$ input of GEMM. Hence both operations induce the same communication volume for a given distribution scheme.

On the other hand, some computations involved in TRTRI are performed using non-symmetric input. This operation features the following data dependencies:

a) each tile $\mathbf{L}(k,k)$, for $k \in \{1,\dots,m_b\}$, output of TRTRI(k) is used as input of:

- TRSM(i,k), for $i \in \{k+1,\dots,m_b\}$, to update tiles $\mathbf{L}(i,k)$;
- TRSM(k,j), for $j \in \{1,\dots,k-1\}$, to update tiles $\mathbf{L}(k,j)$;

b) tiles $\mathbf{L}(i,k)$ and $\mathbf{L}(k,j)$, for $i \in \{k+1,\dots,m_b\}$ and $j \in \{1,\dots,k-1\}$, output of respectively TRSM(i,k) and TRSM(k,j), are used as input of GEMM(i,j,k) to update tiles $\mathbf{L}(i,j)$.

Data dependencies of type (a) require that each tile $\mathbf{A}(k,k)$, $k \in \{1,\dots,m_b\}$ is sent on row k and column k. Data dependencies of type (b) on the other hand imply that

Algorithm 8: Tiled triangular inversion algorithm (TRTRI)

Input: (**L**): **L** is $m_b \times m_b$ lower triangular
Output: ($\mathbf{L}^{-1}$)

```
1 for k = 1 ... m_b do
2     for i = k+1 ... m_b do
3         L(i,k) ← TRSM(i,k): TRSM(L(i,k), L(k,k))
4         for j = k-1 ... m_b do
5             L(i,j) ← GEMM(i,j,k): GEMM(L(i,j), L(i,k), L(k,j))
6     for j = k-1 ... m_b do
7         L(k,j) ← TRSM(k,j): TRSM(L(k,j), L(k,k))
8     L(k,k) ← TRTRI(k): TRTRI(L(k,k))
```

each tile $\mathbf{A}(i,j)$, $1 < j < i < m_b$ is sent to the left on row i and below on column j. Communications generated by dependencies of type (b) are dominant. Hence, keeping the same notation as for Theorem 4, the leading term of communication volume when performing TRTRI is:

- for BC: $S(p+q-2) \sim 2S\sqrt{P}$;
- for SBC extended: $S(2r-2) \sim 2\sqrt{2}S\sqrt{P}$.

Using the BC distribution therefore generates a smaller communication volume than SBC for this operation.

Because of this difference in the communication pattern for the different sub-operations of POTRI, we consider a mixed strategy involving remapping of data between them. We denote it SBC *remap* BC: POTRF and LAUUM are performed using SBC allocations while TRTRI is done with BC. Data redistribution of the whole matrix occurs before and after TRTRI to change the allocation. Higher order terms of the communication volume for the whole POTRI are then given by:

- for BC: $3S(p+q-2) \sim 6S\sqrt{P}$;
- for SBC *remap* BC:

$$\underbrace{2S(r-2)}_{\text{POTRF and LAUUM}} + \underbrace{S(p+q-2)}_{\text{TRTRI}} + \underbrace{2S}_{\text{two remaps}} = S(2r+p+q-4) \sim 2(\sqrt{2}+1)S\sqrt{P}.$$

The remapped version of SBC asymptotically generates a smaller communication volume than BC, though its relative advantage is smaller than for POTRF alone: the reduction ratio is $\frac{3}{\sqrt{2}+1} \simeq 1.24$ instead of $\sqrt{2}$. A small performance gain can thus be expected when comparing this strategy against BC. However, this asymptotic performance gain is achieved for values of P which are large enough so that the cost of redistribution, which does not depend on P, is negligible compared to the communication cost, which is proportional to $\sqrt{P}$. Figure 2.14 presents the case $r = 8$ ($P = 28$), $p = 7$ and $q = 4$, so

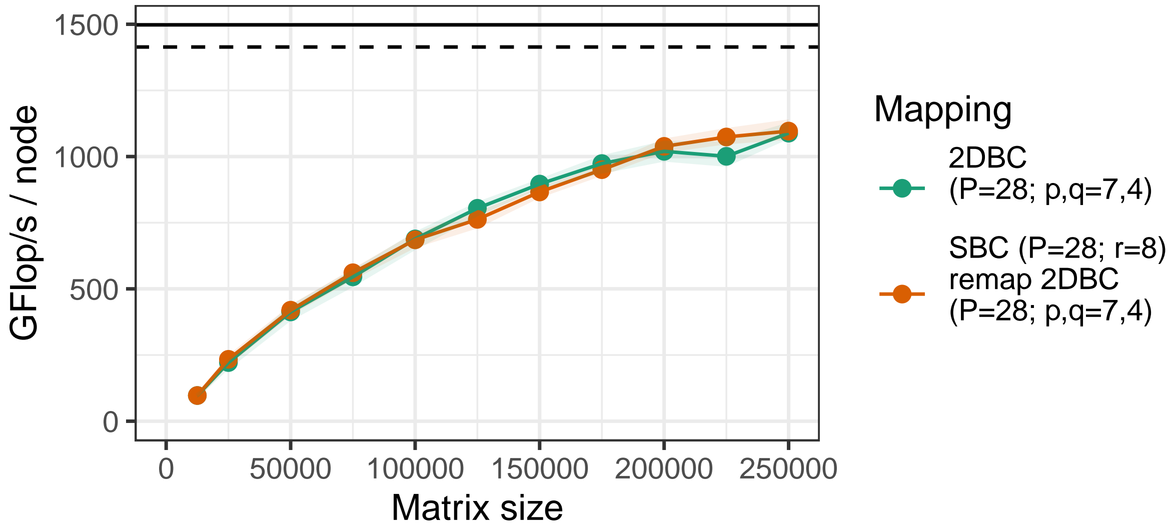


Figure 2.14: Performance per node for POTRI using BC and SBC, $P = 28$.

that the communication volume is reduced by a factor of only $27/23 = 1.17$. In that case, the reduction in communication is too low to obtain a visible improvement in performance. Still, SBC reduces the communication volume, and thus energy consumption and contention with other applications, without degrading performance. In addition, this experimental result shows that SBC data allocation can be seamlessly integrated to multi-operations workflow via data redistribution to reduce the communication volume on specific symmetric steps while leaving the others untouched, hence resulting in global better performance.

2.4 Conclusion

In this chapter we have questioned whether the reduction of overall communication can lead to smaller running time of distributed operation, for linear algebra applications. We have considered the case study of distributed dense Cholesky factorization with minimization of the total volume of inter-node communications as objective function. We have developed an original data distribution, Symmetric Block Cyclic (SBC), that is suited to the symmetry of input matrix $\mathbf{A}$ and aimed at generating few communications. Indeed, we proved analytically that it actually reduces the communication volume by a factor of $\sqrt{2}$ compared to the classical Block Cyclic (BC) distribution. Beyond 2D data distributions that rely on the owner computes rule to distribute tasks among nodes, we propose a 2.5D version of SBC. In a configuration with limited memory of size M per node, we have shown that 2.5D SBC generates a total communication volume of $\frac{1}{2} \cdot \frac{m_b^3}{\sqrt{M}} + o(m_b^3)$ when performing a Cholesky factorization. This improves by a factor of 2 over the previous best result using 2.5D version of the BC distribution. Experimental results obtained with the `Chameleon` library associated with the `StarPU` runtime system show that SBC achieves significantly improved performance for the Cholesky factorization: the total running time

is reduced compared to similar execution using the BC distribution resulting in a higher performance per node. Besides, the communication volume measured match the levels predicted by the theoretical analysis.

Those results show that reducing the total communication volume can lead to additional performance gains, regardless of the computational aspect of the execution. Hence they illustrate how important it is to use a data distribution scheme adapted to the data access pattern: making best use of features specific to the considered operation to reduce communication can lever the potential performance benefit. This work also highlights the effectiveness and flexibility of task-based implementations that enable usage of complex or irregular data distributions while allowing to reach high performance, what would be extremely hard to achieve by explicitly stating tasks and data dependencies using `MPI`

Indeed in our case, the task-based execution model used by the `Chameleon` library associated with `StarPU` allows for a seamless integration of the SBC distribution without changing the Cholesky factorization implementation. Moreover, the experiments with the POSV and POTRI operations show that SBC can be used in a wide variety of operations. It illustrates the fact that, in addition to developing different distributions highly suited for specific operations, it is crucial to work on enabling their efficient integration in complex computation chains.

Chapter 3

Communication Optimal Algorithms for Symmetric Operations

The conclusions from Chapter 2 seem quite promising as they show that inter-node communication reduction is an effective way to improve performance of distributed operations. It is all the more satisfying that it does not require to target a particular subset of communication but simply to seek at minimizing the total communication volume. As illustrated by Symmetric Block Cyclic (SBC), it can be achieved via a data distribution tailored to take advantage of the specific features of the communication pattern of the considered operation. In a task-based context, such distribution can be arbitrarily complex to save communications without interfering with the efficiency of each task execution hence resulting in higher performance.

Such potential performance gains invite to push further the reduction of communications for every possible operation and design data distributions which match existing theoretical bounds on the minimum communication volume required. For the Cholesky factorization, in terms of total communication volume, there still exists a gap between the lower bound [62] which claims that at least $\frac{1}{3\sqrt{2}} \cdot m_b^2\sqrt{P}$ communications are necessary, under the memory scalable assumption, and the best known algorithm, namely 2D Symmetric Block Cyclic (SBC), which performs $\frac{1}{\sqrt{2}} \cdot m_b^2\sqrt{P} + o(m_b^2)$ communications. A gap also exists for the symmetric rank-k update operation which consists in computing $\mathbf{C} = \mathbf{A} \cdot \mathbf{A}^\mathsf{T}$, where $\mathbf{A}$ is $m_b \times n_b$, and whose data dependencies are very similar to those of the Cholesky factorization. It is actually used as sub-routine whithin the Cholesky factorization.

In this chapter, we study both operations under the assumptions of the out-of-core model. In such a setting, any arithmetic operation can only be performed if the required input data is in fast memory. We assume that a given algorithm explicitly controls which data is loaded and removed to and from the fast memory. To tackle the problem, we use the *operational intensity* metric, denoted ρ, which is a useful tool to compare the efficiency of algorithms and their schedules regarding communications. The operational intensity is defined as the ratio of the number of arithmetic operations to the volume of data movement to/from the "slow" memory. For a given operation, the whole set of arithmetic operations is fixed, minimizing communications is therefore equivalent to maximizing the operational intensity. The arithmetic operations can be performed in different orders according to the

algorithm, allowing to generate fewer communications between slow and fast memory, also called *I/O operations*, thus achieving higher operational intensity. Similarly to Chapter 2 for tiled algorithms, communications are counted as individual transfers of elements to and from the slow memory. We qualify as *sequential*, or simply out-of-core, the schedule of an algorithm for a given operation in this context. As mentioned in Section 2.1.1, the theoretical bounds in the out-of-core setting can be immediately extended to the parallel case under the memory scalable assumption. Sequential algorithms developed in this model can also give insight for efficient parallel versions.

In the following we close the gap by improving the best bound by a factor of $\sqrt{2}$ and designing optimal algorithms for the symmetric rank-k update and the Cholesky factorization. In the first section, we review state-of-the-art works regarding bounds and sequential algorithms for the Cholesky factorization and associated operations. Then, in the next section, we provide an insight of the core idea to maximize the operational intensity of symmetric operations and explain why it is not captured by state-of-the-art bounds and algorithms. We also detail useful assumptions and definitions for subsequent derivation of bounds. The third section details the proof of the improved bound for the symmetric rank-k update which is then extended to the Cholesky factorization. Finally, in the last section, we describe an original sequential algorithm, **T**riangle **B**lock **S**YRK (**TBS**), to perform the symmetric rank-k update and show that it generates the minimal amount of communications according to the newly found bound. We then use it as a building block for another sequential algorithm for the Cholesky factorization, **L**arge **B**lock **C**holesky (**LBC**), that is also proven optimal.

3.1 State-of-the-art

3.1.1 Summary of Communication Bounds

Several references that have been mentioned in Section 2.1 are not specific to the parallel model and therefore provide valid bounds in the context of the out-of-core model. As a summary of the results presented there, we can remind:

- Hong and Kung show in [50] that matrix multiplication requires $\Omega(\frac{m^3}{\sqrt{M}})$ communications in the out-of-core setting;
- in [53] Irony *et al.* prove that there exists a memory-communication trade-off for the matrix multiplication; in the out-of-core model it gives a lower bound on the number of communication: $\frac{m_b \cdot n_b \cdot k_b}{2\sqrt{2}\sqrt{M}} - M$;
- this result is emcompassed by a more general formulation valid for several operations, including the matrix multiplication, symmetric rank-k update, LU and Cholesky factorization, presented in [14] by Ballard *et al.*: the minimum communication volume is $\Omega(\frac{\#\text{ total arithmetic operations}}{\sqrt{M}})$;
- in [62] Olivry *et al.* develop an automatic tool called `IOLB`, to derive non-asymptotic bounds for a large set of operations under the assumptions of the out-of-core model;

their results state that performing a symmetric rank-k update requires a minimum of $\frac{1}{2} \cdot \frac{m^2 n}{\sqrt{M}} + o(m^2 n)$ communications and Cholesky factorization $\frac{1}{6} \frac{m^3}{\sqrt{M}} + o(m^3)$.

Recent work from Kwasniewski *et al.* [57] uses explicit enumeration of data reuse to provide an improved bound for both the LU and Cholesky factorization. It is formulated in terms of operational intensity and can thus be applied to the out-of-core model. The bound is primarily derived for LU factorization and the authors claim that it is valid for Cholesky factorization. It states that, for both operations, $\rho \leqslant \frac{\sqrt{M}}{2}$, which implies that LU factorization requires at least $\frac{2}{3} \cdot \frac{m^3}{\sqrt{M}} + o(m^3)$ communications and Cholesky factorization $\frac{1}{3} \cdot \frac{m^3}{\sqrt{M}} + o(m^3)$.

3.1.2 Overview of Sequential Algorithms

In [19] Béreux proposes out-of-core algorithms for the symmetric rank-k update, triangular solve, and Cholesky factorization. Two implementations of the tiled versions of those algorithms exist using either one or two tiles that reside simultaneously in memory. We are only interested in *one-tile* implementations which maximize the operational intensity, as mentioned in Section 3.3 of [19]. A recursive version of the Cholesky factorization algorithm is also developed that we do not consider in this chapter.

The out-of-core algorithms are based on single tile operations that perform updates of the tile residing in memory using *narrow-blocks* of negligible size. The underlying principle is to select the tile size b such that the tile whose values are updated can stay in memory simultaneously with as many narrow-blocks as necessary which are used as input to perform the computations; a narrow-block being a set of b elements of one of the input tiles. The size of the narrow-blocks can be tuned to fit the memory size constraint. As an example of such single tile operations, Algorithm 11 details the matrix multiplication, denoted NBK_GEMM. The algorithm sets the tile size such that $M \geqslant b^2 + 2b$. Hence the input tile of $\mathbf{C}$ can stay in memory during the whole execution of the algorithm with two narrow-blocks of $\mathbf{A}$ and $\mathbf{B}$ that are used to update one of its element. By using such single tile operations, the out-of-core algorithms from Béreux achieve high operational intensity by ensuring that each tile is loaded and flushed from memory only once, thus minimizing I/O data movements. The implementations of single tile symmetric rank-k update, NBK_SYRK, and triangular solve, NBK_TRSM, are detailed respectively in Algorithms 12 and 13. Béreux also makes use of the element-wise version of the Cholesky factorization, denoted ELM_CHOL. It is detailed in Algorithm 10 as a reminder of the same algorithm provided in Section 1.1.1.

Using those single tile algorithms as building blocks, complete out-of-core algorithms for operations on entire matrices are developed by Béreux. In this chapter, we consider the following *one-tile* implementations of out-of-core algorithms as a reference for those operations: OOC_SYRK Algorithm (14) for the symmetric rank-k update, OOC_TRSM Algorithm (15) for the triangular solve, and OOC_CHOL algorithm(16) for the Cholesky factorization.

Algorithm 9: ELM_SYRK, element-wise symmetric rank-k update algorithm

Input: $(\mathbf{C}, \mathbf{A})$: $\mathbf{C}$ is $m \times m$ symmetric, $\mathbf{A}$ is $m \times n$
Output: $(\mathbf{C})$: such that $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{A}^\intercal$

```
1 for k = 1 to n do
2   for i = 1 to m do
3     for j = 1 to i do
4       C_{i,j} += A_{i,k} · A_{j,k}          update operations
5
```

Algorithm 10: ELM_CHOL, element-wise Cholesky factorization algorithm

Input: $(\mathbf{A})$: $\mathbf{A}$ is $m \times m$ symmetric positive definite
Output: $(\mathbf{L})$: $\mathbf{L}$ is $m \times m$ lower triangular such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\intercal$

```
1 for k = 1 to m do
2   A_{k,k} = sqrt(A_{k,k})
3   for i = k + 1 to m do
4     A_{i,k} = A_{i,k} / A_{k,k}
5     for j = k + 1 to i do
6       A_{i,j} -= A_{i,k} · A_{j,k}          update operations
7 L ← lower triangular part of A
```

Algorithm 11: NBK_GEMM, narrow-blocks single tile matrix multiplication

Input: $(\mathbf{C}, \mathbf{A}, \mathbf{B})$: $\mathbf{C}$ is $b \times b$, $\mathbf{A}$ is $b \times n$, $\mathbf{B}$ is $n \times b$
Output: $(\mathbf{C})$: such that $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{B}$
Assumes: memory of size $M \geqslant b^2 + 2b$

1. Load $\mathbf{C}$ in memory.
2. **for** $k = 1$ to n **do**
3. Load $\mathbf{A}_{\cdot,k}$, the kth narrow blocks (column) of $\mathbf{A}$ in memory.
4. Load $\mathbf{B}_{k,\cdot}$, the kth narrow blocks (row) of $\mathbf{B}$ in memory.
5. Perform the outer product of the narrow blocks to update $\mathbf{C}$: $\mathbf{C} \mathrel{+}= \mathbf{A}_{\cdot,k} \cdot \mathbf{B}_{k,\cdot}$
6. Flush the narrow-blocks $\mathbf{B}_{k,\cdot}$ and $\mathbf{A}_{\cdot,k}$ from memory.

These algorithms respectively generate the following number of I/O operations:

$$Q_{\text{OOC_SYRK}}(m, n) = \frac{m^2 n}{\sqrt{M}} + \mathcal{O}(mn)$$

$$Q_{\text{OOC_TRSM}}(m, n) = \frac{m^2 n}{\sqrt{M}} + \mathcal{O}(mn)$$

$$Q_{\text{OOC_CHOL}}(m) = \frac{m^3}{3\sqrt{M}} + \mathcal{O}(m^2)$$

Algorithm 12: NBK_SYRK, narrow-blocks single tile symmetric rank-k update

Input: ($\mathbf{C}$, $\mathbf{A}$): $\mathbf{C}$ is $b \times b$ symmetric, $\mathbf{A}$ is $b \times n$
Output: ($\mathbf{C}$): such that $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{A}^\mathsf{T}$
Assumes: memory of size $M \geqslant b^2 + b$

```
1  Load C in memory.
2  for k = 1 to n do
3      Load A_{.,k}, the k^th narrow blocks (column) of A in memory.
4      Perform the outer product of the narrow block with its transposed self to
         update C: C += A_{.,k} · A^T_{.,k}
5      Flush narrow-block A_{.,k} from memory.
```

Algorithm 13: NBK_TRSM, narrow-blocks single tile triangular solve

Input: ($\mathbf{B}$, $\mathbf{A}$): $\mathbf{B}$ is $b \times b$, $\mathbf{A}$ is $b \times b$ upper triangular
Output: ($\mathbf{X}$): $\mathbf{X}$ is $b \times b$ such that $\mathbf{X} = \mathbf{B} \cdot \mathbf{A}^{-1}$ (solve $\mathbf{X} \cdot \mathbf{A} = \mathbf{B}$)
Assumes: memory of size $M \geqslant b^2 + b$

```
1  Load B in memory.
2  for k = 1 to b do
3      Load A_{k,k→b}, the k^th narrow blocks (row) of A in memory.
4      for i = 1 to b do
5          Update row i of B:
6          B_{i,k} ← B_{i,k} / A_{k,k}
7          for j = k + 1 to b do
8              B_{i,j} ← B_{i,j} − A_{k,j} B_{i,k}
9      Flush narrow-block A_{k,k→b} from memory.
10 X ← B
```

3.2 Preliminaries

3.2.1 Motivation

One can notice that the sequential algorithm for Cholesky factorization in the out-of-core model proposed by Béreux in [19] matches the bound proved by Kwasniewski *et al.* in [57]: the dominant term for the number of communication is in both case $\frac{1}{3}\frac{m^3}{\sqrt{M}}$. However, the methodology used by Kwasniewski *et al.* to prove the upper bound on operational intensity is based on a restrictive assumption.

Let us summarize their methodology and detail its key steps to explain why it leads to a bound which is invalid in the general case. The method used by the authors is applicable to any operation that consists in nested loops of arithmetic operations, each associated with a single instruction in the algorithm describing the operation. It is based on a careful application of Lemma 1 (numbered Lemma 1 in [57]) which states:

Algorithm 14: OOC_SYRK, out-of-core symmetric rank-k update

Input: $(\mathbf{C}, \mathbf{A})$: $\mathbf{C}$ is $m \times m$ symmetric, $\mathbf{A}$ is $m \times n$
Output: $(\mathbf{C})$: such that $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{A}^\intercal$
Assumes: memory of size $M = b^2 + 2b$

1 Let us define: $m_b = \lfloor \frac{m}{b} \rfloor$ and $\ell = m - m_b b$
2 Let us define the subsets of indices that divide $\mathbf{A}$ and $\mathbf{C}$ into tiles:
3 **for** $k \in \{1, \ldots, m_b\}$ **do**
4 $\quad I_k = \begin{cases} [0\,;\,\ell] & \text{if } k = 0 \\ [\ell + (k-1)b + 1\,;\,\ell + kb] & \text{otherwise} \end{cases}$
5 **for** $k \in \{0, \ldots, m_b\}$ **do**
6 $\quad$ Update tile $\mathbf{C}_{I_k,I_k}$ with SYRK operation:
7 $\quad$ NBK_SYRK($\mathbf{C}_{I_k,I_k}, \mathbf{A}_{I_k,\cdot}$) $\quad$ *$\mathbf{C}_{I_k,I_k}$ is in memory*
8 $\quad$ **for** $i \in \{0, \ldots, k-1\}$ **do**
9 $\quad\quad$ Update tile $\mathbf{C}_{I_k,I_i}$ with GEMM operation:
10 $\quad\quad$ NBK_GEMM($\mathbf{C}_{I_k,I_i}, \mathbf{A}_{I_k,\cdot}, \mathbf{A}^\intercal_{I_i,\cdot}$) $\quad$ *$\mathbf{C}_{I_k,I_i}$ is in memory*

Lemma 1. *Fix a constant $X > M$ and assume that any sub-computation H of a cDAG $G = (V, E)$ which reads at most X elements and writes at most X elements performs a number of operations $|H|$ bounded by $|H| \leqslant H_{\max}$. Consider any execution of G with memory M. Its operational intensity ρ is bounded by $\rho \leqslant \frac{H_{\max}}{X-M}$ and its number of I/O operations Q is bounded by:*

$$Q \geqslant \frac{|V|}{\rho} \geqslant \frac{|V|\,(X - M)}{H_{\max}}$$

The main part of the method is thus to find the largest sub-computation H that requires no more than X access, for a given $X > M$. To do so, Kwasniewski *et al.* develop a technique, based on automatic cDAG analysis, that allows to analytically find such an H by checking for two types of data reuse between statements: (i) a set of elements can be used as input for more than one instruction, (ii) a set of elements, output of one instruction, can be the input of another instruction. The technique assumes what is referred to as the *disjoint array access property*, detailed in Section 2.2 of [57], that explicitly states: “a given vertex can be referenced by only one access function vector per statement”. It can be reformulated as: an element cannot be used as multiple input arguments for the same instruction.

Let us illustrate this assumption in the case of the Cholesky factorization and detail its implication. We focus on the instruction line 6 in Algorithm 1 which corresponds to *update operations* (it is the statement `S3` from Listing 1 in Section 6.2 in [57]). The instruction states that the following update operations are performed:

$$\forall i, j, k \in \{1, \ldots, m\},\ k < j < i \ : \quad \mathbf{A}_{i,j} = \mathbf{A}_{i,j} - \mathbf{A}_{i,k} \cdot \mathbf{A}_{j,k}$$

Algorithm 15: OOC_TRSM, out-of-core triangular solve

Input: ($\mathbf{B}$, $\mathbf{L}$): $\mathbf{B}$ is $m \times n$, $\mathbf{L}$ is $m \times m$ lower triangular
Output: ($\mathbf{X}$): $\mathbf{X}$ is $m \times n$ such that $\mathbf{X} = \mathbf{B} \cdot (\mathbf{L}^\mathsf{T})^{-1}$
Assumes: memory of size $M = b^2 + 2b$

1 Let us define: $m_b = \lfloor \frac{m}{b} \rfloor$ and $\ell = m - m_b b$
2 Let us define the subsets of indices that divide $\mathbf{B}$ and $\mathbf{L}$ into tiles:
3 **for** $k \in \{1, \dots, m_b\}$ **do**
4 $\quad I_k = \begin{cases} [0\,;\,\ell] & \text{if } k = 0 \\ [\ell + (k-1)b + 1\,;\,\ell + kb] & \text{otherwise} \end{cases}$
5 **for** $k \in \{0, \dots, m_b\}$ **do**
6 $\quad$ Update tile $\mathbf{B}_{I_k,I_k}$ with GEMM operation:
7 $\quad$ NBK_GEMM($\mathbf{B}_{I_k,I_k}, \mathbf{B}_{I_k,I_{0\to k-1}}, \mathbf{L}^\mathsf{T}_{I_k,I_{0\to k-1}}$) $\qquad$ $\mathbf{B}_{I_k,I_k}$ *is in memory*
8 $\quad$ Update again tile $\mathbf{B}_{I_k,I_k}$ with TRSM operation:
9 $\quad$ NBK_TRSM($\mathbf{B}_{I_k,I_k}, \mathbf{L}^\mathsf{T}_{I_k,I_k}$) $\qquad$ $\mathbf{B}_{I_k,I_k}$ *is still in memory*
10 $\mathbf{X} \leftarrow \mathbf{B}$

Now, for given $k_0, i_0 \in \{1, \dots, m\}$ such that $k_0 + 1 < i_0 < m$, the element $\mathbf{A}_{i_0,k_0}$ can be used as multiple input arguments for the update operations as it can appear at different positions in the right-hand side of the expression: indeed it can be the second argument for all $j \in \{k_0 + 1, \dots, i_0 - 1\}$ and the third one for all $i \in \{i_0 + 1, \dots, m\}$. More generally, each element in $\{\mathbf{A}_{i,k} \;:\; k, i \in \{1, \dots, m\} \mid k + 1 < i < m\}$ is an input of update operations, as second argument for some iterations, as third argument for some others. Nevertheless the disjoint array access property prevents from taking into account such input data reuse for the same instruction and thus restricts the search of the largest H to a subset of all possible solutions. As a consequence, a bound on operational intensity derived assuming disjoint array access property may not be valid for the general case; it may be too restrictive. In this chapter we actually prove that the bound derived by Kwasniewski *et al.*, $\rho \leqslant \frac{1}{2}\sqrt{M}$, is not valid for the Cholesky factorization since the disjoint array access property assumption does not hold for such operation. Indeed, algorithms TBS for the symmetric rank-k update and LBC for Cholesky factorization achieve $\rho = \frac{1}{\sqrt{2}}\sqrt{M}$ precisely via this type of input data reuse for the update operations.

We can observe that such an input data reuse inside the same instruction is enabled by the symmetry of the input matrix $\mathbf{A}$ for the symmetric rank-k update and Cholesky factorization. The third term of the update operation in the case of non symmetric input actually is $\mathbf{A}_{k,j}$ which implies that disjoint array access property does not prevent any input data reuse and thus is a non restrictive assumption. For the non symmetric counterparts of the symmetric rank-k update and Cholesky factorization, namely matrix multiplication and LU factorization, the bound by Kwasniewski *et al.* [57] is therefore valid. Moreover, it is optimal because the algorithm proposed by Béreux [19] for the Cholesky factorization, applied to LU factorization without modification, achieves the

Algorithm 16: OOC_CHOL, out-of-core Cholesky factorization

Input: ($\mathbf{A}$): $\mathbf{A}$ is $m \times m$ symmetric positive definite
Output: ($\mathbf{L}$): $\mathbf{L}$ is $m \times m$ lower triangular such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\mathsf{T}$
Assumes: memory of size $M = b^2 + 2b$

1 Let us define: $m_b = \lfloor \frac{m}{b} \rfloor$ and $\ell = m - m_b b$
2 Let us define the subsets of indices that divide $\mathbf{A}$ into tiles:
3 **for** $k \in \{1, \ldots, m_b\}$ **do**
4 $\quad I_k = \begin{cases} [0\,;\,\ell] & \text{if } k = 0 \\ [\ell + (k-1)b + 1\,;\,\ell + kb] & \text{otherwise} \end{cases}$
5 **for** $k \in \{0, \ldots, m_b\}$ **do**
6 $\quad$ Update tile $\mathbf{A}_{I_k,I_k}$ with SYRK operation:
7 $\quad$ NBK_SYRK($\mathbf{A}_{I_k,I_k}, \mathbf{A}_{I_k,I_{0\to k-1}}$) $\quad$ *$\mathbf{A}_{I_k,I_k}$ is in memory*
8 $\quad$ Perform an element-wise Cholesky factorization to update again tile $\mathbf{A}_{I_k,I_k}$:
9 $\quad$ ELM_CHOL($\mathbf{A}_{I_k,I_k}$) $\quad$ *$\mathbf{A}_{I_k,I_k}$ is still in memory*
10 $\quad$ **for** $i \in \{k+1, \ldots, m_b\}$ **do** $\quad$ *$\mathbf{A}_{I_i,I_k}$ is in memory*
11 $\quad\quad$ Update tile $\mathbf{A}_{I_i,I_k}$ with GEMM operation:
12 $\quad\quad$ NBK_GEMM($\mathbf{A}_{I_i,I_k}, \mathbf{A}_{I_i,I_{0\to k-1}}, \mathbf{A}^\mathsf{T}_{I_k,I_{0\to k-1}}$)
13 $\quad\quad$ Update it again with TRSM operation:
14 $\quad\quad$ NBK_TRSM($\mathbf{A}_{I_i,I_k}, \mathbf{A}^\mathsf{T}_{I_k,I_k}$)
15 $\mathbf{L} \leftarrow$ lower triangular part of $\mathbf{A}$

same operational intensity.

3.2.2 Methodology, Assumptions and Notations

In the following, we use the same general methodology as in [57], that is: for a given $X > M$, find the largest sub-computation H which requires no more than X data access in order to apply Lemma 1.

We consider the symmetric rank-k update and Cholesky factorization, as described in Algorithms 9 and 10. For the Cholesky factorization, as already mentioned in Section 2.2.2, we can restrict our study to the *update operations*, that can be observed line 4 of Algorithm 9 and line 6 of Algorithm 10, and which is the arithmetic operation that generates the dominant part of the communication volume. Then we can describe each operation by a triplet of positive integers (i, j, k), and for both cases we will further ignore the diagonal operations where $i = j$. The sets of arithmetic operations are denoted $\mathcal{S}$ for the symmetric rank-k update and $\mathcal{C}$ for the Cholesky factorization, and are given by:

$$\mathcal{S} = \left\{(i,j,k) \in \{1,\ldots,m\}^2 \times \{1,\ldots,n\} \,|\, i > j\right\}$$
$$\mathcal{C} = \left\{(i,j,k) \in \{1,\ldots,m\}^3 \,|\, i > j > k\right\},$$

We can see that for each statement of these algorithms, the set of written variables is included in the set of read variables, so we only focus on the input data of each operation. In the description of the sets of arithmetic operations $\mathcal{S}$ and $\mathcal{C}$, the third index k actually corresponds to the update index as defined in Section 1.2.2. In the following, if not specified, the sums and unions of sets using this index are performed over all possible values of k. Note also that, for each (i, j, k), the update operation is composed of one multiplication and one addition that are counted as a single arithmetic operation. In the rest of this chapter, H is used to denote a set of arithmetic operations, subset of $\mathcal{S}$ or $\mathcal{C}$. The following definitions and propositions are presented for $H \subset \mathcal{S}$ and the entire process to derive a communication lower bound detailed in Section 3.3.1 is valid for the symmetric rank-k update operation. However those results are then extended to the Cholesky factorization using the argument presented in Section 3.3.2.

Definition 1. *Given a set $H \subset \mathcal{S}$ of arithmetic operations, $H_{|k}$ is the restriction of H to iteration k:*

$$H_{|k} = \left\{(i, j) \in \mathbb{N}^2 \,|\, (i, j, k) \in H\right\}$$

Definition 2. *Given a subset U of $\mathbb{N}^2$, $\tau(U)$ is the* symmetric footprint *of U:*

$$\tau(U) = \{i \in \mathbb{N} \,|\, \exists j, (i, j) \in U \text{ or } (j, i) \in U\}$$

If $i > j$ for all $(i, j) \in U$, then $|U| \leqslant \frac{|\tau(U)|(|\tau(U)|-1)}{2}$. In particular, this holds for any $H_{|k}$.

With these definitions, we can express the number of data accessed by a set H: using the symmetric rank-k update operation as an example, then $\cup_{k\in\{1,\dots,n\}} H_{|k}$ is the set of elements $\mathbf{C}_{i,j}$ accessed by H, and for any $k \in \{1, \dots, n\}$, $\tau\left(H_{|k}\right)$ is the set of $\mathbf{A}_{i,k}$ elements accessed by H.

Proposition 1. *For any set $H \subset \mathcal{S}$ of operations, the number of data accessed by H is*

$$D(H) = \left|\cup_k H_{|k}\right| + \sum_k \left|\tau\left(H_{|k}\right)\right|$$

3.2.3 Triangle Blocks

Many of the following results make use of the concept of *triangle blocks*, which are generalizations of the diagonal tiles in a tile decomposition of a symmetric matrix. In particular, for the symmetric rank-k update operation, we show that accessing the result matrix by triangle blocks minimizes the communications. The TBS algorithm precisely describes how to partition the result matrix in disjoint triangle blocks. Figure 3.1 (page 67) depicts examples of triangle blocks.

Definition 3 (Triangle block)**.** *Given a set R of integer indices, the* triangle block *$TB(R)$ is the set of all subdiagonal pairs of elements of R:*

$$TB(R) = \{(r, r') \,|\, r, r' \in R \text{ and } r > r'\}$$

It is clear that $|\mathrm{TB}(R)| = \frac{|R|(|R|-1)}{2}$. We say that $\mathrm{TB}(R)$ has side length $|R|$.

For any $\ell \in \mathbb{N}$, we define $\sigma(\ell)$ as the smallest possible side length of a triangle block with at least ℓ elements. $\sigma(\ell)$ is thus the smallest element of $\mathbb{N}$ such that $\ell \leqslant \frac{\sigma(\ell)(\sigma(\ell)-1)}{2}$. By solving the quadratic equation, we get:

Lemma 2. *For $\ell \in \mathbb{N}^*$, $\sigma(\ell) = \lceil \sqrt{\frac{1}{4} + 2\ell} + \frac{1}{2} \rceil$, and $\sigma(0) = 0$.*

For any $\ell \in \mathbb{N}$, we define $T(\ell)$ as any size-ℓ subset of $\mathrm{TB}(\{1, \ldots, \sigma(\ell)\})$. We use $T(\ell)$ as a canonical way of performing ℓ arithmetic operations in an iteration, while minimizing the number of data accesses. Indeed, by definition $|T(\ell)| = \ell$, and it is easy to see that $|\tau\left(T(\ell)\right)| = \sigma(\ell)$.

3.3 Lower Bounds

3.3.1 Symmetric Multiplication (SYRK)

As mentioned above, in order to obtain a lower bound on the communications required for the symmetric rank-k update operation, we first provide an upper bound on the largest sub-computation H than can be performed while accessing at most X data elements. We are thus looking for a bound on the optimal value of the following optimization problem:

$$\mathcal{P}(X): \quad \max |H| \quad \text{s.t.} \begin{cases} D(H) = \left|\cup_k H_{|k}\right| + \sum_k \left|\tau\left(H_{|k}\right)\right| \leqslant X \\ H \subseteq \mathcal{S} \end{cases}$$

The main result of this section can be stated as:

Theorem 2. *The optimal value of $\mathcal{P}(X)$ is at most $\frac{\sqrt{2}}{3\sqrt{3}} X^{\frac{3}{2}}$.*

To prove this theorem, we first show that $\mathcal{P}(X)$ admits triangle-shaped optimal solutions, which we call *balanced solutions*. We then compute an upper bound on the size of such a balanced solution.

3.3.1.1 Balanced Solutions

Definition 4. *For given x and ℓ, we define the* balanced solution $B = B(x, \ell)$ *by:*

$$\begin{cases} B_{|k} = T(\ell) & \text{for all } k \in \{0, \ldots, K-1\}, \\ B_{|K} = T(\ell') & \text{for } k = K, \\ B_{|k} = \emptyset & \text{for all } k > K, \end{cases}$$

where $K = \lfloor \frac{x}{\ell} \rfloor$ and $\ell' = x - K\ell < \ell$.

It is clear that $|B(x,\ell)| = x$, since $K\ell + (x - K\ell) = x$. Besides $\left|\cup_k B(x,\ell)_{|k}\right| = \ell$. The next lemma shows that any solution H can be turned into a balanced solution with lower cost.

Lemma 3. *If H is a solution of $\mathcal{P}(X)$, let the corresponding balanced solution be $B = B\left(|H|, \max_k \left|H_{|k}\right|\right)$. Then $D(B) \leqslant D(H)$.*

Proof. Given a solution H, let us define $\ell_k = \left|H_{|k}\right|$ and denote $\ell = \max_k \ell_k$. As mentioned above, we have $|B| = |H|$ and $\left|\cup_k B_{|k}\right| = \ell = \max_k \ell_k \leqslant \left|\cup_k H_{|k}\right|$. Furthermore, since $\sum_k \ell_k = |H| = K \cdot \ell + \ell'$ and since the $\sigma(\cdot)$ function is concave, we have:

$$\begin{aligned}\sum_k \left|\tau\left(B_{|k}\right)\right| &= K\sigma(\ell) + \sigma(\ell') \\ &\leqslant \sum_k \sigma(m_k) \\ &= \sum_k \left|\tau\left(H_{|k}\right)\right|\end{aligned}$$

This shows that $D(B) \leqslant D(H)$. ☐

In particular, if H is an optimal solution, we obtain the following corollary.

Corollary 1. *There exist x and ℓ such that $B(x,\ell)$ is an optimal solution to $\mathcal{P}(X)$.*

3.3.1.2 Optimal Balanced Solution

A balanced solution B can be described with three integer values I, J in $\{1,\ldots,m\}$ with $J \leqslant I$, and $K \in \{1,\ldots,n\}$ such that

$$\begin{cases} \forall k \in \{0,\ldots,K-1\},\ \tau\left(B_{|k}\right) = T(I) \\ \tau\left(B_{|K}\right) = T(J) \end{cases}$$

Such a solution satisfies $|B| = K\frac{I(I-1)}{2} + \frac{J(J-1)}{2}$ and $D(B) = \frac{I(I-1)}{2} + KI + J$. By relaxing integrity constraints and upper bounds on I, J, K, we get that the optimal size of a balanced solution is at most the optimal value of the following problem $\mathcal{P}'(X)$:

$$\begin{aligned}\mathcal{P}'(X):\quad &\max\left(K\frac{I(I-1)}{2} + \frac{J(J-1)}{2}\right) \\ &\text{s.t.}\quad \begin{cases} \frac{I(I-1)}{2} + KI + J \leqslant X \\ J \leqslant I \end{cases}\end{aligned}$$

Lemma 4. *For any (I,J,K) solution to $\mathcal{P}'(X)$, define $K' = K + \frac{J(J-1)}{I(I-1)}$. Then $(I,0,K')$ is a solution to $\mathcal{P}'(X)$ with the same value.*

Proof. The solution $(I, 0, K')$ is feasible:

$$\begin{aligned}\frac{I(I-1)}{2} + K'I &= \frac{I(I-1)}{2} + KI + J\frac{J-1}{I-1} \\ &\leqslant \frac{I(I-1)}{2} + KI + J && \text{since } J \leqslant I \\ &\leqslant X && \text{since } (I,J,K) \text{ is feasible}\end{aligned}$$

Furthermore, its objective value is $K'\frac{I(I-1)}{2} = K\frac{I(I-1)}{2} + \frac{J(J-1)}{2}$, which is the objective value of (I, J, K). □

This lemma shows that the optimum value of $\mathcal{P}'$ is equal to the optimum value of the simpler $\mathcal{P}''$ problem below:

$$\begin{aligned}\mathcal{P}''(X)\text{:} \quad & \max\left(K\frac{I(I-1)}{2}\right) \\ \text{s.t.} \quad & \frac{I(I-1)}{2} + KI \leqslant X\end{aligned}$$

This problem is now simple enough and we can provide a direct bound on its optimum value.

Lemma 5. *The optimum value of $\mathcal{P}''(X)$ is at most $\frac{\sqrt{2}}{3\sqrt{3}}X^{\frac{3}{2}}$.*

Proof. Reformulated as a minimization problem, $\mathcal{P}''(X)$ becomes:

$$\begin{aligned}\min\left(f(K,I) = -K\frac{I(I-1)}{2}\right) \\ \text{s.t.} \quad g(K,I) = \frac{I(I-1)}{2} + KI - X \leqslant 0\end{aligned}$$

Since the regularity conditions are met over the whole definition space of variables I and K, we can write Karush-Kuhn-Tucker necessary conditions: if (K, I) is a local optimum for $\mathcal{P}''(X)$ then

$$\begin{aligned}&\exists u \geqslant 0, \quad \nabla f(K,I) + u\nabla g(K,I) = 0 \\ \Leftrightarrow\ &\exists u \geqslant 0, \quad \begin{cases} -K(I-\frac{1}{2}) + u(I - \frac{1}{2} + K) = 0 \\ -\frac{I(I-1)}{2} + uI = 0 \end{cases}\end{aligned}$$

which implies $u = \frac{I-1}{2}$, and then $KI = (I-1)(I-\frac{1}{2})$.

Let us denote by (K, I) a local minimum of f. Then $KI = (I-1)(I-\frac{1}{2})$. Besides we can select (K, I) such that $\frac{I(I-1)}{2} + KI - X = 0$. This yields $3I^2 - 4I - (2X - 1) = 0$, and we obtain $I = \frac{2}{3} + \frac{\sqrt{1+6X}}{3}$.

An optimal solution of $\mathcal{P}''(X)$ is thus given by

$$\begin{cases} I^* = \frac{2}{3} + \frac{\sqrt{1+6X}}{3} \\ K^* = (I^* - \frac{1}{2})(1 - \frac{1}{I^*}) \end{cases}$$

and its objective value is

$$\begin{aligned} \mathcal{H}''(X) &= K^* \frac{I^*(I^*-1)}{2} \\ &= \frac{1}{4}(I^*-1)^2(2I^*-1) \\ &= \frac{1}{108}(\sqrt{1+6X}-1)^2(2\sqrt{1+6X}+1) \\ &\leqslant \frac{(\frac{\sqrt{6X}}{3})^3}{2} = \frac{\sqrt{2}}{3\sqrt{3}} X^{\frac{3}{2}} \end{aligned}$$

To understand why the last inequality holds, one can observe that the function $X \mapsto \mathcal{H}''(X) - \frac{\sqrt{2}}{3\sqrt{3}} X^{\frac{3}{2}}$ equals 0 for $X = 0$. Besides,

$$\begin{aligned} \frac{\partial}{\partial X}\Big[\mathcal{H}''(X) - \frac{\sqrt{2}}{3\sqrt{3}} X^{\frac{3}{2}}\Big] &= \frac{1}{6}\big(\sqrt{1+6X}-1\big) - \sqrt{\frac{X}{6}} \\ &= \frac{1}{6}\big[\sqrt{1+6X} - (1+\sqrt{6X})\big] \end{aligned}$$

which is obviously negative. □

3.3.1.3 Final Result

Proof of Theorem 2. The result follows directly from Corollary 1, Lemma 4 and Lemma 5. □

Corollary 2. *The number of data accesses required to perform a symmetric rank-k update operation where* $\mathbf{A}$ *has size* $m \times n$*, with memory* M*, is at least:*

$$Q_{SYRK}(m,n,M) \geqslant \frac{1}{\sqrt{2}} \frac{m^2 n}{\sqrt{M}}$$

Proof. Consider the cDAG of the symmetric rank-k update operation, which has $|\mathcal{S}| = \frac{m^2 n}{2}$ vertices. According to Theorem 2, for any X, any sub-computation H of this cDAG which reads at most X elements has size $|H| \leqslant \frac{\sqrt{2}}{3\sqrt{3}} X^{\frac{3}{2}}$.

In particular, for $X = 3M$, we get $|H| \leqslant \sqrt{2} \cdot M^{\frac{3}{2}}$. The value $X = 3M$ is chosen to obtain the strongest possible bound by maximizing the ratio $\frac{|H|}{X-M}$. According to Lemma 1, the maximal operational intensity of the symmetric rank-k update is $\rho = \frac{|H|}{3M-M} \leqslant \sqrt{\frac{M}{2}}$.

This yields the following bound on the number of data accesses for the complete symmetric rank-k update operation:

$$Q_{\text{SYRK}}(m, n, M) \geqslant \frac{|\mathcal{S}|}{\rho} = \frac{1}{\sqrt{2}} \frac{m^2 n}{\sqrt{M}}$$

□

3.3.2 Cholesky Factorization

We now consider the Cholesky factorization, as described by Algorithm 16. As mentioned above, we focus on the update operations, described by the set $\mathcal{C} = \{(i, j, k) \in \{1, \ldots, m\}^3 \mid i > j > k\}$.

For a given X, the largest subset H that accesses at most X elements can be found by solving $\mathcal{P}(X)$, in which the constraint $H \subseteq \mathcal{S}$ is replaced by $H \subseteq \mathcal{C}$. We consider a relaxed version, in which the constraint is instead $H \subseteq \mathcal{C}'$, where $\mathcal{C}' = \{(i, j, k) \in \{1, \ldots, n\}^3 \mid i > j\}$.

Since $\mathcal{C} \subseteq \mathcal{C}'$, the optimal value of this relaxed version is not smaller than the optimal value of the original one. We can now remark that the relaxed version is a special case of $\mathcal{P}(X)$ where $n = m$, so that we can directly apply Theorem 2, which leads to the following corollary:

Corollary 3. *The number of data accesses required to perform a Cholesky operation on a matrix* $\mathbf{A}$ *of size* $m \times m$*, with memory* M*, is at least:*

$$Q_{Chol}(m, M) \geqslant \frac{1}{3\sqrt{2}} \frac{m^3}{\sqrt{M}}.$$

Proof. The cDAG of the Cholesky factorization contains $|\mathcal{C}| = \frac{m^3}{6}$ arithmetic operations (indeed only update operations). According to Theorem 2, for any X, any subcomputation H of this cDAG which reads at most X elements has size $|H| \leqslant \frac{\sqrt{2}}{3\sqrt{3}} X^{\frac{3}{2}}$.

As previously, we apply Lemma 1 to the case where $X = 3M$, and obtain that the maximal operational intensity of the update operations in the Cholesky factorization is $\rho = \frac{|H|}{3M - M} \leqslant \sqrt{\frac{M}{2}}$. Since a Cholesky factorization needs to perform all update operations, this yields the following bound on the number of data accesses:

$$Q_{\text{Chol}}(m, M) \geqslant \frac{|\mathcal{C}|}{\rho} = \frac{1}{3\sqrt{2}} \frac{m^3}{\sqrt{M}}$$

□

3.4 Communication-Optimal Algorithms

In this section, we detail algorithms which perform the same arithmetic operations as Algorithms 9 and 10 with an ordering that allows to achieve a higher operational intensity, *i.e.* perform fewer I/O operations. We start by presenting an algorithm for the symmetric rank-k update operation which we then use to design an algorithm for the Cholesky factorization.

To ease the presentation of the algorithms, we index the matrices elements starting from 0 instead of 1 as it simplifies the understanding of the modulo operation. Our algorithms rely on existing out-of-core algorithms previously proposed by Béreux in [19]. More specifically we make use of the one-tile implementations of the symmetric rank-k update, OOC_SYRK 14, triangular solve, OOC_TRSM 15, and left-looking variant of the Cholesky factorization 16 as presented in Section 3.1.2. For conciseness, we denote them respectively by OCS, OCT and OCC.

The analysis of communication cost in this section is asymptotic in the following sense: we assume that M remains constant and that the sizes m and n of the matrices grow without bounds.

3.4.1 TBS: Triangle Block SYRK

The proof of Theorem 2 shows that the largest operational intensity in the symmetric rank-k update is achieved when computing the elements of $\mathbf{C}$ in a triangle $T(\ell)$, which is located at the top-left of matrix $\mathbf{C}$. The result of Corollary 2 is tight if all, or at least most, parts of the computation have the same operational intensity. But it is not clear whether it is possible to tile the whole computation space with triangles. It is easy around the diagonal, but what about the elements of the matrix away from the diagonal?

Algorithm 17: Pseudo-code of generic out-of-core SYRK algorithm

---

```
1 Partition C in blocks of size M
2 for each block B do
3     Load the corresponding elements of C in memory
4     for i = 0 to n − 1 do
5         Load the required elements of A[·, i]
6         Update block B with these elements
7     Remove block B from memory
```

Our algorithm uses the generic scheme described in Algorithm 17: store a block of elements of the result matrix in memory, and iteratively load elements from $\mathbf{A}$ to update this block. To maximize memory efficiency, it makes sense that blocks would contain M elements. In the OOC_SYRK algorithm proposed by Béreux, the blocks are squares of $\sqrt{M} \times \sqrt{M}$, which is the optimal shape without data reuse; for example squares are the optimal shape for the non-symmetric matrix multiplication. As mentioned above, in order to match the lower bound for the symmetric rank-k update, we need to have blocks shaped as triangles, up to row and column reordering: such blocks are triangle blocks $\mathrm{TB}(R)$, as defined in Definition 3. Indeed, $\mathrm{TB}(R)$ is the set of indices of the elements of $\mathbf{C}$ that can be updated with elements of $\mathbf{A}$ whose row belong in R.

We prove here that it is actually possible to tile almost all the result matrix $\mathbf{C}$ with triangle blocks, each containing roughly M elements.

3.4.1.1 Partitioning the result matrix

We fix k such that $M \geqslant k + \frac{k(k-1)}{2} = \frac{k(k+1)}{2}$. This ensures that the memory can fit a triangle of side length k from the result matrix $\mathbf{C}$ plus a vector of k elements of $\mathbf{A}$ used for the update. It is the same principle as the OOC_SYRK algorithm from Béreux, making use of narrow-blocks of size k but used to update a block, *i.e.* the set of elements in the memory, that is triangle-shaped rather than a square-shaped. Let us assume for the moment that $m = ck$ for some value c. We will see later that not all values of c are eligible, and we will discuss how to choose an appropriate value. We decompose the result matrix $\mathbf{C}$ in $\frac{k(k-1)}{2}$ square *zones* of size $c \times c$. The rest of the matrix, k triangle-shaped zones on the diagonal, will be considered later. In TBS, a block contains exactly one element from each of these square zones, as depicted in Figure 3.1. For $0 \leqslant i, j < c$, we denote by $B^{i,j}$ the block which contains the element (i, j) of the top-most zone, which is the element $(i + c, j)$ of the matrix $\mathbf{C}$.

Let $R^{i,j}$ be the row indices of block $B^{i,j}$. Since we search for blocks with one element per zone, we can write

$$B^{i,j} = \mathrm{TB}(R^{i,j}), \quad \text{with} \quad R^{i,j} = \left\{u \cdot c + f^{i,j}(u) \,|\, 0 \leqslant u < k\right\} \tag{3.1}$$

where $0 \leqslant f^{i,j}(u) < c$ gives the position of the row of $B^{i,j}$ within the u-th row of zones, as can be observed on the left of Figure 3.2. To ensure that $B^{i,j}$ contains $(i+c, j)$, we just need to have $f^{i,j}(0) = j$ and $f^{i,j}(1) = i$. We can thus specify our triangle blocks with an *indexing family*:

Definition 5 (Indexing family)**.** *A* (c, k)-indexing family *is a family of functions* $f^{i,j}(u)$*, defined for* (i, j) *in* $\{0, \ldots, c-1\}^2$*, with:*

$$f^{i,j} : \{0, \ldots, k-1\} \mapsto \{0, \ldots, c-1\}$$
$$\forall i, j, \quad f^{i,j}(0) = j \quad \textit{and} \quad f^{i,j}(1) = i$$

To enforce the validity of the algorithm, triangle blocks $B^{i,j}$ must not overlap. If two functions $f^{i,j}$ and $f^{i',j'}$ are equal for two different values u and v, the corresponding blocks $B^{i,j}$ and $B^{i',j'}$ have two row indices in common and as can be seen on Figure 3.1, these blocks are not disjoint. We thus need to consider *valid* indexing families:

Definition 6 (Validity)**.** *A* (c, k)*-indexing family* f *is* valid *if*

$$\forall u \neq v, \begin{cases} f^{i,j}(u) & = f^{i',j'}(u) \\ f^{i,j}(v) & = f^{i',j'}(v) \end{cases} \implies i = i' \textit{ and } j = j'.$$

It turns out that this condition is sufficient to ensure no collisions:

Lemma 6. *If* f *is a valid* (c, k)*-indexing family, then the sets* $B^{i,j}$ *defined in Equation 3.1 are pair-wise disjoint.*

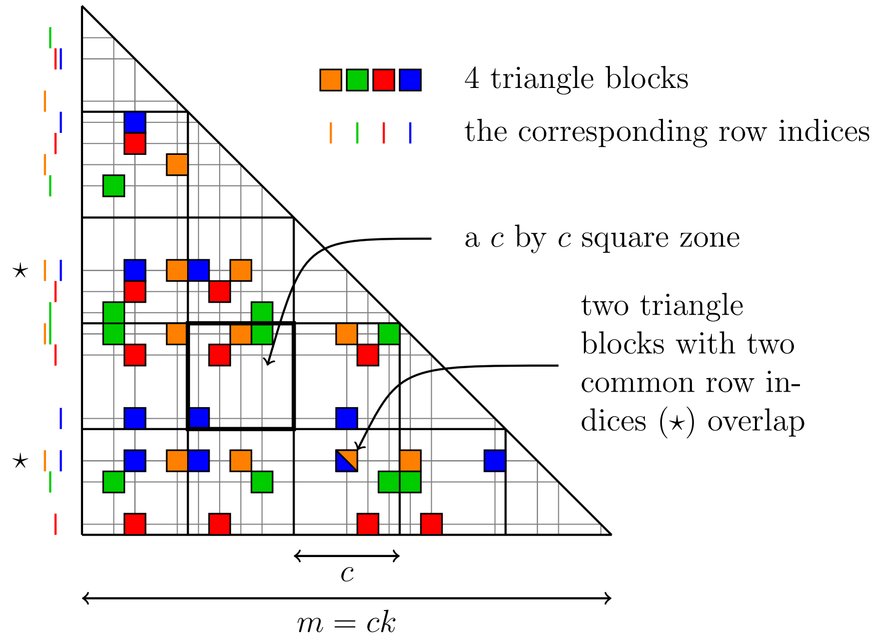


Figure 3.1: Zones and blocks in the TBS algorithm. Each block has one element in each zone.

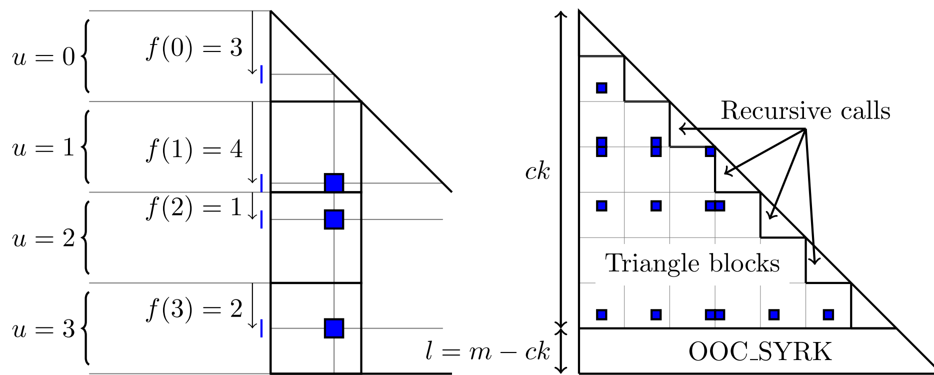


Figure 3.2: *Left:* $f^{i,j}(u)$ gives the position of the row of $B^{i,j}$ within the u-th row of zones. *Right:* which parts of the matrix $\mathbf{C}$ are computed by which method in the TBS algorithm.

Proof. We prove the contrapositive of this statement: if two $B^{i,j}$ sets are not disjoint, then f is not valid. Indeed, let us consider two different pairs (i,j) and (i',j') such that $B^{i,j} \cap B^{i',j'} \neq \emptyset$. There exist (u,v) and (u',v'), with $u \neq v$ and $u' \neq v'$, such that:

$$\begin{aligned} uc + f^{i,j}(u) &= u'c + f^{i',j'}(u') \\ vc + f^{i,j}(v) &= v'c + f^{i',j'}(v') \end{aligned}$$

Since the values of an indexing function are in $\{0,\ldots,c-1\}$, this implies $u = u'$ and $v = v'$.

Thus, there exist $u \neq v$ and i,j,i',j', with $(i,j) \neq (i',j')$, such that $f^{i,j}(u) = f^{i',j}(u)$ and $f^{i,j}(v) = f^{i',j'}(v)$: f is not valid. □

This shows that using a valid indexing family allows to partition the square zones from Figure 3.1 in disjoint triangle blocks. The remaining elements, from the triangular zones close to the diagonal, can be computed by recursive calls to the TBS algorithm. We thus require several valid indexing families for a fixed k and different values of c, since the recursive calls will be made with different value of c. However, we see below that we cannot obtain valid indexing families for all values of c, so we are not yet ready to describe the complete algorithm.

3.4.1.2 Defining a valid indexing family

In this section, we show that it is possible to define a valid indexing family for some values of $c \geqslant k-1$. We do this using the simple modulo operation:

Definition 7. *The* cyclic (c,k)*-indexing family is defined by:*

$$f_c^{i,j}(u) = \begin{cases} j & \text{if } u = 0 \\ i + j(u-1) \mod c & \text{if } u > 0 \end{cases}$$

Lemma 7. *If $c \geqslant k-1$ is coprime with all integers in $\{2,\ldots,k-2\}$, then the cyclic indexing family f_c is valid.*

Proof. Consider any $u,v \in \{0,\ldots,k-1\}$ with $u < v$, and assume that i,j,i',j' in $\{0,\ldots,c-1\}$ are such that $f_c^{i,j}(u) = f_c^{i',j'}(u)$ and $f_c^{i,j}(v) = f_c^{i',j'}(v)$.

We first prove $j = j'$. If $u = 0$, this is direct. Otherwise, we can write:

$$\begin{aligned} &\begin{cases} i + j(u-1) &= i' + j'(u-1) \mod c \\ i + j(v-1) &= i' + j'(v-1) \mod c \end{cases} \\ \Leftrightarrow &\begin{cases} i - i' &= (j'-j)(u-1) \mod c \\ i - i' &= (j'-j)(v-1) \mod c \end{cases} \end{aligned}$$

This implies:

$$
\begin{aligned}
&(j'-j)(u-1) = (j'-j)(v-1) && \mod c \\
\Leftrightarrow\ &(j'-j)(u-v) = 0 && \mod c
\end{aligned}
$$

Since $u < v$, $0 < u, v \leqslant k-1$, we know that $0 < v-u \leqslant k-2$. From our assumption, $v-u$ is coprime with c, so we obtain $j'-j = 0 \mod c$, and thus $j = j'$.

Then, since $i + j(v-1) = i' + j(v-1) \mod c$, we deduce $i = i' \mod c$. Since i, i' are in $\{0, \ldots, c-1\}$, we have $i = i'$. □

We define the constant integer q as the product of all primes no larger than $k-2$: $q = \prod_{p \text{ prime}, p \leqslant k-2} p$. Then c is coprime with all integers in $\{2, \ldots, k-2\}$ if and only if c is coprime with q. Notice that q is constant: it only depends on k, thus on M, but not on m or n.

Now that we know how to build valid indexing families, we are ready to describe the TBS algorithm. However, with the constraints on c imposed by Lemma 7, it is not possible to use triangle blocks on the whole matrix $\mathbf{C}$. Instead, given a matrix size m, we set c to be the largest number coprime with q such that $c \leqslant \frac{m}{k}$. If the obtained c is lower than $k-1$, we can use the simple OOC_SYRK with square blocks. Otherwise, c satisfies the condition of Lemma 7, so we can use triangle blocks to compute the first ck rows of C, and the OOC_SYRK algorithm for the remaining $l = m - ck$ rows, as it is illustrated on the right of Figure 3.2. The resulting algorithm is called TBS and is described in Algorithm 18.

3.4.1.3 Communication cost analysis

Let us first notice that the TBS algorithm loads each entry of $\mathbf{C}$ exactly once, even for the elements computed with OOC_SYRK, so loading these elements has a communication cost of $\frac{m^2}{2}$. In the following, we denote by $\widetilde{Q}_{\text{TBS}}(m, n)$ the communication cost of TBS related to elements of $\mathbf{A}$, for a matrix $\mathbf{A}$ of size $m \times n$.

The definition of c yields $\frac{m}{k} = c + g$, and we need an upper bound on g to estimate the amount of work performed by OOC_SYRK. It is easy to see that for any integer a, $aq+1$ is coprime with q. In particular, $\left\lfloor \frac{m}{kq} \right\rfloor q + 1$ is coprime with q, thus $c \geqslant \left\lfloor \frac{m}{kq} \right\rfloor q + 1$, and $g \leqslant q$. Since q only depends on M and not on m or n, we get $g = \mathcal{O}(1)$. Even though q is a constant, it may be considered very large relative to M. However, the bound $g \leqslant q$ is very pessimistic: sieve methods allow to show that the number of integers coprime with q in any interval $\{(a-1)q, \ldots, aq-1\}$ is exactly $\prod(p-1)$, where p spans the prime numbers below $k-1$ (see Example 1.5 in [38]). In practice, one can expect the value of g to be much lower than q.

We first consider the elements computed with the TBS algorithm, *i.e.* in the first ck rows. There are c^2 triangle blocks, and each triangle block loads kn elements of $\mathbf{A}$. This yields a communication cost $Q_1 = c^2 kn$, and with $c \leqslant \frac{m}{k}$, we obtain $Q_1 \leqslant \frac{m^2 n}{k}$.

Elements computed with OOC_SYRK (in the last $l = gk$ rows) are computed by square $\sqrt{M} \times \sqrt{M}$ blocks, and each block loads $2n\sqrt{M}$ elements from matrix $\mathbf{A}$. Since there are at most gkm such elements, this yields a communication cost $Q_2 \leqslant \frac{gkm}{M} \cdot 2n\sqrt{M} = \mathcal{O}(mn)$.

Algorithm 18: TBS, Triangle Block SYRK algorithm

```
   Input: (A, C): A is m × n and C is m × m symmetric
   Output: (C): such that C += A · A^T
   Assumes: memory of size M = k(k+1)/2
 1 q ← product of all primes in {2, ..., k − 2}
 2 c ← the largest integer coprime with q below m/k
 3 l ← m − ck
 4 if c < k − 1 then                                         c is too small
 5 |   OOC_SYRK (A, C);
 6 else
 7 |   Use OOC_SYRK to compute the last l rows of C
 8 |   for i = 0 to k − 1 do                recursive calls for triangular zones
 9 |   |   R ← {ic, ..., (i + 1)c}
10 |   |   TBS(A_{R,·}, C_{R,R})
11 |   for (i, j) ∈ {0, ..., c − 1}^2 do                      loop over all blocks
12 |   |   R ← {r_u = uc + f_c^{i,j}(u) | 0 ⩽ u < k}                   see Def. 7
13 |   |   Load the elements of C indexed by TB(R)
14 |   |   for i = 0 to n − 1 do                           loop over columns of A
15 |   |   |   Load elements of A indexed by {(r, i) | r ∈ R}
16 |   |   |   for u = 0 to k − 1 do                         loops over elements
17 |   |   |   |   for v = 0 to u − 1 do                            of the block
18 |   |   |   |   |   C_{r_u,r_v} += A_{r_u,i} · A_{r_v,i}
```

Adding the elements covered by the recursive calls, we get:

$$\widetilde{Q}_{\text{TBS}}\left(m,n\right) \leqslant \frac{m^2 n}{k} + k\widetilde{Q}_{\text{TBS}}\left(\frac{m}{k},n\right) + \mathcal{O}(mn)$$

We can iteratively apply this inequality t times, where t is the smallest integer such that $\frac{m}{k^t} < k-1$. We thus have $k^{t-1} < \frac{m}{k}$, and $t = \mathcal{O}(\log(m))$. Then we get:

$$\begin{aligned}
\widetilde{Q}_{\text{TBS}}\left(m,n\right) &\leqslant \sum_{i=1}^{t} \frac{m^2 n}{k^i} + k^t \widetilde{Q}_{\text{OCS}}\left(k,n\right) + t \cdot \mathcal{O}(mn) \\
&\leqslant \sum_{i=1}^{\infty} \frac{m^2 n}{k^i} + m \cdot \frac{k^2 n}{\sqrt{M}} + \mathcal{O}(mn\log(m)) \\
&\leqslant m^2 n\left(\frac{1}{1-\frac{1}{k}} - 1\right) + \mathcal{O}(mn\log(m)) \\
&\leqslant \frac{m^2 n}{k-1} + \mathcal{O}(mn\log(m))
\end{aligned}$$

Remember that k is defined by $M = \frac{k(k+1)}{2}$, so that $k - 1 \simeq \sqrt{2M}$.

In total, with the communications required to load elements of $\mathbf{C}$, we get:

Theorem 3. *The total communication cost $Q_{TBS}(m, n)$ of the TBS algorithm for a matrix $\mathbf{A}$ of size $m \times n$, with a memory of size M, is bounded by:*

$$Q_{TBS}(m, n) \leqslant \frac{1}{\sqrt{2}} \frac{m^2 n}{\sqrt{M}} + \frac{m^2}{2} + \mathcal{O}(mn \log(m))$$

3.4.2 LBC: Large Block Cholesky

The lower bound detailed in Section 3.3.2 is based on the idea that the Cholesky factorization generates at least as many data transfers as the symmetric rank-k update operation. Since the TBS algorithm performs the symmetric rank-k update with the minimum amount of I/O operations, the idea is to use it for the largest possible part of the computation of the Cholesky factorization.

3.4.2.1 Algorithm description

We implement this strategy in the Large Block Cholesky (LBC) algorithm. It is a right-looking, blocked algorithm which performs the Cholesky factorization of any input symmetric positive definite matrix $\mathbf{A}$ making use of OOC_CHOL, OOC_TRSM and TBS algorithms. Note that it would be possible to use a recursive call to LBC instead of OOC_CHOL to perform the factorization, line 3 in Algorithm 19, since LBC performs fewer transfers. However, it turns out that the successive Cholesky factorizations of $\mathbf{A}_{I_0,I_0}$ do not contribute to the higher order term, so we choose to keep OOC_CHOL to simplify the presentation. LBC modifies $\mathbf{A}$ in-place to yield a lower triangular matrix $\mathbf{L}$ as output such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\mathsf{T}$. The steps of the procedure are detailed in Algorithm 19 and described Figure 3.3.

Algorithm 19: LBC, Large Block Cholesky algorithm

Input: ($\mathbf{A}$): $\mathbf{A}$ is $m \times m$ symmetric positive definite
Input: b: block size
Output: ($\mathbf{L}$): such that $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\mathsf{T}$
Assumes: $b|m$
Output: $\mathbf{L}$: $m \times m$ lower triangular matrix s.t. $\mathbf{A} = \mathbf{L} \cdot \mathbf{L}^\mathsf{T}$

```
1 for i = 0 to ⌊m/b⌋ do
2 |   I_0 = {i·b, ..., (i+1)·b}
3 |   A_{I_0,I_0} ← OOC_CHOL (A_{I_0,I_0})
4 |   if (i+1)·b < m then
5 |   |   I_1 = {(i+1)·b, ..., m}
6 |   |   A_{I_1,I_0} ← OOC_TRSM (A_{I_0,I_0}, A_{I_1,I_0})
7 |   |   A_{I_1,I_1} ← TBS (A_{I_1,I_0}, A_{I_1,I_1})
```

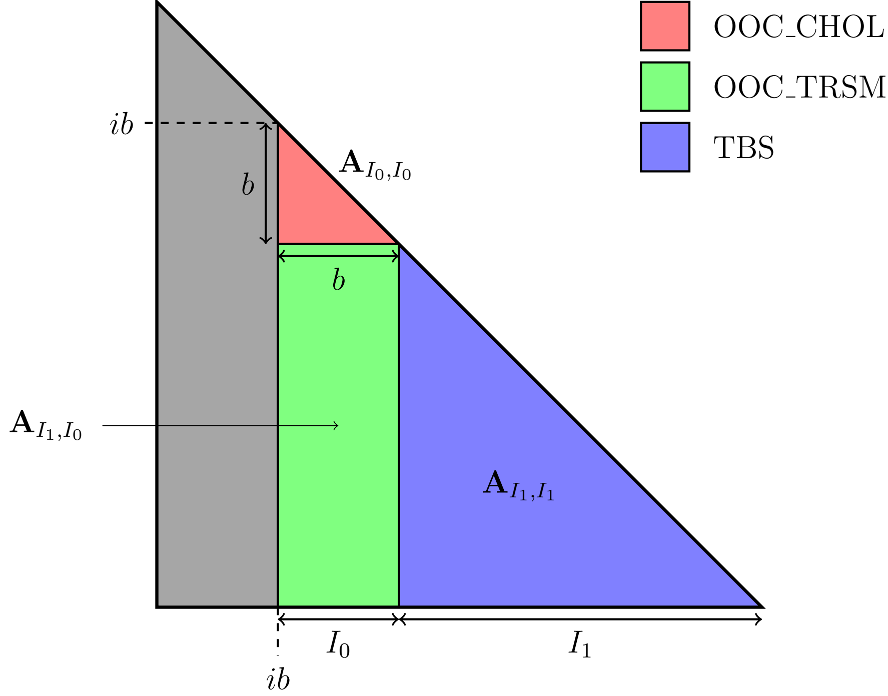


Figure 3.3: Algorithm LBC: updating the three parts of $\mathbf{A}$ at iteration i

LBC is a so-called right-looking variant of the Cholesky factorization. At each iteration, the final values of the two leftmost panels $\mathbf{A}_{I_0,I_0}$ and $\mathbf{A}_{I_1,I_0}$ are computed; $\mathbf{A}_{I_1,I_0}$ is then used to update the right panel $\mathbf{A}_{I_1,I_1}$ whose values are still temporary. By contrast, left-looking variants perform all the update operations of a given value of $\mathbf{A}$ one after the other, allowing to write each element only once.

Right-looking implementations of the Cholesky factorization are known to perform more I/O operations than their left-looking counterparts, because the lower right panel $\mathbf{A}_{I_1,I_1}$ needs to be reloaded at each iteration, so as to be updated using a symmetric rank-k update operation. Nevertheless, this overhead can be rendered negligible. Indeed, the main point of LBC is to use large enough blocks of size $\sqrt{m}$, so that the number of iterations is low ($\sqrt{m}$): then, the communication volume induced by loading $\mathbf{A}_{I_1,I_1}$ remains negligible compared to the one required to update its values.

3.4.2.2 Communication cost analysis

Let us now analyze the total number of I/O operations required by the LBC algorithm on an $m \times m$ matrix $\mathbf{A}$; it is denoted $Q_{\text{LBC}}(m)$. As mentioned above, we get from [19] that $Q_{\text{OCT}}(m,n) = \frac{m^2 n}{\sqrt{M}} + \mathcal{O}(mn)$ and $Q_{\text{OCC}}(m) = \frac{m^3}{3\sqrt{M}} + \mathcal{O}(mn)$. Furthermore, as detailed in Section 3.4.1, we also know that $Q_{\text{TBS}}(m,n) = \frac{1}{\sqrt{2}}\frac{m^2 n}{\sqrt{S}} + \frac{m^2}{2} + \mathcal{O}(mn\log(m))$. Then:

$$
\begin{aligned}
Q_{\mathrm{LBC}}(m) &= \sum_{i=1}^{\frac{m}{b}} Q_{\mathrm{OCC}}(b) + Q_{\mathrm{OCT}}\left(b, \left(\tfrac{m}{b}-i\right)b\right) + Q_{\mathrm{TBS}}\left(\left(\tfrac{m}{b}-i\right)b, b\right) \\
&= \frac{m}{b} Q_{\mathrm{OCC}}(b) + \sum_{i=1}^{\frac{m}{b}} Q_{\mathrm{OCT}}(b, ib) + Q_{\mathrm{TBS}}(ib, b) \\
&= \frac{b^2 m}{3\sqrt{M}} + \mathcal{O}(b^2) + \sum_{i=1}^{\frac{m}{b}} \left(\frac{b^2(ib)}{\sqrt{M}} + \frac{b(ib)^2}{\sqrt{2}\sqrt{M}} + \frac{(ib)^2}{2} + \mathcal{O}(b^2 i \log(ib)) \right)
\end{aligned}
$$

Since $0 < b < m$, $\mathcal{O}(b^2) = \mathcal{O}(m^2)$.
Besides:

$$
\sum_{i=1}^{\frac{m}{b}} \mathcal{O}(b^2 i \log(ib)) \leqslant \sum_{i=1}^{\frac{m}{b}} \mathcal{O}(b^2 \tfrac{m}{b} \log(m)) = \frac{m}{b} \mathcal{O}(mb\log(m)) = \mathcal{O}(m^2 \log(m))
$$

The number of data transfers necessary to perform algorithm LBC is therefore:

$$
\begin{aligned}
Q_{\mathrm{LBC}}(m) &\leqslant \frac{b^2 m}{3\sqrt{M}} + \sum_{i=1}^{\frac{m}{b}} \left(\frac{b^2(ib)}{\sqrt{M}} + \frac{b(ib)^2}{\sqrt{2}\sqrt{M}} + \frac{(ib)^2}{2} \right) + \mathcal{O}(m^2 \log(m)) \\
&\leqslant \frac{b^2 m}{3\sqrt{M}} + \frac{b^3 (\frac{m}{b})^2}{2\sqrt{M}} + \frac{b^3 (\frac{m}{b})^3}{3\sqrt{2}\sqrt{M}} + \frac{b^2 (\frac{m}{b})^3}{6} + \mathcal{O}(m^2 \log(m)) \\
&\leqslant \underbrace{\frac{b^2 m}{3\sqrt{M}}}_{(1)} + \underbrace{\frac{bm^2}{2\sqrt{M}}}_{(2)} + \underbrace{\frac{m^3}{3\sqrt{2}\sqrt{M}}}_{(3)} + \underbrace{\frac{\frac{m^3}{b}}{6}}_{(4)} + \mathcal{O}(m^2 \log(m))
\end{aligned}
$$

As previously discussed the volume of data transfers induced by loading $\mathbf{A}_{I_1,I_1}$ at each step (4) clearly becomes dominant if b is a constant. On the other hand, if the chosen value for b is of order m, the communications required to perform all triangular solve operations (2) becomes dominant. Hence, to ensure that the volume of data transfers used for $\mathbf{A}_{I_1,I_1}$ update (3) is the only dominant term in the formula, we choose to implement LBC using $b = \sqrt{m}$ as block size. Then:

$$
\begin{aligned}
Q_{\mathrm{LBC}}(m) &\leqslant \frac{m^2}{3\sqrt{M}} + \frac{m^2\sqrt{m}}{2\sqrt{M}} + \frac{m^3}{3\sqrt{2}\sqrt{M}} + \frac{m^2\sqrt{m}}{6} + \mathcal{O}(m^2 \log(m)) \\
&= \frac{m^3}{3\sqrt{2}\sqrt{M}} + \mathcal{O}(m^{5/2})
\end{aligned}
$$

Theorem 4. *The total communication cost $Q_{LBC}(m)$ of the Large Block Cholesky algorithm for a matrix $\mathbf{A}$ of size $m \times m$, with a memory of size M, is bounded by:*

$$
Q_{LBC}(m) \leqslant \frac{1}{3\sqrt{2}} \cdot \frac{m^3}{\sqrt{M}} + \mathcal{O}(m^{\frac{5}{2}})
$$

3.5 Conclusion

In this chapter we have explored the complexity in terms of communication of two classical operations, the symmetric rank-k update and Cholesky factorization. We carried out a thorough study of the minimum data movements required to perform those operations in the out-of-core setting using a careful application of a technique by Kwasniewski *et al.* [57] and derive new bounds. In addition, we proposed two out-of-core algorithms, Triangle Block SYRK (TBS) and Large Block Cholesky (LBC), that match the newly derived bounds. By doing so, we provide a definitive answer to the asymptotic communication complexity of both operations: the maximum operational intensity is $\rho \leqslant \frac{1}{\sqrt{2}}\sqrt{M}$ and it is reached by TBS and LBC algorithms, thus achieving explicit optimality regarding communications. Table 3.1a summarizes the expressions in terms of total communication volume of this bound and TBS and LBC algorithms. The main lesson of this work is that the symmetric nature of the symmetric rank-k update and Cholesky factorization can actually be taken advantage of. Hence their operational intensity are intrinsically higher, which implies that they require fewer communications, than those of their non-symmetric counterparts, namely matrix multiplication and LU factorization. The sequential algorithms developed for the out-of-core setting provide insights about how to make use of the symmetry to reach the highest possible operational intensity: the general idea is to perform updates of zones that are triangle shaped in order to benefit from the symmetry of the input. Indeed, this guides the elaboration of the partition of $\mathbf{C}$ into triangle blocks.

The two newly established lower bounds can be extended to the parallel and distributed setting assuming that the total available memory is proportional to the size S of the input matrix, *i.e.* $M = \mathcal{O}(\frac{S}{P})$, as mentioned in Section 2.1.1 under the term of memory scalable assumption. Hence, the quality of such a type of parallel algorithms regarding communications when performing either a symmetric rank-k update or Cholesky factorization can be assessed against this bound. The static data distribution Symmetric Block Cyclic (SBC), presented in Section 2.2.1, is a good candidate to get closer to optimality for both operations. It reduces the communication volume compared to BC and thereby achieves higher performance. As it is a cyclic balanced distribution, it complies with the memory scalable assumption. However, there remains a factor of $\sqrt{2}$ for symmetric rank-k update, respectively $\frac{3}{\sqrt{2}}$ for Cholesky factorization, between the lower bound and the communication volume generated by SBC. Results for the parallel setting are available in Table 3.1b.

In the case of Cholesky factorization, a fraction of the gap between the lower bound and the communication volume achieved by either BC or SBC distributions actually comes from the fact that they are static distributions, and is entirely independent from their characteristics. To illustrate it, let us analyze the value of the operational intensity of SBC over two particular sets of tasks. Let us first consider all the tasks associated with the first iteration of the Cholesky factorization algorithm: it corresponds to $\frac{1}{2}m_b^2$ tasks and, using SBC distribution with an $r \times r$ pattern, requires to transfer rm_b tiles. Asymptotically, it yields the following value of operational intensity: $\rho_{1\text{st}} = \frac{m_b^2}{2rm_b} \underset{P\to+\infty}{\sim} \frac{1}{2\sqrt{2}}\frac{m_b}{\sqrt{P}}$. Since the data stored on each node is $M = \frac{m_b^2}{2P}$, it gives: $\rho_{1\text{st}} = \frac{1}{2}\sqrt{M}$. However, as the factorization progresses, the trailing sub-matrix over which the computations are performed shrinks.

Hence, part of the data stored by each node is no longer used for the computations, and thus the operational intensity decreases. Indeed, when considering the entire operation, the total number of tasks is $\frac{1}{6}m_b^3$ and the communication volume generated by SBC is $\frac{1}{\sqrt{2}}m_b^2P$. This yields a value of the operational intensity: $\rho_{\text{global}} = \frac{1}{3}\sqrt{M}$. We can observe that the intrinsic operational intensity of the SBC strategy, which would apply if the operation was executed in the out-of-core setting, is $\frac{1}{2}\sqrt{M}$ but is reduced by a factor of $\frac{2}{3}$ when considering the parallel and distributed execution of the operation. The same reasoning applies to any static distribution and illustrates the inevitable loss of efficiency of such type of solution for operations performed over a shrinking domain.

The optimality result in the case of the out-of-core model presented in this chapter along with the SBC distribution for the parallel and distributed setting establish a new standpoint for the problem of communication minimization for the symmetric rank-k update and Cholesky factorization in the dense case. It brings a better understanding of the mechanisms underlying data reuse in symmetric operations and provides a solid foundation for subsequent development of theoretical bounds and communication avoiding algorithms. Several promising research directions are therefore open from here on.

- In the context of out-of-core execution, the lower bound on the communication volume may be extended to other linear algebra operations which feature symmetric input or use the same input multiple times. Algorithms achieving explicit optimality based on TBS may be elaborated.

- The techniques developed in the case of out-of-core execution may be used as building principle for communication efficient algorithms in a parallel and distributed setting. In particular, triangle blocks as defined in TBS algorithm may be used as building elements to design static data distribution that generate fewer communications than SBC.

- In the parallel and distributed context, for the symmetric rank-k update and Cholesky factorization, the key characteristics that define the quality of a static distribution regarding the communication volume it generates is the number of different nodes in the union of each row and its associated column, *i.e.* with the same index. It is the number of different nodes on each row and each column independently, for the non-symmetric version of those operations, namely the matrix multiplication and LU factorization. Hence, another direction of research may be to try designing distributions that feature as few different nodes as possible in the union of row and colum, respectively row and column independently for non-symmetric operations. This criterion may then be a relevant objective function to search for communication avoiding data distributions whereas allowing a large flexibility regarding all other characteristics.

- The data distributions elaborated to limit communications in the dense case may be extended to sparse or compressed matrices. In those cases, the workloads associated with the tasks are heterogeneous therefore requiring to take into account load balancing along with communication reduction in the design of static distributions.

The techniques used to develop TBS in the out-of-core context and SBC in the parallel and distributed setting may be adapted to tackle such type of multi-objectives problems.

In the remaining of this work, we focus on two potential ways to extend the results and techniques obtained so far. Chapter 4 presents strategies to design data distributions for the matrix multiplication and LU factorization in the case of compressed matrices. Using the criterion based on the number of different nodes per row and column, we develop data distributions that trade additional communications to improve the load balancing compared to plain BC distribution. Chapter 5 is dedicated to the extension of the BC distribution, on one hand, for matrix multiplication and LU factorization, and of SBC, on the hand, for symmetric rank-k update and Cholesky factorization. The objective is to develop data distributions that generate as few or even fewer communications than BC and SBC but can make an efficient usage of any number of nodes.

	Lower Bound ($Q \geqslant$)		**Algorithms** ($Q \leqslant$)	
	State-of-the-art: from `IOLB` [62]	Contribution	State-of-the-art: narrow-block-based algorithms [19]	Contribution: TBS, LBC algorithms
SYRK	$\frac{1}{2}\frac{m^2n}{\sqrt{M}}$	$\frac{1}{\sqrt{2}}\frac{m^2n}{\sqrt{M}}$	$\frac{m^2n}{\sqrt{M}} + \mathcal{O}(mn)$	$\frac{1}{\sqrt{2}}\frac{m^2n}{\sqrt{M}} + \frac{m^2}{2} + \mathcal{O}(mn\log(m))$
Cholesky	$\frac{1}{6}\frac{m^3}{\sqrt{M}}$	$\frac{1}{3\sqrt{2}}\frac{m^3}{\sqrt{M}}$	$\frac{1}{3}\frac{m^3}{\sqrt{M}} + \mathcal{O}(m^2)$	$\frac{1}{3\sqrt{2}}\frac{m^3}{\sqrt{M}} + \mathcal{O}(m^{\frac{5}{2}})$

(a) Out-of-core setting; narrow-block-based, TBS and LBC algorithms

	Lower Bound ($Q \geqslant$)		**Algorithms** ($Q \leqslant$)	
	State-of-the-art: from `IOLB` [62]	Contribution	State-of-the-art: Block Cyclic	Contribution: Symmetric Block Cyclic
SYRK	$\frac{1}{\sqrt{2}}m_bn_b\sqrt{P}$	$m_bn_b\sqrt{P}$	$2m_bn_b\sqrt{P}$	$\sqrt{2}m_bn_b\sqrt{P}$
Cholesky	$\frac{1}{3\sqrt{2}}m_b^2\sqrt{P}$	$\frac{1}{3}m_b^2\sqrt{P}$	$m_b^2\sqrt{P} + \mathcal{O}(m_b)$	$\frac{1}{\sqrt{2}}m_b^2\sqrt{P} + \mathcal{O}(m_b)$

(b) Parallel and distributed setting using P nodes under the memory scalable assumption ($M = \mathcal{O}(\frac{m_b^2}{2P})$); 2D SBC and BC data distributions

Table 3.1: Summary of the total communication volume for the symmetric rank-k update and Cholesky factorization: theoretical lower bound and value achieved by the algorithms

Chapter 4

Communication Aware Allocation Strategies in Block Low Rank Context

4.1 Introduction

The conclusions of Chapters 2 and 3 provide a thorough insight of the communication minimization problem, especially for symmetric operations. This defines a clear view of the quality of existing algorithms regarding communication reduction in the dense case for both out-of-core and parallel and distributed settings. It is therefore natural to try extending these developed techniques to other configurations. In particular, sparse and compressed matrices are classical solutions to reduce the memory footprint of linear algebra operations and limit the impact of communications in parallel and distributed settings. Those configurations are of high interest as operations using such matrices are commonplace in practical applications [8, 63, 65]. Hence, it is crucial to apply the communication reduction techniques developed in the dense case to the design of data distributions for such types of matrices. Operations applied to sparse matrices are nevertheless very different from dense versions: since large parts of input matrices are zeros, the set of computations of a given operation completely changes from its dense version as most arithmetic operations can be omitted. The techniques elaborated in the dense case therefore cannot be easily adapted to this configuration. The operations applied to compressed matrices require the same set of tasks than in the dense case, though the performance of each varies according to the local compression level of the data to which it is applied. We thus aim at designing algorithms to reduce communications for operations on compressed matrices, based on extensions of the techniques elaborated in the dense case, in particular in parallel and distributed settings.

The compression of data however introduces additional complexity to the communication minimization problem since: (i) communication costs are then heterogeneous and (ii) the performance of a kernel is expected to vary according to the level of compression of the data handled. Then a new problem arises because minimizing the total communication volume can alter the load balancing between nodes and thus degrade the overall perfor-

mance. Therefore optimizing the performance of the distributed execution of an operation in this context implies finding a trade-off between the two objectives, load balancing and communication reduction, which is a difficult problem.

In this chapter we consider two linear algebra operations, the matrix multiplication and LU factorization, performed in a parallel and distributed memory setting using P identical nodes. They can be seen as the non-symmetric versions of the symmetric rank-k update and Cholesky factorization already studied. As in the previous Chapter 2, we assume that input and output matrices are tiled and that the execution of the operations is tiled-based, complying with the owner computes rule, *i.e.* we investigate static data distributions. Besides, we consider that input data matrices are compressed using the Block Low Rank (BLR) format on each tile; heterogeneous compression level between tiles having an impact on the execution efficiency of tasks. In this context, our general goal is to design 2D data distributions that take into account the heterogeneity introduced by BLR compression in order to address the communication versus load balancing trade-off. These distributions should be able to:

- balance the load between the different nodes, knowing the workload associated with each tile and related to the rank of its low-rank decomposition in the BLR format.
- minimize communication volume between nodes.

The quality of a data distribution is assessed according to its overall performance, measured via the total running time of the operation. When talking about workload we thus actually consider the cumulated execution time of tasks according to the performance model detailed in Section 4.3.3.

In the following, we define a general strategy to elaborate data distributions that trade additional communications for improved workload balance among nodes. We propose two heuristics based on this strategy that provide data distributions that ensure load balancing and meet some constraints on the maximum number of different nodes on each row and column:

- Block Cyclic Extended (BCE) is an extension of Block Cyclic (BC) allowing more flexibility by using a repeated pattern in which each node can appear several times;
- Random Subsets (RSB) is a heuristic that directly provides a complete distribution for the whole matrix using subsets of nodes randomly built in a previous step.

For the two considered operations, we compare the performance of the distributions provided by BCE and RSB to BC. Experimental results show that both BCE and RSB perform significantly better than BC in all cases. RSB consistently achieves better load balancing than BCE and BC. However for LU factorization which features numerous task dependencies, the best gain in terms of total running time goes either to BCE or RSB depending on the configuration of the test case, in terms of problem size compared to the number of nodes, and in terms of constraints on the maximum acceptable number of communications.

The rest of the chapter is organized as follows: in Section 4.2, we review the related work regarding tile-based data compression methods, heterogeneous partitioning problems arising in linear algebra and we provide some examples of practical solutions used to deal with compressed matrices from an applied perspective. Then we define some notations and detail the linear algebra operations we are considering in Section 4.3. We also formalize the BLR format, present a simple performance model for tasks on compressed tiles and detail the metrics used to evaluate the quality of data distributions. Finally we describe the load balancing versus communications trade-off, derive an optimization problem for obtaining efficient data distributions and point out connections to existing bin packing variants studied in the literature. In Section 4.4, we detail BCE and RSB heuristics to solve the constrained allocation problem. The experimental framework and a comparison of each distribution scheme relative performance is described in Section 4.5. Finally, a more in-depth analysis of the behavior of BCE is provided in Section 4.6 along with a discussion about practical issues of using such data distribution heuristics.

4.2 Related Work

4.2.1 Data Compression

Reducing the size of data handled when executing a distributed algorithm is a way to limit the cost of communications. In the context of linear algebra operations, several approaches have been proposed to store data using the opportunity of representing the matrices in a compressed form. For example, the $\mathcal{H}$-matrix representation introduced in [48] is now widely used to reduce storage costs and also execution time significantly, when the matrix can actually be compressed. For instance, it is known [44] that in the context of LU factorization, the number of floating point operations can be reduced from $\Theta(\frac{2}{3}m^3)$ to $\Theta(mk^2\log^2(m))$, where k is a parameter that represents the compression that can be achieved. Similar results have been proved in [7] for semi-separable matrices. Recently, these techniques have also been extended to sparse matrices, for example in [37, 63].

Complementary approaches have been proposed to avoid issues related to complex and irregular data accesses induced by $\mathcal{H}$-matrix representation. In Block Low Rank (BLR) decomposition, matrices are regularly tiled and, when compression is possible, the data of each tile is approximated using a low-rank decomposition. This technique is detailed in Section 4.3.2. BLR decomposition has been advocated for instance for multi-frontal methods in [8, 60] or finite-element matrices in [9]. Another approach is *lattice* $\mathcal{H}$-matrices presented in [51,70]. The work in [28] follows the same line of research. In all cases, the idea is to increase simplicity and to keep a high compression ratio, even at the price of a slightly higher flop complexity. Indeed, keeping the regular tiled structure makes it possible to use runtime schedulers, that have proven their efficiency in making use of heterogeneous resources (CPU cores, GPUs) at the level of a single node by applying dynamic scheduling strategies [5], and achieving high performance in a parallel and distributed context via task-based execution of the operations [4].

4.2.2 Heterogeneous Allocation Problems in Linear Algebra

The problem of designing alternative placement strategies has been considered in the case of heterogeneous resources. In this framework, in the most general situation, P processors are given with relative speeds $v_1, \ldots, v_P$ and the goal is to allocate tiles to these different resources in a way that balances the load (each node receives a number of tiles proportional to its speed) and minimizes the communication volume.

In the case of matrix multiplication, the original matrix is partitioned into P rectangles, whose areas are proportional to the relative speeds of processors, and such that the sum of all rectangle perimeters is minimal. Indeed, using Cannon-type algorithms for multiplying matrices, the volume of data exchanged by a node at each step is proportional to the perimeter of its assigned area in the matrix . This approach was initially proposed in [55] and [16]. It led to developments in two orthogonal directions, one concerned with solving the optimal problem for a small number of resources [18,34,59] and the other concerned with the design of approximation algorithms [17,39,61]. A survey on the results obtained with these different approaches has been proposed in [15].

A closely related problem is the one of allocating a sparse matrix to a set of homogeneous processors in order to balance the load between processors while keeping a regular allocation structure. It has been introduced by Manne *et al.* under the name of *rectilinear partitioning* or *generalized block distribution* in [45] and is still the object of an active literature, such as [71]. In rectilinear partitioning, the problem consist in partitioning the rows and the columns into groups, the intersection between a group of columns and a group of rows being in turn assigned to a given processor. In this context, the main goal is to build groups such as the load, *i.e.* the number of non-zero elements allocated to a processor, is well balanced.

Most of the work related to partitioning algorithms however focuses on the operation of matrix multiplication with few exceptions that are interested in the LU factorization for sparse matrices [63,65]. Besides, because of the difficulty of the problem when dealing with heterogeneous resources and/or workloads, the developed solutions generally rely on approximation algorithms that perform worse in the homogeneous case than 2D BC distributions.

4.2.3 Applied Perspective

Compression techniques, such as BLR, are actually used in several scientific applications where the reduction of very large input data is necessary to allow reasonable execution time. In [12] for example, the *Boundary Elements Method* (BEM) is applied to the surface of an aircraft to model the propagation of electro-magnetic waves. It yields a symmetric linear system, equivalent to a matrix, whose values represent the interactions between any pair of elements. Since far-field interaction are weak, the resulting matrix features off-diagonal regions which can be compressed using BLR format. The authors then propose an implementation for a distributed memory platform to perform the Cholesky factorization of the matrix using plain BC method for the distribution of tiles to the computing nodes.

In the field of weather forecast, maximum-likelihood estimation method also requires the solution of large scale linear systems, as illustrated in [1] The authors tackle the problem using Cholesky decomposition of a matrix whose tiles are compressed using BLR. A similar technique is applied to LU factorization in [6] to study the propagation of sound waves on the surface of 3D objects. Both implementations make use of a hybrid data distribution method based on BC and detailed in [27]: since full rank diagonal tiles are not compressed, and therefore associated with longer execution time, they are treated specifically to ensure a better load balancing between nodes in a distributed setting. Diagonal tiles are distributed in a round robin fashion while a plain BC distribution scheme is applied for off diagonal ones.

4.3 Problem Description and Modeling

4.3.1 Communication Scheme of the Operations

The tiled version of the operations considered in this chapter, the matrix multiplication and LU factorization, are presented in Section 1.1.2.4 and reminded here in Algorithms 20 and 21. To analyze their communication scheme, we follow the same methodology as detailed in Section 2.2.2 for the Cholesky factorization: considering the data dependencies implied by each type of task in the algorithm, we count the number of different nodes to which each tile need to be sent. This allows to determine which type of tasks is responsible for the dominant part of the communication volume.

4.3.1.1 Matrix Multiplication

Algorithm 20: Tiled matrix multiplication algorithm (GEMM)

Input: ($\mathbf{C}$, $\mathbf{A}$, $\mathbf{B}$): $\mathbf{C}$ is $m_b \times n_b$, $\mathbf{A}$ is $m_b \times k_b$, $\mathbf{B}$ is $k_b \times n_b$
Output: ($\mathbf{C}$): such that $\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$

1 **for** $k = 1 \dots k_b$ **do**
2   **for** $i = 1 \dots m_b$ **do**
3   **for** $j = 1 \dots n_b$ **do**
4   GEMM(i, j, k): GEMM$(\mathbf{C}(i, j), \mathbf{A}(i, k), \mathbf{B}(k, j))$

Matrix multiplication algorithm computes the matrix product $\mathbf{C} = \mathbf{C} + \mathbf{A} \cdot \mathbf{B}$. It only makes use of GEMM type of task and is completely parallel since there are only dependencies between two successive GEMM applied to the same tile, *i.e.* for each $(i, j) \in \{1, \dots, m_b\} \times \{1, \dots, n_b\}$ and $k \in \{2, \dots, k_b\}$, the task GEMM$(i, j, k-1)$ is a parent task for GEMM(i, j, k), because they modify the same tile $\mathbf{C}(i, j)$. Data dependencies can be summarized as follows:

1. each tile $\mathbf{A}(i, k)$, for $(i, k) \in \{1, \dots, m_b\} \times \{1, \dots, k_b\}$ is used as input of GEMM(i, j, k), for $j \in \{1, \dots, n_b\}$, to update tiles $\mathbf{C}(i, j)$;

2. each tile $\mathbf{B}(k, j)$, for $(k, j) \in \{1, \ldots, k_b\} \times \{1, \ldots, n_b\}$ is used as input of $\mathrm{GEMM}(i, j, k)$, for $i \in \{1, \ldots, m_b\}$, to update tiles $\mathbf{C}(i, j)$.

Dependencies (1) imply that each tile $\mathbf{A}(i, k)$ is sent to all nodes owning tiles of $\mathbf{C}$ on the same row i. Similarly, dependencies (2) imply that each tile $\mathbf{B}(k, j)$ is sent to all nodes owning tiles of $\mathbf{C}$ on the same column j.

4.3.1.2 LU Factorization

Algorithm 21: Tiled LU factorization algorithm (LU)

```
   Input: (A): A is m_b × m_b
   Output: (L, U): such that A = L · U
 1 for k = 1 ... m_b do
 2     GETRF(k): GETRF(A(k,k))
 3     for i = k+1 ... m_b do
 4         TRSM(k,i): TRSM(A(k,i), A(k,k))
 5     for i = k+1 ... m_b do
 6         TRSM(i,k): TRSM(A(i,k), A(k,k))
 7         for j = k+1 ... m_b do
 8             GEMM(i,j,k): GEMM(A(i,j), A(i,k), A(k,j))
 9 L ← lower-triangular part of A
10 U ← upper-triangular part of A
```

For a given square matrix $\mathbf{A}$, the LU factorization computes a lower-triangular matrix $\mathbf{L}$ and an upper-triangular matrix $\mathbf{U}$ such that: $\mathbf{A} = \mathbf{L} \cdot \mathbf{U}$. The description of the operation according to Algorithm 21 corresponds to the right looking variant as detailed in [42]. Besides, we consider the version of LU factorization without pivoting. This operation makes use of three types of task, namely GETRF, TRSM and GEMM. Data dependencies can be described at each iteration step, line 1 of Algorithm 21, as follows:

1. each tile $\mathbf{A}(k, k)$, for $k\{1, \ldots, m_b\}$ is used as input of $\mathrm{TRSM}(i, k)$ and $\mathrm{TRSM}(k, i)$, for $i \in \{k+1, \ldots, m_b\}$, respectively to update tiles $\mathbf{A}(i, k)$ and $\mathbf{A}(k, i)$;

2. each tile $\mathbf{A}(i, k)$, for $i \in \{1, \ldots, m_b\}$ is used as input of $\mathrm{GEMM}(i, j, k)$, for $j \in \{k+1, \ldots, m_b\}$, to update tiles $\mathbf{A}(i, j)$;

3. similarly each tile $\mathbf{A}(k, j)$, for $j \in \{1, \ldots, m_b\}$ is used as input of $\mathrm{GEMM}(i, j, k)$, for $i \in \{k+1, \ldots, m_b\}$, to update tiles $\mathbf{A}(i, j)$.

Dependencies (1) imply that each tile $\mathbf{A}(k, k)$ is sent to all nodes owning tiles of $\mathbf{A}$ on the row k and column k. Dependencies (2) imply that each tile $\mathbf{A}(i, k)$ is sent to all nodes owning tiles of $\mathbf{A}$ on the same row i. Similarly, dependencies (3) imply that each tile $\mathbf{A}(k, j)$ is sent to all nodes owning tiles of $\mathbf{A}$ on the same column j.

4.3.2 Block Low Rank Compression

The BLR format is based on a low rank approximation of individual tiles. According to the considered application, some tiles of the input matrix, especially off-diagonal ones, generally correspond to far field interactions that can be approximated or even omitted without degrading too much the linear system representation of the underlying physical model. Such tiles, if considered compressible, can be approximated using a truncated version of their *Singular Value Decomposition* (SVD). More precisely, let us assume that tile $\mathbf{A}(i,j)$, $(i,j) \in \{1,\ldots,m_b\}^2$ of matrix $\mathbf{A}$ is considered compressible, then there exists $r_{i,j} \in \{1,\ldots,b\}$ and two rectangular matrices $\mathbf{U}_{i,j}$, $\mathbf{V}_{i,j}$ of size $b \times r_{i,j}$ such that $\mathbf{A}(i,j)$ can be approximated by the outer product of $\mathbf{U}_{i,j}$ and $\mathbf{V}_{i,j}$:

$$\mathbf{A}(i,j) \approx \mathbf{U}_{i,j} \cdot \mathbf{V}_{i,j}^{\mathsf{T}}$$

The value of the approximation rank $r_{i,j}$ is tile dependent and is derived from the accuracy required by the end user. It is selected according to the application. Generally compression is applied to all off-diagonal tiles while diagonal ones are considered full rank and not compressed at all.

The selection of a threshold for the rank of each compressed tile according to the expected accuracy of the end result is outside of the scope of our study. Extensive discussion about this question can however be found in the literature. In the following, we assume that the input matrix is provided in its compressed form and the rank for each tile is known.

4.3.3 Performance Model and Evaluation Metrics

For each operation we are looking for data distributions $\mathbf{D}$ to P identical nodes. For the LU factorization the distribution $\mathbf{D}$ applies to the input matrix $\mathbf{A}$ which is factorized. For matrix multiplication, $\mathbf{C} \mathrel{+}= \mathbf{A} \cdot \mathbf{B}$, it applies to $\mathbf{C}$ while the distribution of $\mathbf{A}$ and $\mathbf{B}$ may be different.

Let us introduce some additional notations:

- For each operation, we define $\mathcal{I}$ the set of valid tile positions of the output matrix:

$$\mathcal{I} = \begin{cases} \{1,\ldots,m_b\} \times \{1,\ldots,n_b\} & \text{for matrix multiplication} \\ \{1,\ldots,m_b\}^2 & \text{for LU factorization} \end{cases}$$

- For $(i,j) \in \mathcal{I}$, the set of valid iteration index for the tile in position (i,j) is denoted $\mathcal{K}_{i,j}$. It corresponds to the index of the outermost loop in the algorithm describing the operation.

$$\mathcal{K}_{i,j} = \begin{cases} \{1,\ldots,k_b\} & \text{for matrix multiplication} \\ \{1,\ldots,\min(i,j)\} & \text{for LU factorization} \end{cases}$$

- For $(i,j) \in \mathcal{I}$ and $k \in \mathcal{K}_{i,j}$, the task modifying the tile in position (i,j) at iteration k is denoted $T_{i,j}^{[k]}$. In the following, we use the expression $T_{i,j}^{[\cdot]} = \{T_{i,j}^{[k]} \ : \ k \in \mathcal{K}_{i,j}\}$.

- The type of operation, named after `LAPACK` terminology, associated with a task T is denoted $\tau(T)$. We define the simplified expression: $\tau(T_{i,j}^{[k]}) = \tau_{i,j}^{[k]}$.
- Its execution time is denoted $w(T)$, for "working" time. Similarly, we define the simplified expression: $w(T_{i,j}^{[k]}) = w_{i,j}^{[k]}$.

In the BLR format, a compressed tile $\mathbf{A}(i,j)$ is approximated by the outer product of two matrices $\mathbf{U}_{i,j}$ and $\mathbf{V}_{i,j}$ of size $b \times r_{i,j}$ that can be considered tall and skinny, *i.e.* $b \gg r_{i,j}$. Operations performed using such approximation require fewer arithmetic operations than if performed on uncompressed tiles and therefore have different execution times. In order to reflect the variability of the execution time according to the compression, we use a simple performance model based on the reasonable assumption: the more compressed a tile is, the faster the execution of a task applied on it. Formally we assume that the execution time of an operation performed on a compressed tile is proportional to its compression ratio:

$$\forall (i,j) \in \mathcal{I},\ \forall k \in \mathcal{K}_{i,j} : \quad w_{i,j}^{[k]} = \frac{r_{i,j}}{b} C_{\tau_{i,j}^{[k]}}$$

where C_X represents the estimated execution time of the type of operation X when performed on an uncompressed tile.

Because operations are performed according to the owner computes rule, it is useful to exhibit the cumulated execution time associated with all the tasks applied to a single tile, since they all are performed by the same node. Therefore we define the matrix $\mathbf{W}$ of such cumulated execution time:

$$\forall (i,j) \in \mathcal{I} \ : \ \mathbf{W}(i,j) = \sum_{k \in \mathcal{K}_{i,j}} w_{i,j}^{[k]} \tag{4.1}$$

Note that we do not consider the phenomenon of "rank-filling", *i.e.* the fact that tiles become less and less compressed as tasks update their values, as it is very difficult to predict and model. In our model, the rank of each tile remains unchanged over the entire operation. In practice, to avoid a prohibitive increase in memory consumption during the execution, this phenomenon is often dealt with by *ad hoc* re-compression strategies, using a fixed number of updates or a maximum rank as threshold.

We consider the total running time to complete an operation using a given data distribution as the objective function to optimize. However, finding a closed form formula to estimate the running time associated with a distribution is impossible in practice because it would require taking into account the evolution of the ranks of the tiles, the communication costs including the contention on the communication medium and estimating the overlap with computations. Hence to evaluate the quality of a distribution for a given operation we use three different metrics that act as surrogate values for the expected achievable performance in terms of total running time, with increasing predictive capacity.

1. $\mathfrak{B}$ is the **load balancing** metric. It is a proxy to the total running time assuming that the nodes are never idle and that communications can always be overlapped

with computations. For a given data distribution $\mathbf{D}$, $\mathfrak{B}(\mathbf{D})$ corresponds to the total execution time associated with any node:

$$\mathfrak{B}(\mathbf{D}) = \max_{p \in \{1,\dots,P\}} \left(\sum_{\substack{(i,j)\in\mathcal{I} \\ \mathbf{D}(i,j)=p}} \mathbf{W}(i,j) \right)$$

2. $\mathfrak{D}$ is the **no communication running time**. It corresponds to the simulated running time assuming that communications are instantaneous or can always be overlapped with computations. However it takes into account the idle time that can be induced by task dependencies. $\mathfrak{D}(\mathbf{D})$ value is calculated using a simple discrete events based algorithm where each node is fed by a queue of ready tasks, ordered according to their priority. Task priorities are computed as the longest path in the cDAG from this task to the end task, each edge being weighted according to the estimated execution time of the origin task. The chosen implementation of this algorithm allows preemption of incomplete tasks, so as to model a real task-based scheduler behavior. Indeed, since a real task applied to a tile consists in many arithmetic operations, stopping its execution at the end of one of these arithmetic operations can be assimilated to the preemption of the task.

3. $\mathfrak{C}$ is the **simulated running time**. It is the simulated total running time using the communication model provided by `SimGrid` [29]. It takes into account the communication time that cannot be overlapped by computations and the idle time induced by task dependencies. This simulation is based on an actual implementation in the `Chameleon` library, using the `SimGrid` based simulation backend of `StarPU`, which allows to obtain realistic results.

4.3.4 General Problem Modeling: Load Balancing versus Communication Trade-Off

Since $\mathfrak{D}$ and $\mathfrak{C}$ metrics require simulating the complete execution of an operation to be evaluated they are not suitable as guiding criterion to search for data distributions. To guide the search for efficient distributions we therefore make use of the load balancing metric $\mathfrak{B}$ which is straightforward to evaluate. The values $\mathfrak{D}(\mathbf{D})$ and $\mathfrak{C}(\mathbf{D})$ are nevertheless used to evaluate the quality of a distribution $\mathbf{D}$ for a given operation because they give a more accurate estimation of the real total running time which is the objective function.

Section 4.3.1 details the communication scheme associated with the parallel and distributed execution of matrix multiplication and LU factorization. For both operations, the communications required to transfer the input of GEMM tasks represent the dominant part of the total communication volume. Those types of communications imply that each tile is sent to all nodes owning a tile on the same row and on the same column. We can observe that this communication scheme corresponds to the non-symmetric version of the Cholesky factorization, that is detailed in Section 2.2.2. The key difference is that, in the symmetric case, the communication volume induced by a distribution is controlled

by the number of different nodes on each union of a row and its corresponding column while, in the non-symmetric case, it depends on the number of different nodes on rows and columns independently.

Therefore, we try to design data distribution that feature few different nodes on each row and each column. More specifically a distribution $\mathbf{D}$ is considered valid if no more than $p_{\max}$ different nodes are allocated on any row or any column. The formal optimization problem we seek to solve can be stated as:
given a weight matrix $\mathbf{W}$, a number of nodes P and an integer $p_{\max} \leqslant P$

1. find a data distribution $\mathbf{D}$ of the output matrix $\mathbf{A}$ for LU factorization, $\mathbf{C}$ for matrix multiplication, such that:

$$\begin{cases} (1) \quad \forall i \quad |\{\mathbf{D}(i,\ell) \ : \ (i,\ell) \in \mathcal{I}\}| \leqslant p_{\max} \\ (2) \quad \forall j \quad |\{\mathbf{D}(\ell,j) \ : \ (\ell,j) \in \mathcal{I}\}| \leqslant p_{\max} \end{cases} \tag{4.2}$$

2. which minimizes the load balancing metric $\mathfrak{B}(\mathbf{D})$

Given the communication scheme of matrix multiplication and LU factorization, and according to the discussion in Section 1.1.2.3, the BC distribution using a square pattern $\sqrt{P} \times \sqrt{P}$ is the optimal solution to minimize the communication volume for those operations. It can thus be considered as a limit case for the load balancing versus communication trade-off formalized above. Therefore, we carry out the analysis of the problem in the heterogeneous case by defining the constraints (1) and (2) of System 4.2 as follows: let $\alpha \in [1; +\infty[$, then: $p_{\max} = \left\lceil \alpha \lceil \sqrt{P} \rceil \right\rceil$. α is parameter controlling the tightness of communication restriction; $\alpha = 1$ corresponds to the square BC distribution. In the following, we assess this trade-off using several values for $\alpha = 2$ or 3.

4.3.5 Bin Packing Problem Variant

The heterogeneity of task execution times when applied to low-rank compressed tiles makes the load balancing versus communications trade-off particularly difficult to solve. Indeed, the optimization problem as defined in the previous paragraph (4.3.4) can be considered as a variant of the *bin packing* problem, where values of the matrix $\mathbf{W}$ are weighted items to be associated with the different nodes, considered as bins.

The bin packing problem is known to be NP-hard, even in its most plain form. Including additional constraints (1) and (2) of System 4.2 to limit the communication volume leads to an even harder version of the problem. Those constraints actually reduce the set of the feasible packings. Thus the problem we are considering can be seen as a generalization of some related literature on bin packing: bin packing with class constraints [36, 54] in which there is a limit to the number of classes allowed in a bin, and bin packing with minimum color fragmentation [20] in which the number of bins containing a given class is limited. Since we consider constraints on both rows and columns, our case is a two-dimensional version of these problems. However the techniques developed in those papers can not be generalized to our context.

The same modeling is directly applicable in the dense case. For pattern-based distributions, as considered in Chapter 2 and 5, the problem is simpler because balancing the load is equivalent to balancing the number of positions in the pattern associated with each node, as discussed in Section 1.1.2.3. However, when designing patterns where each node may appear several times, we face a similar constrained bin packing problem with items of identical weights. In Chapter 5 where such a strategy is proposed, the problem is actually solved using a heuristic strategy, as here.

4.4 Data Distribution Schemes

In this section, we present the strategies developed to address the load balancing versus communication trade-off, formalized as an optimization problem and detailed in Section 4.3.4. For a given number of nodes P, a constraint parameter α and an input matrix of cumulated workload $\mathbf{W}$, the methods provide a data distribution $\mathbf{D}$ of the output matrix which is feasible, in the sense that it respects the constraints of System 4.2, and which is designed to minimize $\mathfrak{B}(\mathbf{D})$, *i.e.* balance the workloads among nodes. We first briefly review the Block Cyclic (BC) distribution in Section 4.4.1 to explain how it can be adapted to heterogeneous tasks in the case of compressed matrices. In Section 4.4.2, we consider a natural extension of the BC algorithm, called Block Cyclic Extended (BCE). Variants of this scheme have been proposed in [21] for example, but we propose here a formalization of its design and a simple algorithm to solve the associated optimization problem. Finally, we propose a randomized strategy in Section 4.4.3, Random Subsets (RSB), that does not rely on a repeated pattern.

Both BC and BCE distributions are based on a repeated pattern, denoted $\mathbf{G}$ of size $r \times c$. To ease the evaluation of the load balancing metric for those distributions, we define the aggregated grid of workload, denoted $\overline{\mathbf{W}}$. It has the same dimensions as $\mathbf{G}$ and depends on the input matrix of workloads $\mathbf{W}$: the value at each position of $\overline{\mathbf{W}}$ corresponds to the total execution time of all tasks assigned to the node at the same position in $\mathbf{G}$:

$$\forall (k,\ell) \in \{1,\dots,r\} \times \{1,\dots,c\} : \quad \overline{\mathbf{W}}(k,\ell) = \sum_{\substack{(i,j)\in\mathcal{I} \\ k=i \mod r \\ \ell=j \mod c}} \mathbf{W}(i,j) \tag{4.3}$$

The value of the load balancing metric is then:

$$\mathfrak{B}(\mathbf{D}) = \max_{p\in\{1,\dots,P\}} \left(\sum_{\substack{\mathbf{G}_{k,\ell}=p \\ (k,\ell)\in\{1,\dots,r\}\times\{1,\dots,c\}}} \overline{\mathbf{W}}(k,\ell) \right)$$

4.4.1 Block Cyclic

As already mentioned, BC distribution using a square pattern of size $\sqrt{P} \times \sqrt{P}$ is optimal for matrix multiplication and LU factorization regarding the generated communication volume. However, in the case of heterogeneous tasks that we are considering in this chapter, it no longer provides an optimal distribution regarding the load balancing among

nodes. Besides, the compressed matrices that we are considering typically come from the discretization of physical equations in practice, as mentioned in Section 4.2.3. As a consequence, they often feature full rank diagonal tiles, that are not compressed, and low rank off diagonal ones, that are highly compressed. Hence, the workloads associated with diagonal tiles are significantly larger that those associated with off diagonal ones. To allow a better load balancing among all nodes in this situation, the shape of choice for the pattern of BC distribution is such that $|p-q|=1$ as it ensures a wide variety of nodes present on the diagonal tiles. To compare our strategies to the best possible variant of BC, we therefore carry out the experimental tests using values of P such that $P=p(p-1)$.

4.4.2 Block Cyclic Extended

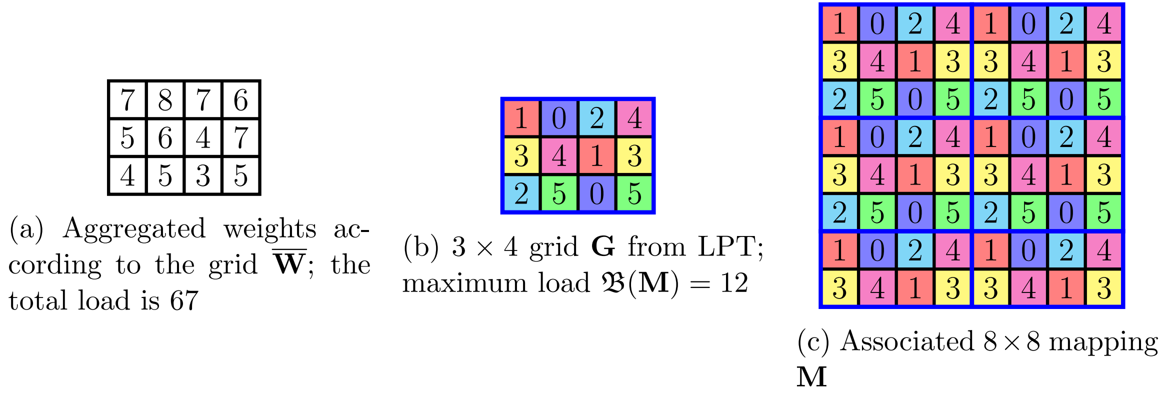
Since BC is optimal regarding communications but does not necessarily achieve a good load balancing among nodes in the heterogeneous case, a quite natural idea is to relax its characteristics regarding the number of different nodes on rows and columns, allowing potentially more communications, so as to better control the load allocated to each node. This strategy has already been proposed in [21] but with a non-optimal computation algorithm. We formalize it here into a data distribution strategy called Block Cyclic Extended (BCE). The key point of the method is to allow $p_{\max} > \sqrt{P}$ different nodes per column and per row. As mentioned in Section 4.3.4, $p_{\max}$ is defined according to a constraint tightness parameter $\alpha \geqslant 1$: $p_{\max} = \left\lceil \alpha \lceil \sqrt{P} \rceil \right\rceil$. The whole BCE strategy is then carried out following two steps:

1. For a given number of nodes P and parameter α, define a grid of dimensions (r,c) with $r=p_{\max}$ and c the largest integer smaller than or equal to $p_{\max}$ such that P divides rc. Then compute the aggregated grid of workload $\overline{\mathbf{W}}$ of dimension (r,c) according to Equation 4.3: the value associated with each position in $\overline{\mathbf{W}}$ is the sum of the workloads of all corresponding positions in $\mathbf{W}$ when replicating the grid over the entire matrix.

2. Create a pattern $\mathbf{G}$ of dimensions (r,c) by allocating each position to a node based on the aggregated workload values in $\overline{\mathbf{W}}$. The final distribution $\mathbf{D}$ of the output matrix is then derived the same way as when using BC method, simply replicating the pattern $\mathbf{G}$ over the entire matrix.

The whole process can be seen as a solution according to BC method using $\alpha^2 P$ "virtual" nodes since $\mathbf{G}$ features approximately α^2 times more positions than P. The procedure is then completed by associating those virtual nodes with real ones.

Relaxing the constraint on the number of different nodes on rows and columns allows many pairs of dimensions (r,c) for the pattern, thus leading to as many different feasible distributions after step 2. One may expects that all of such potential dimensions are worth testing. However, the matrix of cumulated workload $\overline{\mathbf{W}}$ is obtained by aggregating many values of execution times in $\mathbf{W}$. Hence, the values in $\overline{\mathbf{W}}$ are very homogeneous. It is therefore very difficult to achieve a good load balancing between nodes in step 2 if



(a) Aggregated weights according to the grid $\overline{\mathbf{W}}$; the total load is 67

(b) 3×4 grid $\mathbf{G}$ from LPT; maximum load $\mathfrak{B}(\mathbf{M}) = 12$

(c) Associated 8×8 mapping $\mathbf{M}$

Figure 4.1: Example of BCE allocation: $P = 6$, $p_{\max} = 4$, $\mathbf{G}$ is 3×4

not all of them are associated with the same number of positions in the pattern. It is therefore very difficult to achieve a good load balancing between nodes in step 2 if not all of them are associated with the same number of positions in the pattern. That is why we select c as the largest possible integer such that P divides rc. This issue is discussed in more details in Section 4.6.

Once the aggregated grid of workload $\overline{\mathbf{W}}$ is calculated, building up the pattern $\mathbf{G}$ in step 2 reduces to a well known scheduling problem: the makespan minimization problem on P homogeneous machines, denoted $P\,||\,C_{\max}$ according to Graham notation (introduced in [43], application can be found at [76]). This problem is also closely related to the bin packing problem. In our context, each aggregated workload value in $\overline{\mathbf{W}}$ must be allocated to a node and the objective is to minimize the load of the most loaded one. Several solution methods exist for such problem: exact ones such as Integer Linear Programming, and approximated ones. With a naive formulation, the linear program which has to be solved when using such exact method is quite large, featuring $\mathcal{O}(\alpha^2 P)$ variables and $\mathcal{O}(\alpha\sqrt{P})$ constraints, thus leading to long solution time. On the other hand, some simple heuristics provide, in virtually no time, good quality approximated solutions to this problem, though this should be tampered according to the specific cases mentioned above. We chose to perform step 2 of BCE method using the **L**argest **P**rocessing **T**ime (**LPT**) greedy heuristic well known for its good average performance and its guaranteed approximation ratio of $\frac{4}{3}$. Its implementation is detailed in Algorithm 22.

4.4.3 Random Subsets Algorithm

Both BC and BCE impose a very regular pattern for allocating tiles to nodes. This has the advantage of allowing direct control of the number of nodes allocated to each row and each column and ease of implementation. On the other hand, such a regular pattern (i) imposes constraints on P because of the fixed size of the pattern and (ii) limits the search for a well balanced solution since the complete distribution is build from a solution of a restricted load balancing problem using $\overline{\mathbf{W}}$ as input. BCE distribution is however expected to limit those issues somewhat.

In the following we define another data distribution scheme called Random Subsets

Algorithm 22: Largest Processing Time algorithm

Input: P, $\overline{\mathbf{W}}$
Output: $\mathbf{G}$, L

1 (r', c') are the dimensions of $\overline{\mathbf{W}}$.
2 Initialize the total load of each node: $L(p) = 0,\ \forall p \in \{1, \ldots, P\}$
3 **for** $(i, j) \in \{1, \ldots, r'\} \times \{1, \ldots, c'\}$ considered in decreasing order of $\overline{\mathbf{W}}(i, j)$ **do**
4   Find least loaded node: $p_{\text{least}} = \underset{p \in \{1,\ldots,P\}}{\operatorname{argmin}} (L(p))$
5   Allocate node p_{least} to position (i, j): $\mathbf{G}(i, j) = p_{\text{least}}$
6   Update the load of node p_{least}: $L(p_{\text{least}}) \leftarrow L(p_{\text{least}}) + \overline{\mathbf{W}}(i, j)$

(RSB) which attempts to overcome those issues. The underlying idea in RSB is to step away from distributions based on regular pattern replication and their inherent constraints, and try to apply the non expensive and relatively efficient LPT heuristic on the whole matrix of cumulated workload $\mathbf{W}$. However, trying to directly apply this method to solve the load balancing problem under the constraints of System 4.2 may lead to "dead end" configurations where the algorithm cannot complete the allocation process because the selection of any node would increase the number of different nodes per row and column beyond the constraint limits. An illustration of such a configuration can be seen on Figure 4.2. The probability of the algorithm getting stuck in such configuration dramatically increases with the size of the problem m_b and the number of processors P. Because of the highly combinatorial nature of the problem, backtracking to the last feasible configuration in order to continue the exploration of the solution space would lead to prohibitively long resolution time. Hence direct application of LPT is unusable in practice for almost all use cases.

3	2	1	2	1	0	0	1
2	4	4	3	1	1	0	0
2	4	4	5	5	3	2	0
1	3	4	7	3	2	0	0
0	2	2	7	7	8	5	3
0	1	3	8	14	12	7	11
0	2	6	10	7	19	18	15
0	0	4	9	16	17	7	20

(a) 8×8 matrix of estimated execution time W

–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–
–	–	–	–	–	–	–	–
–	–	–	4	–	–	–	–
–	–	–	?	5	3	–	2
–	–	–	1	–	1	2	5
–	–	–	0	4	3	–	0

(b) Incomplete allocation: next position to allocate is $(6, 4)$ but no suitable processor can be found since the current allocation allows processors $\{2, 3, 5\}$ on row 6 and $\{0, 1, 4\}$ on column 4

Figure 4.2: Example of "dead end" configuration ($m_b = 8$, $P = 6$, $p_{\max} = 3$)

To circumvent this problem RSB method is based on predefined subsets of nodes

associated with either rows or columns. Nodes allocated on each row, respectively column, all belong to one subset. Those subsets have the following properties:

1. each has no more than $p_{\max}$ elements;
2. each row, respectively column, subset is *compatible*, *i.e.* has a non empty intersection with each column, respectively row, subset.

Using row and column subsets, a modified version of LPT can be applied to directly allocate nodes to tiles of the output matrix. This greedy heuristic, detailed in Algorithm 23, necessarily provides a feasible distribution.

Having available row and column subsets of nodes is a prerequisite to the above mentioned heuristic. Finding subsets featuring the required properties (1) and (2) in a deterministic way seems at least tedious for small use cases and extremely challenging for large values of P. Therefore, in the practical implementation of RSB we chose to generate those subsets using a randomized strategy.

More precisely, we select a number Q of row and column subsets that needs to be created, along with $K \geqslant 1$, the minimum accepted intersection size between any two row and column subsets. Then the subsets are generated using random sampling according to the procedure described in Algorithm 24.

This process produces the two necessary families of row ($\mathcal{R}$) and column ($\mathcal{C}$) subsets of nodes which feature the desired properties (1) and (2). Although quite straightforward, the subsets generation procedure may require a long time to complete due to the fact that, given the $\mathcal{R}$ family of row subsets, each randomly sampled column subset C is tested against a compatibility criterion, line 5 of Algorithm 24, that, if not met, imposes to re-sample C thus lengthening the execution. The procedure can be tuned by using different numbers of row and column subsets to accelerate the generation process. However this would make the following statistical analysis more complex. Using the same number of row and column subsets, the difficulty to sample a suitable C subset can then be linked to the input parameters Q and K values.

Assuming, for each subset, unbiased and independent samplings of nodes according to a uniform probability distribution, the probability of C being compatible with each subset in $\mathcal{R}$ is:

$$\Pr(\{R_1 \cap C \geqslant K\}; ...; \{R_Q \cap C \geqslant K\})$$

which reduces to:

$$\Pr(\{R \cap C \geqslant K\})^Q \tag{4.4}$$

since all subsets are supposed identically and independently sampled, R being any of the row subsets in family $\mathcal{R}$. This probability can be calculated from the following expressions:

$$\Pr(\{R \cap C \geqslant K\}) = 1 - \sum_{k \in \{0,...,K-1\}} \Pr(\{R \cap C = k\}) \tag{4.5}$$

Algorithm 23: Random Subsets greedy algorithm

Input: P, $\mathbf{W}$, $\mathcal{R} = \{R_1, ..., R_Q\}$, $\mathcal{C} = \{C_1, ..., C_Q\}$
Output: $\mathbf{D}$, L

1 Initialisation
2 Total load for each node: $L(p) = 0,\ \forall p \in \{1, \ldots, P\}$
3 Initially empty $m_b \times m_b$ mapping of tiles: $\mathbf{M}$
4 Initial set of usable subsets for each row: $\mathcal{R}_i = \mathcal{R},\ \forall i \in \{1, \ldots, m_b\}$
5 Initial set of usable subsets for each column: $\mathcal{C}_j = \mathcal{C},\ \forall j \{1, \ldots, m_b\}$
6 Initial set of usable processors for each position:
$\mathcal{P}_{i,j} = \{1, \ldots, P\},\ \forall (i,j) \in \{1, \ldots, m_b\}^2$
7 Initial list of tiles to allocate: $\mathcal{B} = \{(i,j)\ :\ \forall (i,j) \in \{1, \ldots, m_b\}^2\}$
8 Allocation
9 **while** $\mathcal{B} \neq \emptyset$ **do**
10 Get the *heaviest* tile's position to allocate: $(i,j) = \underset{(i,j)\in\mathcal{B}}{\texttt{argmax}}(\mathbf{W}(i,j))$,
$w = \mathbf{W}(i,j)$
11 Remove it from the list: $\mathcal{B} \leftarrow \mathcal{B} \setminus \{(i,j)\}$
12 Find least loaded processor among usable ones: $p_{\text{least}} = \underset{p\in\mathcal{P}_{i,j}}{\text{argmin}}(L(p))$
13 Allocate processor p_{least} to position (i,j): $\mathbf{D}(i,j) = p_{\text{least}}$
14 Update the load of processor p_{least}: $L(p_{\text{least}}) \leftarrow L(p_{\text{least}}) + w$
15 Update the set of still usable subsets on row i:
16 **for** $R \in \mathcal{R}_i$ **do**
17 **if** $\exists p \in \{\mathbf{D}(i,\ell)\ :\ \ell \in \{1, \ldots, m_b\}\}\ :\ p \notin R$ **then**
18 $\mathcal{R}_i \leftarrow \mathcal{R}_i \setminus R$
19 Update the set of still usable subsets on column j, $\mathcal{C}_j$, the same way as $\mathcal{R}_i$
20 Update the sets of still usable processors on row i:
$\forall \ell \in \{1, \ldots, m_b\}\ :\ \mathcal{P}_{i,\ell} = \left(\cup_{R\in\mathcal{R}_i} R\right) \cap C_\ell$ — $\mathcal{P}_{i,\ell} \neq \emptyset$ *because subsets are* compatibles
21 Update the sets of still usable processors on column j:
$\forall k \in \{1, \ldots, m_b\}\ :\ \mathcal{P}_{k,j} = \left(\cup_{C\in\mathcal{C}_j} C\right) \cap R_k$
22 **for** $(k,\ell) \in \mathcal{B}$ such that $|\mathcal{P}_{k,\ell}| = 1$ **do**
23 Allocate the only possible processor $p \in \mathcal{P}_{k,\ell}$ to position (k,ℓ): $\mathbf{M}_{k,\ell} = p$
24 Update its load: $L(p) \leftarrow L(p) + \mathbf{W}(k,\ell)$
25 Remove position (k,ℓ) from $\mathcal{B}$: $\mathcal{B} \leftarrow \mathcal{B} \setminus \{(k,\ell)\}$

Algorithm 24: Subsets Generation algorithm

Input: P, $p_{\max}$, Q, K
Output: $\mathcal{R}$, $\mathcal{C}$

1 Randomly sample Q sets of $p_{\max}$ indices among $\{1, \ldots, P\}$ without replacement; they are the *row* subsets $\{R_1, ..., R_Q\}$ gathered in $\mathcal{R} = \{R_1, ..., R_Q\}$
2 Initialize the family of column subsets: $\mathcal{C} = \emptyset$
3 **while** $|\mathcal{C}| < Q$ **do**
4 Randomly sample a subset C of $p_{\max}$ indices among $\{1, \ldots, P\}$ without replacement
5 **if** C is *compatible*: $\forall i \in \{1, \ldots, Q\}$: $|C \cap R_i| \geqslant K$ **then**
6 Add C to the family of *column* subsets: $\mathcal{C} \leftarrow \mathcal{C} \cup \{C\}$

and:

$$\Pr(\{R \cap C = k\}) = \sum_{1 \leqslant \ell_1 < \ldots < \ell_k \leqslant P} \left[\left(\prod_{t=1}^{k} \frac{p_{\max}}{P - \ell_t + 1} \right) \times \left(\prod_{\substack{\ell=1 \\ \ell \neq \ell_1, \ldots, \ell \neq \ell_k}}^{p_{\max}} 1 - \frac{p_{\max}}{P - \ell + 1} \right) \right] \quad (4.6)$$

On the one hand, parameters Q and K can be selected so that this probability is large enough to ensure that the subsets generation process is "easy". Given a target lower bound $0 < \eta < 1$ for this probability, the number of subsets in each family must not exceed:

$$Q \leqslant \left\lfloor \frac{\log(\eta)}{\log(\Pr(\{R \cap C \geqslant K\}))} \right\rfloor \quad (4.7)$$

On the other hand, the value of Q must ensure a sufficient expected number of occurences of each node in the subsets so that the variety of nodes allows the subsequent allocation process using RSB algorithm to provide a good load balancing. For instance, requiring an average of $\beta > 0$ occurences of each proc in the subsets imposes to select Q such that:

$$Q \geqslant \frac{\beta P}{p_{\max}} \quad (4.8)$$

In turns, this leads to a probability of a successful column subset sampling η:

$$\eta \leqslant \Pr(\{R \cap C \geqslant K\})^{\frac{\beta P}{p_{\max}}} \quad (4.9)$$

There is therefore a trade-off to handle between the number of subsets and the sampling difficulty and associated generation time. However, the generation process is only specific to a pair of (P, α) parameters but not to the matrix size.

In the following experiments, for each (P, α) pair, subsets are generated using the value of parameter Q derived form Equation 4.8 with $\beta = 10$, and simply $K = 1$. 10

families $(\mathcal{R}, \mathcal{C})$ are generated using those parameters and for each test case, resolutions are performed using each of the 10 families of subsets. The best result according to the load balancing metric $\mathfrak{B}$ is kept as final result for the test case.

The RSB algorithm, derived from the LPT heuristic, is expected to have good performance regarding load balancing inasmuch as it allows choice between different resources at each iteration. Hence, one may want to enforce such behavior by generating subsets using a parameter $K \geqslant 2$. However, it makes the sampling process significantly longer, because of a tighter compatibility criterion, for only slight improvement: we indeed observed that the generation process using $K = 1$ leads to an average intersection size between any two row and column subsets R and C generally larger than 1 and, in practice, often close to 2.

Compared to BC and BCE, RSB is expected to produce better load balancing. On the other hand, the distribution of tiles allocated to a node is not a priori regular in the output matrix, contrary to pattern-based allocations. It is likely to cause load balancing problems in the course of execution for LU factorization in which, on one hand, all tasks are not available from the beginning and, on the other hand, only a few set of tasks remain available during the last phase of the execution.

4.5 Experiments

4.5.1 Test Cases

To evaluate the performance of the data distribution strategies developed in this chapter, it is relevant to try them out on test cases that feature a large diversity of compression levels among the tiles, *i.e.* their ranks, so that the load balancing problem is challenging enough to observe differences between solutions provided by the tested methods. In order to have a complete control on this property for the test matrices, we relied for our experiments on synthetic test cases. Nevertheless, those synthetic matrices are generated in a way that reproduces the main characteristics of actual matrices that can be encountered in various large scale simulation problems, such as in [2, 12]. An example of such a randomly generated matrix is illustrated on Figure 4.3: Figure 4.3a shows the distribution of the tile ranks while Figure 4.3b shows the cumulated execution time associated with each tile in the LU factorization. Note that for matrix multiplication, since each tile is associated with the same set of tasks, the relative distribution of cumulated execution time is essentially the same as the distribution of the ranks.

For each selected matrix size $m_b \times m_b$, we generated a pool of five square matrices for which the rank of each tile is defined according to a randomized procedure described below. The rank distribution of each synthetic matrix is used to simulate both input matrices $\mathbf{A}$ and $\mathbf{B}$ in the case matrix multiplication and matrix $\mathbf{A}$ in the case of LU factorization. The generation of those synthetic matrices requires to define the distribution of the ranks of the tiles according to their position and in particular on their distance from the diagonal. The following randomized process is used:

- All diagonal tiles are associated with a rank value $r_{i,i} = b$ for $i \in \{1, \ldots, m_b\}$, which means that diagonal tiles are assumed to be non-compressible, or *full rank*.

- For all other tiles: $(i, j) \in \{1, \ldots, m_b\}^2$, $i \neq j$, the value of $\frac{r_{i,j}}{b}$ is set to:

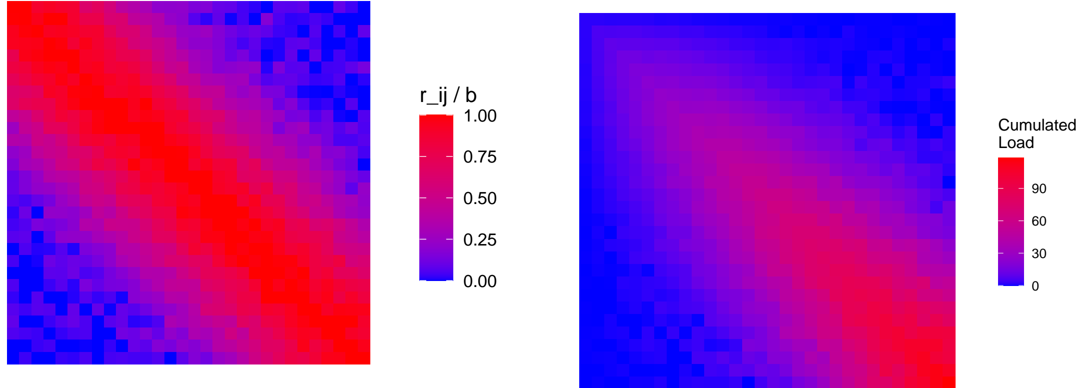
$$\frac{r_{i,j}}{b} = \max(\min(v(i,j) + \gamma, 1), 0)$$

  where:

  - $v(i,j)$ follows an exponential distribution as a function of the distance to the diagonal: $v(i,j) = \exp^{-\delta(\frac{i-j}{m_b-1})^2}$;
  - γ is a noise that follows a normal distribution: $\gamma \sim \mathcal{N}(0, \frac{1}{20})$.

  To generate synthetic matrices featuring a large variety of compression levels, slowly decreasing from diagonal positions, we used $\delta = 4$ as parameter of the exponential law of v.

- In addition to diagonal tiles, a number θ of off-diagonal tiles are assumed to be full rank. They are located uniformly randomly in the matrix, and θ follows a normal distribution: $\theta \sim \mathcal{N}(\sqrt{m_b}, \frac{\sqrt{m_b}}{2})$.



(a) Relative distribution of ranks on a 30×30 matrix; values are normalized to the tile size b

(b) Matrix W of cumulated execution time for LU factorization

Figure 4.3: Example of rank distribution and cumulated execution time for LU factorization on a 30×30 matrix using individual execution time for each type of task: $C_{GETRF} = 2$, $C_{TRSM} = 2$, $C_{GEMM} = 4$

Regarding the execution times of each type of task, they are chosen as follows: $C_{GETRF} = 2$, $C_{TRSM} = 2$ and $C_{GEMM} = 4$. It corresponds roughly to the relative execution times of the kernels associated with those types of task on dense tiles, as observed in the experimental setup described in Section 2.3.1. The exact values can be found in Table 2.1.

Then the actual execution time of each individual task is estimated using the rank of the tile to which it is applied, according to the performance model described in Section 4.3.3.

4.5.2 Results

The performance of the three strategies described in Section 4.4 has been evaluated using the synthetic test matrices for different values of the parameters (m_b, P, α). Each pair of values (P, α) is associated with a maximum number $p_{\max}$ of different nodes on each row and column whose values are summarized in Table 4.1. For all test cases, we choose values for P such that $P = q(q-1),\ q \in \mathbb{N}^*$ so that we can concentrate on the load balancing itself and not on rounding errors. Note that this situation favors BC and BCE distributions with respect to RSB, as discussed in more details in Sections 4.6.

All cases defined by the pair of values (P, α) are tested on matrices of size $m_b \times m_b$ with $m_b = 30, 60, 90$. For each triplet of parameters, all three methods are applied to the five synthetic matrices of the corresponding size $m_b \times m_b$ providing a data distribution for each. Then the evaluation metrics (maximum load, total simulated running time with and without communications) are computed for each distribution. The values of those metrics for the distribution obtained using each method are the results of our experimental evaluation. They are all normalized with respect to ideal load balancing where all processors would receive exactly the same load.

Those results are depicted for matrix multiplication in Figure 4.4 and for LU factorization in Figure 4.5. Each color corresponds to one method: BC in red, BCE in purple and RSB in green. The different metric values are represented by different shapes: the maximum load, $\mathfrak{B}$, is a dot, the running time without communications, $\mathfrak{D}$, is a triangle and the simulated running time with communications, $\mathfrak{C}$, is a square. Each point corresponds to the mean value of one metric over the five test cases; the error-bar represents the minimum and maximum values. Note that results for BC are shown on $\alpha = 2$ and $\alpha = 3$ plots as well to ease the comparison with the other methods but actually correspond to $\alpha = 1$ case.

		α	
		2	3
P	12	8 (8×6)	12 (12×9)
	30	12 (12×10)	18 (18×15)
	90	20 (20×18)	30 (30×27)

Table 4.1: Values of $p_{\max}$ (the maximum number of different nodes per row and column) according to the values of the parameters (P, α) and associated pattern sizes (r, c) used for BCE

Results show that both BCE and RSB lead to better solutions that plain BC distribution in specific regions of the parameters space. The general trends can be summarized as follow:

- The values of the load balancing criteria $\mathfrak{B}$ is significantly lower for both BCE and RSB compared to BC regardless of the problem size, number of nodes and constraint

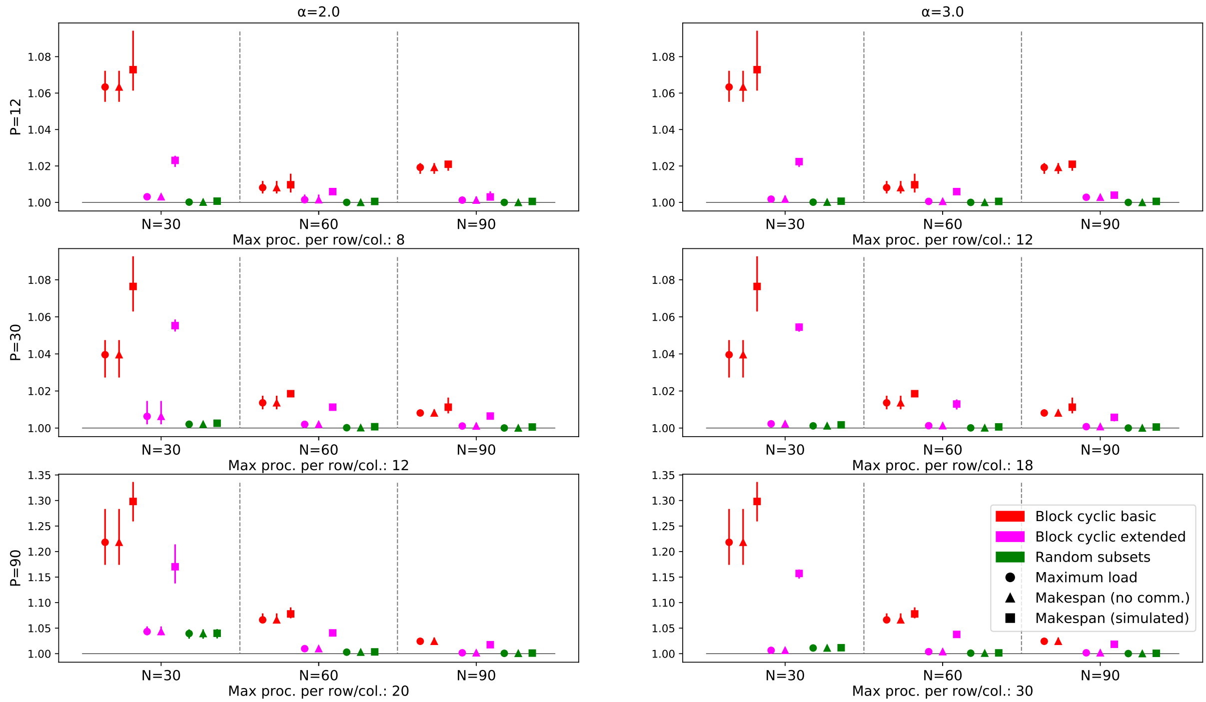


Figure 4.4: Results for matrix multiplication

on communications, controlled by the parameter α, for matrix multiplication and LU factorization alike.

- The values of the total simulated running time with, $\mathfrak{C}$, and without communications, $\mathfrak{D}$, show more mitigated results: though being generally lower for BCE and RSB than those achieved by BC, they become larger with the ratio $\frac{P}{m_b}$. This effect is particularly visible for both methods for $P = 90$ and $m_b = 30$ in the case of LU factorization and affects in a lesser manner BCE only in the case of matrix multiplication.

First, we can draw some interesting conclusions for the case of matrix multiplication whose results are depicted on Figure 4.4. Since the matrix multiplication consists of a set of independent tasks, there is in general no difference between pure load balancing and simulation without communications. For this operation, the best heuristic is clearly RSB for all values of P and m_b, provided that α is large enough. This observation is true even though the chosen P values benefit BC and BCE. With $\alpha = 3$, for all the configurations tested here, the value of the total running time with communications is indeed always less than 1% of the value that would be obtained without communications and with optimal load balancing.

The conclusions that can be drawn in the case of LU factorization, Figure 4.5, are quite similar, except that the dependencies between tasks generate a difference between total running time values with and without communications. We can notice that BC distribution achieves a better load balancing value for matrix multiplication than for LU factorization. This situation is related to the fact that the loads associated with the tiles

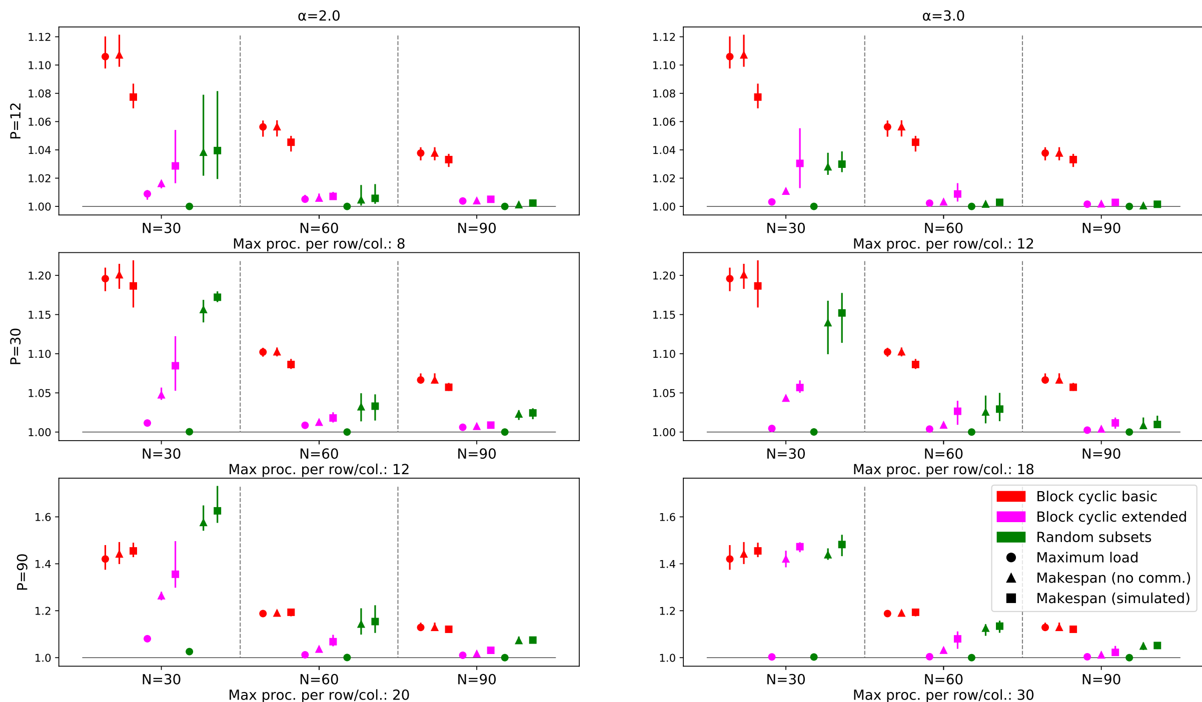


Figure 4.5: Results for LU factorization

are less homogeneous: indeed the weight of a tile increases with i and j, which makes load balancing more difficult. In this context, both BCE and RSB are strategies of choice for data distributions and both give excellent results. Nevertheless, the performance degradation between pure load balancing and running time without communications is more important in the case of RSB than in the case of BCE. This can be explained by the fact that, for distributions based on BCE, just like BC, the tiles assigned to a given node are always distributed regularly by construction, whereas this property is not automatically met with RSB. This can lead to load unbalance in the course of the execution of the operation, *i.e.* at certain instants, a subset of nodes may be overloaded compared to the rest of the nodes, which in turn can slow down the whole execution.

In all cases, we can observe that BCE behavior is very robust, as soon as α is greater than 2. For example, with $\alpha = 3$, BCE returns allocations that are within 5% of the lower bound. BCE often performs at least as well as RSB, which is a more expensive and sophisticated heuristic. As general conclusion from those results, we can therefore state that a meta-strategy based on selecting the solution with minimum maximum load among all three methods is already a significant improvement over BC and over each method alone. Such a strategy provides a good quality distribution for almost any combination of parameters.

4.6 Analysis of Block Cyclic Extended Algorithm

In this section, we analyze the behavior of BC and BCE for two types of tile weight distribution. In the case of the matrix multiplication, Figure 4.6, the tile weights in the matrix

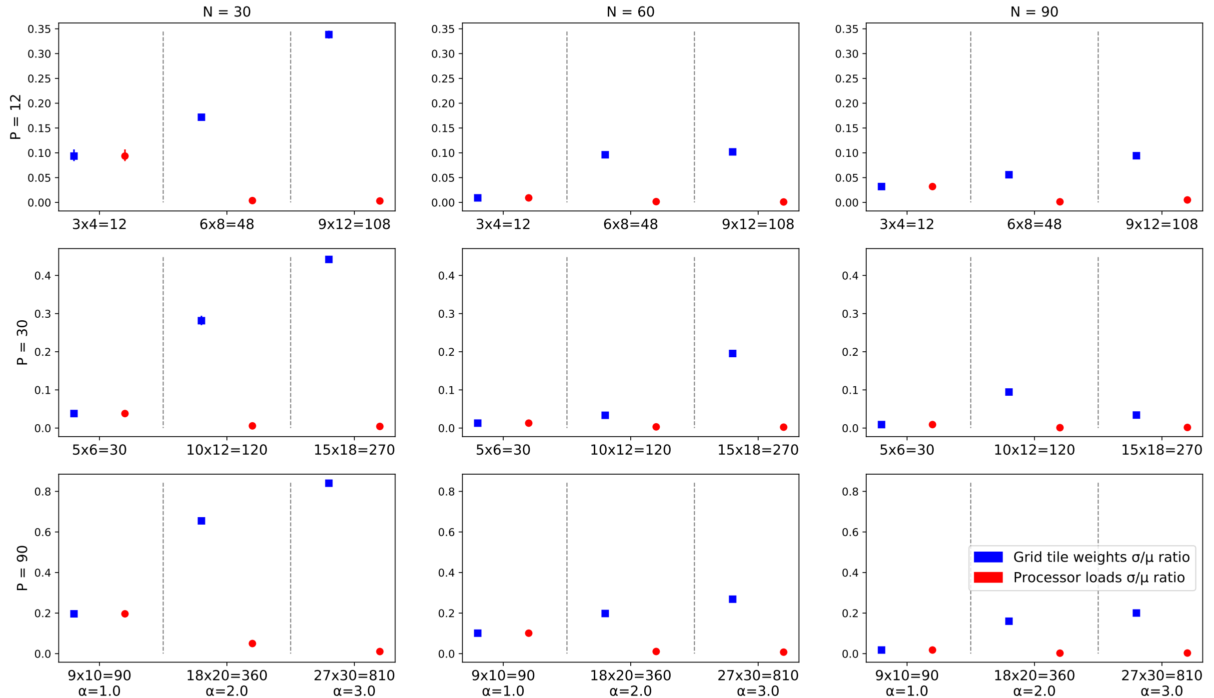


Figure 4.6: BCE load balancing for matrix multiplication

depend only on their distance to the diagonal. In the case of LU factorization, Figure 4.7, the weights of the tiles in the matrix depend on both their distance to the diagonal and to the bottom right corner, since the bottom right tiles induce a naturally higher computation cost because many more tasks are performed to update them. The heterogeneity of tile weights is therefore higher for LU factorization than for matrix multiplication.

In our experimental setup, we consider different sizes of matrices ($m_b = 30, 60, 90$), different values of the number of nodes ($P = 12, 30, 90$) and different values of α ($\alpha = 1, 2, 3$), where $p_{\max} = \left\lceil \alpha \lceil \sqrt{P} \rceil \right\rceil$ is the maximum number of different processors that can be present on the same row or column. $\alpha = 1$ corresponds to BC, whereas $\alpha = 2, 3$ correspond to two different variants of BCE.

Let us measure the quality of the load balancing between the nodes for pattern-based distributions BC and BCE. On Figures 4.6 and 4.7, blue points are associated with grid tiles, *i.e.* show the balance between the loads of the virtual nodes, whereas red points are associated with actual nodes. When $\alpha = 1$, values represented by the red and blue dots are the same since the numbers of virtual and real nodes are the same whereas the number of virtual nodes is α^2 times higher in general. Each point represents the dispersion of the load of virtual nodes (the blue points) or real nodes (red points), where the dispersion of nodes load is defined as $\frac{\text{standard deviation of the distribution of loads}}{\text{mean value of the distribution of loads}}$ so that a lower value represents a better load balancing. A ratio of 0 corresponds to perfect load balancing.

The first conclusion that can be drawn is related to the observation of the blue dots on any cell for $\alpha = 1, 2, 3$. We can notice that the first phase of tile aggregation following

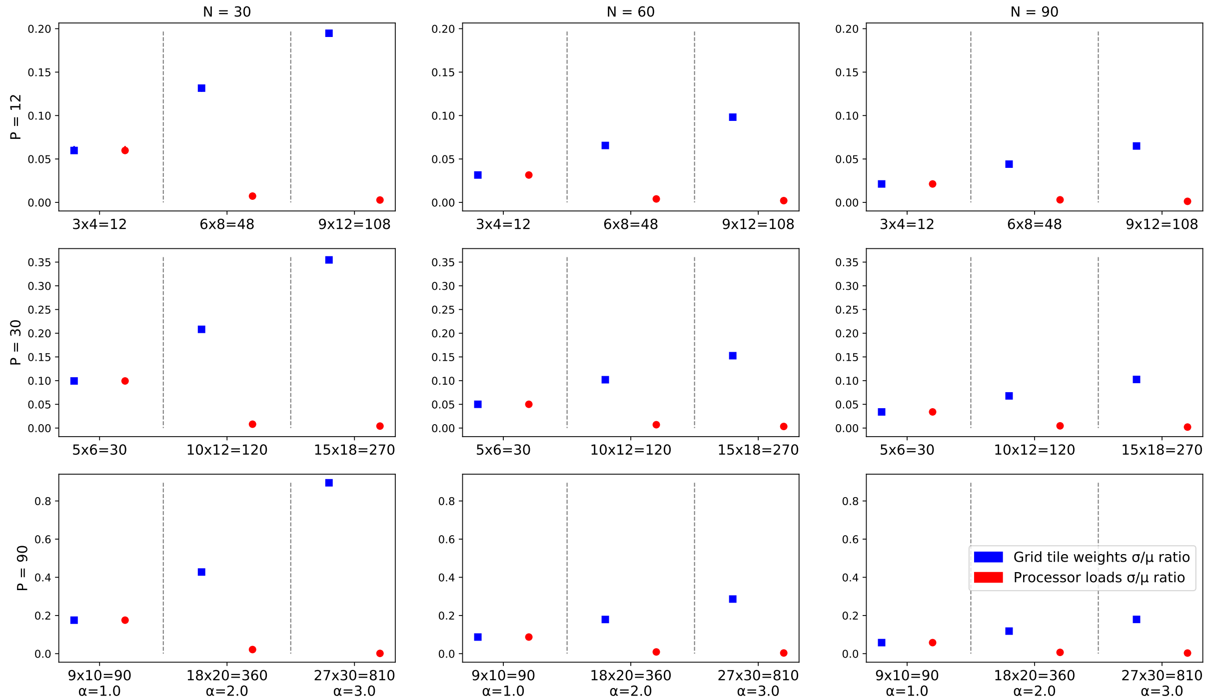


Figure 4.7: BCE load balancing for LU factorization

the regular pattern is efficient enough to limit the dispersion and that the dispersion is much greater when the number of virtual nodes is increased and therefore the number of tiles allocated to each virtual nodes is decreased.

The second conclusion is related to the observation of the relative values of the blue and red dots when $\alpha = 2, 3$. It shows that the second phase of packing virtual nodes into a pattern using LPT heuristic, as in BCE, is very efficient. Indeed, we can observe that the dispersion is almost null as soon as we group the virtual nodes by groups of 4 ($\alpha = 2$) or 9 ($\alpha = 3$).

The intuition is as follows: if we use for example $\alpha = \sqrt{2}$, we will obtain $2P$ virtual nodes, whose respective loads are already rather homogeneous since each virtual node corresponds to the aggregation of a certain number of tiles. In this case, the packing algorithm results in the most loaded k^{th} virtual node being matched with the least loaded k^{th} virtual node inside the real k^{th} node, resulting in a very homogeneous distribution of weights.

We have tried to establish this result theoretically but it is in fact notoriously difficult. Indeed, even if we assume that the initial weight distribution is known and simple, uniform or normal for example, then the distribution of the mean value and standard deviation for the kth element are only known by approximate and non-closed formulas [10,32]. This is why we relied on simulations to establish these properties.

4.7 Conclusion

In this chapter, we have considered the problem of designing data distributions to perform the matrix multiplication and LU factorization in a parallel and distributed setting in the context of compressed matrices with heterogeneous tile compression levels. We have considered the Block Low Rank (BLR) format which is largely used in the community of numerical linear algebra to reduce the memory footprint of operations involving large matrices. It is based on a regular per-tile compression that approximates each tile by the outer product of two low-rank dense matrices. In such a compressed case, the total execution time, or "workload", associated with the set of all tasks to be performed on a given tile varies according to its compression level which is generally higher for tiles away from the diagonals in actual applications. Thus, contrary to the dense case, workloads among tiles are no longer homogeneous. We presented two original data distribution strategies designed to handle this heterogeneity and ensure a good load balancing between nodes. The first one, Block Cyclic Extended (BCE), is an extension of Block Cyclic (BC). It makes use of a pattern, $\mathbf{G}$, whose size is such that there are more positions than available nodes. The workloads associated with each tile of the matrix are aggregated to each position in $\mathbf{G}$, each "virtual" node, by replicating the pattern over the entire matrix, as for classic BC method. Each position of the pattern along with its aggregated workload is then associated with a real node using a greedy heuristic. The use of a pattern with more positions than actual nodes allows to achieve a better load balancing between nodes than BC at the cost of a slightly higher number of processors per row and per column, which is however entirely controlled by an input parameter α of the method. The second one is a randomized strategy called Random Subsets (RSB) that is not pattern-based but rather directly provides an entire distribution of the matrix. It is a greedy algorithm that allocates each tile of the matrix one by one in order to balance the load among nodes. To make sure that the procedure finishes and provides a feasible solution according to the constraints on the maximum number of different nodes per row and column, it makes use of precomputed subsets of nodes that ensure that there is always a at least one valid candidate node when considering the allocation of each tile. Both of these strategies significantly improve the results compared to BC, achieving better load balancing and total running time. RSB show excellent results regarding load balancing, although the irregularity of its solutions seem to hinder its performance in terms of running time. BCE proves to be particularly efficient, in particular the good load balancing consistently translates into short running times, illustrating the importance of the distribution regularity to maintain load balance among all nodes in the course of the execution. RSB, on the other hand, has the advantage of giving very interesting results regardless of the value of P. The developments carried out in this chapter therefore open several perspectives. First of all, one can try to improve the RSB strategy so that it better takes into account the specific dependencies of the application's tasks, in order to better balance the load throughout the entire computation. It may also be possible to hybridize the different strategies by adding a randomized component to BCE so that it can run on any number of processors.

Chapter 5

Data Distribution Schemes for Dense Linear Algebra Factorizations on Any Number of Nodes

5.1 Introduction

As already mentioned, the results presented in Chapters 2 and 3 define a new interesting standpoint regarding the problem of communication minimization for dense linear algebra operations. As it is summarized in Section 3.5, a new set of simple and efficient algorithms is now readily available for the two operations symmetric rank-k update and Cholesky factorization, along with bounds that enable to evaluate their absolute performance regarding communications. In the out-of-core context, both those operations and their non-symmetric counterparts, matrix multiplication and LU factorization, can therefore be performed with the minimum possible number of communications. On the other hand, BC and SBC distributions provide the best known solutions to the load balancing and communication minimization problem in a parallel and distributed setting, though SBC is not optimal in terms of communications. Furthermore, while BC is already used in almost all linear algebra libraries, SBC can be easily implemented under a task-based execution model.

However, regardless of the optimality of the distribution, there remains a limitation to the practical usage of both BC and SBC methods related to the number of nodes used to carry out a computation. As mentioned in Sections 1.1.2.3 and 4.3.4, the BC pattern produced for $P = p^2$ nodes has many qualities: it ensures a good load balancing, both globally and in the course of the execution of an operation, as it is the smallest possible pattern where all nodes are present. Besides, in the case of non-symmetric operations, it minimizes the total communication volume because the number of different nodes per row and column, $\sqrt{P}$, is minimum, which is proven to be optimal for matrix multiplication [53]. The situation is more complex for many specific values of the number of available nodes P, the worst case being if it is a prime number. Let us consider for example that 23 nodes are available. Using BC distribution with a 23×1 pattern is very unbalanced and generates many communications as broadcasts along the rows requires sending messages

to 22 receiver nodes. A classical solution is to use 22 nodes instead, in a 11×2 pattern, which is still unbalanced, or 21 in a 7×3 pattern, which is better, or even only 20 in a 5×4 pattern. Sample results for LU factorization can be observed on Figure 5.1 which were obtained using the `Chameleon` library and the `StarPU` runtime system (details about the experimental setting used can be found in Section 5.6). These results show that, as expected, the performance per node increases when the pattern becomes closer to a square. However, the raw performance gains, which are equivalent to the time to solution, are limited by the fact that fewer nodes are used, so that all these solutions obtain roughly similar results, as can be observed on Figure 5.1 for patterns of size 11×2, 7×3 and 5×4. In practice, being able to run an operation on any number of nodes is of great practical interest because it is common that, given already scheduled reservations on a computing platform, the number of available nodes is not, or not even close, of the form $P = p^2$. Generally, in this case, users reserve fewer nodes than are actually available, as in the example above, but in the form $P = p \times q$, where p and q are of the same order of magnitude.

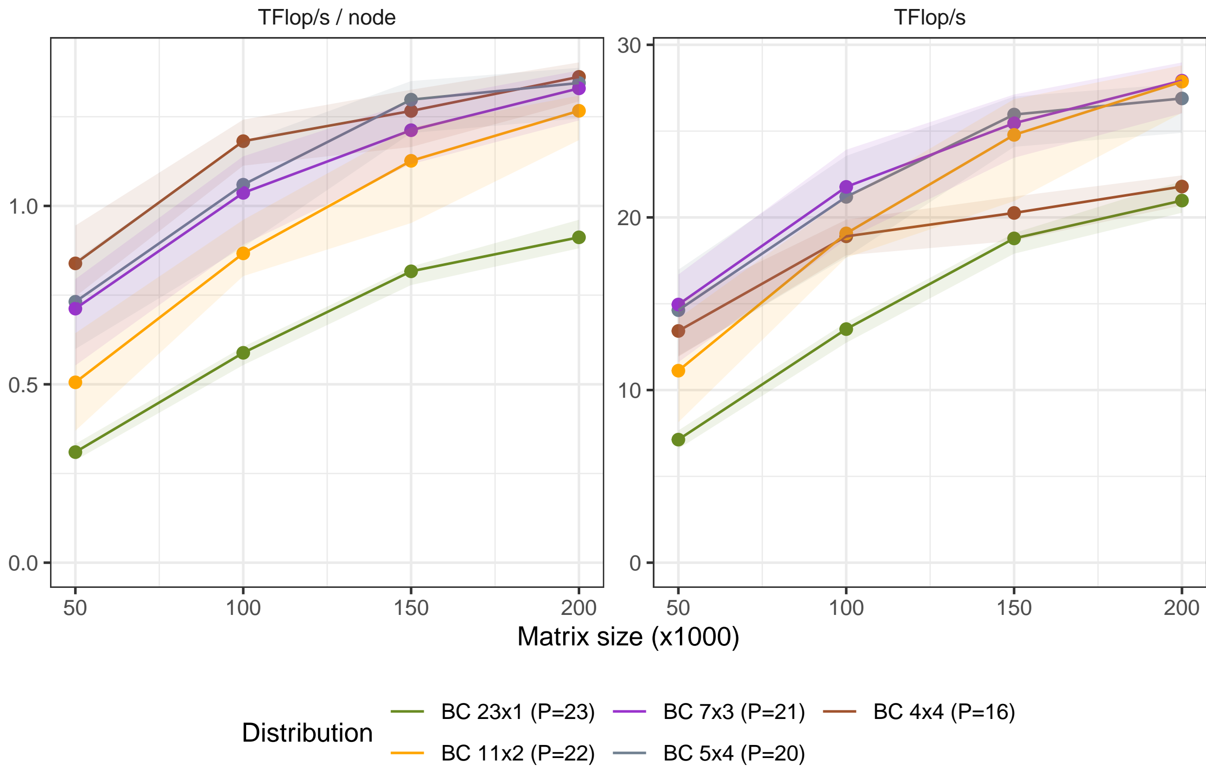


Figure 5.1: Sample results for LU factorization of matrix A of size $m \times m$, $m = 50,000$ to $200,000$, distributed using 2DBC with different pattern shapes.

In this chapter, we attempt to tackle this type of limitation and show that it is possible to use any number of nodes without sacrificing on the efficiency per node. Our objective is thus to build distribution patterns that can be used on an arbitrary number of nodes, while keeping good properties in terms of communications. The general approach is to

build larger patterns, in which each node may appear several times. This allows us to explore a larger solution space than plain BC, and we propose several new solutions, in both the non-symmetric and symmetric contexts.

We first study the non-symmetric case, where we propose a systematic way to construct, for all possible values of P, a perfectly balanced pattern of size $b(b-1) \times P$, where $b = \lceil \frac{P}{\lceil \sqrt{P} \rceil} \rceil$. Hence, $b(b-1)$ is of order P, and $\lceil \sqrt{P} \rceil$ different nodes appear in each row and in each column of the pattern. This pattern is therefore optimal in terms of load balancing, since each node appears exactly $P-1$ times, and in terms of communications, because any pattern using P nodes would require at least $\lceil \sqrt{P} \rceil$ different nodes per row and per column. The pattern is larger in general than the BC pattern but it is valid for all values of P.

We then consider the case of distributions tailored to symmetric problems. The Symmetric Block Cyclic distribution (SBC) introduced in Section 2.2.1 is valid for specific values of P, of the form either $\frac{r^2}{2}$ for an even integer r, or $\frac{r(r-1)}{2}$ for any integer r. SBC induces a communication volume lower by a factor of $\sqrt{2}$ than BC, but however remains within a factor of $\frac{3}{\sqrt{2}}$ of the lower bound, as summarized in Section 3.5. In the following, we propose an extension of this symmetric distribution to all possible values of P. Unlike for the non-symmetric case, it is very challenging to analytically build an elegant, optimal and valid distribution scheme for all P. Instead, we propose a greedy-based heuristic called Greedy ColRow & Matching (GCR&M) that builds an efficient pattern for a given P and a pattern size $r \times r$. The found solutions are not optimal, but yield better performance than SBC on average, with the additional benefit that GCR&M is able to provide solutions for all values of P.

Note that all the strategies elaborated in this chapter are dedicated to static data distributions, assuming that the execution of the considered operations is then performed using the owner computes rule. This assumption is not restrictive at all regarding the cases where additional memory is available because, as already mentioned in Section 2.2.3 for BC and SBC, the distributions presented here can as easily be extended to 2.5D or 3D versions. Such additional developments are simply orthogonal to the problem dealt with in this chapter and therefore left out of its scope.

The content of this chapter is organized as follows. In Section 5.2, we provide reminders of the context and point out already mentioned references about task-based runtime systems, lower bounds on communications induced by different linear algebra operations, and data distributions in the dense case and the sparse case, with homogeneous or heterogeneous resources. In Section 5.3 we introduce the model and notations used to evaluate the quality of data distributions and guide the design of more efficient ones, regarding total running time. In Section 5.4, we show how to build, in the non-symmetric case, a generic data distribution scheme that is valid for any number of nodes, while providing an optimal overall communication volume. In Section 5.5, we consider the symmetric case, and we propose an algorithm that, despite not being optimal, allows us to build efficient distribution schemes for any number of nodes. Section 5.6 present the results of an experimental evaluation of those two new distributions using the `Chameleon` library. It shows that the theoretical reduction of the communication volume induced by our distributions

actually translates into a reduction of the execution time of the factorization operations. Finally, we provide concluding remarks in Section 5.7.

5.2 Context

We investigate here the possibility of designing data distributions aimed at reducing the total communication volume for parallel execution of linear algebra operations in a task-based model. Prior research works related to this problem are already largely referenced in this document: the principles of the task-based model are described in Section 1.2; reviews of the existing work on communication bounds and algorithms can be found in Sections 2.1 and 3.1 for both out-of-core and parallel and distributed settings; Section 4.2 provides references regarding the problem of load balancing between nodes in parallel executions, in particular when considering heterogeneous computing resources or heterogeneous tasks. A quick summary of the main existing results is provided below; the reader may refer to those previous sections for additional details.

In this entire chapter, we consider the execution of linear algebra operations under the task-based model, *i.e.* the application performing the computations is associated with a runtime system which manages tasks and data dependencies as well as communications. Such a tool allows to perform operations using very irregular data distributions since the runtime system automatically enforces dependencies and infers the required inter-node communications from the distribution. The task-based model is therefore a requirement to actually implement the solutions elaborated in this chapter. This paradigm has been shown very efficient and scalable for many linear algebra applications [4] even when dealing with non-standard data distribution, as illustrate the results for SBC presented in Section 2.3.4.

Regarding communication lower bounds in the out-of-core context, the state-of-the-art results for LU and Cholesky factorization are respectively from Olivry *et al.* [62] and the work presented in Chapter 3. In this setting, with a memory of size M, the LU factorization of an $m \times m$ matrix requires a minimum of $\frac{2}{3}\frac{m^3}{\sqrt{M}} + \mathcal{O}(m^2)$ communications while Cholesky factorization requires $\frac{1}{3\sqrt{2}}\frac{m^3}{\sqrt{M}} + \mathcal{O}(m^2)$. As any bound in the out-of-core context, they can be extended to the parallel and distributed case using P nodes under the memory scalable assumption which states that the size of the total available memory is proportional to the size of the input matrix. This assumption is valid when using all the nodes in pattern-based data distributions. It thus exactly corresponds to the situation considered in this chapter. In such a case, the lower bounds translate as: the parallel and distributed LU factorization of an $m \times m$ matrix generates at least $\frac{2}{3}m^2\sqrt{P} + o(m^2)$ inter-nodes communications; Cholesky factorization generates $\frac{1}{3}m^2\sqrt{P} + o(m^2)$. The 2.5D algorithm proposed by Kwasniewski *et al.* in [56], which is actually based on BC, generates $m^2\sqrt{P} + o(m^2)$ communications. For the symmetric case, the best known solution is SBC which generates $\frac{1}{\sqrt{2}}m^2\sqrt{P} + o(m^2)$ communications when used to perform Cholesky factorization, as stated in Theorem 1 in Section 2.2.2.

In parallel and distributed settings, the problem of balancing the load among nodes has been extensively studied, particularly in the heterogeneous case, from two perspectives: (i) assigning heterogeneous tasks among identical resources, as it occurs in the context

of sparse or compressed matrices, such as illustrated in Chapter 4 or (ii) distributing homogeneous tasks among heterogeneous resources with different computing power. In both cases, however, the solutions provided focus on the matrix multiplication and do not consider symmetric problems and factorization operations. Moreover, because of the additional difficulty introduced by heterogeneity, they usually have to rely on approximation algorithms that perform worse in the homogeneous case than plain BC distributions.

5.3 Model and Notations

The objective of this chapter is essentially to build patterns that reduce the communication volume generated when performing for LU and Cholesky factorization. In this section, we analyze the communication volume induced by an arbitrary pattern when replicated onto the entire matrix and used for those two operations. The formal method to do so generalizes the analysis carried out for SBC pattern in Section 2.2.2. It allows us to define a *communication cost* metric for an arbitrary pattern.

We consider a matrix $\mathbf{A}$ divided into $m_b \times m_b$ tiles and distributed according to a pattern $\mathbf{G}$, of dimension $p \times q$. To avoid any ambiguity, we use *tile* to denote a position in the matrix, and *cell* to denote a position in a pattern. A pattern completely defines the data distribution of the matrix: indeed, just like for BC distribution, for all $(i, j) \in \{1, \ldots, m_b\}^2$, the tile at position (i, j) is owned by the node which is present in the cell $(i \mod p, j \mod q)$ in $\mathbf{G}$. Therefore the set of nodes in a row or column of $\mathbf{A}$ is the same as the set of nodes in the corresponding row or column of $\mathbf{G}$.

5.3.1 Case of LU factorization

To complete the analysis, we denote by:

- x_i the number of different nodes on row $i \in \{1, \ldots, p\}$ of $\mathbf{G}$;
- y_j the number of different nodes on column $j \in \{1, \ldots, q\}$ of $\mathbf{G}$.

First, let us analyze the communications scheme in the case of the tiled LU factorization as described in Algorithm 21 in Chapter 4. Let us assume that matrix $\mathbf{A}$ is distributed according to a cyclic distribution using P nodes. Its values are overwritten by $\mathbf{L}$ and $\mathbf{U}$ during the operation. An illustration with a BC distribution with $m_b = 12$ and $P = 6$ is depicted in Figure 5.2a. The replicated pattern is of size $p \times q$, with $(p, q) = (2, 3)$ in the example.

According to the analysis of the data dependencies of this operation detailed in Section 4.3.1.2, at each iteration $\ell \in \{1, \ldots, m_b\}$ of the algorithm, each node owning a tile in column ℓ of $\mathbf{A}$ sends it to all other nodes on the same row to the right, and each node owning a tile in row ℓ sends it to all other nodes on the same column below, as illustrated by the black zones on Figure 5.2a.

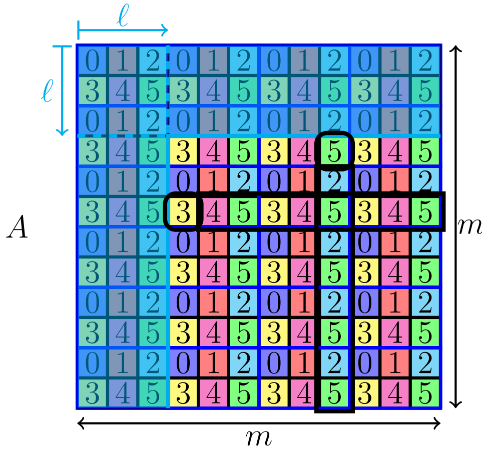
There are x_i different nodes on a row i of the pattern, respectively y_j on a column j, and one among them is the sender. Hence, for each replicated pattern, the communication

volume generated is $\underbrace{\sum_{i=1}^{p} x_i - 1}_{\text{row-wise}} + \underbrace{\sum_{j=1}^{q} y_j - 1}_{\text{column-wise}}$. The pattern is replicated $\frac{m_b - \ell}{p}$ times vertically and $\frac{m_b - \ell}{q}$ times horizontally over the trailing sub-matrix of $\mathbf{A}$. Assuming that $\ell \leqslant m_b - \max(p,q)$ at least one full pattern appears both vertically and horizontally. Therefore the total number of communications at iteration ℓ is:

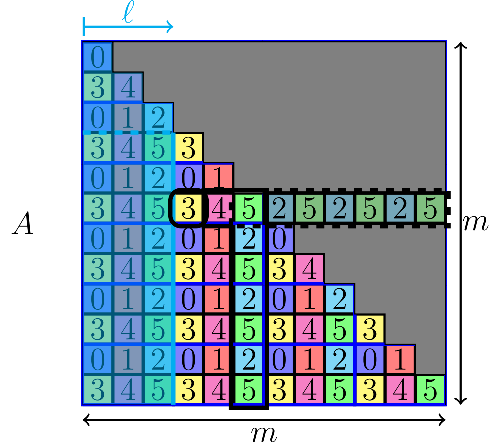
$$Q_\ell^{\mathrm{LU}} = (m_b - \ell)\Big(\frac{1}{p}\sum_{i=1}^{p}(x_i - 1) + \frac{1}{q}\sum_{j=1}^{q}(y_j - 1)\Big)$$

If we denote by $\bar{x}$ and $\bar{y}$ the average values of x_i and y_j, so that $\bar{x} = \frac{1}{p}\sum_{i=1}^{p} x_i$ and $\bar{y} = \frac{1}{q}\sum_{j=1}^{q} y_j$, and the total communication number over the complete factorization is given by:

$$Q^{\mathrm{LU}}(\mathbf{G}) = \frac{m_b(m_b+1)}{2}\big(\bar{x} + \bar{y} - 2\big) \tag{5.1}$$



(a) Example for LU factorization: solid black rectangles highlight two tiles of $\mathbf{A}$ sent by nodes 3 and 5 at iteration $\ell = 3$ and the corresponding sets of receiver nodes



(b) Example for Cholesky factorization: one tile of $\mathbf{A}$ is sent by node 3 at iteration $\ell = 3$ to receiver nodes in its *col-row*, as highlighted by the solid black shape. This communication scheme comes from the symmetry of $\mathbf{A}$ as illustrated by the dashed black rectangle

Figure 5.2: Communication scheme for LU and Cholesky factorization using BC distribution over an $m_b \times m_b$ matrix, $m_b = 12$, using $P = 6$ nodes laid out as a 2×3 pattern.

As soon as $\ell > m_b - \max(p,q)$, the trailing matrix of $\mathbf{A}$ becomes smaller than a full pattern, either vertically or horizontally. If $\ell > m_b - p$, the term of the first sum in Equation 5.1 is not x_i but the number of different nodes on row i of the pattern allocated to the last $m_b - \ell$ columns of matrix $\mathbf{A}$, which is smaller than x_i, and similarly if

$\ell > m_b - q$ for y_j. Equation 5.1 therefore overestimates the row-wise communications for the last q iterations, respectively column-wise communications for the last p iterations, because the domains where elementary operations are performed shrinks as iterations progress. Besides, partial pattern replication may occur at the edges of the matrix if p does not divide m_b (and/or q does not divide m_b) which would change one term of each sum in Equation 5.1. In either cases, it modifies the non-dominant parts of the total communication estimate and can therefore be neglected.

5.3.2 Case of Cholesky factorization

The case of Cholesky factorization is depicted in Figure 5.2b. Since $\mathbf{A}$ is symmetric, only half of it need to be handled; we assume that it is the lower triangular part. The data dependencies associated with the Cholesky factorization, described in Algorithm 4 in Chapter 1, have already been analyzed in Section 2.2.2. In this case, during iteration ℓ, tiles of column ℓ are sent to all nodes along the same row to the right, *and* along the column with the same index. In the following, we denote such a set of tiles as a *colrow*:

Definition 8 (colrow)**.** *Given a square pattern or a square matrix, we denote as* colrow *i the union of the column i and of the row i of the pattern or matrix. In addition, for a square pattern $\mathbf{G}$, we denote by z_i the number of nodes belonging to colrow i of $\mathbf{G}$.*

Thus, in the case of Cholesky factorization, tiles are sent along a colrow instead of a row or a column. Furthermore, when the pattern $\mathbf{G}$ is square, a colrow of the matrix always corresponds to exactly one colrow of the pattern. If this is the case, the previous reasoning to compute the number of communications at a given iteration remains valid if applied with the number of different nodes in a colrow, *i.e.* z_i for colrow i of the pattern. The number of communications generated at a given iteration for this kernel is therefore given by:

$$Q^{\mathrm{Chol}}(\mathbf{G}) = \frac{m_b(m_b+1)}{2}(\bar{z}-1). \tag{5.2}$$

The same restrictions apply as for LU factorization regarding domain shrinking and partial model replications.

5.3.3 Communication cost metric

In both cases (Equations 5.1 and 5.2), neither the multiplicative factor $\frac{m_b(m_b+1)}{2}$ nor the additive -1 depend on the pattern. In the following, we thus aim at designing patterns $\mathbf{G}$ that minimize the following metric $T(\mathbf{G})$, which we call *communication cost*:

$$T(\mathbf{G}) = \begin{cases} \bar{x}+\bar{y} & \text{for LU factorization} \\ \bar{z} & \text{for Cholesky factorization, if } p = q \end{cases}$$

In addition, in order to enforce a good load balancing of the computations, we require that the patterns are *balanced*, *i.e.* that all nodes appear the same number of times in the pattern. Note that contrarily to the BC pattern, we allow a node to appear several times in the pattern.

5.4 Generalized Block Cyclic

In this section, we present the Generalized Block Cyclic (G-BC) distribution, which extends the original BC distribution to any number of nodes, maintaining the good properties in terms of communication volume of the square BC distribution with $P = p^2$ nodes. First, we show in Section 5.4.1 how to build the pattern. Then, in Section 5.4.2, we prove some properties associated with G-BC. We then provide a comparative evaluation of the cost of G-BC and BC in terms of communication in Section 5.4.3. Finally experimental results illustrating the performance of G-BC are presented in Section 5.6.

5.4.1 Pattern Construction

In order to approximate a square pattern, we first build a quasi-square pattern with P cells, if there exists p, $P = p^2$, or slightly more. Let us denote $a = \left\lceil \sqrt{P} \right\rceil$, $b = \left\lceil \frac{P}{a} \right\rceil$ and $c = ab - P$. By construction, we can first observe that $0 \leqslant c < a$. Indeed, $\frac{P}{a} \leqslant b < \frac{P}{a} + 1$ so that $P \leqslant ab < P + a$ and $0 \leqslant c < a$. We can now define an incomplete $b \times a$ pattern $\mathcal{IP}$ which contains the elements from 1 to P and whose c elements of the last row are left undefined. An example for $P = 10$ is given on the left of Figure 5.3.

From $\mathcal{IP}$, we now build $b-1$ different $b \times a$ patterns $\mathcal{P}_i$, for $1 \leq i \leq b-1$. For a given $1 \leq i \leq b-1$, the pattern $\mathcal{P}_i$ is a copy of $\mathcal{IP}$, where the undefined elements are replaced with the last c elements of row i in $\mathcal{IP}$. These c elements are thus present twice in $\mathcal{P}_i$.

Additionally, we also build a last pattern of size $b \times (a-c)$, denoted $\mathcal{LP}$, which consists of the $a - c$ first columns of $\mathcal{IP}$.

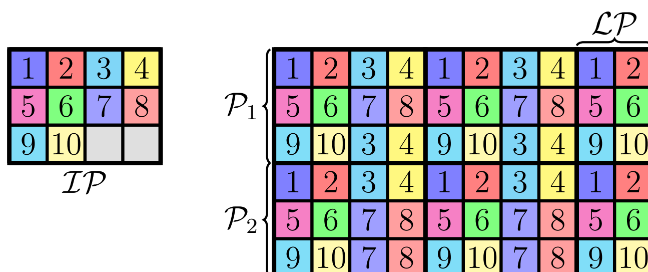


Figure 5.3: Example of the G-BC pattern for $P = 10$, thus $a = 4$, $b = 3$ and $c = 2$. Left: incomplete pattern $\mathcal{IP}$, right: full G-BC pattern.

Finally, the complete G-BC pattern $\mathcal{P}$ has size $b(b-1) \times P$, and is built as follows. The first b rows of $\mathcal{P}$ contain $b-1$ copies of $\mathcal{P}_1$, followed by a copy of $\mathcal{LP}$. This leads to a total of $a(b-1) + a - c = ab - c = P$ columns. The following rows of $\mathcal{P}$ are built in the same way, successively using copies of $\mathcal{P}_2, \ldots, \mathcal{P}_{b-1}$. The result pattern for $P = 10$ can be seen on the right of Figure 5.3.

5.4.2 Pattern Properties

We first show that the pattern $\mathcal{P}$ is well balanced: each node appears exactly $b(b-1)$ times.

Lemma 8. *In the G-BC pattern $\mathcal{P}$ defined above, each node is assigned to exactly $b(b-1)$ cells.*

Proof. We distinguish two cases: nodes that appear in $\mathcal{LP}$ and the others.

- Nodes in $\mathcal{LP}$ appear exactly once in $\mathcal{LP}$ and in each of the patterns $\mathcal{P}_i$. Furthermore, the pattern $\mathcal{P}$ contains exactly $b-1$ copies of $\mathcal{LP}$ and $b-1$ copies of each $\mathcal{P}_i$. Hence, a node in $\mathcal{LP}$ appears $b(b-1)$ times $\mathcal{P}$.

- Let us now consider a node that does not appear in $\mathcal{LP}$, and denote by u its row in $\mathcal{IP}$. This node appears in exactly two cells of $\mathcal{P}_u$, and in exactly one cell in each $\mathcal{P}_i$ for $i \neq u$. Therefore, this node appears in exactly $2(b-1)+(b-2)(b-1) = b(b-1)$ cells in $\mathcal{P}$.

□

Let us now analyze the communication cost $T(\mathcal{P})$ of this pattern. For this purpose, let us compute the number of different nodes that appear in each row and each column of $\mathcal{P}$. By construction, in each row of $\mathcal{P}$, exactly a different nodes are present, so that $\bar{x} = a$. In terms of columns, the situation differs depending on whether the column is part of $\mathcal{LP}$ or not. A column of $\mathcal{LP}$ contains exactly b different nodes, there are $a-c$ such columns, and the complete pattern contains b copies of each of these columns, one as a column of $\mathcal{LP}$, and $b-1$ as copies of the $\mathcal{P}_i$. A column not in $\mathcal{LP}$ contains $b-1$ different nodes, since the c undefined cells of $\mathcal{IP}$ are filled with nodes already present in the column. There are c such columns, and they are copied $b-1$ times in pattern $\mathcal{P}$.

In total, the pattern $\mathcal{P}$ has P columns, so that the average number of nodes per column in $\mathcal{P}$ is

$$\bar{y} = \frac{1}{P}\big(\overbrace{b \cdot (a-c) \cdot b}^{\text{columns in } \mathcal{LP}} + \overbrace{(b-1) \cdot c \cdot (b-1)}^{\text{columns not in } \mathcal{LP}}\big)$$

We can thus provide a bound on the total cost of the G-BC pattern.

Lemma 9. *For any P, the Generalized Block Cyclic pattern $\mathcal{P}$ with P nodes has a total cost $T(\mathcal{P})$ bounded by $2\sqrt{P} + \frac{2}{\sqrt{P}}$.*

Proof. By definition, $T(\mathcal{P}) = \bar{x} + \bar{y}$. The above considerations yield $\bar{x} = a$ and $\bar{y} = \frac{1}{P}(b^2(a-c) + (b-1)^2 c)$. We start by bounding $\bar{y}$:

$$
\begin{aligned}
\bar{y} &= \frac{b^2a - 2bc + c}{P} \\
&= \frac{b(ab-c) - cb + c}{P} && \text{and since } ab - c = P\text{:} \\
&= b\left(1 - \frac{c}{P}\right) + \frac{c}{P} && \text{we now replace } b = \frac{P+c}{a}\text{:} \\
&= \frac{(P+c)(P-c)}{aP} + \frac{c}{P} \\
&= \frac{P^2 - c^2}{aP} + \frac{c}{P} \\
&\leqslant \frac{P}{a} + \frac{c}{P}.
\end{aligned}
$$

This yields $T(\mathcal{P}) = \bar{x} + \bar{y} \leqslant a + \frac{P}{a} + \frac{c}{P}$. Let us write $a = \sqrt{P} + \mu$, where $0 \leqslant \mu < 1$. If we develop μ^2, we obtain:

$$
\frac{\mu^2}{a} = \frac{(a - \sqrt{P})^2}{a} = a - 2\sqrt{P} + \frac{P}{a},
$$

thus $a + \frac{P}{a} = 2\sqrt{P} + \frac{\mu^2}{a}$. We obtain $T(\mathcal{P}) \leqslant 2\sqrt{P} + \frac{\mu^2}{a} + \frac{c}{P}$. And finally, since $\mu < 1$, $a \geqslant \sqrt{P}$, and $c \leqslant a - 1 \leqslant \sqrt{P}$, we can conclude $T(\mathcal{P}) \leqslant 2\sqrt{P} + \frac{2}{\sqrt{P}}$.

□

As a comparison, the square BC pattern for $P = p^2$ obtains a cost exactly equal to $2\sqrt{P}$. In addition, we can remark that whenever $c = 0$, the G-BC pattern reduces to the standard BC pattern; this happens if $P = p^2$ or if $P = p(p+1)$.

5.4.3 Evaluation of G-BC

We can observe on Figure 5.4 a representation of the cost $T(\mathcal{P})$ of the G-BC pattern, compared to the standard BC pattern. For each value of P, we show the cost of the best available BC pattern for this value of P, using all possible ways to write P as $P = pq$, and the cost of the corresponding G-BC pattern. As we can see, the cost of G-BC closely follows the $2\sqrt{P}$ value, and allows to significantly improve the communication volume over BC for many values of P.

5.5 Data Distributions for Symmetric Matrices

In this section, we consider the symmetric case and search for patterns to generalize the SBC distribution. A mentioned above, in the symmetric case it is necessary to use square patterns of size r. Having a balanced pattern of size r imposes a strong constraint on r: since there are r^2 cells to distribute among P nodes, a balanced pattern would require that r^2 is a multiple of P. To soften this constraint, we use a property of the diagonal

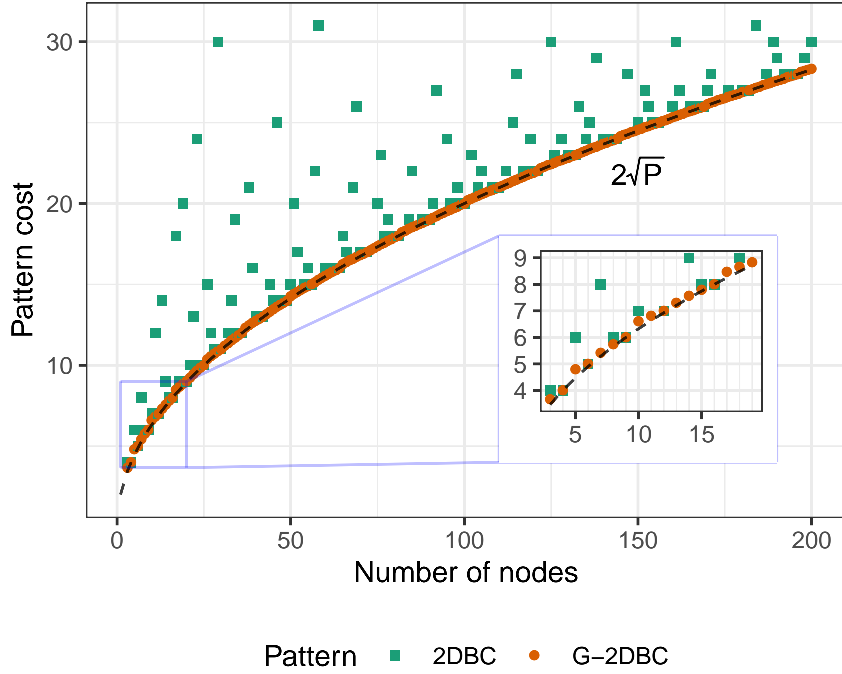


Figure 5.4: Total cost T of G-2DBC and the best 2DBC for varying values of P.

cells of the pattern: each diagonal cell belongs to a unique colrow. It can thus be assigned to any node on its colrow without changing the communication cost, and its different replicas on the complete matrix might even be assigned to different nodes as long as these nodes belong to the colrow of the pattern.

It is thus possible to design incomplete patterns, where the diagonal cells remain undefined, and assign them only when the pattern is replicated on the complete matrix. This can be done greedily, by successively assigning undefined tiles to the least loaded node among those present in the colrow. This is a generalization of the extended version of SBC detailed in Section 2.2.1.

Still, there are limitations to the possible values of r. On one hand, the load balancing procedure above can only be performed successfully if all nodes are assigned at most $\frac{r^2}{P}$ cells of the pattern. Otherwise, when the pattern is replicated over the entire matrix, one node would be assigned too many tiles, even not considering the undefined tiles which correspond to diagonal cells of the pattern. Then, assigning all those undefined tiles to other nodes would not be enough to compensate the imbalance. On the other hand, the average number of non-diagonal cells to assign to each node is $\frac{r(r-1)}{P}$, so that at least one node receives $\left\lceil \frac{r(r-1)}{P} \right\rceil$ cells. Hence, the following condition is necessary to ensure that there exists a pattern which is balanced or which can lead to a balanced distribution once replicated over the entire matrix:

$$\left\lceil \frac{r(r-1)}{P} \right\rceil \leq \frac{r^2}{P} \tag{5.3}$$

5.5.1 Greedy ColRow & Matching

Our greedy algorithm, Greedy ColRow & Matching (GCR&M), has two phases. The first phase assigns colrows to each node, trying to balance the load among all nodes. In this process however, it may happen that the colrow assignment allows several nodes to be assigned the same cell. Then, the second phase of the greedy algorithm applies a matching algorithm to perform the assignment of cells to the nodes.

Algorithm 25: Greedy ColRow & Matching Algorithm

Input: Number of nodes P, Pattern Size r

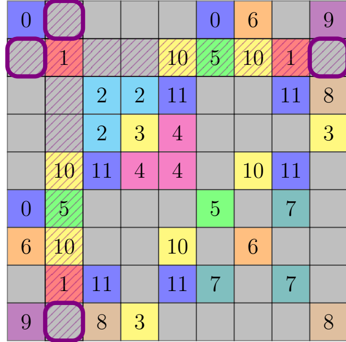
1 $\mathcal{U} \leftarrow \{(i,j) | 1 \leqslant i \neq j \leqslant r\}$
2 **for** $i \leftarrow 1 \ldots r$ **do**
3 $\quad \mathcal{A}[i \mod P] \leftarrow \{i\}$
4 **while** $\mathcal{U} \neq \emptyset$ **do**
5 $\quad p \leftarrow$ least loaded node
6 $\quad$ **for** each colrow r **do**
7 $\quad\quad \mathcal{C}[r] \leftarrow \mathcal{U} \cap \{(r,i) | i \in \mathcal{A}[p]\}$
8 $\quad b \leftarrow \text{argmax}_r(\text{Card}(\mathcal{C}[r]))$ *(tie-break: lowest usage)*
9 $\quad \mathcal{A}[p] \leftarrow \mathcal{A}[p] \cup \{b\}$
10 $\quad$ remove $\mathcal{C}[b]$ from $\mathcal{U}$
11 $k \leftarrow \left\lfloor \frac{r(r-1)}{P} \right\rfloor$
12 Compute a matching between all cells and k duplicates per node
13 Compute a matching between unassigned cells and 1 duplicate per node
14 **if** unassigned cells $c = (i,j)$ remain **then**
15 $\quad$ Assign (i,j) to the least loaded node p s.t. $\mathcal{A}[p]$ contains i or j

First phase. The first phase of the GCR&M algorithm computes an assignment $\mathcal{A}$ of colrows to nodes, so that $\mathcal{A}[p]$ is the set of colrows on which node p can appear. A cell (i,j) is *covered* by a node p if p can appear on this cell, *i.e.* i and j are both in $\mathcal{A}[p]$. In Algorithm 25, the set $\mathcal{U}$, defined line 1, contains all *uncovered* cells.

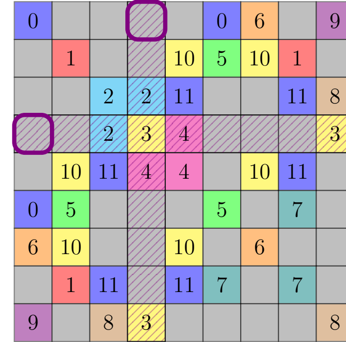
The algorithm starts by assigning one node to each colrow, in a round robin way if $P < r$, as shown line 3. The rest of the assignment is performed with a greedy procedure: as long as an uncovered cell remains, we assign an additional colrow to the least loaded node. The colrow is chosen so as to maximize the number of newly covered cells, as can be observed line 8. In case of equality, the least used colrow is selected, *i.e.* the one which appears in the lowest number of $\mathcal{A}[p]$ sets. Further ties are broken randomly. Once a colrow has been selected, the set of covered cells is updated.

An illustration of the first phase of Greedy ColRow & Matching algorithm is presented on Figure 5.5 for a pattern size of $r = 9$ and $P = 12$ nodes. The figure shows the set of uncovered cells, in grey, and the other cells colored according to the nodes that cover them. It illustrates the step of assigning an additional colrow to the least loaded node, line 5 in Algorithm 25. In this case, node 9 is the least loaded one, as it covers 2 cells while

all others cover at least 3. Node 9 is already assigned colrows 1 and 9: $\mathcal{A}[9] = \{1, 9\}$. The algorithm can assign to node 9 the colrow 1 because it allows to cover 4 additional cells (Figure 5.5a), which is the maximum, but not the colrow 4 which only allows to cover 2 additional cells (Figure 5.5b).



(a) Assigning colrow 2 to node 9 allows to cover 4 additional cells.



(b) Assigning colrow 4 to node 9 only allows to cover 2 additional cells.

Figure 5.5: Illustration of the first phase of GCR&M algorithm on a 9×9 pattern using $P = 12$ nodes. Gray cells are uncovered, the striped zone is the additional colrow, thick purple highlighted cells are newly covered cells.

Second phase. The first phase of the algorithm provides an assignment of colrows to the nodes, and could also be used to obtain an assignment of cells. For example, one could assign a cell to the first node that covers it. However, this does not guarantee a good load balancing. To solve this, the second phase of the GCR&M algorithm uses a bipartite graph matching approach. We build a bipartite graph with all cells on one side, and k copies of each node on the other side, where $k = \left\lfloor \frac{r(r-1)}{P} \right\rfloor$. There are edges between a cell and all copies of all nodes that cover this cell. A bipartite matching algorithm is used to perform an assignment of cells to nodes.

The reasoning behind this choice of k is the following: the total number of copies of nodes is lower than the number of cells, so that in a perfect matching all nodes receive k cells. Using $k' = \left\lceil \frac{r(r-1)}{P} \right\rceil$ would result in more copies of nodes than cells, so that all cells would be assigned. However, there would be no guarantee on the load balance between nodes: it could happen that many nodes receive k' cells, and some nodes receive few or even zero cells.

Hence, this first matching assigns k cells per node but may leave some cells unassigned. Another matching procedure is performed next, between all unassigned cells and a single copy per node. This ensures that all nodes have at least $\left\lfloor \frac{r(r-1)}{P} \right\rfloor$ and at most $\left\lceil \frac{r(r-1)}{P} \right\rceil$ cells. If some cells remain unassigned after both matching procedures, each remaining cell is assigned greedily to the least loaded node p that can cover it by adding only one colrow to $\mathcal{A}[p]$, it is the statement on line 15 of the algorithm.

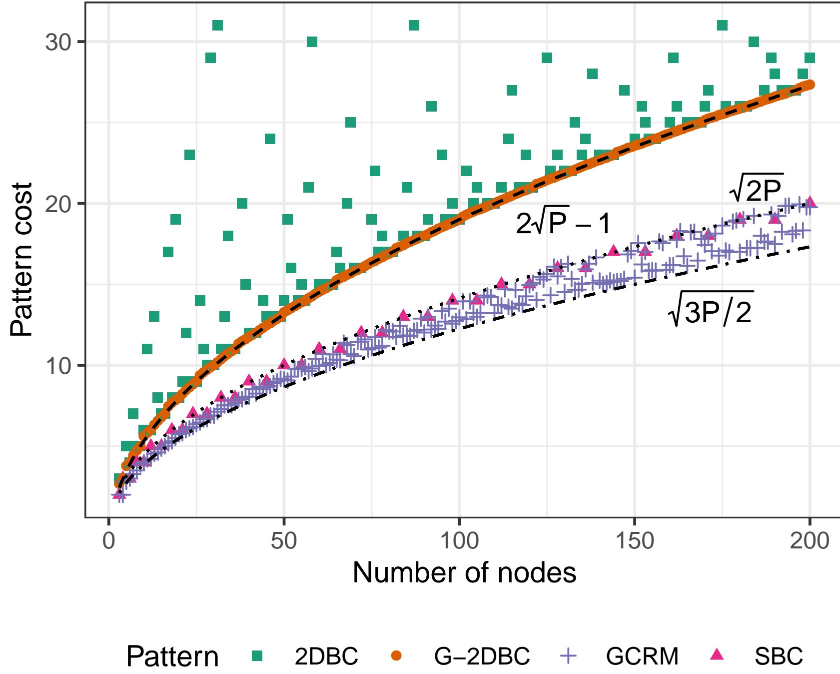


Figure 5.6: Total cost T of the symmetric and non-symmetric patterns for varying values of P.

Random choices The result of Algorithm 25 depends on random choices made when breaking ties. To obtain better results, the algorithm is run several times, and the pattern with the lowest cost is kept. 5 resolutions for the same case is enough in practice and this value is used later in the experiments.

5.5.2 Evaluation of GCR&M

We perform the evaluation of GCR&M as follows: for each value of P, we apply Algorithm 25 for all values of $r \leqslant 6\sqrt{P}$ which satisfy Equation 5.3. The algorithm is run 5 times with different random seeds. For each value of P, we keep the pattern with the lowest cost. These thresholds, the $6\sqrt{P}$ limit and the 5 re-executions of the algorithm, are sufficient in practice to obtain good patterns.

A comparison of the cost of all patterns for the symmetric case can be seen on Figure 5.6. As before, for each value of P the plot shows the cost of the best pattern of each type for this value of P. Note that for BC and G-BC, the symmetric cost is equal to the non-symmetric cost minus 1: indeed, the intersection between nodes in a row and in a column of the pattern always contains exactly one node. Thus, the number of nodes on a colrow is the number of nodes on some column, plus the number of nodes on some row, minus 1 for the intersection. The plot shows the SBC pattern, whose cost grows as $\sqrt{2P}$ for its basic version, and $\sqrt{2P} - \frac{1}{2}$ for the extended version. The GCR&M algorithm obtains patterns with a cost either similar to SBC, or even lower in many cases, with a lower limit that we empirically observe to be $\sqrt{\frac{3P}{2}}$.

An intuition for this limit can be provided. Let us consider a regular pattern in which each node is present on v colrows, and is assigned l non-diagonal cells. For example, on the SBC pattern, $v = 2$ and $l = 2$. If such a regular pattern contains P nodes, then its size r satisfies $r(r-1) = Pl$, so that $r \sim \sqrt{Pl}$. In addition, since each node is present on v colrows, we can compute the total number of nodes present on the colrows: $\sum_i z_i = Pv$. Thus, $\bar{z} \sim \frac{Pv}{\sqrt{Pl}} = \frac{v}{\sqrt{l}}\sqrt{P}$.

As mentioned, for SBC we have $v = 2$. The next possible value for v is $v = 3$, in which case the maximum possible number of non-diagonal cells is $v(v-1) = 6$. A regular pattern with these values would have a communication cost $T \sim \frac{3}{\sqrt{6}}\sqrt{P} = \sqrt{\frac{3P}{2}}$. The results obtained by GCR&M show that it is able to produce patterns where most of the nodes obtain an assignment of cells with a similar efficiency as what can be done with $v = 3$.

5.6 Experimental Performance Evaluation

In this section, we present sample results of experiments to test out the connection between the patterns cost and the actual performance of the associated cyclic distribution. We performed LU and Cholesky factorization on four test cases, using $P = 23$, 31, 35 and 39 nodes, which correspond to situations where there is no satisfying BC and SBC pattern using all the resources. For each test case, we compare the execution of the LU factorization with the G-BC distribution using all nodes and one or several BC distributions using a reasonably similar number of nodes, smaller or equal to P. Similarly, for Cholesky factorization, we compare the execution with the best GCR&M distribution found and the SBC distribution using the largest number of nodes smaller or equal to P. We observe the global and per-node performance in terms of GFlop/s.

5.6.1 Experimental Setup

The experiments were performed using the same platform as the one used to test SBC distribution. The complete description of the experimental setup can be found in Section 2.3.1 and is briefly summarized below:

- all the operations are performed in double precision on a cluster of 42 nodes, each equipped with 36 *Intel Xeon Skylake Gold 6240* cores;
- the nodes are connected with a 100Gb/s OmniPath network;
- experiments are carried out on randomly generated matrices which size ranges from $50,000$ to $300,000$ divided in tiles of size 500×500;
- 5 runs are performed for each matrix size and each tested method; on result plots each dot represents the average value over the 5 runs and the shaded zone the range between the minimum and maximum value;
- we use `Chameleon` version 1.1.0 associated with `StarPU` version 1.3.8;

- 2 cores per node are used to by `StarPU` for the scheduling and management of MPI communications;

- the *Intel MKL 2020* implementation of `BLAS` routines is used for the computations;

- `Open MPI` version 4.0.3 is used for inter process communications.

5.6.2 Results for Generalized Block Cyclic

For the test case using $P = 23$ and 39 nodes, we observe on Figures 5.7 and 5.10 that G-BC distribution consistently achieves a larger raw performance for all tested matrix sizes. More precisely, for $P = 23$, we can see on Figure 5.7 that BC using all the nodes performs poorly because of the tall and narrow shape of the pattern used, 23×1, which induces many communications. Its global throughput is even smaller than the version using only 16 nodes laid out as a perfect square, though the trend seems to reverse for larger matrix sizes. On the other hand, G-BC shows better global performance for all matrix sizes, which immediately translates into smaller running time. On a per-node basis, its performance is comparable to the BC distribution using 21 nodes with a 7×3 pattern. The observations are quite similar for $P = 39$: though using all the available nodes, the performance of the second BC distribution is hindered by the rectangular shape of its 13×3 pattern which generates many communications. As in the previous case, it is less efficient than its square-pattern counterpart using 36 nodes. G-BC consistently achieves the highest throughput of all three distributions for all matrix sizes and manages to reach the same per-node efficiency than BC with a 6×6 patterns while using roughly 10% more resources. As a conclusion, those experiments illustrate that G-BC distribution enables to efficiently use all available resources in parallel while not sacrificing on the number of inter-node communications thanks to its almost periodic and square pattern. Indeed, even if the G-BC pattern is significantly larger (22×23 for G-BC with $P = 23$ against 4×5 for BC for $P = 20$), it is in fact itself quasi-periodic since it is built from 20 quasi-copies of the 5×5 incomplete pattern $\mathcal{IP}$ defined in Section 5.4.1. The workload between the processors in the trailing matrix remains very well balanced, even if the pattern is larger. This allows to achieve higher global performance than BC distribution for specific values of the number of nodes P and in turn translates into faster global execution of the operation.

For the other test cases using $P = 31$ and 35 nodes, illustrated on Figures 5.8 and 5.9, the G-BC distribution performs as well as the BC distribution using exactly (for $P = 35$) or slightly fewer (for $P = 31$) nodes, with an almost square pattern. In the case $P = 31$, we can observe again that the BC distribution using all available nodes but laid out as a narrow shaped pattern of size 31×1 generates many more communications than the other methods and therefore achieves significantly inferior performance. Those results show that, in configurations where P is favorable to efficient BC distributions, G-BC can also be used indifferently without degrading the performance.

5.6.3 Results for Greedy ColRow & Matching

Table 5.1 summarizes the characteristics of the best patterns found using the GCR&M algorithm for the considered test cases and used for the experiments. It also provides the size of the SBC patterns to which it is compared. The key property $\bar{z}$ associated with each pattern, which corresponds to its cost $T(\mathbf{G})$, is shown along with the pattern size r. We can observe that the cost of each GCR&M pattern is either lower, for $P = 35$, or roughly similar, in all other cases, to the cost associated with the SBC patterns. Hence, the GCR&M distributions are expected to generate no more communications than the SBC ones. The size of the GCR&M patterns are relatively large, equal or almost equal to P for all cases except $P = 35$. The load balancing among nodes in those patterns, *i.e.* the number of cells allocated to each node, is perfect except in the case $P = 23$ where there is an imbalance of approximately 10% between the most and least loaded nodes.

Results for Cholesky factorization are quite similar to those obtained for LU factorization for the test cases $P = 31$ and 35. They can be observed on Figures 5.12 and 5.13. In those cases, the GCR&M distributions outperforms the SBC one in terms of raw performance and achieves very close performance per node for all the matrix sizes considered. Results for the BC distribution, for both cases, and the G-BC distribution, for the case $P = 31$ only, are also show on the figures in order to stress again the clear disadvantage of those distributions compared to the SBC and GCR&M distributions that are tailored for symmetric operations. Indeed, as they induce more communications when performing the Cholesky factorization, both BC and G-BC show significantly lower performance than SBC and GCR&M in the two test cases. This extends to the two new distributions the conclusions from the comparative results between BC and SBC presented in Section 2.3.4.

Results for the two others test cases are less clear. For $P = 39$, GCR&M distribution using all the nodes achieves roughly the same raw performance as SBC using only 36 nodes, as can be observed on Figure 5.14. However, the general trend seems to be that GCR&M performs better for larger matrix sizes; from $m \geqslant 200,000$, the difference of performance between the two methods becomes significant. The results for the test case with $P = 23$ nodes, illustrated on Figure 5.11, does not show any comparative advantage of GCR&M over SBC, although it makes uses of 2 additional nodes. The lower than expected performance of the GCR&M distribution may comes from a problem of load balancing: since its associated pattern features a small imbalance between nodes, as shown in Table 5.1, the greedy procedure, mentioned in the beginning of Section 5.5, used to transform the incomplete pattern into a complete distribution compensates the imbalance by allocating many tiles to the least loaded node. Those tiles correspond to the diagonal cells of the pattern. Therefore, the distribution obtained from this pattern may show a poor local load balancing, which translates into an imbalance in the course of the execution In turn, it can affect the overall performance of the distribution.

5.7 Conclusion

In this chapter, we investigated the possibility to extend the state-of-the-art data distributions for the parallel and distributed execution of linear algebra operations to any number

Symmetric Block Cyclic			Greedy ColRow & Matching				
P	r	$\bar{z}$	P	r	$\bar{z}$	min. load	max. load
21	7	6	23	22	6.045	19	21
28	8	7	31	31	7.065	30	30
32	8	8	35	15	7.4	6	6
36	9	8	39	27	7.926	18	18

Table 5.1: Sizes and characteristics of the SBC and GCR&M patterns used for the experiments; the SBC pattern for $P = 32$ is the basic version of size 8×8

of identical nodes. Indeed, in a parallel and distributed context, the classic Block Cyclic (BC) and recently developed Symmetric Block Cyclic (SBC) distributions show very good performance respectively for LU factorization and Cholesky factorization. However, the SBC pattern can only be defined for specific values of the number of available nodes P while the BC pattern obtains poor results regarding communications for many values of P.

To overcome this limitation, we developed two improved, but more irregular, distribution schemes that allow to perform those operations using all available nodes, regardless of their number P, and which feature the same quality as BC and SBC regarding the load balancing and the communication volume they generate. In the non-symmetric case, Generalized Block Cyclic (G-BC) is a pattern-based distribution whose pattern is perfectly balanced and features the same property as BC pattern when used with a square number of nodes, $P = p^2$, *i.e.* it minimizes the number of different nodes per row and column, reaching $\left\lceil \sqrt{P} \right\rceil$. Hence, it achieves explicit optimality regarding the total communication volume generated for LU factorization. In the symmetric case, we propose a randomized greedy heuristic, Greedy ColRow & Matching (GCR&M), which provides patterns for any number of available nodes P. The procedure is designed to search for perfectly balanced patterns which minimize the number of different nodes on all colrows, *i.e.* unions of row and column with the same index, which is the key property to reduce communications for Cholesky factorization. Sample results actually show that, for P up to a few hundreds, GCR&M algorithm can effortlessly provide patterns expected to generate as few or even fewer communications than SBC according to this criterion. These patterns can therefore be seen as a generalization of the SBC method. The irregular distributions provided by these two methods can be easily used under the task-based execution paradigm. An experimental evaluation carried out with `Chameleon` linear algebra library associated with `StarPU` as runtime system actually shows that G-BC and GCR&M achieve improved performance respectively compared to BC and SBC. In particular, they manage to efficiently use all available nodes in situations where the value of P forces BC and SBC distribution to either use fewer resources or use them all at the cost of extra communications. It leads to shorter running time in those situations while both G-BC and GCR&M remain competitive with BC and SBC in situations where the value of P is more favorable.

This work opens several interesting perspectives. The question of whether it is possible to find an explicit description of an efficient pattern in the symmetric case, instead of relying on a heuristic, remains open. It would also be interesting to assess how large a

pattern needs to be to obtain good communication efficiency, or to explore the trade-off between pattern size and communication efficiency, similarly to what has been attempted with the elaboration of BCE in Chapter 4. Note that, for GCR&M which is a randomized procedure, the computational cost of providing patterns is not a problem, since they do not need to be computed at each execution. On the contrary, one could imagine to provide in database containing, for each possible value of P, a very efficient pattern for the symmetric case. Another avenue of research could be to extend those results to the case of heterogeneous nodes, as it seems that both methods can be adapted to this case without substantial modifications.

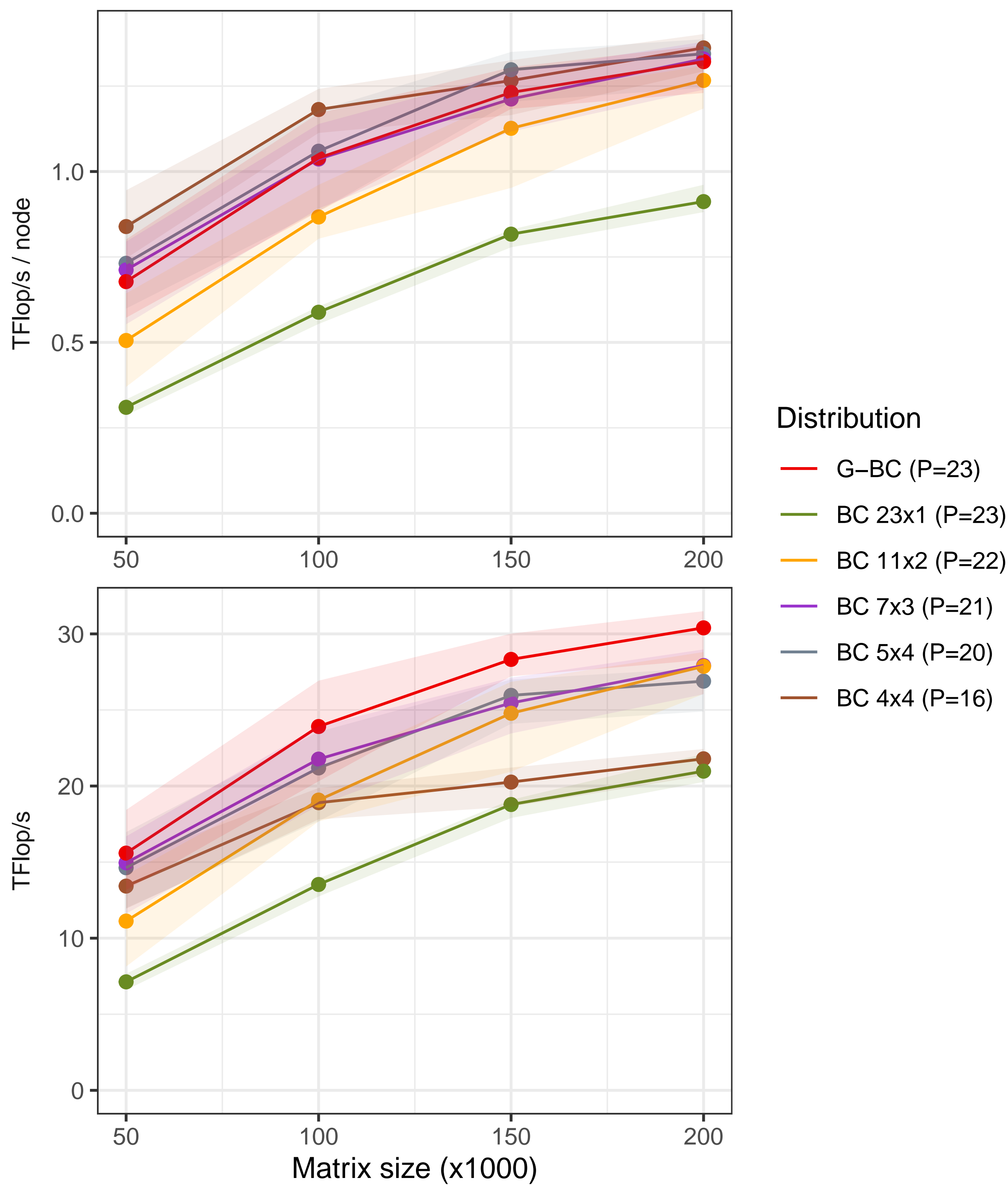


Figure 5.7: Performance results for LU factorization using $P = 23$ nodes

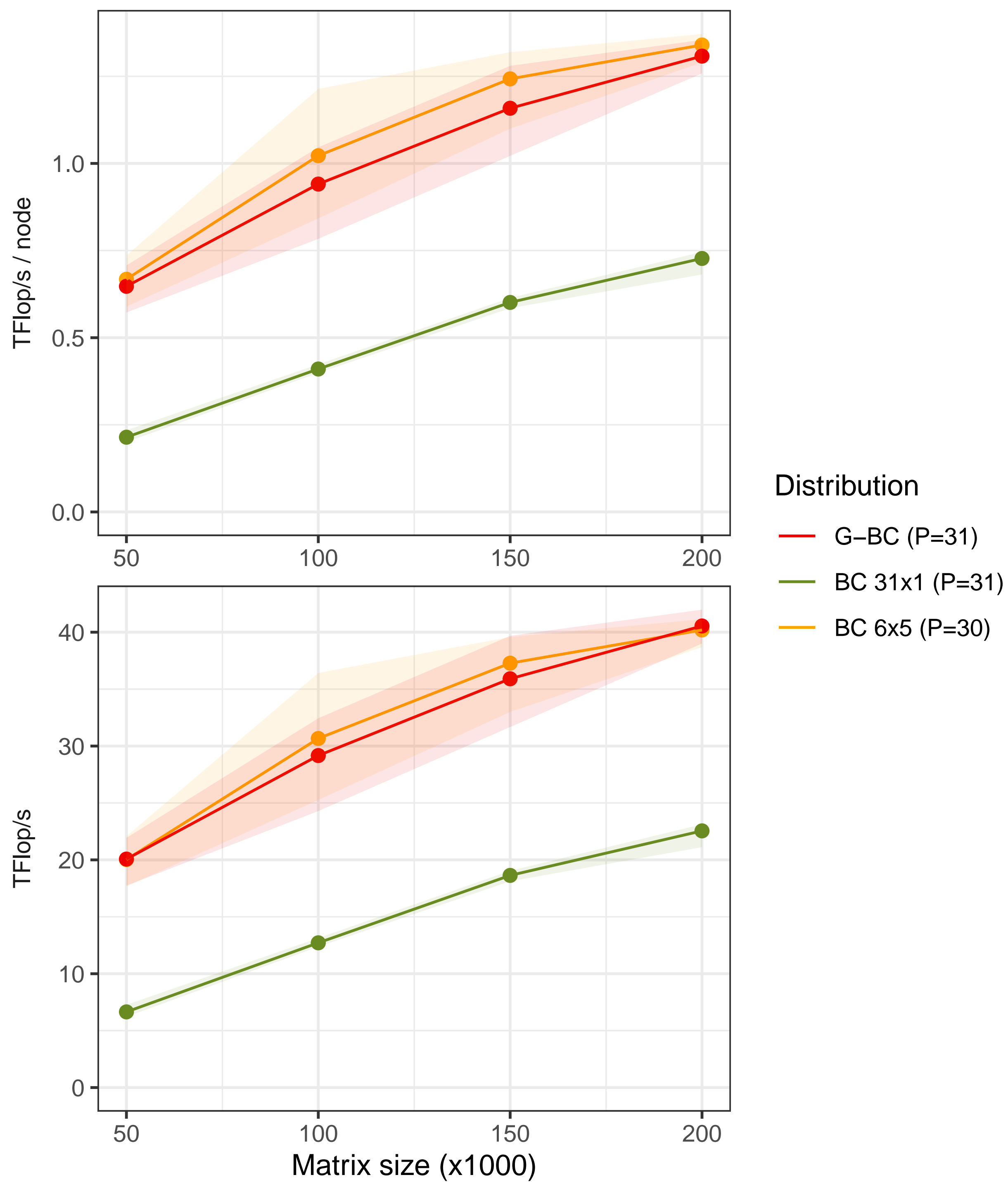


Figure 5.8: Performance results for LU factorization using $P = 31$ nodes

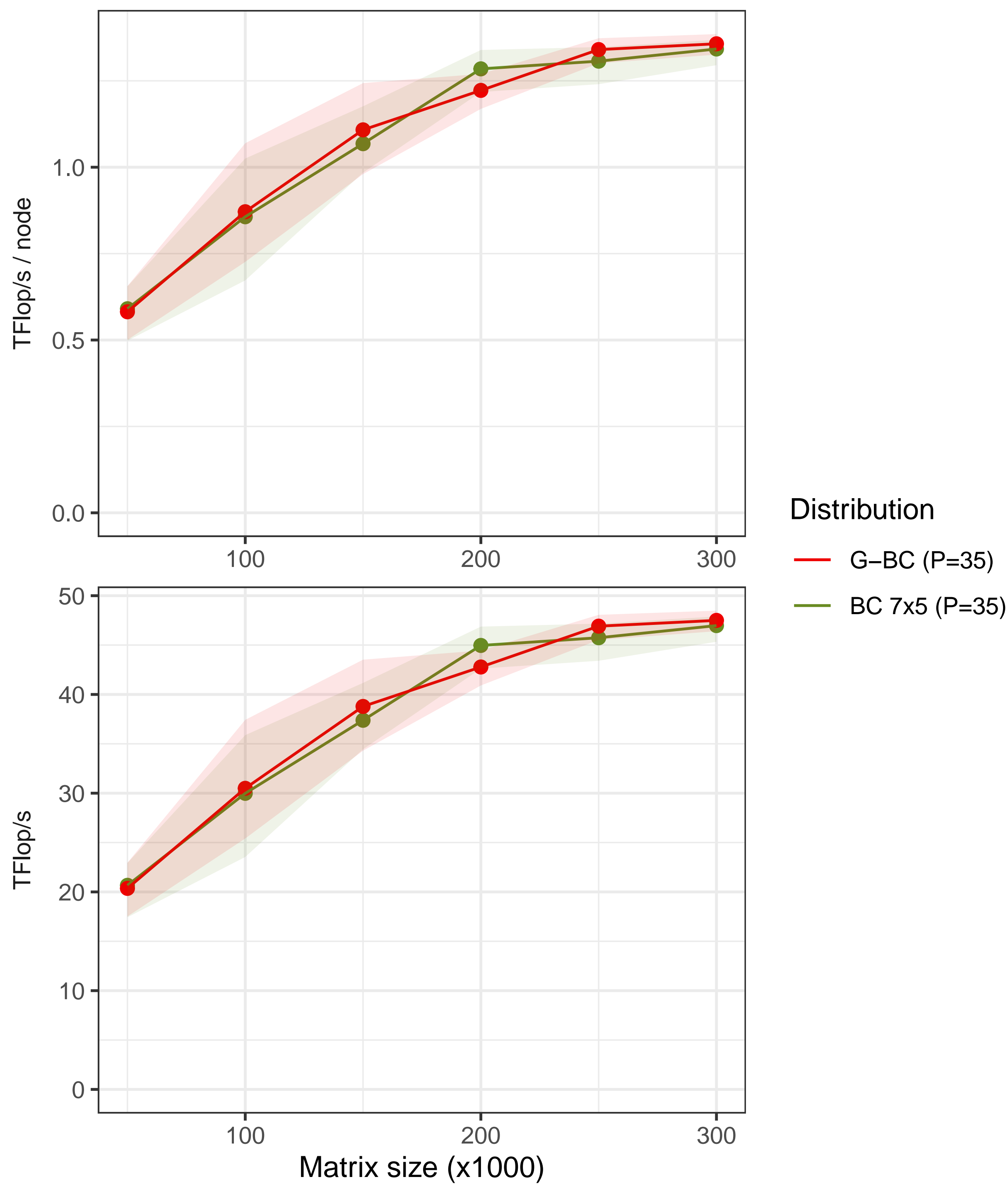


Figure 5.9: Performance results for LU factorization using $P = 35$ nodes

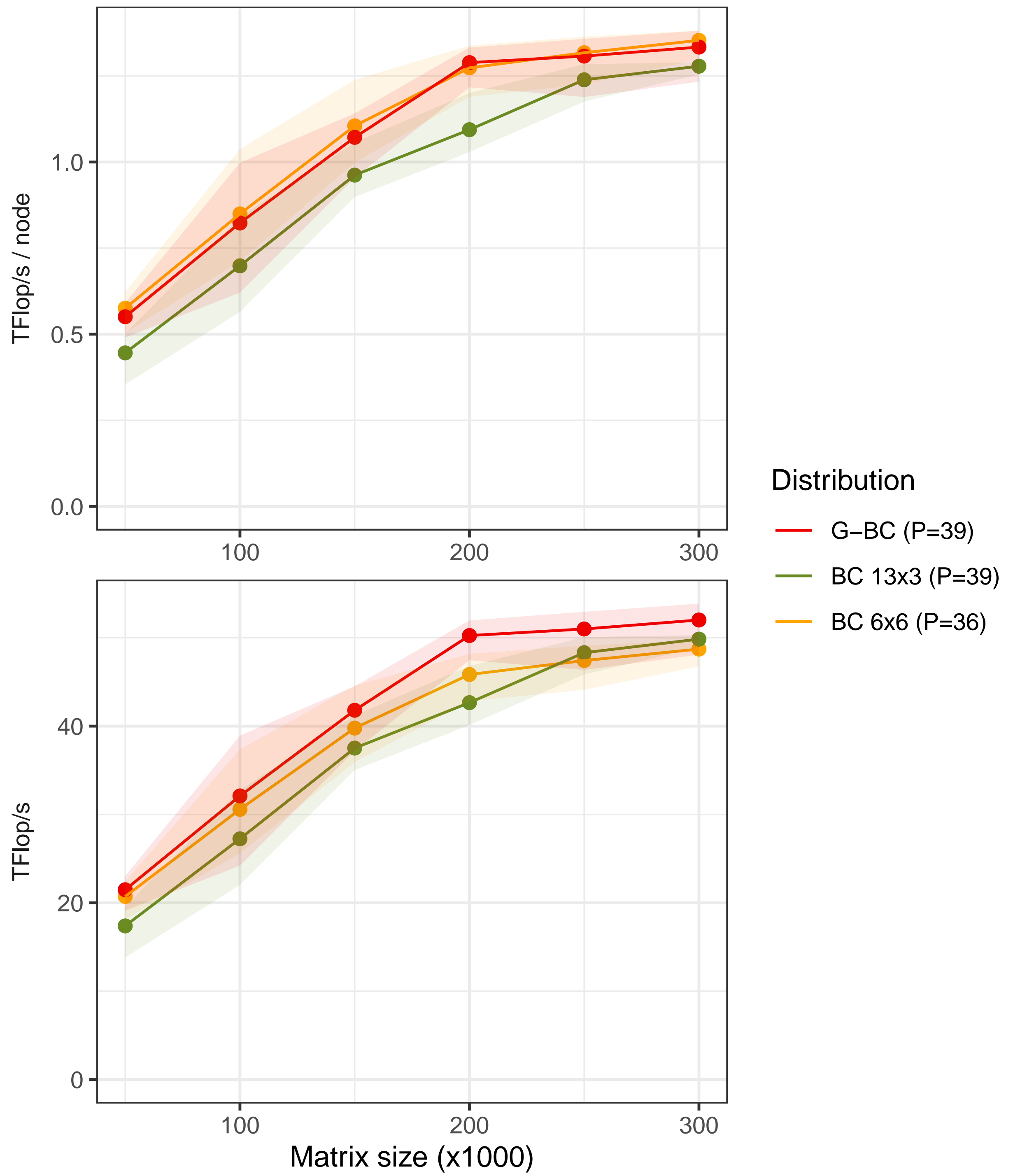


Figure 5.10: Performance results for LU factorization using $P = 39$ nodes

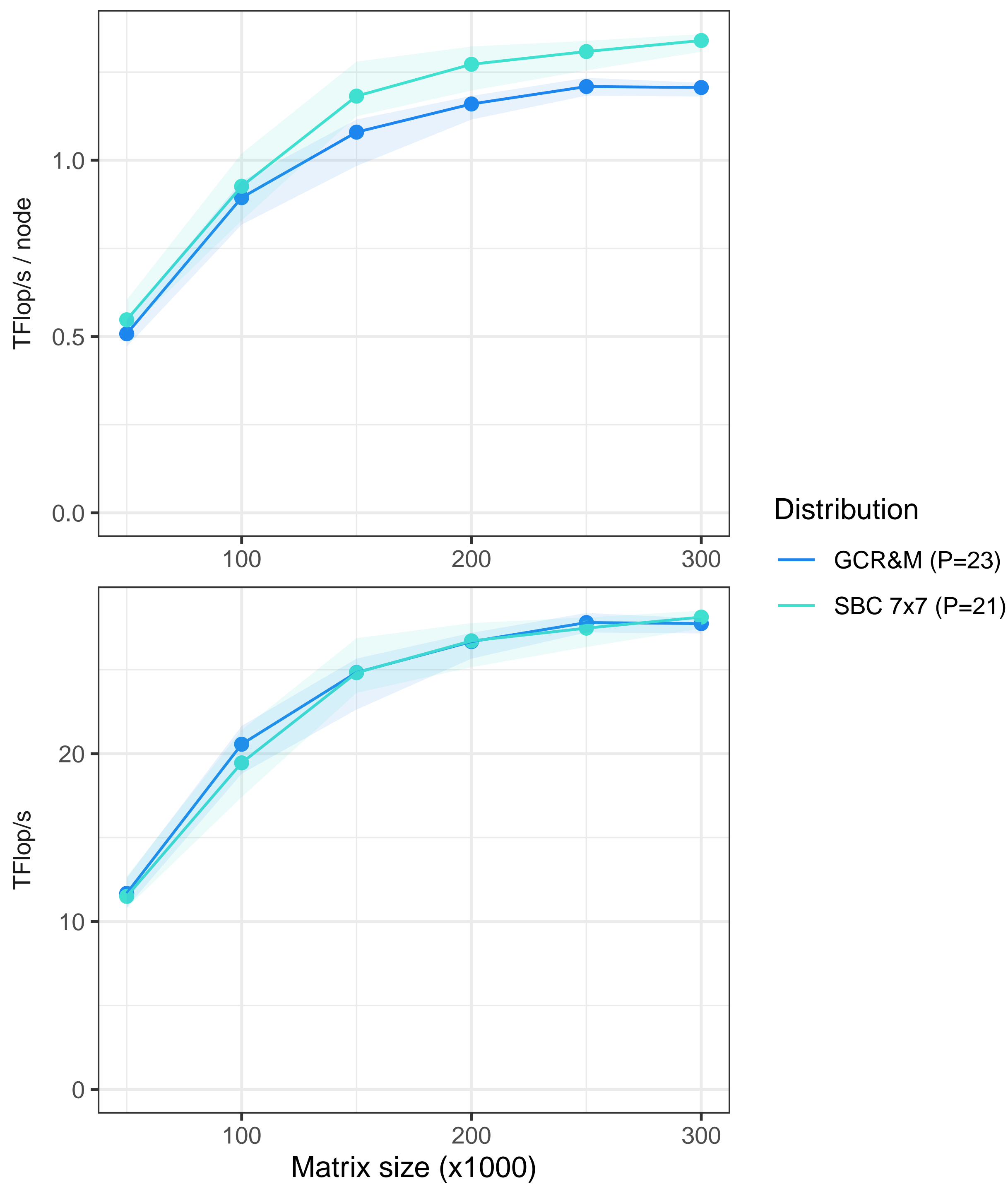


Figure 5.11: Performance results for Cholesky factorization using $P = 23$ nodes

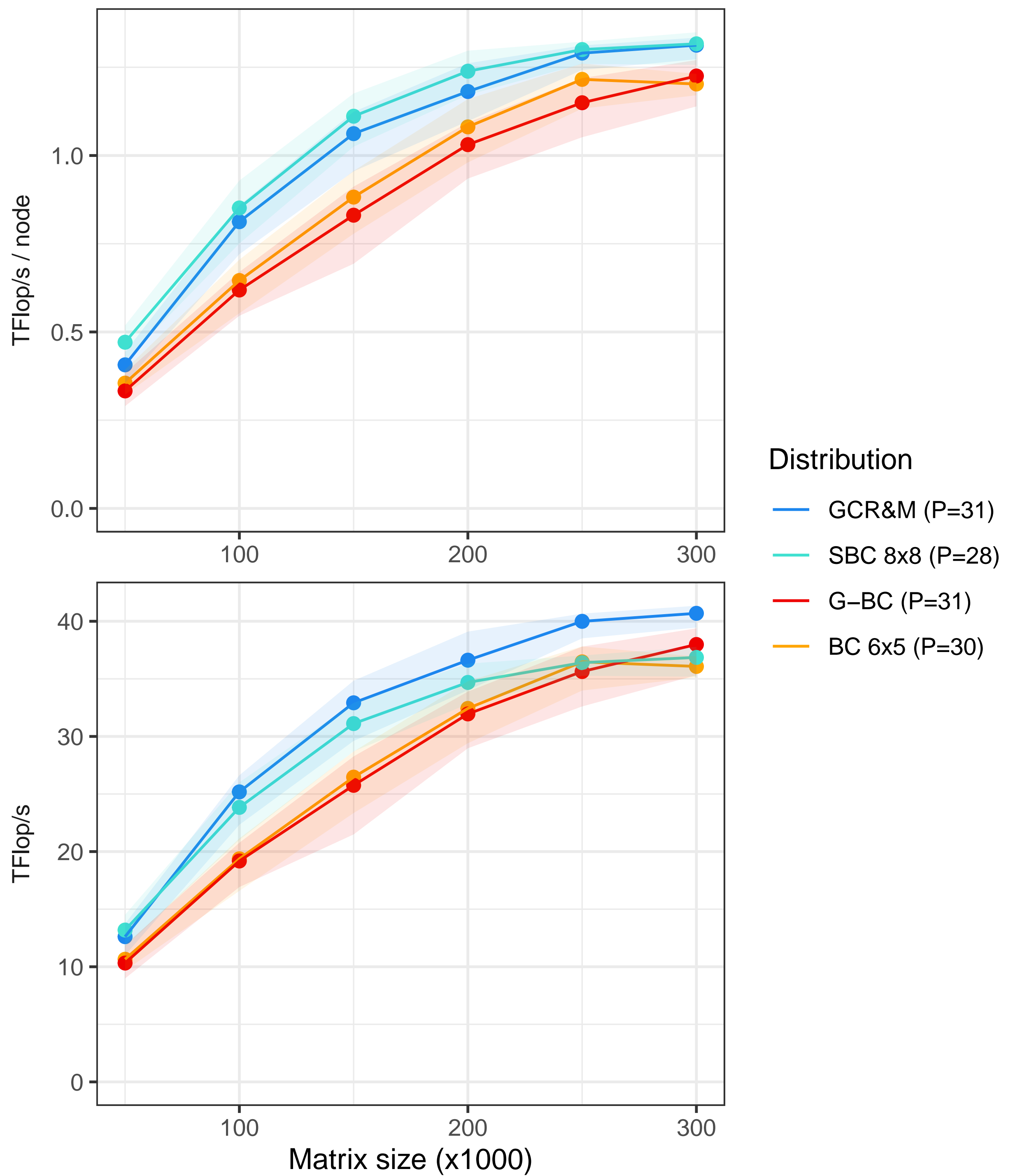


Figure 5.12: Performance results for Cholesky factorization using $P = 31$ nodes

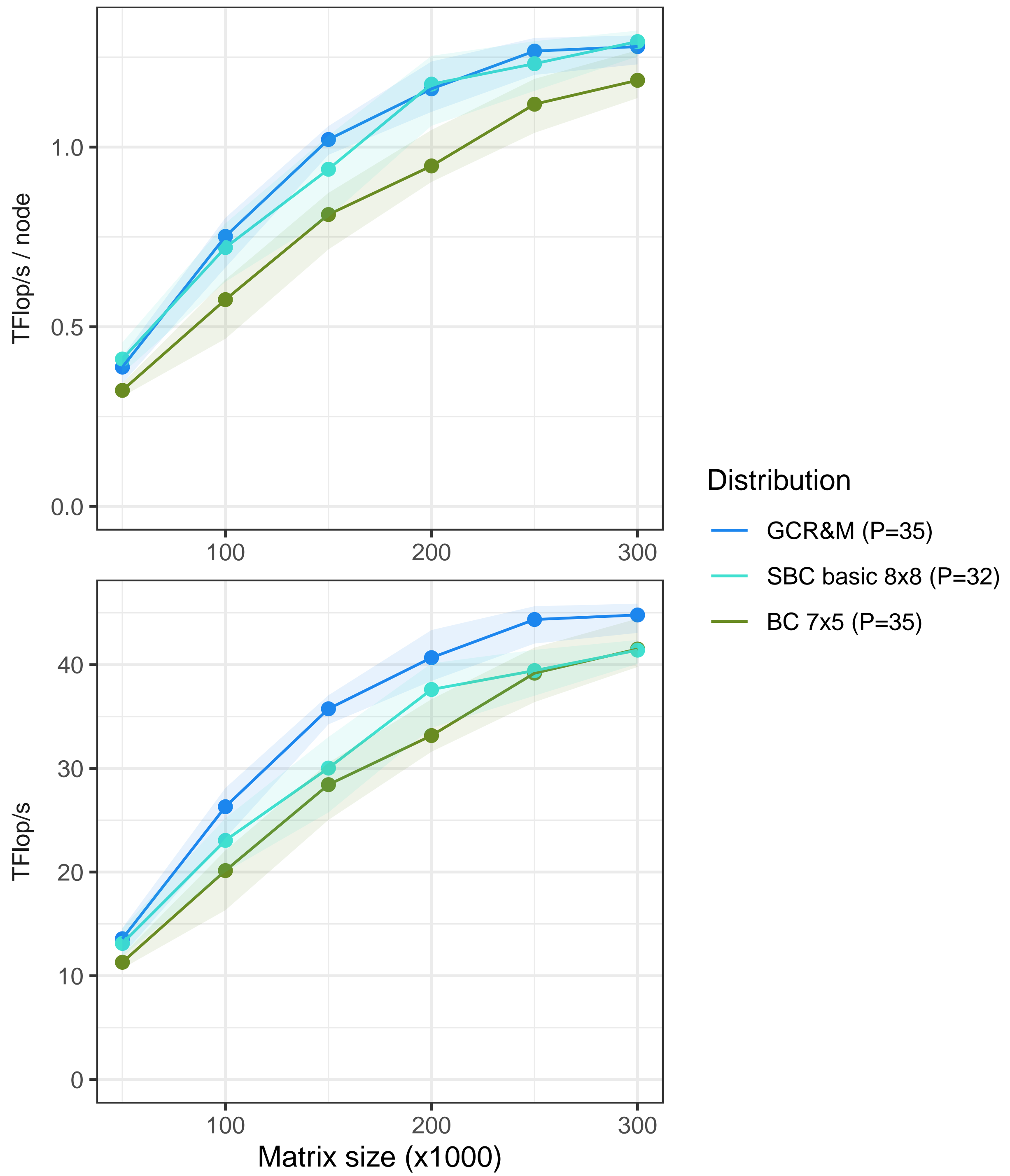


Figure 5.13: Performance results for Cholesky factorization using $P = 35$ nodes

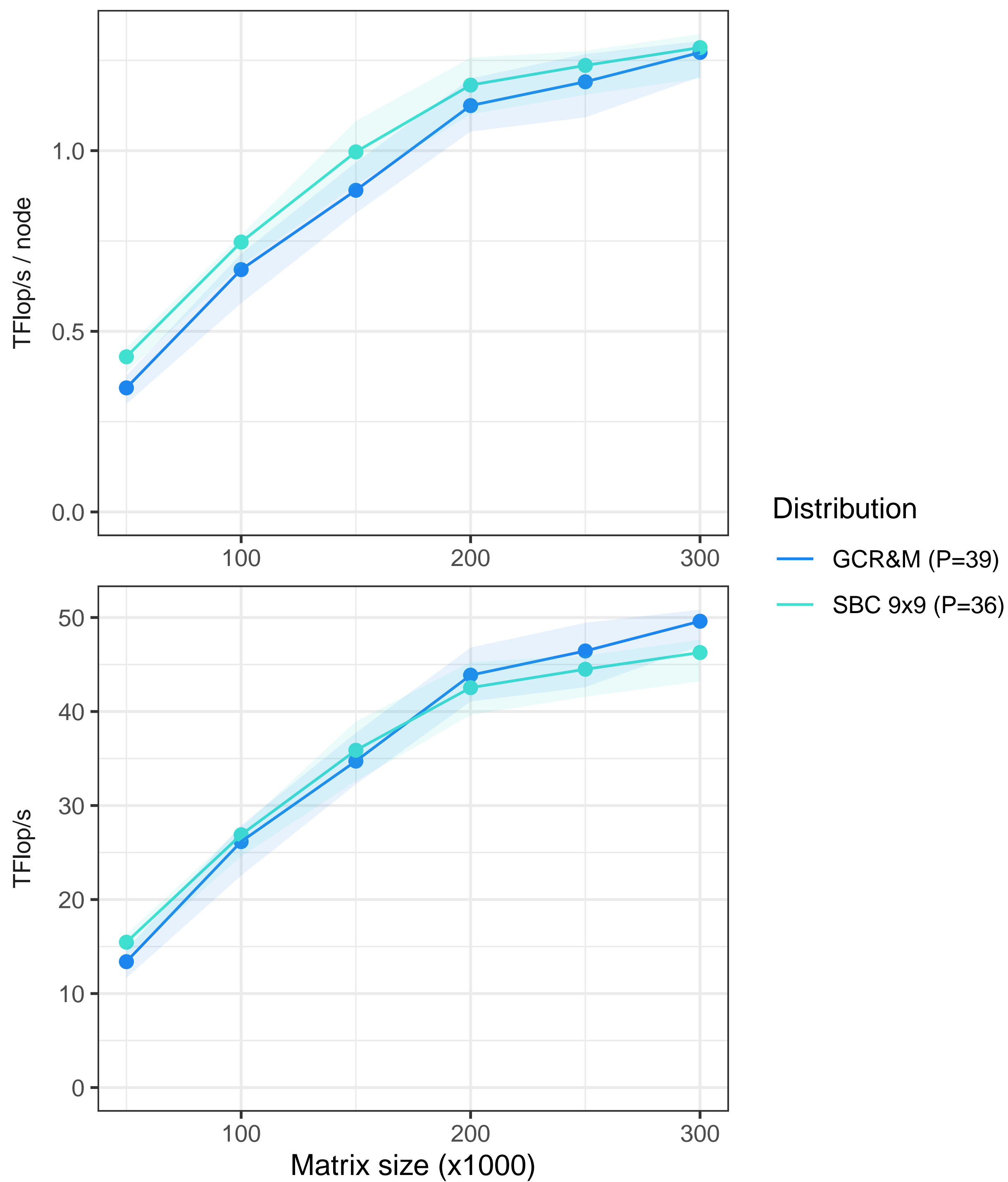


Figure 5.14: Performance results for Cholesky factorization using $P = 39$ nodes

Conclusion and Perspectives

Linear algebra applications are central for the solution of many large scale scientific problems that cannot be handled analytically. Performing linear algebra operations in parallel on distributed platforms is a necessity to provide solutions in a reasonable time. As the scale and the complexity of computing platforms grow, so is the difficulty of using efficiently available resources. The recent development of the task-based execution model and associated runtime systems is a paradigm shift for such applications as it provides a level of abstraction for the development of efficient data distributions.

In this work, we consider the parallel and distributed execution of four linear algebra operations: matrix multiplication, symmetric rank-k update, LU and Cholesky factorization, in the context of task-based model. We explore the possibility offered by runtime systems to seamlessly perform those operations using any arbitrary data distribution, focusing on static distributions. As many existing linear algebra libraries implementing the task-based execution model and relying on runtime systems still make almost exclusive use of the traditional 2D Block Cyclic (BC) distribution, taking advantage of such possibility appears as a relevant way for potential improvements.

In Chapters 2 and 3, we focus on symmetric operations, symmetric rank-k update and Cholesky factorization, performed on dense matrices.

The parallel and distributed execution of Cholesky factorization using P identical nodes is considered in Chapter 2. For this operation, we design a static pattern-based distribution, Symmetric Block Cyclic (SBC), aimed at reducing inter-node communications. By arranging the nodes in a specific layout which takes advantage of the symmetry of the input matrix, the SBC distribution generates a factor of $\sqrt{2}$ fewer communications than plain 2D BC distribution when performing Cholesky factorization. Moreover, experimental results using the `Chameleon` library associated with the `StarPU` runtime system show that the reduction of communication directly translates into faster executions of the operation. This advocates the use of the task-based execution model as a mean to achieve higher performance via adapted data distributions. The easy integration of SBC distribution in `Chameleon` library also illustrates the advantage of the task-based model to investigate and develop such sophisticated distributions. The integration of SBC method into multi-operations workflow and reallocation from and to BC are also tested, along with 2.5D and 3D variants.

Chapter 3 provides a theoretical foundation for the *ad hoc* SBC distribution. It presents original results regarding the communications volume required for symmetric rank-k update and Cholesky factorization in the out-of-core setting, *i.e.* using a single

resource which fetches and stores data from a distant memory. New lower bounds for the communication volume are proven for both operations that improve previous results by a factor of $\sqrt{2}$. In addition, two optimal algorithms are developed: Triangle Block SYRK for symmetric rank-k update and Large Block Cholesky for Cholesky factorization. Extending the idea underlying the SBC distribution, they are both designed to take advantage of the symmetry of the input matrix to maximize the operational intensity. Thereby, each generates a communication volume that matches the associated lower bound for the operation.

Chapters 4 and 5 explore two different research directions to extend the techniques elaborated in the previous chapters.

Chapter 4 deals with the problem of data distribution for the parallel and distributed execution of non-symmetric operations, matrix multiplication and LU factorization, in the case of heterogeneous tasks. Such heterogeneity arises when dealing with compressed matrices; in this case we considered the regular Block Low Rank (BLR) format. In this context, we develop two heuristic methods that provide data distributions adapted to the set of heterogeneous tasks and which aim at balancing the workload among nodes while controlling the number of communications: (i) Block Cyclic Extended (BCE) is a simple yet robust extension of the classic Block Cyclic distribution; (ii) Random Subsets (RSB) is a greedy procedure which generates non-regular distributions. Results of simulated executions using those two methods show that they perform significantly better than 2D BC in terms of load balancing for the majority of the tested configurations. This in turn implies shorter total running time, although this must be tempered for RSB, as its irregularity induces some inefficiency for LU factorization.

In Chapter 5, we consider again the parallel and distributed executions of LU and Cholesky factorizations in the dense case. We investigate techniques to directly extend and improve BC and SBC strategies for any number of nodes. We develop two pattern-based data distribution schemes, Generalized Block Cyclic (G-BC) and Greedy ColRow & Matching (GCR&M) which preserve the quality of BC and SBC methods regarding the load balancing and the reduction of communications and which can provide solutions for any number of nodes. While G-BC is defined analytically and is proven optimal regarding the communication volume generated for LU factorization, GCR&M is a randomized greedy algorithm. According to the results of sample experiments performed using `Chameleon` and `StarPU`, both G-BC and GCR&M show improved performance in terms of total running time compared respectively to BC and SBC. Overall, they perform as well or even slightly better than BC and SBC while being able to maintain the same performance on any number of resources.

The techniques developed in this work and the results they allowed to obtain act as a practical argument in favor of the extensive usage of the task-based model ability to perform parallel and distributed operations according to any data distribution. Indeed, the data distributions presented in this work clearly allow performance gains over the classic and widely used BC distribution. Nevertheless, they are easy to implement in the `Chameleon` library and automatically used by the `StarPU` runtime system to perform the required operations. It therefore definitely shows that the task-based model is currently the best tool for the development of efficient and versatile linear algebra libraries able to

handle the specificities of each operation.

This opens wide perspectives for additional research as the results obtained in this work let us expect that there remains a lot of potential performance gains for linear algebra applications that are still unexploited.

Regarding the theoretical aspects, existing communication lower bounds may be improved by taking into account more carefully the many possible ways of data reuse. Our results on the lower bounds for symmetric rank-k update and Cholesky factorization in the out-of-core setting from Chapter 3 illustrates that such data reuse may not be captured even by sophisticated techniques. One first step towards the integration of our theoretical results into a broader formal framework would be the adaptation of the `IOLB` tool, proposed by Olivry *et al.* in [62], so that it is able to handle the type of data reuse that we have been exploiting using *ad hoc* techniques. Such theoretical results also show the importance of understanding the underlying key aspects of the operations to help guiding the elaboration of efficient solutions.

For parallel and distributed linear algebra operations in the dense case, the methods presented in Chapters 2 and 5 open the door to a large exploration of potential techniques to provide efficient communication-avoiding distributions for all existing operations. The very good results obtained by the SBC, G-BC and GCR&M strategies are very encouraging. They suggest that, for each operation, specifically tailored designs of distribution are likely to lead to significant performance gains. It is therefore worthy to question the quality of already existing strategies, especially BC, for each specific linear algebra application. Furthermore, the ease of implementation of solutions as irregular as those we proposed, allows to envision the possibility of combining highly tuned data distributions and reallocation strategies for very complex applications featuring multiple operations.

The core of this work has been targeted toward communications reduction for parallel and distributed operations in the context of dense matrices. The introduction of heterogeneities of tasks and/or resources to the problem adds a complementary objective of load balancing in the search of efficient data distributions, which then becomes much harder. Our attempt to tackle such problem in the case of compressed matrices in Chapter 4 illustrates the difficulty of designing data distribution schemes that ensure both load balancing and communication reduction at once. Future research in that direction should probably be oriented towards the case of sparse matrices because of its practical importance in the linear algebra community and in the industry. Although dealing with multiple objectives is very challenging, a wide range of heuristic methods and their combination can be tried on such problems. Hence, we can expect that extensions of the type of resolution methods that we explored could be efficiently adapted and provide solutions that allow to achieve significant performance improvements.

Bibliography

[1] Sameh Abdulah, Hatem Ltaief, Ying Sun, Marc G. Genton, and David E. Keyes. Parallel Approximation of the Maximum Likelihood Estimation for the Prediction of Large-Scale Geostatistics Simulations. In *2018 IEEE International Conference on Cluster Computing (CLUSTER)*, pages 98–108, 2018.

[2] Sameh Abdulah, Hatem Ltaief, Ying Sun, Marc G. Genton, and David E. Keyes. Geostatistical Modeling and Prediction Using Mixed Precision Tile Cholesky Factorization. *HIPC*, pages 152–162, 2019.

[3] Emmanuel Agullo, Cédric Augonnet, Jack Dongarra, Hatem Ltaief, Raymond Namyst, Samuel Thibault, and Stanimire Tomov. Faster, Cheaper, Better – a Hybridization Methodology to Develop Linear Algebra Software for GPUs. In Wen mei W. Hwu, editor, *GPU Computing Gems*, volume 2. Morgan Kaufmann, September 2010.

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

[5] Emmanuel Agullo, Olivier Beaumont, Lionel Eyraud-Dubois, Julien Herrmann, Suraj Kumar, Loris Marchal, and Samuel Thibault. Bridging the gap between performance and bounds of cholesky factorization on heterogeneous platforms. In *IPDPSW*, pages 34–45. IEEE, 2015.

[6] Noha Al-Harthi, Rabab Alomairy, Kadir Akbudak, Rui Chen, Hatem Ltaief, Hakan Bagci, and David Keyes. *Solving Acoustic Boundary Integral Equations Using High Performance Tile Low-Rank LU Factorization*, pages 209–229. 06 2020.

[7] Sivaram Ambikasaran and Eric Darve. An $O(N \log N)$ Fast Direct Solver for Partial Hierarchically Semi-Separable Matrices. *Journal of Scientific Computing*, 57(3):477–501, 2013.

[8] Patrick Amestoy, Cleve Ashcraft, Olivier Boiteau, Alfredo Buttari, Jean-Yves L'Excellent, and Clément Weisbecker. Improving multifrontal methods by means of block low-rank representations. *SIAM Journal on Scientific Computing*, 37(3):A1451–A1474, 2015.

[9] Amirhossein Aminfar, Sivaram Ambikasaran, and Eric Darve. A fast block low-rank dense solver with applications to finite-element matrices. *Journal of Computational Physics*, 304:170–188, 2016.

[10] Barry C Arnold, Narayanaswamy Balakrishnan, and Haikady Navada Nagaraja. *A first course in order statistics*. SIAM, 2008.

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

[12] Cédric Augonnet, David Goudin, Matthieu Kuhn, Xavier Lacoste, Raymond Namyst, and Pierre Ramet. A hierarchical fast direct solver for distributed memory machines with manycore nodes. Research report, CEA;Total; Université de Bordeaux, October 2019.

[13] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Communication-optimal parallel and sequential Cholesky decomposition. *SIAM Journal on Scientific Computing*, 32(6):3495–3523, 2010.

[14] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. Minimizing Communication in Numerical Linear Algebra. *SIAM J. Matrix Anal. Appl.*, 32:866–901, 2011.

[15] Olivier Beaumont, Brett A Becker, Ashley Deflumere, Lionel Eyraud-Dubois, Thomas Lambert, and Alexey Lastovetsky. Recent advances in matrix partitioning for parallel computing on heterogeneous platforms. *IEEE Transactions on Parallel and Distributed Systems*, 30(1):218–229, 2018.

[16] Olivier Beaumont, Vincent Boudet, Fabrice Rastello, Yves Robert, et al. Partitioning a square into rectangles: NP-completeness and approximation algorithms. *Algorithmica*, 34(3):217–239, 2002.

[17] Olivier Beaumont, Lionel Eyraud-Dubois, and Thomas Lambert. A New Approximation Algorithm for Matrix Partitioning in Presence of Strongly Heterogeneous Processors. In *IPDPS*, 2016.

[18] Brett A Becker and Alexey Lastovetsky. Towards data partitioning for parallel computing on three interconnected clusters. In *ISPDC'07*, pages 39–39. IEEE, 2007.

[19] Natacha Béreux. Out-of-Core Implementations of Cholesky Factorization: Loop-Based versus Recursive Algorithms. *SIAM Journal on Matrix Analysis and Applications*, 30(4):1302–1319, 2009.

[20] David Bergman, Carlos Cardonha, and Saharnaz Mehrani. Binary decision diagrams for bin packing with minimum color fragmentation. In *CPAIOR*, pages 57–66. Springer, 2019.

[21] Paul Beziau. *Rééquilibrage de charge dans les solveurs hiérarchiques pour machines massivement parallèles.* Theses, Université de Bordeaux, December 2021.

[22] L. S. Blackford, J. Choi, A. Cleary, E. D'Azevedo, J. Demmel, I. Dhillon, J. Dongarra, S. Hammarling, G. Henry, A. Petitet, K. Stanley, D. Walker, and R. C. Whaley. *ScaLAPACK Users' Guide.* Society for Industrial and Applied Mathematics, 1997.

[23] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, A. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible Development of Dense Linear Algebra Algorithms on Massively Parallel Architectures with DPLASMA. In *IEEE International Symposium on Parallel and Distributed Processing Workshops*, pages 1432–1441, May 2011.

[24] G. Bosilca, A. Bouteiller, A. Danalis, T. Herault, P. Lemarinier, and J. Dongarra. DAGuE: A generic distributed dag engine for high performance computing. *Parallel Computing*, 38((1-2)):37–51, 2012.

[25] George Bosilca, Aurelien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack J. Dongarra. PaRSEC: Exploiting Heterogeneity to Enhance Scalability. *Computing in Science and Engineering*, 15(6):36–45, 2013.

[26] Herm Jan Brascamp and Elliott H. Lieb. Best Constants in Young's Inequality, Its Converse, and Its Generalization to More than Three Functions. *Advances in Mathematics*, 20:151–173, 1976.

[27] Qinglei Cao, Yu Pei, Thomas Herault, Kadir Akbudak, Aleksandr Mikhalev, George Bosilca, Hatem Ltaief, David Keyes, and Jack Dongarra. Performance Analysis of Tile Low-Rank Cholesky Factorization Using PaRSEC Instrumentation Tools. In *ProTools*. ACM, 2019.

[28] Rocío Carratalá-Sáez, Mathieu Faverge, Grégoire Pichon, Guillaume Sylvand, and Enrique Quintana-Ortí. Tiled Algorithms for Efficient Task-Parallel H-Matrix Solvers. In *PDSEC*, 2020.

[29] Henri Casanova, Arnaud Legrand, and Martin Quinson. Simgrid: A generic framework for large-scale distributed experiments. In *Tenth International Conference on Computer Modeling and Simulation (uksim 2008)*, pages 126–131. IEEE, 2008.

[30] E. Chan, F. G. Van Zee, P. Bientinesi, E. S. Quintana-Ortí, G. Quintana-Ortí, and R. van de Geijn. SuperMatrix: a multithreaded runtime scheduling system for algorithms-by-blocks. In *PPoPP '08*, pages 123–132. ACM, 2008.

[31] Michael Christ, James Demmel, Nicholas Knight, Thomas Scanlon, and Katherine A. Yelick. Communication lower bounds and optimal algorithms for programs that reference arrays - part 1. Technical Report UCB/EECS-2013-61, EECS Department, University of California, Berkeley, May 2013.

[32] E.G. Coffman, G.N. Frederickson, and G.S. Lueker. Probabilistic analysis of the LPT processor scheduling heuristic. In *Deterministic and stochastic scheduling*, pages 319–331. Springer, 1982.

[33] Terry Cojean, Abdou Guermouche, Andra Hugo, Raymond Namyst, and Pierre-André Wacrenier. Resource aggregation for task-based Cholesky Factorization on top of modern architectures. *Parallel Computing*, 83:73–92, 2019.

[34] A. DeFlumere and A. Lastovetsky. Optimal data partitioning shape for matrix multiplication on three fully connected heterogeneous processors. In *HeteroPar*, Porto, Portugal, 25 August 2014.

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

[36] Leah Epstein, Csanád Imreh, and Asaf Levin. Class constrained bin packing revisited. *Theoretical Computer Science*, 2010.

[37] Mathieu Faverge, Grégoire Pichon, Pierre Ramet, and Jean Roman. On the use of H-Matrix Arithmetic in PaStiX: a Preliminary Study. In *Workshop on Fast Solvers, Toulouse, France*, 2015.

[38] John Friedlander and Henryk Iwaniec. *Opera de cribro*, volume 57 of *American Mathematical Society Colloquium Publications*. American Mathematical Society, Providence, RI, 2010.

[39] Armin Fügenschuh, Konstanty Junosza-Szaniawski, and Zbigniew Lonc. Exact and approximation algorithms for a soft rectangle packing problem. *Optimization*, 63(11):1637–1663, 2014.

[40] Mark Gates, Jakub Kurzak, Ali Charara, Asim YarKhan, and Jack Dongarra. Slate: Design of a modern distributed and accelerated linear algebra library. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC19)*, Denver, CO, 2019-11 2019. ACM, ACM.

[41] Evangelos Georganas, Jorge González-Domínguez, 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*, pages 1–11. IEEE, 2012.

[42] Gene H Golub and Charles F Van Loan. *Matrix computations*, volume 3. JHU press, 2012.

[43] Ron Graham, Eugene L. Lawler, Jan Karel Lenstra, and Alexander H. G. Rinnooy Kan. Optimization and approximation in deterministic sequencing and scheduling: a survey. *Annals of discrete mathematics*, 5:287–326, 1977.

[44] Lars Grasedyck and Wolfgang Hackbusch. Construction and arithmetics of H-matrices. *Computing*, 70(4):295–334, 2003.

[45] Michelangelo Grigni and Fredrik Manne. On the complexity of the generalized block distribution. In *International Workshop on Parallel Algorithms for Irregularly Structured Problems*, pages 319–326. Springer, 1996.

[46] Laura Grigori, James W. Demmel, and Hua Xiang. Communication Avoiding Gaussian elimination. In *SC '08: Proceedings of the 2008 ACM/IEEE Conference on Supercomputing*, pages 1–12, 2008.

[47] John A. Gunnels, Fred G. Gustavson, Greg M. Henry, and Robert A. van de Geijn. FLAME: Formal Linear Algebra Methods Environment. *ACM Trans. Math. Softw.*, 27(4):422–455, December 2001.

[48] Wolfgang Hackbusch. A sparse matrix arithmetic based on H-matrices. *Computing*, 62(2):89–108, 1999.

[49] Pascal Hénon, Pierre Ramet, and Jean Roman. PaStiX: A High-Performance Parallel Direct Solver for Sparse Symmetric Definite Systems. *Parallel Computing*, 28(2):301–321, 2002.

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

[51] Akihiro Ida. Lattice H-matrices on distributed-memory systems. In *2018 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*, pages 389–398. IEEE, 2018.

[52] Dror Irony and Sivan Toledo. Trading Replication for Communication in Parallel Distributed-Memory Dense Solvers. *Parallel Processing Letters*, 12(01):79–94, 2002.

[53] Dror Irony, Sivan Toledo, and Alexander Tiskin. Communication Lower Bounds for Distributed-memory Matrix Multiplication. *J. Parallel Distrib. Comput.*, 64(9):1017–1026, September 2004.

[54] Klaus Jansen, Alexandra Lassota, and Marten Maack. Approximation Algorithms for Scheduling with Class Constraints. *arXiv preprint arXiv:1909.11970*, 2019.

[55] Alexey Kalinov and Alexey Lastovetsky. Heterogeneous distribution of computations solving linear algebra problems on networks of heterogeneous computers. *Journal of Parallel and Distributed Computing*, 61(4):520–535, 2001.

[56] Grzegorz Kwasniewski, Tal Ben-Nun, Alexandros Nikolaos Ziogas, Timo Schneider, Maciej Besta, and Torsten Hoefler. On the parallel I/O optimality of linear algebra kernels: Near-optimal LU factorization. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2021.

[57] 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, New York, NY, USA, 2021. Association for Computing Machinery.

[58] L. H. Loomis and H. Whitney. An inequality related to the isoperimetric inequality. *Bulletin of the American Mathematical Society*, 55(10):961 – 962, 1949.

[59] Tania Malik and Alexey Lastovetsky. Towards optimal matrix partitioning for data parallel computing on a hybrid heterogeneous server. *IEEE Access*, 9:17229–17244, 2021.

[60] Théo Mary. *Block Low-Rank multifrontal solvers: complexity, performance, and scalability.* PhD thesis, 2017.

[61] Hiroshi Nagamochi and Yuusuke Abe. An approximation algorithm for dissecting a rectangle into rectangles with specified areas. *Discrete applied mathematics*, 155(4):523–537, 2007.

[62] Auguste Olivry, Julien Langou, Louis-Noël Pouchet, P Sadayappan, and Fabrice Rastello. Automated derivation of parametric data movement lower bounds for affine programs. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 808–822, 2020.

[63] Grégoire Pichon, Eric Darve, Mathieu Faverge, Pierre Ramet, and Jean Roman. Sparse supernodal solver using block low-rank compression: Design, performance and analysis. *Journal of computational science*, 27:255–270, 2018.

[64] Jack Poulson, Bryan Marker, Robert A. van de Geijn, Jeff R. Hammond, and Nichols A. Romero. Elemental: A New Framework for Distributed Memory Dense Matrix Computations. *ACM Trans. Math. Softw.*, 39(2), feb 2013.

[65] Hadi Pouransari, Pieter Coulier, and Eric Darve. Fast hierarchical solvers for sparse matrices using extended sparsification and low-rank approximation. *SIAM Journal on Scientific Computing*, 39(3):A797–A830, 2017.

[66] E. S. Quintana-Ortí, G. Quintana-Ortí, R. A. van de Geijn, F. G. Van Zee, and E. Chan. Programming matrix algorithms-by-blocks for thread-level parallelism. volume 36, 2009.

[67] Edgar Solomonik, Erin Carson, Nicholas Knight, and James Demmel. Trade-Offs Between Synchronization, Communication, and Computation in Parallel Linear Algebra Computations. *ACM Trans. Parallel Comput.*, 3(1), jan 2017.

[68] Edgar Solomonik and James Demmel. Communication-optimal parallel 2.5D matrix multiplication and LU factorization algorithms. In *EuroPar*. Springer, 2011.

[69] Zhang Xianyi, Wang Qian, and Zhang Yunquan. Model-driven level 3 blas performance optimization on loongson 3a processor. In *2012 IEEE 18th International Conference on Parallel and Distributed Systems*, pages 684–691, 2012.

[70] Ichitaro Yamazaki, Akihiro Ida, Rio Yokota, and Jack Dongarra. Distributed-memory lattice H-matrix factorization. *The International Journal of High Performance Computing Applications*, 33(5):1046–1063, 2019.

[71] Abdurrahman Yaşar and Ümit V Çatalyürek. Heuristics for Symmetric Rectilinear Matrix Partitioning. *arXiv preprint arXiv:1909.12209*, 2019.

[72] Intel MKL webpage: `https://www.intel.com/content/www/us/en/developer/tools/oneapi/onemkl.html`.

[73] Netlib webpage: `https://netlib.org`.

[74] Openblas webpage: `http://www.openblas.net/`.

[75] PLAFRIM webpage: `https://www.plafrim.fr`.

[76] The Scheduling Zoo webpage: `http://schedulingzoo.lip6.fr`.