Factorisation incomplète et résolution de systèmes triangulaires pour des machines exploitant un parallélisme à grain fin

Aboul-Karim Mohamed El Maarouf

▶ **To cite this version:**

Aboul-Karim Mohamed El Maarouf. Factorisation incomplète et résolution de systèmes triangulaires pour des machines exploitant un parallélisme à grain fin. Autre [cs.OH]. Université de Bordeaux, 2023. Français. NNT : 2023BORD0051 . tel-04429547

HAL Id: tel-04429547
https://theses.hal.science/tel-04429547v1

Submitted on 31 Jan 2024

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

ECOLE DOCTORALE MATHEMATIQUES ET INFORMATIQUE

Par **Aboul-Karim MOHAMED EL MAAROUF**

Factorisation incomplète et résolution de systèmes triangulaires pour des machines exploitant un parallélisme à grain fin

Sous la direction de **Luc GIRAUD**
Co-directeur **Abdou GUERMOUCHE**

Soutenue le 17 mars 2023

Membres du jury :

M. Damien Tromeur-Dervout	Professeur	Université Lyon	Rapporteur
M. Pierre Jolivet	Chargé de Recherche CNRS	Université Paris Sorbonne	Rapporteur
M. Brice Goglin	Directeur de Recherche	Inria Bordeaux	Président de jury
Mme. Jocelyne Erhel	Directrice de Recherche Émérite	Inria Rennes	Examinatrice
M. David Goudin	Ingénieur de Recherche	Atos	Examinateur
M. Luc Giraud	Directeur de Recherche	Inria Bordeaux	Directeur
M. Abdou Guermouche	Maître de Conférences	Université Bordeaux	Co-Directeur
M. Thomas Guignon	Ingénieur de Recherche	IFP Energies Nouvelles	Membre Invité

Remerciements

Je souhaite exprimer ma gratitude envers toutes les personnes qui ont contribué de quelque manière que ce soit à cette thèse. Ces trois années ont été remplies de rebondissements et d'émotions. Heureusement, de nombreuses personnes ont participé à cette aventure, la rendant possible.
Je tiens à exprimer ma reconnaissance envers mes directeurs, Luc Giraud et Abdou Guermouche, pour m'avoir permis de réaliser cette thèse dans les meilleures conditions possibles. Merci pour votre soutien et votre implication tout au long de ces trois années. Vos connaissances, votre pédagogie et votre enthousiasme ont été source d'une stimulation constante pour moi. Malgré une situation inédite (une petite pandémie et des confinements répétés) vous avez toujours été disponibles. J'ai apprécié vos nombreux conseils, et travailler avec vous a été un réel plaisir.
Je souhaite également adresser des remerciements particuliers à Thomas Guignon, mon encadrant à l'IFPEN. Merci pour ton implication et ta rigueur. Ta passion pour la transmission de la connaissance et ton expertise ont été et restent des sources de motivation pour moi. J'ai beaucoup apprécié travailler à tes côtés.
Je tiens à remercier les rapporteurs de ce manuscrit, Damien Tromeur-Dervout et Pierre Jolivet, ainsi que les autres membres du jury, Jocelyne Erhel, David Goudin, et Brice Goglin d'avoir présidé ce jury. Merci à tous pour vos remarques et vos nombreuses questions pertinentes durant la soutenance.
Je remercie également chaleureusement Ani Anciaux-Sedrakian pour tes conseils, ton professionnalisme et ta bienveillance. Ce fut un réel plaisir de travailler avec toi. Ma reconnaissance s'étend également à Raphael Gayno et à tous les collègues du département Informatique Scientifique et du département Mathématiques Appliquées de l'IFPEN.
Une mention spéciale pour l'équipe des doctorants, Joelle, Morgane, Jana, Arthur et Antoine. Merci pour ces soirées d'after work à refaire le monde et ces bons moments de rigolade et de partage. On se souviendra longtemps du fameux exposé de Joelle sur la courbe de fertilité et des théories farfelues d'Arthur. À nos fous rires passés et à cette amitié naissante. Un petit clin d'œil aussi aux "anciens" doctorants, Sabrina, Alexis, Guissel et Thoi. Merci pour votre aide et vos conseils de "grand frère".
À mon père, merci pour tes paroles fortes et aimantes à notre dernière rencontre. J'aurais aimé te

dire au revoir une dernière fois.
À ma mère, merci pour ton soutien inconditionnel et ton amour sans faille.
À ma tante Moinour et mon oncle Mirdane, merci d'avoir toujours été là, pour tous vos sages conseils et vos encouragements.
À ma famille de la Grande Comore, de Mayotte et de l'Hexagone.
Depuis les bancs de la fac jusqu'à maintenant, rien n'a changé, toujours la même équipe : Anaïs, Stéphanie, Kevin, Pauline, Pierre et Guillaume. Merci pour les magnifiques soirées, les fous rires et vos encouragements. À Anaïs, cette aventure aurait sans doute été bien différente sans toi à mes côtés.

Résumé

Factorisation incomplète et résolution de systèmes triangulaires pour des machines exploitant un parallélisme à grain fin

Résumé :

Comment calculer efficacement avec des processeurs intégrant un grand nombre de cœurs, une hiérarchie mémoire complexe et des unités de calcul avec plusieurs niveaux de parallélisme ? Cette évolution des moyens de calcul introduit trois niveaux de parallélisme : multi-coeur : plusieurs unités de calcul indépendantes partagent une partie de la hiérarchie mémoire dont la mémoire principale, SIMD (*Single Instruction stream Multiple Data stream*) : chaque unité de calcul via son jeu d'instruction a la capacité des traiter des valeurs par paquets plutôt qu'individuellement et hyperthreading : chaque unité de calcul peut exécuter plusieurs flots d'instructions simultanément. Dans le cadre de cette thèse, nous allons nous concentrer sur la parallélisation SIMD appliquée à la résolution de systèmes triangulaires résultants des méthodes de factorisations incomplètes de grands systèmes linéaires creux issus de la discrétisation de systèmes d'équations aux dérivées partielles (en particulier ceux issus des simulateurs IFPEN). Il existe différentes méthodes de factorisations incomplètes : les plus simples sont sans remplissage (ILU(0)), i.e., qu'aucun coefficient non nul n'est ajouté dans les facteurs. Ces méthodes sont utilisées comme préconditionneurs des solveurs itératifs basés sur les méthodes de sous-espaces Krylov. Dans ce cadre la capacité du préconditionneur à accélérer la convergence du solveur de Krylov, est primordiale.
De nombreuses études se sont intéressées à la renumérotation des équations et des inconnues. En particulier à l'effet que des méthodes comme Reverse Cuthill-McKee (RCM), Minimum Degree, Nested Dissection, Multi-Coloriage peut avoir sur l'efficacité de la factorisation incomplète ILU(0). Le problème est que les méthodes qui permettent d'avoir le parallélisme à grain fin recherché (Multi-Coloriage) ont tendance à réduire l'efficacité du préconditionneur.
L'apport original de la thèse est de proposer une méthode de renumérotation basée sur du coloriage de graphe qui permet d'obtenir le parallélisme à grain fin adapté aux nouvelles architectures exploitant le SIMD et une convergence du solveur de Krylov efficace. Nous avons proposé une méthode qui combine une renumérotation avec RCM avec un coloriage de graphe (ColorRCM). Cette méthode permet d'avoir une convergence du solveur de Krylov performante par rapport à l'ordre naturel et une convergence significativement plus rapide que celle obtenue avec un coloriage de graphe classique. Ensuite, nous avons exploité la structure creuse de la matrice du système linéaire pour la résolution des systèmes triangulaires avec un format de stockage permettant d'utiliser des instructions SIMD. Avec ce format notre résolution des systèmes triangulaires comparée au solveur de systèmes triangulaires proposé par la librairie Intel MKL 21.4 montre des temps de calculs au moins 3 fois inférieures.
Dans la deuxième partie de la thèse, nous abordons les synchronisations et les réductions pour les méthodes de Krylov parallèles en mémoire partagée. Nous avons proposé une barrière de synchronisation *Extended Butterfly* en mémoire partagée avec une complexité de $O(\log_2(\mathbf{P}))$ quelque soit **P** (**P** est le nombre de cœurs de calcul). Nos tests montrent que cette barrière *Extended Butterfly* est comparable à une barrière *Butterfly* ou *Dissemination* et a une latence plus faible que les barrières OpenMP avec les compilateurs Intel 21.4 et GCC 11.1. Cette barrière est ensuite utilisée comme support pour les opérations de réductions et l'on exploite l'atomicité des écritures SIMD pour échanger des valeurs de réductions entre les cœurs sans transfert additionnel par rapport à la barrière *Extended Butterfly*. La réduction proposée est jusqu'à 4 fois plus rapides que les réductions OpenMP avec GCC 11.1 sur le processeur milan et jusqu'à 6 fois plus rapides que les réductions OpenMP avec Intel 21.4 sur le processeur knl.

Mots-clés : Permutation symétrique de matrices, Coloriage de graphe, Résolution de systèmes linéaires triangulaires creux, Mécanismes de synchronisations, Calcul parallèle, SIMD

Incomplete factorization and solution of triangular systems for fine-grained parallelism computers

Abstract:

How to calculate efficiently with processors integrating a large number of cores, a complex memory hierarchy, and sophisticated computational units? This evolution of processor architecture introduces three levels of parallelism: multithreading (several cores simultaneously), SIMD (several data streams simultaneously), and hyperthreading (several instruction streams simultaneously). In this thesis, we will focus on SIMD (Single Instruction stream Multiple Data stream) parallelization applied to sparse triangular systems resulting from incomplete factorization methods of large sparse linear systems arising from the discretization of systems of partial differential equations (in particular, those coming from IFPEN simulators). There are different methods of incomplete factorization: the simplest ones are without fill-in (ILU(0)), i.e., no non-zero coefficient is added in the factors. These methods are used as preconditioners for iterative solvers based on Krylov subspace methods. In this context, the ability of the preconditioner to accelerate the convergence of the Krylov solver is essential.
Many studies have focused on the reordering of equations. In particular, the effect that methods like Reverse Cuthill-McKee (RCM), Minimum Degree, Nested Dissection, Multi-Coloriage can have on the efficiency of incomplete factorization ILU(0). The problem is that the methods which allow for the desired fine-grained parallelism (Multi-Coloriage) tend to reduce the efficiency of the preconditioner.
The original contribution of the thesis is to propose reordering methods based on graph coloring that allow for the fine-grained parallelism, adapted to the new architectures using SIMD, and an efficient convergence of the Krylov solver. We have proposed a method that combines RCM reordering with a graph coloring (ColorRCM). This method allows for good convergence of the Krylov solver with respect to the natural order and a convergence significantly faster than the one obtained with a classical graph coloring. Furthermore, we exploited the sparse structure of the matrix from the linear system for solving triangular systems with a storage format allowing for the use of SIMD instructions. With this format, our solution for triangular systems compared to the solver for triangular systems proposed by the Intel MKL 21.4 library shows computational times at least three times lower.
In the second part of the thesis, we discuss synchronization and reduction for parallel Krylov methods in shared memory. We present a synchronization barrier (*Extended Butterfly*) in shared memory with a complexity of $O(\log_2(\mathbf{P}))$ for any $\mathbf{P}$ ($\mathbf{P}$ is the number of cores). Our tests show that this barrier is comparable to a *Butterfly* or *Dissemination* barrier and has lower latency than the OpenMP barriers with the Intel 21.4 and GCC 11.1 compilers. The barrier is then used as support for reduction operations, and the atomicity of SIMD writes is exploited to exchange reduction values between cores without any additional transfer compared to the *Extended Butterfly* barrier. The proposed reduction is up to four times faster than OpenMP reductions with GCC 11.1 on the milan processor and up to six times faster than OpenMP reductions with Intel 21.4 on the knl processor.

Keywords: Symmetric permutation of matrices, Graph coloring, Sparse triangular linear systems, Synchronization primitves, Parallel computing, SIMD

Unité de recherche
UMR xxxx Université, 33000 Bordeaux, France.

Table des matières

Remerciements **iii**

Résumé **v**

INTRODUCTION **1**

1 PRÉAMBULE **7**

1.1 Algèbre linéaire 7

1.1.1 Notions de base d'un graphe 8

1.1.2 Permutation symétrique de matrices 9

1.1.3 Les solveurs linéaires 10

1.2 Les machines parallèles 13

1.2.1 Les architectures parallèles 13

1.2.2 Niveau de parallélisme 14

1.2.3 Évolution des architectures 15

1.2.4 Systèmes à mémoire partagée 16

2 PRÉCONDITIONNEMENT POUR UN SOLVEUR DE KRYLOV **17**

2.1 Introduction 17

2.2 État de l'art 18

2.2.1 Renumérotations et factorisation LU incomplète 21

2.3 Coloriage de graphe . . . 27
2.3.1 Le concept de ColorRCM . . . 27
2.3.2 Mise-en-oeuvre ColorRCM . . . 29
2.4 Expérimentations numériques . . . 31
2.4.1 Impact de la renumérotation . . . 32
2.5 Discussion . . . 37

3 RÉSOLUTION DE SYSTÈMES TRIANGULAIRES **39**
3.1 Introduction . . . 39
3.2 Formats de stockage creux . . . 41
3.3 État de l'art pour la résolution de systèmes triangulaires creux . . . 44
3.3.1 Level Scheduling . . . 45
3.3.2 Coloriage de graphe . . . 47
3.4 Parallélisation SIMD . . . 49
3.4.1 Multiplication Matrice-Vecteur . . . 49
3.4.2 Résolutions parallèle de systèmes triangulaires . . . 51
3.5 Expérimentations numériques . . . 53
3.5.1 Multiplication Matrice-Vecteur . . . 53
3.5.2 Résolution des systèmes triangulaires . . . 55
3.5.3 Performance du solveur de Krylov . . . 56
3.6 Discussion . . . 57

4 LES MÉCANISMES DE SYNCHRONISATION **61**
4.1 Introduction . . . 62
4.2 État de l'art des barrières de synchronisation . . . 65
4.2.1 La barrière *Sense-reversing Centralized* . . . 65
4.2.2 La barrière *Linear Centralized* . . . 65
4.2.3 La barrière *Software Combining Tree* . . . 66
4.2.4 La barrière *Tournament* . . . 67
4.2.5 La barrière *Butterfly* . . . 68
4.2.6 La barrière de *Dissemination* . . . 69
4.3 La barrière *Extended Butterfly* . . . 70
4.3.1 False sharing . . . 72

4.4 Réduction combinée avec une barrière 73
4.4.1 *Padding* des flags de synchronisation 74
4.4.2 Écriture SIMD atomique 75
4.4.3 Effet des communications redondantes lors de la réduction 76
4.4.4 Mise en œuvre de la réduction *Extended Butterfly* 78
4.5 Résultats expérimentaux 80
4.5.1 Environnement test 80
4.5.2 Résultats sur les barrières et réductions 81
4.5.3 Benchmark EPCC 85
4.6 Discussion 86

CONCLUSIONS GÉNÉRALES ET PERSPECTIVES **87**

A ANNEXE **91**
A.1 Résultats des tests avec le solveur ILU(0)-BiCGStab 91
A.2 Algorithmes pour la réduction des valeurs avec *Extended Butterfly* 98

Bibliographie **101**

Table des figures

1.1 Représentation d'une matrice creuse et de son graphe. 8
1.2 Permutation de la matrice flèche. 9

2.1 Illustration de Reverse Cuthill-McKee avec la matrice cage4. 24
2.2 Illustration de Nested Dissection avec la matrice cage4. 25
2.3 Illustration du coloriage de graphe avec la matrice cage4. 26
2.4 Illustration du coloriage de graphe avec ColorRCM sur la matrice cage4. 28
2.5 Illustration des renumérotations RCM, GreedyMC, GreedyMC(8) et ColorRCM(8) sur la matrice cage5. 30
2.6 Boîte à moustache des nombres d'itérations des différentes renumérotations. 33
2.7 Représentations des *performance profils* par rapport à leur nombre d'itérations sur l'ensemble des matrices. 34
2.8 Représentations des *performance profils* des matrices issue de la collection *Suite Sparse Matrix* par rapport à leur nombre d'itérations. 35
2.9 Représentations des *performance profils* des matrices SPE10 par rapport à leur nombre d'itérations. 36

3.1 Des exemples de représentation des formats de stockage CSR, EllPack et SELL-2 . . 43
3.2 Illustration de la matrice cage5 étendue avec ColorRCM(4). 45
3.3 Illustration du Level Scheduling avec la partie triangulaire inférieur de la matrice cage4. 46

3.4 Illustration du coloriage de graphe avec la partie triangulaire inférieur de la matrice cage4. . . . 47

3.5 Illustration d'une multiplication matrice-vecteur $\mathbf{A} \times \mathbf{z} = \mathbf{y}$ avec $\mathbf{A}$ stocké au format SELL-P. . . . 51

3.6 Représentations du nombre d'opérations flottante par seconde du SIMD-SPMV . . 54

3.7 Représentation du nombre d'opérations flottante par seconde de la résolution des systèmes triangulaires . . . 55

3.8 Illustration des échanges de valeurs pour la factorisation ILU(0) avec $\mathbf{A}$ stocké au format SELL-P . . . 59

4.1 Exemple de schéma d'échanges de notifications dans la barrière *Linear Centralized* avec 7 threads. . . . 66

4.2 Exemple de schéma d'échanges de notifications dans une barrière *Software Combining Tree* avec 7 threads. . . . 67

4.3 Exemple de schéma d'échanges de notifications dans la barrière *Butterfly* avec 8 threads. . . . 68

4.4 Exemple de schéma d'échanges de notifications dans la barrière *Dissemination* avec 7 threads. . . . 69

4.5 Exemple de schéma d'échanges de notifications dans la barrière *Extended Butterfly* avec 7 threads. . . . 70

4.6 Exemple de ligne de cache avec une charge utile non utilisée. . . . 73

4.7 Exemple de ligne de cache stockant le flag de synchronisation et des valeurs de réduction (7 valeurs en double précision). . . . 74

4.8 Exemple de communication lors d'une réduction avec une somme impliquant 5 threads. . . . 76

4.9 Exemple de communications entre le thread T_0 et T_1 du groupe 0 de la réduction *Extended Butterfly* . . . 79

4.10 Résultat comparatif des barrières avec le test synthétique utilisant le compilateur GCC 11.1 sur le processeur MILAN. . . . 82

4.11 Résultat comparatif des barrières avec le test synthétique utilisant le compilateur Intel 21.4 sur le processeur KNL. . . . 82

4.12 Résultat comparatif du test synthétique des réductions avec le compilateur Intel 21.4 sur le processeur SKL. . . . 83

4.13 Résultat comparatif du test synthétique des réductions avec le compilateur Intel 21.4 sur le processeur KNL. . . . 83

4.14 Résultat comparatif du test synthétique des réductions avec le compilateur GCC 11.1 sur le processeur MILAN. . . . 83

4.15 Coût moyen de la réductions de 1, 3 et 7 valeurs sur le processeur KNL avec le compilateur Intel 21.4. 84
4.16 Résultats du benchmark EPCC sur SKL, KNL et MILAN. 85

Liste des tableaux

2.1 Comptabilisation de tous les tests effectués avec ILU(0)-BiCGStab toutes configurations confondues. 32

2.2 Nombre d'échecs de ILU(0)-BiCGStab pour chaque renumérotation sur tout le jeu de matrice. 32

3.1 Pourcentage des temps moyens par itération des différentes opérations dans BiCGStab préconditionné en séquentiel. 40

3.2 Informations sur les matrices de tests. 44

3.3 Accélération de la version AVX2 de la résolution de systèmes triangulaires 55

3.4 Temps de calcul moyen par itération du solveur 56

4.1 Résumé des propriétés des barrières 71

4.2 Description des états du protocole MESI 72

4.3 Le taux d'échec pour million de tests sur 1000 lignes de cache entre 2 threads sur 2 cœurs d'un même socket. 76

4.4 Les caractéristiques des processeurs. 80

A.1 Résultats des tests ILU(0)-BiCGStab avec les matrices de la collection *Suite Sparse Matrix*. 92

A.2 Résultats des tests ILU(0)-BiCGStab avec les matrices SPE10. 95

List of Algorithms

1 BiCGStab préconditionné . . . 13
2 Factorisation incomplète ILU(0) . . . 19
3 Reverse Cuthill-McKee . . . 23
4 Multi-Coloriage Glouton . . . 25
5 Multi-Coloriage Glouton avec une limite de la taille des couleurs . . . 29
6 Multi-Coloriage suivant l'ordre définit par RCM. . . . 29
7 Descente pour SpTRSV . . . 45
8 Descente avec Level Scheduling pour SpTRSV . . . 46
9 Descente avec une renumérotation GreedyMC pour SpTRSV . . . 48
10 CSR - Multiplication Matrice-Vecteur . . . 49
11 CSR - Multiplication Matrice-Vecteur . . . 50
12 Descente AVX2-SpTRSV en double précision . . . 52
13 Réduction centralisée . . . 73
14 Réduction Extended Butterfly avec une somme pour AVX512. . . . 78
15 Test synthétique pour les opérations de réductions . . . 81
16 Calcul du coût de la réduction avec le benchmark EPCC . . . 85
17 Phase entrante dans la barrière *Extended Butterfly* . . . 98
18 Phase *Butterfly* dans la barrière *Extended Butterfly* . . . 98
19 Phase sortante dans la barrière *Extended Butterfly* . . . 99

Introduction

Contexte générale et objectifs

Le calcul haute performance ou HPC (High-Performance Computing) est un champ scientifique qui concerne à la fois les mathématiques et l'informatique et qui est transversal à de nombreux autres domaines tels que le climat, l'énergie, la mobilité durable, etc. Dans ces domaines nous avons souvent besoin de simuler des phénomènes physiques à grande échelle avec des modèles mathématiques qui sont de grand consommateurs de temps de calcul et d'espace de stockage. Un modèle mathématique est un ensemble d'équations permettant de décrire, sous certaines hypothèses, une situation donnée et la simulation numérique est un outil couramment utilisé pour trouver une solution qui satisfait certaines conditions ou permet de prédire l'évolution d'un système physique donné. Le but du calcul haute performance est d'utiliser de manière la plus optimale possible les ressources : machines, logiciels, algorithmes, méthodologies de développement..., pour permettre à des super-calculateurs de traiter de manière précise et en peu de temps, des problèmes complexes et massifs issus d'applications très variées. Le calcul haute performance a eu une évolution très rapide dans tous les domaines concernés, en lien direct avec l'évolution des technologies du numérique et en particulier pour la puissance de calcul rassemblée au sein d'une même machine. Récemment, un nouveau cap a été franchi avec le super-calculateur Frontier du Laboratoire national d'Oak Ridge (ORNL) aux États-Unis, enregistré avec une puissance de 1.102 EFLOP/s[*] dans le classement TOP500 de novembre 2022. Il détrône le super-calculateur Fugaku du Riken Center for Computational Science (R-CCS) au Japon avec ses 0.442 EFLOP/s. Ces machines ont dépassé la barre du milliard de milliard d'opérations par seconde, avec 8 730 112 cœurs de calcul pour Frontier issus de 9 472 noeuds CPUs et 37 888 noeuds GPUs, et 7 630 848 cœurs de calculs pour Fugaku répartis sur 158 976 noeuds CPUs. À titre de comparaison en 1993 le premier super-calculateur du top500 développe une puissance de 0,0000000597 EFLOP/s avec 1024 unités de calcul (TOP June 1993).
Au-delà de la puissance de calcul que proposent les super-calculateurs, la consommation éner-

*. 1 EFLOP/s : 10^{18} opérations flottantes par seconde

gétique de ces machines est aussi un grand enjeu. Et dans la catégorie du Green500, la machine Adastra du Grand Équipement National de Calcul Intensif - Centre Informatique National de l'Enseignement Supérieur (GENCI-CINES) en France est dans le top 3 des super-calculateurs avec une efficacité énergétique de 58.02 GFLOPs/Watt pour une puissance de calcul de 46.1 PFLOP/s. Le super-calculateur Adastra propose une partition composée des processeurs EPYC Genoa d'AMD embarquant 96 cœurs cadencés jusqu'à 4.4 GHz au maximum. Ces nouvelles générations de processeurs dites *many-cores* allient la performance des calculs et l'efficacité énergétique.
D'autre part, le côté utilisateur joue aussi un rôle majeur dans l'optimisation de la consommation énergétique de ces super-calculateurs. Bien optimiser nos applications et exploiter efficacement les ressources des machines que nous utilisons est primordial. Une première question que pose la thèse est : Comment calculer efficacement avec des processeurs intégrant un grand nombre de cœurs, une hiérarchie mémoire complexe et des unités de calcul SIMD et SIF (Simultaneous Instruction Flow : hyperthreading) ? Doit-on reformuler ou changer les algorithmes classiques pour exploiter efficacement ces processeurs ? Cette évolution des moyens de calcul introduit trois niveaux de parallélisme (multithreading, SIMD et hyperthreading), dont 2 relativement nouveaux dans leurs impacts sur les méthodes et algorithmes :

— Les processeurs multi-cœurs se sont généralisés et de nouveaux processeurs intégrant un nombre très important de cœurs apparaissent. Initié par le processeur Intel Xeon Phi qui contient 72 cœurs ou, plus récemment, la 4ème génération de processeur AMD EPYC 9004 avec ses 96 cœurs ou encore le processeur Ampere Altra embarquant 80 cœurs. On les appelle des processeurs *many-cores*. Des modèles de programmation parallèle comme les *runtime systems* adaptés à ces nouveaux processeurs font l'objet de nombreux travaux dans la communauté HPC. Ils permettent, avec un graphe de dépendance des calculs en mémoire partagée (souvent appelées tâches) d'alimenter efficacement les unités de calcul. Un avantage de cette approche est qu'elle permet de faire apparaître facilement un parallélisme non-classique. Cette approche pourrait cependant montrer ses limites si, par exemple, la topologie d'interconnexion des cœurs n'est pas prise en compte dans les mécanismes de synchronisation ou que le temps de gestion du graphe de dépendance est trop important relativement à la durée des tâches. L'importance des mécanismes de synchronisation en mémoire partagée est d'autant plus importante que le nombre de coeurs d'un processeur a fortement augmenté.

— Le SIMD impose d'avoir des algorithmes présentant un degré de parallélisme suffisant au niveau des instructions. Cependant, de nombreux algorithmes de résolution de systèmes linéaires en parallèle sont des transformations d'algorithmes séquentiels qui permettent de créer un parallélisme à gros grain, adapté au modèle classique de parallélisation en mémoire distribuée avec des unités de calcul séquentielles. Par exemple, pour les méthodes de factorisation incomplètes, la parallélisation possible dépend de l'ordre de numérotation des équations : une parallélisation à grain fin de ces méthodes nécessite un ordre particulier qui peut être incompatible avec les bonnes propriétés de convergence des méthodes de résolution utilisées.

— Le SIF (hyperthreading) est un élément assez nouveau : il consiste en ce que le cœur de calcul n'exécute plus un flot d'instructions mais plusieurs "simultanément". Du point de vue macroscopique (système d'exploitation, programmeurs) ces flots sont vus comme de véritables processeurs, on est donc tenté de les utiliser ainsi via par exemple un runtime system comme pour la parallélisation multi-cœurs. Mais cette approche fait l'impasse sur

le fait que les processeurs représentant ces flots partagent la majeure partie des différentes unités du cœur de calcul (cache de données L1, cache d'instruction, lecture et décodage des instruction, etc.). Il faut donc être capable de faire travailler ces flots sur des calculs fortement corrélés : mêmes instructions et "données proches". Il y a donc une similitude avec la parallélisation SIMD, mais à un grain supérieur.

Les éléments précédents sont assez généraux et, dans le cadre de ce travail de recherche, nous allons nous concentrer sur les méthodes de factorisation incomplètes (en particulier ILU(0)) de grands systèmes linéaires creux issus de la discrétisassions de systèmes d'équations aux dérivées partielles (en particulier ceux issus des simulateurs IFPEN). Ces méthodes sont utilisées comme pré-conditionneurs des solveurs itératifs basés sur les méthodes de Krylov. Dans ce cadre, la "qualité" du pré-conditionneur, c'est-à-dire sa capacité à faire converger rapidement le solveur de Krylov, est primordiale. Ces méthodes sont connues depuis longtemps et ne constituent plus, dans la plupart des cas, le pré-conditionneur principal, mais sont plutôt utilisées comme solveurs locaux de méthodes de décomposition de domaines ou CprAMG. Ces méthodes restent donc d'une grande importance dans les solveurs linéaires actuels. Nous allons donc nous intéresser à la parallélisation à grain très fin, donc adaptée aux unités de calcul SIMD, des méthodes de factorisation incomplètes. L'objectif principal de la thèse est de proposer des méthodes de renumérotation des équations du système linéaire permettant d'avoir un bon pré-conditionnement et un parallélisme à grain fin pour la phase de résolution. Le principe d'une méthode de factorisation incomplète appliquée, comme pré-conditionneur, à un système linéaire $\mathbf{Ax} = \mathbf{y}$ consiste à effectuer une factorisation incomplète de $\mathbf{A}$ telle que $\mathbf{A} = \tilde{\mathbf{L}}.\tilde{\mathbf{U}} + \mathbf{R}$ où $\tilde{\mathbf{L}}$ et $\tilde{\mathbf{U}}$ sont des facteurs triangulaires pour ILU et à résoudre le système $\tilde{\mathbf{L}}.\tilde{\mathbf{U}}\mathbf{x} = \mathbf{y}$ au lieu du système $\mathbf{Ax} = \mathbf{y}$ avec l'idée que $(\tilde{\mathbf{L}}.\tilde{\mathbf{U}})^{-1}$ est proche de $\mathbf{A}^{-1}$ et que la solution obtenue est suffisante pour assurer une bonne convergence du solveur itératif. Il existe différentes méthodes de factorisation incomplètes : les plus simples sont sans remplissage (ILU(0)), c'est-à-dire qu'aucun coefficient non nul n'est ajouté dans les facteurs. De nombreux articles se sont intéressés à la renumérotation des équations du système linéaire et à l'effet que l'ordre peut avoir sur la stabilité des méthodes ILU [9, 10, 18, 26, 67]. Différentes renumérotations sont étudiées comme par exemple RCM (Reverse Cuthill-McKee), MD (Minimum Degree). Leur but est en général de réduire le nombre de coefficients ajoutés par la factorisation, soit pour des raisons de consommation mémoire, soit pour tenter de minimiser l'erreur induite par l'aspect incomplet de la factorisation. Le problème est que dans la plupart des cas, la structure creuse des facteurs triangulaires ne fait pas apparaître naturellement assez de parallélisme à grain fin dans la résolution de $\tilde{\mathbf{L}}.\tilde{\mathbf{U}}\mathbf{x} = \mathbf{y}$ pour être exploité avec des processeurs SIMD. Les renumérotations par coloriage du graphe d'adjacence de $\mathbf{A}$ (GreedyMC) permettent de faire apparaître le degré de parallélisme requis pour ces unités de calcul. Ces méthodes de coloriage permettent de former des groupes d'équations indépendantes, appelés couleurs, qui peuvent donc être résolues simultanément. Le nombre d'équations dans chaque groupe définit le degré de parallélisme et on cherche donc à colorier le graphe en utilisant un nombre minimal de couleurs pour maximiser ainsi le degré de parallélisme. Le problème est que ces renumérotations peuvent réduire l'efficacité du préconditionneur et conduire à une très mauvaise convergence du solveur itératif. Les travaux portant sur les permutations des systèmes linéaires pour la parallélisation des factorisations incomplètes sont maintenant assez anciens. En particulier, les permutations par coloriage pour la parallélisation à grain fin sont un peu tombées en désuétude avec l'abandon des machines SIMD et des modèles de parallélisme associés apparues dans les années 90. De plus, les travaux sur les pré-conditionnements se sont tournés vers des méthodes plus avancées comme les méthodes de décomposition de domaine ou les méthodes multi-grilles, pourtant les

méthodes ILU restent d'actualité, car elles peuvent être utilisées comme solveurs locaux des méthodes précédentes. Maintenant l'évolution des moyens de calcul avec la généralisation du SIMD et des processeurs *many-cores* nous impose de revenir sur les méthodes de permutation par coloriage pour les factorisations incomplètes. On constate dans la littérature que le sujet de la parallélisation de la résolution des systèmes triangulaires est de nouveau d'actualité, mais que l'impact des permutations obtenues sur la qualité du pré-conditionneur et donc sur la convergence du solveur itératif n'est pas encore vraiment abordé. Pour cette partie, l'apport original de la thèse est de proposer des méthodes de permutation permettant d'obtenir le parallélisme adapté aux nouvelles architectures utilisant des jeux d'instruction SIMD et une convergence du solveur de Krylov non dégradée ou peu dégradée.
Dans le cadre d'une parallélisation à grain fin au niveau des coeurs de calcul la performance parallèle des solveurs de Krylov est aussi conditionnée par la capacité à effectuer des opérations de synchronisation des coeurs nécessaires, par exemple, aux calculs des produits scalaires et normes intervenant dans ces méthodes. Nous nous intéressons aux synchronisation en mémoire partagée avec un grand nombre de coeurs. La synchronisation en mémoire partagée est aussi un sujet ancien et de nombreuses méthodes ont été proposées. On retrouve souvent ces méthodes dans les "runtimes" OpenMP des différents compilateurs utilisés dans le monde HPC. Dans le cadre d'une parallélisation classique en mémoire partagée, c'est à dire non basée sur un graphe de dépendance de tâches, l'objectif des synchronisations entre coeurs est d'assurer d'une part la cohérence des résultats produits par chaque coeur vus par les autres coeurs et d'autre part l'ordre des calculs. Les synchronisations font soit intervenir 2 coeurs de calcul : les synchronisations point à point, soit des groupes de coeurs pour les synchronisations collectives que sont les barrières et réductions. Dans tous les cas il est nécessaires que le temps passé dans ces synchronisation soit le plus petit possible (relativement au temps passé dans le calcul) pour assurer une bonne performance parallèle du calcul qui nous intéresse, ici un solveur de Krylov. Dans ce contexte l'apport original de ce travail de recherche est de proposer une méthode de réduction qui permet d'effectuer une réduction combinée à une barrière en utilisant les propriétés de certaines instructions SIMD pour réduire le coût de la réduction.

Plan de la thèse

Ce manuscrit de thèse se décompose en quatre chapitres. Le chapitre 1 introduit les notions de base dont nous aurons besoin tout au long du manuscrit. Ce chapitre rend compte aussi de notre double intérêt pour l'algèbre linéaire des solveurs et les machines modernes. Les trois chapitres suivants décrivent les contributions de la thèse.
Le chapitre 2 a pour objectif de comparer différentes méthodes de renumérotations issues de l'état de l'art. Certaines sont connues pour apporter beaucoup de parallélisme dans les calculs et d'autres apportent un pré-conditionneur performant pour le solveur itératif. Dans un second temps, ce chapitre présente une nouvelle méthode alliant du parallélisme à grain fin et une certaine efficacité du pré-conditionneur pour faire converger plus rapidement le solveur. Les performances numériques du solveur configuré avec ces différentes renumérotations ont été analysées au travers d'un ensemble de matrices issus de la collection SuiteSparse Matrix[*].
Dans le chapitre 3, l'objectif est d'utiliser la structure apportée par le coloriage de graphe pour le

*. Collection SuiteSparse Matrix (https://sparse.tamu.edu/)

calcul SIMD lors de la résolution de systèmes triangulaires. Nous avons mis en œuvre des versions du solveur de résolution de systèmes triangulaires en AVX2 et AVX512 qui ont été comparées avec la routine de `mkl_cspblas_dbsrtrsv` de Intel MKL 21.4. Puis nous avons mesuré les performances du solveur itératif configuré avec la nouvelle méthode de renumérotation et accéléré avec le jeux d'instruction SIMD AVX2 des processeurs x86.
Dans le chapitre 4, nous nous sommes intéressés aux processeurs modernes embarquant beaucoup de cœurs de calculs. L'objectif est de proposer une opération de réduction combinée à une barrière performante pour un grand nombre de cœurs. Pour atteindre cet objectif, il est nécessaire d'avoir une barrière efficace et adaptée au type de l'opération de réduction (généralement une somme arithmétique). Nous avons étudié plusieurs barrières issues de la littérature et en avons proposé une nouvelle qui étend une méthode existante. Basée sur cette barrière, nous avons mis en œuvre une opération de réduction combinée . Ensuite nous avons comparé cette nouvelle opération de réduction et la réduction d'OpenMP 4.5 sur différents processeurs SKL, KNL, MILAN.
Le document sera finalisé par une conclusion et des perspectives.

1 PRÉAMBULE

Contents

1.1 Algèbre linéaire **7**
1.1.1 Notions de base d'un graphe 8
1.1.2 Permutation symétrique de matrices 9
1.1.3 Les solveurs linéaires 10
1.2 Les machines parallèles **13**
1.2.1 Les architectures parallèles 13
1.2.2 Niveau de parallélisme 14
1.2.3 Évolution des architectures 15
1.2.4 Systèmes à mémoire partagée 16

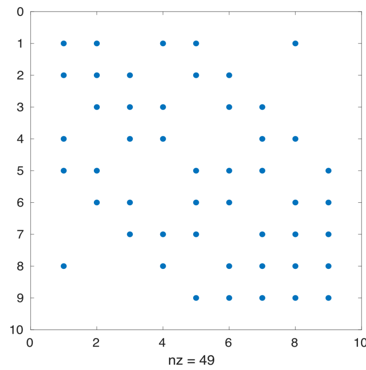
Ce chapitre a pour objectif d'introduire et définir des notions essentielles à la compréhension du manuscrit. Dans un premier temps, nous allons nous intéresser à la résolution de grands systèmes linéaires creux par des méthodes itératives. Cela permettra de situer nos travaux par rapport à l'état de l'art. Nous définirons aussi certaines notions de base de la théorie des graphes qui nous seront utiles par la suite. Cette première partie parlera donc d'algèbre linéaire. Dans la seconde partie, nous parlerons de l'architecture des processeurs. Durant les 20 dernières années, des processeurs embarquant de plus en plus de coeurs avec plusieurs niveaux de parallélismes ont progressivement été créés. Ce chapitre présentera les architectures parallèles de ces processeurs.

1.1 Algèbre linéaire

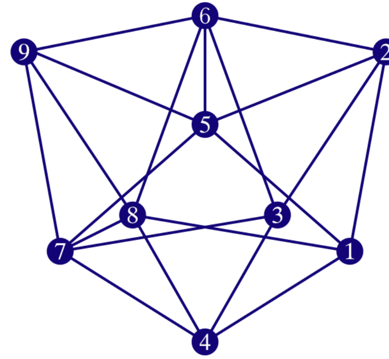
L'algèbre linéaire est en filigrane de nombreuses simulations numériques issues de problèmes aussi variés que complexes. En général, ces simulations utilisent la discrétisation de systèmes d'équations aux dérivées partielles et utilisent la résolution de systèmes linéaires issus de cette discrétisation. Les matrices des systèmes linéaires résolus sont dites "creuses" car le nombre d'éléments non nuls est très inférieur à n^2 où n est le nombre d'équations/inconnues. Il y a donc

un intérêt à ne stocker que les éléments non nuls des matrices. Se restreindre aux éléments non nuls permet un gain pour le stockage de la matrice mais aussi un gain sur les calculs effectués avec une telle matrice. Cette définition n'est pas rigoureuse, mais elle nous permet d'avoir une idée globale de ce qu'une matrice creuse peut être. Une manière visuelle de représenter et exploiter certaines propriétés des matrices est de passer par la théorie des graphes. Nous allons introduire quelques définitions de base concernant les graphes.

1.1.1 Notions de base d'un graphe



(a) *Matrice creuse cage4*



(b) *Graphe non-orienté cage4*

Figure 1.1 – *(1.1a) représente le profile de la matrice creuse cage4. (1.1b) est un exemple de représentation du graphe d'adjacence non-orienté de la matrice (1.1a) avec* $\{1,2,3,4,5,6,7,8,9\}$ *l'ensemble des sommets et* $\{\{1,2\},\{1,4\},\{1,5\},\{1,8\},\{2,3\},\{2,5\},...,\{7,8\},\{7,9\},\{8,9\}\}$ *l'ensemble des arêtes.*

Définition 1.1 (Graphe non-orienté/orienté)**.** Un graphe non orienté G signifie que les arêtes de $\mathcal{E}$ ne sont pas ordonnées, c'est-à-dire que, $\forall u, v \in \mathcal{V}$ et $u \neq v$ alors $u, v \in \mathcal{E}$. À l'inverse, G est orienté lorsque les sommets u et v ($u \neq v$) des arêtes de $\mathcal{E}$ sont ordonnés, c'est-à-dire que (u, v) et/ou $(v, u) \in \mathcal{E}$.

Définition 1.2 (Graphe d'adjacence d'une matrice)**.** Le graphe d'adjacence de **A** est le graphe décrivant la relation binaire suivante entre les sommets de **A** : $\forall i, j \in \mathcal{V}$, $1 \leq i, j \leq n$ et $i \neq j$, $(i, j) \in \mathcal{E}$ si et seulement si $a_{i,j} \neq 0$. $\mathcal{V}$ est l'ensemble fini des sommets (ou nœuds) de **A** tel que $\mathcal{V} = v_1, v_2, ..., v_n$. $\mathcal{E}$ est l'ensemble des arêtes formées par les paires des éléments de $\mathcal{V}$, c'est-à-dire $\mathcal{E} = \{(v_i, v_j)\}_{1 \leq i,j \leq n\,;\, i \neq j}$.

Notez que si **A** est symétrique ou structurellement symétrique alors le graphe d'adjacence est un graphe non-orienté. Dans ces travaux, nous ne manipulons que des matrices qui sont au moins structurellement symétriques et donc des graphes non-orientés.

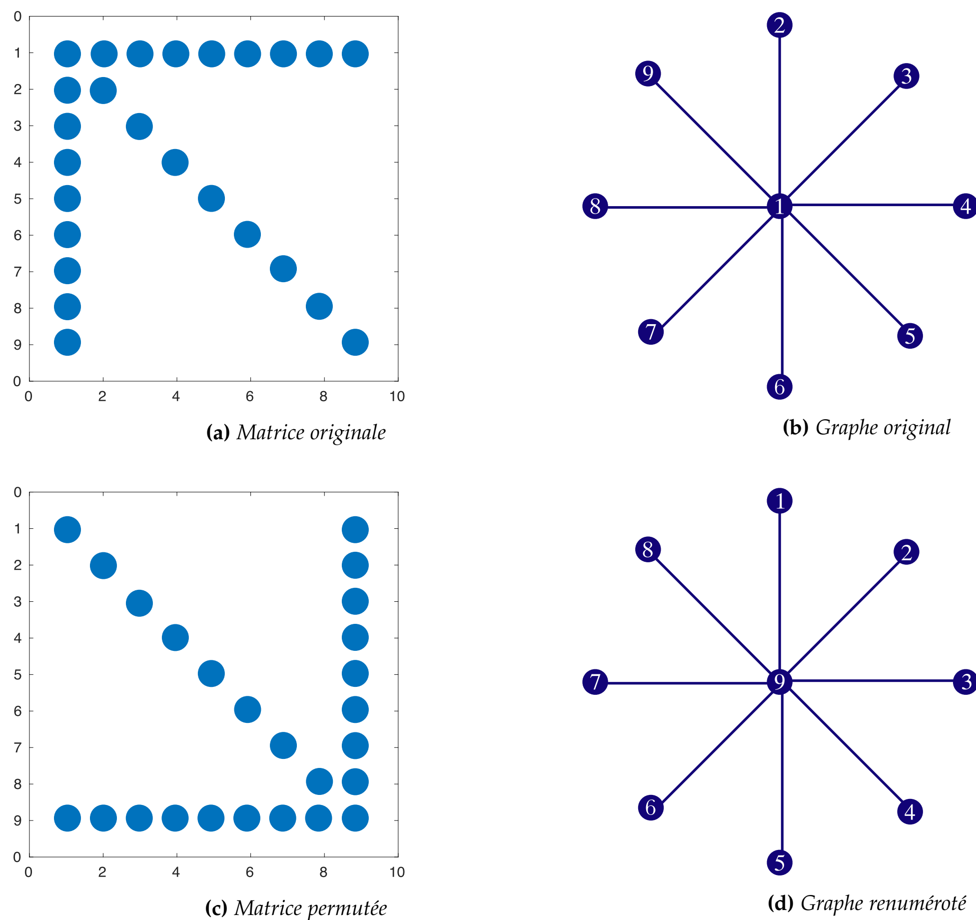
Définition 1.3 (Sommet adjacent)**.** Un sommet $v \in \mathcal{V}$ est adjacent à un sommet $u \in \mathcal{V}$ si et seulement si $u, v \in \mathcal{E}$. L'ensemble des sommets adjacents à u sera noté $Adj(u)$ et défini comme $Adj(u) = \{v | u, v \in \mathcal{E}, \forall v \in \mathcal{V}\}$.

Définition 1.4 (Degré d'un sommet)**.** Le degré d'un sommet $u \in \mathcal{V}$ est défini comme le cardinal de l'ensemble $Adj(u)$ c'est-à-dire $|Adj(u)|$.

Définition 1.5 (Excentricité)**.** L'excentricité d'un sommet v dans un graphe, notée $\epsilon(v)$, est la plus grande distance entre v et n'importe quel autre sommet du graphe.

1.1.2 Permutation symétrique de matrices

La permutation symétrique d'une matrice du point de vue de la théorie des graphes revient à renuméroter les noeuds du graphe d'adjacence associé à la matrice. Ainsi, la structure du graphe de la matrice permutée sera la même que celle de la matrice d'origine, mais seule la numérotation aura changé. Si $\mathbf{A}'$ est la matrice permutée de $\mathbf{A}$, le graphe de $\mathbf{A}'$ peut être défini comme $G' = (\pi(\mathcal{V}), \mathcal{E})$ avec $\pi : \mathcal{V} \to \mathcal{V}, i \mapsto \pi(i) = j$. Autrement dit, la fonction π nous donne pour l'ancienne numérotation i d'un noeud de G la nouvelle numérotation j pour le graphe G'.



(a) *Matrice originale*

(b) *Graphe original*

(c) *Matrice permutée*

(d) *Graphe renuméroté*

Figure 1.2 – *Un exemple de l'effet d'une renumérotation des noeuds d'un graphe. 1.2a est la matrice flèche originale. 1.2b est le graphe de la matrice flèche originale. 1.2c est le résultat de la permutation symétrique de la matrice flèche originale et 1.2d est le graphe de la matrice permutée.*

La permutation qui a été appliquée à la matrice flèche originale consiste à inverser l'ordre des noeuds de la matrice originale, en passant de (1,2,3,4,5,6,7,8,9) à (9,8,7,6,5,4,3,2,1). On obtient la matrice permutée (1.2c) avec la flèche dirigée vers le bas. Comme on peut le voir, les arrêtes entre

les noeuds n'ont pas été modifiées, seule la numérotation des noeuds a changé. Pour la suite du manuscrit, lorsque nous parlerons de méthode de renumérotation, cela désignera une méthode permettant d'obtenir un certain ordre des noeuds du graphe. Cet ordre appliqué à l'ensemble des noeuds du graphe permettra d'avoir une renumérotation des noeuds, c'est-à-dire, une permutation symétrique de la matrice originale.

1.1.3 Les solveurs linéaires

Prenons le système linéaire suivant :

$$\mathbf{Ax} = \mathbf{b} \tag{1.1}$$

A est une matrice carrée non singulière à valeurs réelles, **b** est le vecteur second membre du système à valeurs réelles et **x** est le vecteur contenant les inconnues du système à valeurs réelles. L'approche directe pour résoudre ce système (1.1) est basée sur l'élimination de Gauss (ou l'une de ses variantes). L'idée est de réduire **A** en un système triangulaire pour simplifier la résolution. Toujours dans le même esprit de simplification, on peut chercher une factorisation de la matrice **A** en produit de deux matrices avec une factorisation LU (ou Cholesky dans le cas d'une matrice symétrique définie positive). La factorisation LU décompose la matrice **A** comme le produit de deux matrices **L** et **U** respectivement triangulaires inférieures à diagonale unité et triangulaires supérieures. Dès lors, pour calculer les inconnues de (1.1), cela revient à Descendre le système $\mathbf{Ly} = \mathbf{b}$ puis à Remonter le système $\mathbf{Ux} = \mathbf{y}$. Les méthodes de cette catégorie sont connues pour leur robustesse. Néanmoins, elles sont généralement coûteuses en temps de calcul avec une complexité en $\mathcal{O}(n^3)$ avec n la dimension du système pour une matrice dense. Si la matrice **A** est creuse, c'est-à-dire qu'elle comporte beaucoup plus d'éléments nuls que d'éléments non nuls la complexité peut être moindre. Elle est en particulier en $\mathcal{O}(n^2)$ pour des matrices issues de la discrétisation d'EDP sur une grille 3D. Cependant, ces méthodes de décomposition exacte peuvent engendrer beaucoup de remplissage, c'est-à-dire que la différence entre le nombre d'éléments non nuls de la matrice **A** et le nombre d'éléments non nuls des facteurs **L** et **U** est grande, l'ajout d'éléments non nuls allant parfois jusqu'à remplir totalement les facteurs **L** et **U**. Donc, les facteurs **L** et **U** peuvent perdre le caractère creux de la matrice **A**, et le coût de stockage est de n^2 avec un stockage en place des facteurs denses **L** et **U**. Dans un contexte industriel où l'on manipule de grands systèmes linéaires creux, il n'est pas envisageable de perdre la structure creuse des systèmes. Pour réduire cet effet de remplissage, il existe différentes techniques, notamment les méthodes de renumérotation des équations qui sont conçues pour limiter le remplissage dans des zones parfois prédéfinies. L'idée de ces méthodes est de chercher une permutation particulière de la matrice. Certaines méthodes de renumérotation peuvent exhiber une structure de la matrice pour la parallélisation des calculs. Cependant, à cause du remplissage parfois important à l'issue de la décomposition, on peut perdre la structure conférée par la méthode de renumérotation, rendant difficile l'exploitation du parallélisme lors de la résolution du système, réduisant ainsi fortement le degré de parallélisme des opérations. Pour la suite du manuscrit, nous nous concentrerons uniquement sur une catégorie de méthodes de résolution des systèmes linéaires, à savoir les méthodes itératives.

Les méthodes itératives

L'approche itérative pour résoudre le système linéaire (1.1) consiste à partir d'une solution initiale $\mathbf{x}^{(0)}$ et à calculer une suite de solutions approchées $\mathbf{x}^{(k)}$ qui converge vers la solution exacte $\mathbf{x}$ à mesure que le nombre d'itérations k augmente. Différentes méthodes ont été mises en place pour calculer la suite $\mathbf{x}^{(k)}$. Dans cette section, nous allons introduire les principes généraux de deux familles de méthodes itératives : les méthodes stationnaires et les méthodes par projection dans des sous-espaces de Krylov.

Les méthodes stationnaires

Cette famille de méthode est basée sur la décomposition de la matrice $\mathbf{A}$ en une somme de deux matrices telle que $\mathbf{A} = \mathbf{M} + \mathbf{N}$ avec $\mathbf{M}$ une matrice non singulière. Ainsi, le système (1.1) devient :

$$(\mathbf{M} + \mathbf{N})\mathbf{x} = \mathbf{b} \ \mathbf{M}\mathbf{x} \qquad = \mathbf{b} - \mathbf{N}\mathbf{x} \tag{1.2}$$

On peut approcher une solution de ce problème (1.2) par une méthode de point fixe : la suite $\mathbf{x}^{(k)}$ des solutions approchées du problème (1.2) équivalent au problème (1.1) s'exprime de la manière suivante :

$$\mathbf{x}^{(k+1)} = \mathbf{M}^{-1}(\mathbf{b} - \mathbf{N}\mathbf{x}^{(k)}) = \mathbf{x}^{(k)} + \mathbf{M}^{-1}(\mathbf{b} - \mathbf{A}\mathbf{x}^{(k)}) \tag{1.3}$$

Les méthodes itératives par des schémas stationnaires diffèrent dans le choix de la matrice $\mathbf{M}$. Si on décompose $\mathbf{A} = \mathbf{L} + \mathbf{D} + \mathbf{U}$, avec $\mathbf{L}$ étant la partie triangulaire inférieure stricte de $\mathbf{A}$, $\mathbf{D}$ étant la diagonale de $\mathbf{A}$ et $\mathbf{U}$ étant la partie triangulaire supérieure stricte de $\mathbf{A}$. Voici une liste non exhaustive des plus connues d'entre elles :

$$\text{Jacobi} : \mathbf{M}_J = \mathbf{D} \tag{1.4}$$

$$\text{Gauss-Seidel} : \mathbf{M}_{GS} = \mathbf{D} + \mathbf{L} \tag{1.5}$$

$$\text{Successive Over Relaxation (SOR)} : \mathbf{M}_{SOR} = \frac{1}{\omega}(\mathbf{D} + \omega\mathbf{L}) \tag{1.6}$$

Remarque, ici les matrices $\mathbf{L}$ et $\mathbf{U}$ ne sont pas les mêmes que celles issues de la factorisation LU introduite plus haut. La convergence de ces méthodes est garantie pour toute solution initiale $\mathbf{x}^{(0)}$ si et seulement si le rayon spectral de la matrice d'itération $(\mathbf{I} - \mathbf{M}^{-1}\mathbf{A})$ est strictement inférieur à 1. Nous redirigeons le lecteur à la référence [79] pour une description et une analyse complètes de ces méthodes.

Les méthodes de Krylov

Reprenons l'équation (1.3) :

$$(1.3) \Leftrightarrow \mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \mathbf{M}^{-1}\mathbf{r}^{(k)}, \text{ with } \mathbf{r}^{(k)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(k)} \tag{1.7}$$

Si l'on calcule les quatre premières approximations de la solution x, on obtient :

$$\begin{aligned}
&\mathbf{x}^{(0)}\\
&\mathbf{x}^{(1)} = \mathbf{x}^{(0)} + \mathbf{M}^{-1}\mathbf{r}^{(0)}\\
&\mathbf{x}^{(2)} = \mathbf{x}^{(1)} + \mathbf{M}^{-1}\mathbf{r}^{(1)} = \mathbf{x}^{(0)} + 2\mathbf{M}^{-1}\mathbf{r}^{(0)} - (\mathbf{M}^{-1}\mathbf{A})\mathbf{M}^{-1}\mathbf{r}^{(0)}\\
&\mathbf{x}^{(3)} = \mathbf{x}^{(2)} + \mathbf{M}^{-1}\mathbf{r}^{(2)} = \mathbf{x}^{(0)} + 3\mathbf{M}^{-1}\mathbf{r}^{(0)} - 3(\mathbf{M}^{-1}\mathbf{A})\mathbf{M}^{-1}\mathbf{r}^{(0)} + (\mathbf{M}^{-1}\mathbf{A})^2\mathbf{M}^{-1}\mathbf{r}^{(0)}
\end{aligned}$$

En d'autres termes, pour un $\mathbf{x}^{(0)}$ donné, on observe que tous les $\mathbf{x}^{(k)}$ sont contenus dans le sous-espace vectoriel :

$$x^{(0)} + sev\{\mathbf{M}^{-1}\mathbf{r}^{(0)}, (\mathbf{M}^{-1}\mathbf{A})\mathbf{M}^{-1}\mathbf{r}^{(0)}, ..., (\mathbf{M}^{-1}\mathbf{A})^{k-1}\mathbf{M}^{-1}\mathbf{r}^{(0)}\}$$

L'espace $\mathcal{K}(k;\mathbf{T};\mathbf{v})$ est défini par $sev\mathbf{v}, \mathbf{T}\mathbf{v}, ..., \mathbf{T}^{k-1}\mathbf{v}$ et est appelé sous-espace de Krylov de dimension k avec $\mathbf{T}$ une matrice à valeurs réelles de dimension $n \times n$ et $\mathbf{v}$ un vecteur à valeurs réelles de taille n. La méthode des sous-espaces de Krylov commence par une solution initiale $\mathbf{x}^{(0)}$ (qui peut être le vecteur nul), un résidu initial $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$ et calcule les différentes itérations $\mathbf{x}^{(k)}$ telles que les $\mathbf{x}^{(k)} - \mathbf{x}^{(0)}$ appartiennent à l'espace $\mathcal{K}(k;\mathbf{A};\mathbf{r}^{(0)})$. Cela revient à définir les itérées $\mathbf{x}^{(k)}$ par un polynôme sur $\mathbf{A}$ de degré au plus k tel que $\mathbf{x}^{(k)} = \mathbf{x}^{(0)} + q_{k-1}(A)\mathbf{r}^{(0)}$. Les différentes méthodes de résolution de systèmes linéaires basées sur les sous-espaces de Krylov diffèrent par la manière de choisir les poids de $q_{k-1}(A)$. Plus de détails peuvent être trouvés dans les références [3, 24, 68, 79]. Il a été développé différents algorithmes pour rechercher les solutions $\mathbf{x}^{(k)}$ comme élément du sous-espace $\mathcal{K}(k;\mathbf{T};\mathbf{v})$, notamment la méthode du gradient conjugué (CG) pour les matrices symétriques définies positives [43], la méthode du gradient bi-conjugué (BiCG) [30, 53], la méthode du gradient bi-conjugué stabilisé (BiCGStab) [76], la méthode Orthomin [78] ou encore la méthode Generalized Minimal RESidual (GMRES) [69] pour les systèmes non symétriques. De nombreuses études ont traité de ces méthodes et le lecteur trouvera plus de détails théoriques et d'algorithmes dans [49, 68, 70]. Nous introduisons ici l'algorithme BiCGStab préconditionné (Algorithme 1), qui sera notre solveur principal dans la partie expérimentale.

Précisons que dans certains cas, cette algorithme peut avoir des arrêts prématurés ou *Breakdown*. La raison a cela est que certaines quantité comme α, β, ω peuvent devenir infiniment grandes ou égalent au zéro machine. Dans l'étude mené dans [36], l'auteur détaille les principales raisons qui peuvent causer un arrêt prématuré du solveur BiCGStab.

Certaines spécificités de la méthode BiCGStab ont été présentées dans [68]. D'un point de vue numérique, l'exécution de l'Algorithme 1 nécessite de calculer à chaque itération quatre résolutions triangulaires éparses (SpTRSV), deux multiplications matrice-vecteur éparses (SpMV), six combinaisons linéaires (AXPY c.-à-d. une multiplication scalaire suivie d'une addition vectorielle) et cinq produits scalaires. En tenant compte du fait que le temps de calcul ou la complexité de calcul peuvent être améliorés à l'aide de méthodes de permutation de matrices, le préconditionnement du système ou du type de stockage creux des matrices, alors les méthodes itératives deviennent plus compétitives que les méthodes directes dans certains cas de figures.

Algorithm 1 BiCGStab préconditionné

1: **procedure** BiCGStab($\mathbf{A}, \mathbf{M}, \mathbf{b}, \mathbf{x}^{(0)}$)
   Inputs : Matrice $\mathbf{A}$, préconditionneur $\mathbf{M}$, second membre $\mathbf{b}$, solution initiale $\mathbf{x}^{(0)}$
   Inputs : Vecteur $\hat{\mathbf{r}}^{(0)}$ tel que $\hat{\mathbf{r}^{(0)}}^T \cdot \mathbf{r}^{(0)} \neq 0$.
   Calculer $\mathbf{r}^{(0)} = \mathbf{b} - \mathbf{A}\mathbf{x}^{(0)}$
   $\rho_0 = \alpha = \omega_0 = 1\,;\ \mathbf{v}^{(0)} = \mathbf{p}^{(0)} = 0\,;$
2:   **for** $k = 1, 2, ...$ **do**
3:     $\rho_k = \hat{\mathbf{r}^{(0)}}^T \cdot \mathbf{r}^{(k-1)}$
4:     $\beta = \dfrac{\alpha \rho_k}{\omega_{k-1} \rho_{k-1}}$
5:     $\mathbf{p}^{(k)} = \mathbf{r}^{(k-1)} + \beta(\mathbf{p}^{(k-1)} - \omega_{k-1} \mathbf{v}^{(k-1)})$
6:     $\mathbf{y} = \mathbf{M}^{-1} \mathbf{p}^{(k)}$
7:     $\mathbf{v}^{(k)} = \mathbf{A}\mathbf{y}$
8:     $\alpha = \dfrac{\rho_k}{\hat{\mathbf{r}^{(0)}}^T \cdot \mathbf{v}^{(k)}}$
9:     $\mathbf{s} = \mathbf{r}^{(k-1)} - \alpha \mathbf{v}^{(k)}$
10:     **if** $\|\mathbf{s}\|$ est assez petite **then**
11:       $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha \mathbf{y}$
12:       break
13:     **end if**
14:     $\mathbf{z} = \mathbf{M}^{-1} \mathbf{s}$
15:     $\mathbf{t} = \mathbf{A}\mathbf{z}$
16:     $\omega_k = \dfrac{\mathbf{t}^T \cdot \mathbf{s}}{\mathbf{t}^T \cdot \mathbf{t}}$
17:     $\mathbf{x}^{(k+1)} = \mathbf{x}^{(k)} + \alpha_k \mathbf{p}^{(k)} + \omega \mathbf{z}$
18:     **if** $\mathbf{x}^{(k+1)}$ est assez précis **then**
19:       break
20:     **end if**
21:     $\mathbf{r}^{(k+1)} = \mathbf{s} - \omega \mathbf{t}$
22:   **end for** k
23: **end procedure**

1.2 Les machines parallèles

1.2.1 Les architectures parallèles

L'optimisation des processeurs est un sujet en constante évolution depuis les années 1980. Dans [41], nous avons une représentation des différents processeurs de 1978 à 2012 en fonction de leurs performances. Aujourd'hui, les processeurs à parallélisme multi-niveaux sont devenus un standard. Le parallélisme applique le principe de "diviser pour mieux régner". Pour résoudre un problème donné, ce dernier est divisé en parties indépendantes qui seront traitées de manière indépendante. Quand on parle de calcul parallèle, il y a deux notions fondamentales, les données et les instructions. Les différentes machines parallèles peuvent être classées en quatre catégories selon la façon dont ces deux notions sont traitées, . Ceci a été proposé dans [31] et connu sous le nom de taxonomie de Flynn. En jouant sur les deux états, simple ou multiple, des deux notions

introduites ci-dessus (données ou instructions), on obtient la classification suivante :

— **Single Instruction stream Single Data stream (SISD) :** Cette architecture concerne les processeurs uniques qui sont conçus pour gérer des programmes séquentiels. Les premiers modèles d'ordinateurs étaient équipés de ce type d'architecture.

— **Single Instruction stream Multiple Data stream (SIMD) :** Cette architecture est conçue pour le calcul parallèle. Une même instruction sera exécutée sur plusieurs données. Cela signifie que plusieurs unités de calcul seront capables d'exécuter la même instruction sur différentes données en même temps. Les processeurs utilisant ce type d'architecture sont appelés processeurs vectoriels.

— **Multiple Instruction stream Single Data stream (MISD) :** Chaque unité de calcul exécute indépendamment plusieurs instructions sur les mêmes données. Même si cette architecture est conçue pour le calcul parallèle, il n'existe aucun processeur commercialisé basé sur cette architecture.

— **Multiple Instruction stream Multiple Data stream (MIMD) :** Avec cette architecture, chaque processeur sera capable d'exécuter différentes instructions sur ses propres données. Ces exécutions peuvent être synchrones ou asynchrones. On a donc une architecture plus flexible mais elle est généralement plus coûteuse à utiliser si l'on ne fait pas attention à la taille du grain. C'est une architecture conçue pour le calcul parallèle et qui est utilisée par de nombreux supercalculateurs modernes.

Cette classification donne une idée générale des différentes architectures utilisées sur les processeurs. Cependant, dans la pratique, il s'agit plutôt de l'hybridation de ces différentes classes. Par exemple, les architectures MIMD ou SISD peuvent inclure des unités capables de gérer du SIMD.

1.2.2 Niveau de parallélisme

Dans certains cas, la manipulation des données peut être hiérarchisée pour le calcul en parallèle. Il existe deux types de niveaux de parallélisme dans les applications :

— **Parallélisme des données (DP) :** La manipulation d'une grande quantité de données peut être effectuée en même temps. Le parallélisme est ici induit par la répartition des données sur les différents processeurs. Ce type de parallélisme est utilisé dans les machines à mémoire distribuée.

— **Parallélisme au niveau des tâches (TLP) :** Ici, l'ensemble du travail est divisé en unités de travail appelées tâches. Ces tâches peuvent être exécutées en parallèle ou gérées dans un certain ordre. L'accomplissement de toutes ces tâches fournit le résultat final attendu.

Le même principe est appliqué aux instructions :

— **Parallélisme au niveau des instructions (ILP) :** Il consiste à utiliser des pipelines pour chevaucher l'exécution des instructions. Ainsi, si les données ne sont pas dépendantes les unes des autres, plusieurs instructions peuvent être exécutées en parallèle. Ce parallélisme d'instructions est possible physiquement sur le matériel et virtuellement par le compilateur. Le calcul parallèle avec SIMD est un exemple de ILP.

1.2.3 Évolution des architectures

Pour pouvoir simuler des problèmes vastes et complexes, nous avons besoin d'une grande puissance de calcul. Cette puissance est principalement liée au nombre d'unités de calcul pouvant travailler en même temps et à la fréquence à laquelle elles sont synchronisées. Cette puissance est mesurée en FLOP/s, c'est-à-dire le nombre d'opérations par seconde qu'un ordinateur peut effectuer sur des nombres à virgule flottante. Les processeurs actuels intègrent de plus en plus de cœurs de calcul, plusieurs dizaines de cœurs. On les appelle des processeurs multi-cœurs ou *many-cores* pour les Intel Xeon Phi ou l'AMD GENOA par exemple. Pour obtenir encore plus de parallélisme, les cœurs sont indépendants, chacun ayant son propre pipeline d'instructions, ses registres et ses unités arithmétiques. Ainsi, d'un point de vue macroscopique, chaque cœur est comme un processeur. Néanmoins, les cœurs vont partager entre eux un ou plusieurs niveaux de cache et l'accès au bus mémoire. Les caches sont des zones de mémoire proches des cœurs qui vont permettre au processeur de stocker certains éléments de la mémoire de manière à minimiser la latence d'accès aux données. Tous les cœurs partagent le cache L3. Dans certains processeurs, le cache L2 est également partagé entre les cœurs, sinon il est privé, tout comme le cache L1. Leurs numéros indiquent leur proximité avec le cœur et plus le cache est proche du cœur, plus l'accès est rapide. Mais sa capacité diminue à mesure que le cache se rapproche du cœur. Ainsi, le cache L1 est le plus rapide mais avec la plus petite capacité de stockage et le cache L3 est le plus lent des trois mais avec la plus grande capacité de stockage. On a l'habitude de les symboliser dans une pyramide avec le cache L3 à la base de la pyramide, le cache L2 au milieu et le cache L1 au sommet de la pyramide. Nous avons mentionné l'amélioration des performances du processeur grâce à la multiplication des cœurs au sein du processeur. Les cœurs ont également la capacité de traiter des instructions SIMD telles que *Streaming SIMD Extension* (SSE), *Advanced Vector Extension* (AVX) ou *Scalable Vector Extension* (SVE). De plus, la génération actuelle de multiprocesseurs offre également la possibilité à un cœur de traiter plusieurs flots d'instructions simultanément. C'est ce qu'on appelle l'hyperthreading [1]. Cette technologie multiplie encore les performances de calcul des processeurs en augmentant le nombre de cœurs. En résumé, nous avons donc des processeurs qui possèdent plusieurs dizaines de cœurs, chacun d'entre eux pouvant gérer jusqu'à quatre flots d'instructions simultanément combiné à la capacité d'effectuer des calculs vectoriels. Du point de vue du programmeur, une attention particulière doit être portée aux codes parallèles s'exécutant sur ces processeurs modernes. En effet, afin de tirer le meilleur parti de ces processeurs, nous avons besoin d'un grain de parallélisme plus fin que celui déjà requis par les unités SIMD.

Définition 1.6. La granularité dans le calcul parallèle est un concept important qui indique une mesure qualitative du rapport entre les calculs et les communications. On parle généralement de deux types de granularité, la granularité fine et la granularité grossière.

— **Granularité grossière** signifie qu'une quantité relativement importante de calcul est effectuée entre les événements de communication.

— **Granularité fine** signifie qu'une quantité relativement faible de calcul est effectuée entre les communications.

Pour accélérer encore plus les calculs, la résolution des problèmes s'effectue aujourd'hui sur des *Graphical Processing Units* (GPU). Le GPU regroupe un grand nombre d'unités de calcul simples (unités arithmétiques et logiques) qui sont capables d'effectuer des opérations. Chaque cœur des GPU exécute un flot d'instructions identiques mais sur des données différentes. C'est le

modèle *Single Instruction Multiple Data* (SIMD). Ainsi, le même programme est exécuté n fois sur n ensembles de données différents, en parallèle. Les GPU sont organisés avec les mêmes systèmes de cache que les CPU, mais avec une hiérarchie différente. Les architectures les plus connues proviennent de Nvidia, comme Nvidia Tesla [62].

1.2.4 Systèmes à mémoire partagée

Jusqu'à présent, nous avons mis en évidence l'évolution des performances des processeurs et leurs architectures embarquant de plus en plus d'unités de calcul. Mais les performances des processeurs dépendent également de la manière dont un processeur peut accéder aux données. Nous avons évoqué une certaine hiérarchie de la mémoire et leur latence respective. Par conséquent, comprendre comment la mémoire est gérée dans le calcul parallèle est aussi important que les performances de calcul du processeur. Si nous effectuons les calculs rapidement mais que nous perdons tout notre temps à chercher et charger des données, cela rend l'application totalement inefficace. Pour le calcul parallèle, il existe généralement deux types de systèmes de gestion de la mémoire. Soit le calcul parallèle est effectué en mémoire partagée, soit en mémoire distribuée.

— **Mémoire partagée :** Cela signifie que tous les processeurs ont une vue directe sur l'ensemble de la mémoire physique. Ils peuvent adresser ou accéder au même emplacement mémoire en utilisant les bus d'accès mémoire.
— **Mémoire distribuée :** Les processeurs ne peuvent adresser que la mémoire locale de la machine et doivent utiliser des communications pour accéder à la mémoire d'autres machines. Ils doivent utiliser des réseaux d'intercommunication pour accéder à la mémoire des autres machines où d'autres processeurs s'exécutent.

Dans ce travail, nous nous concentrons sur la programmation en mémoire partagée. Dans ce cas, la mémoire est visible par tous.

— *Symmetric Multi-Processing* **(SMP) :** Avec cette architecture, chaque processeur dispose de sa propre mémoire cache et accède de manière symétrique à une mémoire commune à l'aide de bus mémoire. La mise en oeuvre de ce type d'architecture avec beaucoup d'unités de calcul pourrait avoir un impact négatif sur les performances des applications. Aujourd'hui, on a délaissé l'architecture SMP au profit de l'architecture NUMA.
— *Non-Uniform Memory Access* **(NUMA) :** Avec cette architecture, les processeurs sont organisés en groupes et chaque groupe accède à une partie de la mémoire. Ces groupes sont appelés nœuds NUMA et l'accès à leur mémoire est symétrique, comme pour l'accès SMP. La mémoire est donc distribuée au sein des groupes et si un nœud veut accéder à une autre zone de mémoire, il doit utiliser un réseau d'interconnexion. Cependant, la latence de cette architecture dépendra du fait que la donnée adressée soit locale ou distante du processeur.

De nos jours, ces architectures NUMA sont largement utilisées dans les processeurs multi-cœurs. Les cœurs sont organisés en nœuds NUMA et ils ont un accès local à la mémoire qui est attachée à leur nœud NUMA. Nous avons maintenant une vue globale de ce qui se passe, au niveau matériel, pour le calcul parallèle.

2

Préconditionnement pour un solveur de Krylov : Factorisation LU Incomplète

Contents

2.1 Introduction **17**
2.2 État de l'art **18**
2.2.1 Renumérotations et factorisation LU incomplète 21
2.3 Coloriage de graphe **27**
2.3.1 Le concept de ColorRCM 27
2.3.2 Mise-en-oeuvre ColorRCM 29
2.4 Expérimentations numériques **31**
2.4.1 Impact de la renumérotation 32
2.5 Discussion **37**

2.1 Introduction

Depuis la fin des années 30, le concept « d'accélération » des méthodes itératives a commencé à être introduit, notamment par Cesari référencé dans [11]. Dans les années 70, le sujet du préconditionnement avec une factorisation incomplète a commencé à être largement exploré, notamment dans le cas des matrices symétriques définies positives (SPD) en particulier dans [19, 57]. Depuis, de nombreuses variantes de la factorisation incomplète ont été introduites pour résoudre le problème de la stabilité des factorisations incomplètes, parmi lesquelles [29, 39, 82]. Dans [70], nous trouvons une étude historique sur les avancées majeures des méthodes itératives au cours du 20ème siècle, notamment sur le thème du préconditionnement par factorisation incomplète. En 2002, une étude menée par Benzi [8], offre un large éventail des techniques de préconditionnement des solveurs itératifs.
En pratique, le préconditionnement de (1.1) peut être effectué de différentes manières. Considérons

$\mathbf{M}$ une matrice carrée non singulière à valeurs réelles telle que $\mathbf{M}$ soit proche de $\mathbf{A}$. Supposons que $\mathbf{M}$ s'écrive sous la forme du produit $\mathbf{M} = \mathbf{M}_L\mathbf{M}_R$. Alors (1.1) peut être remplacé par le système équivalent suivant :

$$\text{(Préconditionnement à droite et à gauche)} \qquad \mathbf{M}_L^{-1}\mathbf{A}\mathbf{M}_R^{-1}\mathbf{u} = \mathbf{M}_L^{-1}\mathbf{b}, \text{ avec } \mathbf{x} = \mathbf{M}_R^{-1}\mathbf{u} \tag{2.1}$$

Donc, le préconditionnement d'un système (1.1) implique la transformation du système original en un système de la forme de (2.1). Si nous considérons les cas de figure suivants : $\mathbf{M} = \mathbf{M}_R$ ou $\mathbf{M} = \mathbf{M}_L$ (c'est-à-dire $\mathbf{M}_R = \mathbf{I}$ ou $\mathbf{M}_L = \mathbf{I}$ avec $\mathbf{I}$ la matrice d'identité), nous avons respectivement

$$\text{(Préconditionnement à gauche)} \qquad \mathbf{M}_L^{-1}\mathbf{A}\mathbf{x} = \mathbf{M}_L^{-1}\mathbf{b} \tag{2.2}$$

$$\text{(Préconditionnement à droite)} \qquad \mathbf{A}\mathbf{M}_R^{-1}\mathbf{u} = \mathbf{b}, \text{ avec } \mathbf{x} = \mathbf{M}_R^{-1}\mathbf{u} \tag{2.3}$$

Malgré le coût calculatoire par itération plus élevé lorsque l'on applique une méthode de Krylov à un système préconditionné, le temps de résolution de ce dernier peut être nettement inférieur au temps de résolution du système non préconditionné grâce à une convergence plus rapide. En effet, un bon préconditionneur doit aider le solveur à converger rapidement (en quelques itérations, bien moins que la résolution sans préconditionnement). La difficulté reste cependant dans le choix du préconditionneur $\mathbf{M}$. Ceux-ci sont catégorisés en deux familles : les préconditionneurs explicites et implicites.

2.2 État de l'art

Les préconditionneurs sont utilisés dans les solveurs itératives pour réduire le nombre d'itérations nécessaire pour converger. Parmi les préconditionneurs, il y a les méthodes de préconditionnement par inverse approcher, telles que les méthodes d'approximation de l'inverse. L'idée de ces méthodes est de chercher une certaine matrice $\mathbf{K}$ qui sera une approximation de $\mathbf{A}^{-1}$ pour le préconditionnement à gauche. Dans le livre [68], l'auteur a consacré un chapitre à l'explication de ces méthodes. Dans certains cas, la matrice calculée $\mathbf{K}$ tend à être plus dense que la matrice $\mathbf{A}$ [28]. Des variantes de cette méthode, telles que les SPAI (Sparse Approximate Inverse), ont été développées pour prendre en compte le caractère creux de la matrice $\mathbf{A}$. Elles sont basées sur des critères de seuil pour choisir si le nouvel élément doit être ajouté ou ignoré [37]. D'autres méthodes, comme la FSAI (Factorized Sparse Approximate Inverse) [51], fournissent une factorisation de la matrice $\mathbf{K}$ en facteurs triangulaires. Dans cette catégorie, on peut également citer les préconditionneurs polynomiaux, qui visent à représenter l'inverse de $\mathbf{A}$ par un polynôme sur $\mathbf{A}$. Ce polynôme est ensuite utilisé comme préconditionneur [25].

Les préconditionneurs implicites

Une autre approche consiste à chercher une matrice $\mathbf{M}$ qui sera une approximation de $\mathbf{A}$. Une première approche est d'utiliser le même principe que les méthodes itératives stationnaires. On définit donc $\mathbf{M} = \mathbf{M}_J$ (1.4) ou $\mathbf{M} = \mathbf{M}_{GS}$ (1.5) ou $\mathbf{M} = \mathbf{M}_{SOR}$ (1.6). Ces méthodes ont l'avantage d'être peu coûteuses à construire et à stocker.
Une autre classe importante de préconditionneur consiste à approximer la matrice $\mathbf{A}$ par un

produit de matrices ayant des structures simples comme des matrices diagonales ou des matrices triangulaires. Il s'agit principalement de méthodes basées sur des factorisations incomplètes. L'idée ici est d'approximer les facteurs triangulaires **L** et **U** de la factorisation LU exacte en respectant un critère donné, comme préserver la structure creuse de la matrice **A**. On a alors $\mathbf{A} = \tilde{\mathbf{L}}\tilde{\mathbf{U}} - \mathbf{R}$ où **R** est la matrice résiduelle. Par conséquent, plus la norme de **R** sera petite et plus $\tilde{\mathbf{L}}$, $\tilde{\mathbf{U}}$ tendront vers **L**, **U**. Le principe reste le même pour une factorisation Cholesky incomplète. Nous allons pour la suite de la rédaction nous intéresser exclusivement à la Factorisation LU Incomplète. Une factorisation incomplète réduit le nombre d'opérations flottantes et le coût mémoire par rapport à une factorisation exacte *LU* en ignorant tout ou partie du remplissage introduit dans les facteurs **L** et **U**. Par exemple, une manière simple de contrôler ce remplissage consiste à conserver strictement la structure creuse de la matrice **A** dans chacun des facteurs. Considérons $\mathcal{P}$ un sous-ensemble d'indices de lignes et colonnes de **A** tel que

$$\mathcal{P} \subset (i,j) \quad | \quad i \neq j; \quad 1 \leq i,j \leq n \tag{2.4}$$

La factorisation ILU effectuée sur l'ensemble $\mathcal{P}$ va négliger toutes les opérations sur les éléments $a_{i,j}$ tels que $(i,j) \notin \mathcal{P}$. On a alors $\mathbf{A} = \tilde{\mathbf{L}}\tilde{\mathbf{U}} - \mathbf{R}$ tels que les éléments de **R** sont les valeurs de **A** ignorées lors de la factorisation. Lorsque l'ensemble $\mathcal{P}$ est égal à l'ensemble des indices des éléments non nuls de **A** que nous notons $\mathcal{P}0$:

$$\mathcal{P}0 \stackrel{\text{def}}{=} (i,j) \quad | \quad i \neq j \text{ et } a_{i,j} \neq 0; \quad 1 \leq i,j \leq n \tag{2.5}$$

La factorisation ILU effectuée sur $\mathcal{P}_0$ est notée ILU(0).

Algorithm 2 Factorisation incomplète ILU(0)

```
 1: procedure ILU0(A)
    Inputs : Matrice A ∈ ℝ^{n,n}
 2:   for (i = 2 ; i < n ; i = i + 1) do
 3:     for (k = 1 ; k < i − 1 ; k = k + 1) do
 4:       if (i, k) ∈ P_0 then
 5:         a_{i,k} = a_{i,k} / a_{k,k}
 6:         for (j = k + 1 ; j < n ; j = j + 1) do
 7:           if (i, j) ∈ P_0 then
 8:             a_{i,j} = a_{i,j} − a_{i,k} ∗ a_{k,j}
 9:           end if
10:         end for j
11:       end if
12:     end for k
13:   end for i
14: end procedure
```

Avec une factorisation en place, cet algorithme ILU(0) a le même coût de stockage que la matrice **A**. Le fait de se limiter strictement à la structure creuse de la matrice **A** implique parfois de négliger un trop grand nombre de valeurs dans les facteurs. Cela se reflète généralement par un nombre élevé d'itérations du solveur. Pour augmenter la performance numérique de la factorisation incomplète, plusieurs variantes de cet algorithme ont été proposées. Par exemple, la factorisation

ILU(k) permet plus de remplissage que ILU(0) lorsque k est strictement supérieur à zéro. Le facteur k est considéré comme le niveau de remplissage de la factorisation. Pour ILU(k), tout élément $a_{i,j}$ qui a un niveau de remplissage supérieur à k sera négligé. Le niveau de remplissage des éléments est défini de la manière suivante :
Initialement, $lev(a_{i,j}) = \begin{cases} 0, \text{ si } a_{i,j} \neq 0 \text{ ou } i = j \\ \infty, \text{ sinon} \end{cases}$, puis à chaque étape de ILU(k), le niveau de l'élément $a_{i,j}$ est évalué comme étant $lev(a_{i,j}) = \min lev(a_{i,j}), lev(a_{i,k}) + lev(a_{k,j}) + 1$. Donc, par définition, les éléments initialement non nuls de la matrice $\mathbf{A}$ auront un niveau de remplissage égal à 0, de sorte que si $k = 0$ alors cela est équivalent à calculer une factorisation ILU(0). Cette variante ILU(k), tout comme son homologue ILU(0), est basée uniquement sur la structure de la matrice. D'autres variantes prennent aussi en compte les valeurs des facteurs pour décider si le nouvel élément peut être conservé ou non. C'est le cas par exemple de la méthode de factorisation incomplète à double seuil ILUT(τ, p) introduite dans [68]. L'idée de cette approche est d'ignorer tous les éléments de la ligne (colonne) i du facteur $\tilde{\mathbf{L}}$ (resp. $\tilde{\mathbf{U}}$) dont la magnitude est inférieure à la tolérance seuil $\tau_i = \tau \|a_{i,*}\|_2$ et de ne conserver qu'au plus les p plus grands. L'une des difficultés qui se pose avec l'utilisation de ILUT est le choix de la valeur de τ. Celle-ci est souvent liée au type de problème manipulé. Dans la littérature, on observe souvent un seuil de tolérance compris entre $[10^{-4}, 10^{-2}]$. En règle générale, il n'y a aucune garantie que la factorisation ILU puisse être calculée jusqu'au bout. L'échec de la factorisation ILU peut être dû à différentes causes, comme des pivots trop petits (proches du zéro machine). Pour éviter ce type de situation, il y a la variante ILUTP(τ, p) qui propose une factorisation incomplète à double seuil comme ILUT(τ, p), mais avec une stratégie de pivotage en plus [68]. Pour ne pas perdre certaines bonnes propriétés de la matrice de départ, des approches comme le ILUM (ILU Modifié) ont été introduites. L'idée est de garantir que la somme des lignes des facteurs $\tilde{\mathbf{L}}$ et $\tilde{\mathbf{U}}$ est égale à la somme des lignes de $\mathbf{A}$, c'est-à-dire $\mathbf{A}e = \mathbf{LU}e$ ou $e = (1, ..., 1)^T$, en ajoutant aux pivots des éléments de remplissage négligés lors de l'élimination. Cela peut être intéressant par exemple pour des matrices à diagonale dominante. L'ensemble des techniques introduites ci-dessous date généralement de la fin du 20ème siècle. Plus récemment, une approche originale pour calculer les facteurs incomplets $\tilde{\mathbf{L}}$ et $\tilde{\mathbf{U}}$ a été proposée dans [16]. L'idée est de calculer les éléments des facteurs $\tilde{\mathbf{L}}$ et $\tilde{\mathbf{U}}$ en résolvant le système suivant :

$$\begin{cases} \tilde{l}_{i,j} &= \dfrac{1}{u_{i,i}} \left(a_{i,j} - \sum\limits_{k=1}^{j-1} \tilde{l}_{i,k} \tilde{u}_{k,j} \right), \forall (i,j) \in \mathcal{P} \text{ et } i > j \\ \tilde{u}_{i,j} &= a_{i,j} - \sum\limits_{k=1}^{i-1} \tilde{l}_{i,k} \tilde{u}_{k,j}, \forall (i,j) \in \mathcal{P} \text{ et } i \leq j \end{cases}$$

$\mathcal{P}$ est le sous-ensemble des indices de $\mathbf{A}$. En posant $\mathbf{z}$ le vecteur contenant les inconnues $\tilde{l}_{i,j}$ et $\tilde{u}_{i,j}$, on peut réécrire le problème sous la forme $\mathbf{z}^{(n+1)} = g(\mathbf{z}^{(n)})$. Ainsi, on approxime les valeurs des facteurs $\tilde{\mathbf{L}}$ et $\tilde{\mathbf{U}}$ avec quelques itérations de la méthode de point fixe, appelée *sweep*. L'avantage de cette approche réside dans la possibilité de mettre à jour simultanément toutes les composantes des facteurs incomplets [5, 16]. Les algorithmes ILU classiques introduits précédemment n'ont pas naturellement ce parallélisme à grain fin. Pour cela, on utilise communément des méthodes de renumérotation des équations du système pour exhiber une structure particulière dans la matrice permutée. Ces renumérotations sont utilisées en pré-traitement du solveur. La majorité des factorisations incomplètes classiques engendrent du remplissage, ce qui peut avoir pour effet de détériorer tout ou partie de la structure apportée par la permutation. Soulignons que certaines méthodes de renumérotation des équations permettent de prédire a priori la zone du remplissage

sur la matrice. Cependant, le préconditionneur ILU(0) nous garantit que tout le remplissage sera ignoré, préservant ainsi la structure de la matrice permutée. Pour la suite du manuscrit, nous travaillerons exclusivement avec ce préconditionneur.

2.2.1 Renumérotations et factorisation LU incomplète

Les méthodes de renumérotation des équations d'un système sont maintenant des techniques couramment utilisées en pré-traitement des méthodes itératives préconditionnées. Néanmoins, leur usage a été originellement introduit dans le contexte des méthodes directes pour répondre à des contraintes de stockage ou de parallélisme. C'est pour ce deuxième aspect que ces méthodes vont particulièrement nous intéresser. Les méthodes de renumérotation ont un intérêt particulier pour nous pour les raisons suivantes :

1. Améliorer l'efficacité numérique du préconditionneur pour accélérer la convergence du solveur.
2. Introduire du parallélisme pour la construction du préconditionneur et son application.

L'idée générale de ces méthodes est de trouver une permutation de la matrice $\mathbf{A}$ en changeant la numérotation des sommets de $\mathcal{G}$. Pour nous, cette permutation sera toujours symétrique pour conserver la structure symétrique de la matrice. Ainsi, au lieu de résoudre (1.1), nous résolvons :

$$\mathbf{PAP}^T\mathbf{z} = \mathbf{Py}, \quad \mathbf{x} = \mathbf{P}^T\mathbf{z} \tag{2.6}$$

où $\mathbf{P}$ est une matrice de permutation et $\mathbf{P}^T\mathbf{P} = \mathbf{PP}^T = \mathbf{I}$.

Quelques méthodes de renumérotations célèbres

À l'origine, les méthodes de renumérotation des équations du système ont été principalement étudiées dans le contexte des factorisations creuses pour les méthodes directes, car c'est un moyen de réduire le remplissage qui apparaît dans les facteurs triangulaires ou d'améliorer l'accès aux données en optimisant leur localité en mémoire. Les méthodes de renumérotation telles que Cuthill-McKee (CM) [20] et Reverse Cuthill-McKee (RCM) [34], Sloan [72] ou Gibbs-Poole-Stockmeyer (GPS) [35] sont connues pour leur capacité à réduire la largeur de bande et l'enveloppe des matrices. Parmi ces heuristiques, RCM est la plus référencée et étudiée [56]. RCM est une amélioration de l'algorithme CM, qui est l'un des premiers algorithmes proposés pour résoudre la problématique de la réduction de la largeur de bande et du profil de la matrice. Une solution à ces problèmes est intéressante pour l'optimisation de la localisation spatiale des données en mémoire. Historiquement, les méthodes réduisant la largeur de bande des matrices étaient utilisées dans un format de stockage appelé Skyline pour la factorisation incomplète et la résolution du système [17].

Le principe de RCM consiste à parcourir le graphe en largeur en partant d'un sommet privilégié (un sommet "périphérique") pour trouver une nouvelle numérotation des sommets qui sera basée sur l'ordre inverse des visites des sommets par l'algorithme. La différence entre CM et RCM se situe à la dernière étape de RCM. Celui-ci va inverser l'ordre obtenu par CM (voir Algorithme 3 étape (23)). D'autres méthodes telles que Minimum Degree (MD) [33], Nested Dissection (ND) [32], connues pour leur efficacité à minimiser le remplissage dans la factorisation exacte des matrices, ont également été largement étudiées dans la littérature. De plus, Nested Dissection est

principalement utilisée dans les solveurs itératifs pour sa capacité à révéler une structure par blocs de la matrice permutée qui permet d'exploiter du parallélisme gros grain pour les applications. D'autres méthodes qui apportent un bon degré de parallélisme avec une granularité plus fine sont les méthodes basées sur la coloration de graphe. Le concept du MULTI-COLORIAGE de graphe pour les problèmes sur grille non structurée est référencé dans [27] comme étant popularisé par [50]. Le principe de base des MULTI-COLORIAGE (GREEDYMC) est de donner une couleur (un label) à chaque sommet du graphe de telle sorte que la couleur d'un sommet soit différente de ses voisins.

Définition 2.1 (Degré de parallélisme)**.** Le degré de parallélisme est le nombre minimum d'opérations qui peuvent être calculées simultanément par des processeurs lors de l'exécution parallèle d'une application.

Ces méthodes de coloriage permettent de former des groupes d'équations indépendantes qui peuvent donc être résolues simultanément. Généralement, on cherche à colorier le graphe en utilisant un nombre minimal de couleurs pour maximiser le nombre d'équations qui pourront être résolues simultanément. La matrice permutée suite à une renumérotation par MULTI-COLORIAGE présente une structure par blocs, avec sur la diagonale, des sous-matrices diagonales. Avec certains problèmes sur grille structurée 2D ou 3D, on peut colorier l'ensemble du graphe avec seulement 2 couleurs, c'est ce qu'on appelle la renumérotation Rouge-Noir (ou Red-Black). C'est l'un des MULTI-COLORIAGE les plus connus, car il offre le degré de parallélisme le plus élevé.

Mise en oeuvre de certaines renumérotation

Reverse Cuthill-McKee : Le parcours en largeur du graphe de RCM est basé sur l'algorithme BFS (Breath First Search) [60, 71]. Cet algorithme consiste à visiter l'ensemble des sommets d'un graphe connexe et non orienté de voisin en voisin et en partant d'un sommet initial. Il va donc construire une arborescence de tous les sommets du graphe accessible en partant du sommets initial, c'est-à-dire l'ensemble des sommets du graphe pour un graphe connexe. La structure de données la mieux adaptée pour cet algorithme est la structure de file. La file est une structure de données abstraite avec la propriété suivante : *FIFO (First In, First Out)*. Le premier élément inséré dans la file sera le premier élément à sortir de la file, aussi désigné par l'acronyme FIFO.

1. Enfiler le sommet initial dans la file.
2. Défiler le premier élément de la file.
3. Enfiler tous les voisins non visités de l'élément qui vient d'être défilé.
4. Si la file n'est pas vide alors, aller à l'étape 2

L'algorithme BFS tel que décrit ci-dessus montre qu'à chaque étape, on sera en présence d'un ensemble de sommets (les voisins des sommets précédents) sans relation d'ordre particulière entre eux. Par défaut, BFS prend les sommets par ordre croissant de leur numérotation. RCM lève cette indécision en ordonnant les sommets par ordre croissant de leur degré. Cependant, il arrive qu'on puisse avoir plusieurs sommets de même degré. Dans ce cas, RCM choisira l'ordre ascendant des numérotations des sommets pour résoudre l'indécision entre les sommets de même degré. À noter que l'ordre de visite des sommets dépend du choix du sommet initial. Donc, l'efficacité de RCM dépend aussi du choix de ce sommet initial. Pour faire ce choix, il existe différentes approches. L'une d'elles consiste à choisir un sommet périphérique du graphe. Pour cela, il faut

explorer plusieurs fois le graphe jusqu'à converger vers ce ou ces sommets périphériques. Cette approche étant très coûteuse en temps, dans [35] les auteurs ont proposé de rechercher un sommet pseudo-périphérique. Leur méthode est moins coûteuse et la convergence est rapide. Une autre approche beaucoup moins coûteuse consiste à prendre simplement le sommet de plus petit degré comme sommet initial. C'est cette approche que nous utiliserons par la suite.

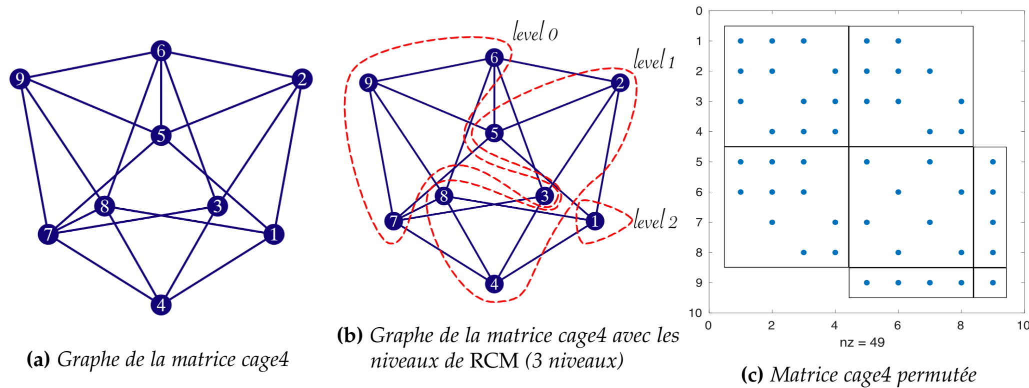
Algorithm 3 REVERSE CUTHILL-MCKEE

```
1:  procedure RCM(G, u)
    Inputs : Graphe : G = {V, E} et Sommet initial : u
2:      Créer une file Q
3:      Vecteur booléen de taille |V| : visite
4:      Vecteur d'entiers de taille |V| : R  ▷ R va contenir la nouvelle numérotation des sommets
5:      Entier : idx = 1
6:      Initialiser visite[:] = Faux
7:      Initialiser R[:] = −1
8:      Q.ENFILER(u)
9:      visite[u] = Vrai
10:     R[0] = u
11:     while !Q.VIDE() do
12:         u = Q.tete()
13:         Q.DEFILER()
14:         Trier les voisins non-visité de u dans l'ordre croissant de leur degré
15:         for v voisin trier de u do
16:             if !visite[v] then
17:                 Q.ENFILER(v)
18:             end if
19:         end for
20:         R[idx] = u
21:         idx = idx + 1
22:     end while
23:     Inverser l'ordre des sommets dans R
24: end procedure
```

Lors de la manipulation des structures en file, les opérations d'enfilement et défilement s'exécutent en $O(1)$. L'initialisation du vecteur **visite** et **R** se fait en $O(n)$, $n = |\mathcal{V}|$ étant le nombre de sommets de G. Dans la boucle **While** la condition qu'on ne puisse insérer dans la file **Q** que les sommets non visités garantit que chaque sommet est enfilé et défilé au plus une fois. Donc le coût de ces opérations est de l'ordre de $O(n)$. Comme chaque sommet est visité au plus une fois. Donc la liste de ses voisins est aussi parcourue et triée au plus une fois. Ce qui correspond au plus au cardinal de l'ensemble des arêtes du graphe, $M = |\mathcal{E}|$. Le coût de balayage des voisins de chaque sommet du graphe est donc de $O(M)$. Cependant le coup de tri des voisins de chaque sommet est dans le pire des cas de $O(M^2)$. Finalement, dans le pire des scénarios la complexité de l'Algorithme 3 tel que décrit dans est de $O(M^2)$. À noter qu'avec un bon algorithme de tri, par exemple la fonction *sort()* de la bibliothèque standard C, la complexité de RCM descend à $O(M \log_2(M) + n)$.

Nested Dissection : L'idée de la dissection emboîtée (ND) est la suivante :

1. Étant donné un graphe non orienté G.



(a) *Graphe de la matrice cage4*

(b) *Graphe de la matrice cage4 avec les niveaux de* RCM *(3 niveaux)*

(c) *Matrice cage4 permutée*

Figure 2.1 – *(2.1a) représente le graphe de la matrice cage4 (le profil de la matrice est représenté par la Figure 1.1a) et (2.1b) est une représentation des niveau de* RCM*. La Figure 2.3c est une représentation de la permutation symétrique de la matrice cage4 à l'issue de* RCM*.*

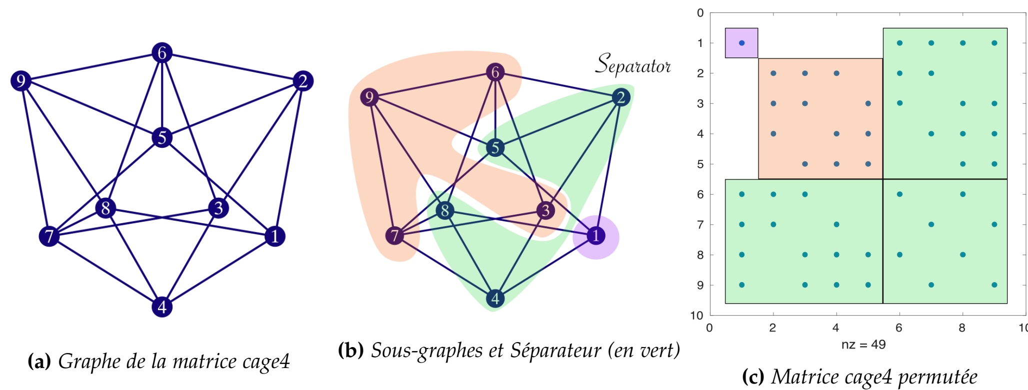
2. Parcourir G pour trouver des séparateurs qui vont partitionner le graphe en deux sous-graphes disjoints.
3. Répéter l'étape 2 dans chaque sous-graphe jusqu'à ce que certains critères soient validés.

Le séparateur est un ensemble de sommets dont la suppression entraîne la décomposition du graphe en deux sous-graphes disjoints. D'un point de vue matriciel, si l'on regroupe les inconnues associées aux deux sous-graphes ensemble et leurs séparateurs, on obtient une matrice par bloc 3×3, le bloc 2×2 est bloc diagonal associé aux 2 sous-graphes disjoints et un dernier bloc, celui des séparateurs, avec des termes non-nuls de connection avec les deux sous-graphes. Le ND introduit de cette manière un parallélisme naturel de traitement entre les blocs des sous-graphes. En revanche, tous les séparateurs dépendent de leurs partitions. On a donc une méthode de renumérotation mettant en évidence un parallélisme intrinsèque même si le niveau de parallélisme est fortement limité par la taille des séparateurs. Les méthodes comme NESTED DISSECTION ou MULTI-COLORIAGE sont également connues pour leur capacité à révéler une structure par blocs particulière dans la matrice permutée qui permet d'exploiter du parallélisme dans les calculs. NESTED DISSECTION est connue pour apporter un parallélisme gros grain alors que MULTI-COLORIAGE apporte un parallélisme grain fin dans les calculs. Les propriétés de la renumérotation ND ont été formalisées dans [32].

Coloriage de graphe : L'idée de la coloration des graphes est l'affectation de couleurs aux sommets du graphe $\mathcal{G}$, de manière à ce que deux sommets adjacents n'aient pas la même couleur. La conséquence est que la matrice résultant de l'ordre multicolore aura des blocs diagonaux qui sont des sous-matrices diagonales. Les blocs diagonaux représentent les classes de couleur, qui sont calculées comme suit :

$$\mathcal{C}_j = \{v \in \mathcal{V} \mid color(v) = j\}, \text{ avec } color(v) = \min(c \geq 0 \mid c \neq color(u), \forall u \in Adj(v)) \qquad (2.7)$$

En d'autres termes, trouver une coloration de $\mathcal{G}$, c'est trouver une partition $\mathcal{C}_0, \mathcal{C}_1, ..., \mathcal{C}_{ncolor-1}$ de $\mathcal{V}$, telle que les sommets dans les $\mathcal{C}_l$ sont indépendants les uns des autres. Habituellement, le problème de coloration des graphes revient à trouver le nombre chromatique $\chi(\mathcal{G})$, c'est-à-dire



(a) *Graphe de la matrice cage4*
(b) *Sous-graphes et Séparateur (en vert)*
(c) *Matrice cage4 permutée*

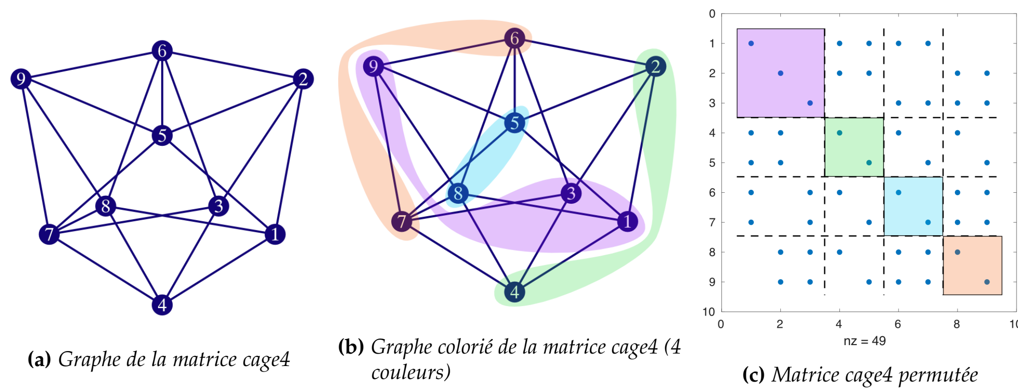
Figure 2.2 – *(2.2a) représente le graphe de la matrice cage4 (le profil de la matrice est représenté par la Figure 1.1a) et (2.2b) est une représentation des sous-graphes construit avec* ND *et du séparateur. La Figure 2.2c est une représentation de la permutation symétrique de la matrice cage4 à l'issue de* ND*.*

le nombre minimal de couleurs. Le problème consistant à trouver $\chi(\mathcal{G})$ peut ne pas garantir l'équilibre des couleurs, et il existe généralement un fort déséquilibre entre la première et la dernière couleur.Alors que les premières couleurs formées sont composées d'un grand nombre de sommets, on a remarqué que les dernières couleurs formées sont souvent composées de seulement quelques sommets (largement inférieurs à la taille des premières couleurs). Chercher à équilibrer la taille des couleurs est un domaine de recherche très actif et en lien direct avec le domaine du Calcul Haute Performance. Nous présenterons ici un algorithme de coloriage glouton pour colorier un graphe. En général, cet algorithme ne donne pas le nombre chromatique $\chi(\mathcal{G})$, mais essaie de minimiser le nombre de couleurs utilisées. L'algorithme est le suivant :

Algorithm 4 MULTI-COLORIAGE GLOUTON

1: **procedure** GREEDY_GRAPH_COLOR(Graph G,π)
2: **Inputs :** Graphe : $G = \{\mathcal{V}, \mathcal{E}\}$ et Vecteur de permutation : π
3: **for** $u \in \mathcal{V}$ dans l'ordre apporter par π **do**
4: **for** v voisin de u **do**
5: $color(u) = \min(c \geq 0 \mid c \neq color(v))$
6: **end for**
7: **end for**
8: **end procedure**

Nous considérerons les sommets dans un ordre spécifique $\pi = (u_1, ..., u_n)$ et attribuons à u_i, i un entier compris entre 1 et n, la plus petite couleur disponible non utilisée par ses voisins. En utilisant le même raisonnement que pour calculer la complexité algorithmique de RCM, l'algorithme `greedy_graph_color` parcourt tous les sommets ainsi que leurs voisins au plus une fois. Donc nous avons une complexité de $O(n + M)$ avec M et n étant respectivement le nombre d'arêtes du graphe et le nombre de sommets du graphe.



(a) *Graphe de la matrice cage4* **(b)** *Graphe colorié de la matrice cage4 (4 couleurs)* **(c)** *Matrice cage4 permutée*

Figure 2.3 – *(2.3a) représente le graphe de la matrice cage4 (le profil de la matrice est représenté par la Figure 1.1a) et (2.3b) est une représentation de la coloration du graphe (2.3a). La Figure 2.3c est une représentation de la permutation symétrique de la matrice cage4 à l'issue de la coloration du graphe.*

L'effet des renumérotations sur la convergence des solveurs de Krylov

L'impact des renumérotations dans la méthode du sous-espace préconditionné a été étudié pour la première fois dans [26] pour les matrices symétriques définies positives (SDP). Les auteurs ont signalé que l'utilisation de renumérotations comme MD, ND ou Red-Black (un Multi-Coloriage avec 2 couleurs) pour une factorisation de Cholesky incomplète sans remplissage (IC(0)) peut parfois fortement dégrader la convergence du solveur Gradient Conjugué (CG). Selon les observations faites dans [26], RCM est l'ordre qui se dégrade le moins, et parfois il est même meilleur que l'ordre original de la matrice. Dans de nombreuses études pour des cas non SDP, RCM est souvent l'ordre qui améliore le plus la convergence du solveur de Krylov [9, 10, 18, 67]. Les auteurs de [10] le recommandent comme renumérotation par défaut avec une Factorisation LU Incomplète si nous n'avons pas de connaissance préalable du problème à résoudre. Cependant, RCM ne fournit pas naturellement un parallélisme pour le calcul de la Factorisation LU Incomplète. L'effet des méthodes de coloriage de graphe sur les solveurs itératifs pour les méthodes sur grille structurée a été largement étudié depuis l'étude de [26] qui met en évidence le lien entre la convergence du solveur Gradient Conjugué préconditionné avec IC(0) (IC(0)-CG) et le degré de parallélisme. Il semble que plus on a de parallélisme, plus on dégrade la convergence du solveur [26]. De nombreuses raisons peuvent expliquer cette réduction de performance, voire l'échec de la convergence. En particulier, l'instabilité des facteurs, qui peut être causée par des pivots trop petits ou des facteurs mal conditionnés, ou simplement lorsqu'il y a eu trop de valeurs négligées lors de la factorisation incomplète. Les deux premiers points sont souvent symptomatiques de matrices non diagonales dominantes. Pour plus de détails, nous renvoyons les lecteurs à [8, 10, 18]. Dans [47], les auteurs ont proposé la méthode *Algebraic Block Multi Coloring* (BMC) pour la parallélisation multi-thread de la résolution triangulaire dans IC(0)-CG . Plus récemment, [48], la méthode *Hierarchical Block Multi Color* pour la parallélisation SIMD du solveur triangulaire dans IC(0)-CG a montré de meilleures performances numériques que BMC alors que les deux méthodes sont équivalentes. Cependant, la méthode *Hierarchical Block Multi Color* dégrade encore fortement la convergence du solveur IC(0)-CG . Dans la recherche d'une parallélisation de la Factorisation LU Incomplète, une nouvelle approche originale pour calculer les facteurs $\tilde{\mathbf{L}}$ et $\tilde{\mathbf{U}}$ a été proposée dans [16]. Avec cette approche, on peut naturellement

avoir une parallélisation à grain fin des calculs. Leur méthode est une approximation des facteurs **L** et **U** en résolvant un système d'équations non linéaires pour chaque élément non nul. Les auteurs ont montré que leur méthode ne nécessite que quelques itérations, appelées *sweeps*, pour fournir un préconditionneur ILU efficace. Néanmoins, dans nos travaux, nous nous sommes focalisés sur l'approche classique de la FACTORISATION LU INCOMPLÈTE. Dans ce contexte, très peu d'algorithmes ont été développés permettant une parallélisation à grain fin des calculs avec GREEDYMC tout en fournissant une FACTORISATION LU INCOMPLÈTE numériquement efficace.

2.3 Coloriage de graphe

En observant l'Algorithme 3, nous remarquons que RCM construit des ensembles de sommets à chaque étape qui sont les voisins des sommets précédents. On peut alors décrire RCM de la manière suivante :

1. Choisir un sommet initial u (par exemple un des sommets de plus petit degré).
2. $\mathcal{R} = \mathcal{L}_0 = u$.
3. Pour $i = 1$: *nlevel*
   $\mathcal{L}_i = Adj(\mathcal{L}_{i-1}) \backslash \mathcal{R}$ et trier les éléments de $\mathcal{L}_i$ dans l'ordre croissant de leur degré.
   $\mathcal{R} = \mathcal{R} \bigcup \mathcal{L}_i$.
4. Inverser l'ordre des sommets dans $\mathcal{R}$.

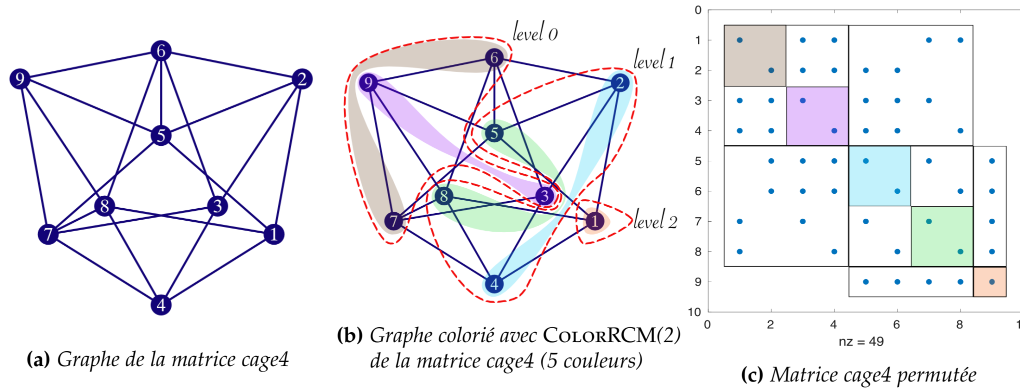
Alors l'ensemble $\mathcal{L}_i$ est juste connecté avec $\mathcal{L}_{i-1}$ et $\mathcal{L}_{i+1}$. Par conséquent, la matrice permutée avec RCM sera une matrice tridiagonale par bloc, mais les blocs n'ont pas nécessairement une structure particulière. L'idée est de chercher une coloration des sommets des sous-graphes $G_{\mathcal{L}_i}$ composés par les sommets dans $\mathcal{L}_i$. Ce qui se traduit de la manière suivante :

1. Calculer l'ensemble de niveaux $\mathcal{L} = \bigcup_{0=i}^{nlevel} \mathcal{L}_i$ à partir de RCM.
2. Calculer $\mathcal{C}_{i,j} = v \in \mathcal{L}_i \mid color(v) = j, 0 \leq i < nlevel,$ avec $color(v) = \min(c \geq 0 \mid c \neq color(u), \forall u \in Adj(v) \bigcap \mathcal{L}_i)$

$\mathcal{C}_{i,j}$ est la classe de couleur j du niveau i, c'est-à-dire l'ensemble des sommets du niveau i ayant la même couleur. Après la coloration des sommets de chaque ensemble $\mathcal{L}_i$, nous obtiendrons une matrice permutée tridiagonale par bloc et des sous-matrices diagonales à l'intérieur des blocs diagonaux (voir Figure 2.4c). Nous appelons cette renumérotation COLORRCM. La factorisation incomplète est sensible à l'ordre des sommets du graphe d'adjacence. L'ordre fourni par une coloration gloutonne du graphe affecte souvent négativement la convergence numérique du solveur. Ainsi, la coloration des sommets des sous-graphes $G_{\mathcal{L}_i}$ suivra l'ordre donné par RCM.

2.3.1 Le concept de COLORRCM

La première utilisation de COLORRCM est pour vectoriser les calculs pendant la résolution de systèmes triangulaires dans le solveur BICGSTAB. L'approche consistant à minimiser le nombre de couleurs est couramment utilisée pour obtenir un parallélisme à grain fin avec un grand degré de parallélisme. Il en résulte de grandes tailles de couleurs, c'est-à-dire de nombreux sommets dans les couleurs. Cependant, nous avons constaté deux inconvénients :



(a) *Graphe de la matrice cage4*

(b) *Graphe colorié avec* ColorRCM*(2) de la matrice cage4 (5 couleurs)*

(c) *Matrice cage4 permutée*

Figure 2.4 – *(2.4a) représente le graphe de la matrice cage4 (le profil de la matrice est représenté par la Figure 1.1a) et (2.4b) est une représentation du coloriage de graphe par* ColorRCM*. Les lignes pointillés rouge illustrent les ensembles de nœds (i.e. les niveaux) construit avec* RCM*. La Figure 2.4c est une représentation de la permutation symétrique de la matrice cage4 à l'issue de la coloration du graphe par* ColorRCM*(2).*

1. La convergence du solveur est souvent impactée négativement (beaucoup d'itérations).
2. Les dernières couleurs contiennent trop peu de sommets par rapport aux premières couleurs.

Pour répondre à l'observation (2), il faut chercher à colorier le graphe de manière équitable, c'est-à-dire que la différence des cardinales de deux couleurs soit au plus de 1. Ce problème a été introduit dans [59] et sa résolution est généralement plus difficile que le problème de coloriage standard. Dans [55], on trouve une étude sur le coloriage équitable de graphes et certaines situations sont citées pour lesquelles le coloriage équitable est NP-difficile. Une petite astuce pour réduire ce fort déséquilibre des couleurs consiste à définir une taille maximale de couleur, c'est-à-dire un nombre maximal de sommets qu'une couleur peut contenir. En partant du constat que la parallélisation des calculs se fait avec un nombre défini de coeurs, nous pouvons simplement limiter la taille des couleurs au nombre de coeurs ou threads fixés par l'utilisateur, par exemple. Dans notre cas, cette taille sera limitée à la taille d'un vecteur SIMD (pour la manipulation de valeurs en double précision : 8 en AVX512 et 4 en AVX2). Ainsi, la coloration de graphe ColorRCM avec la taille de couleur restreinte, notée ColorRCM(p), peut être décrite comme suit :

1. Calculer les ensembles de niveau $\mathcal{L} = \bigcup_{0=i}^{nlevel} \mathcal{L}_i$ à partir de RCM.
2. Calculer $\mathcal{C}_{i,j} = \{v \in \mathcal{L}_i \mid j = color(v) \text{ et } |\mathcal{C}_{i,j}| \leq p\}, \quad 0 \leq i < nlevel$ et $p \in \mathbb{N}$

Avec p le nombre maximum de sommets pour un ensemble $\mathcal{C}_{i,j}$, c'est-à-dire la taille de la couleur. L'idée est simplement d'essayer de minimiser le nombre de couleurs, mais dès qu'une classe de couleur atteint le nombre maximum de sommets p fixé par l'utilisateur, nous passons à une autre couleur. Cela aura pour effet d'augmenter significativement le nombre de couleurs. Pour notre objectif de réduire la détérioration de la convergence du solveur introduite par la multi-coloration (1), l'augmentation du nombre de couleurs est plutôt bénéfique. En effet, elle augmente la dépendance globale pour la Factorisation LU Incomplète [22] car les couleurs sont dépendantes les unes des autres. La Figure 2.4 illustre le coloriage de graphe avec ColorRCM sur la matrice cage4. La taille des couleurs a été limité à 2, ce qui résulte en un nombre de couleur total égale à 5.

2.3.2 Mise-en-oeuvre COLORRCM

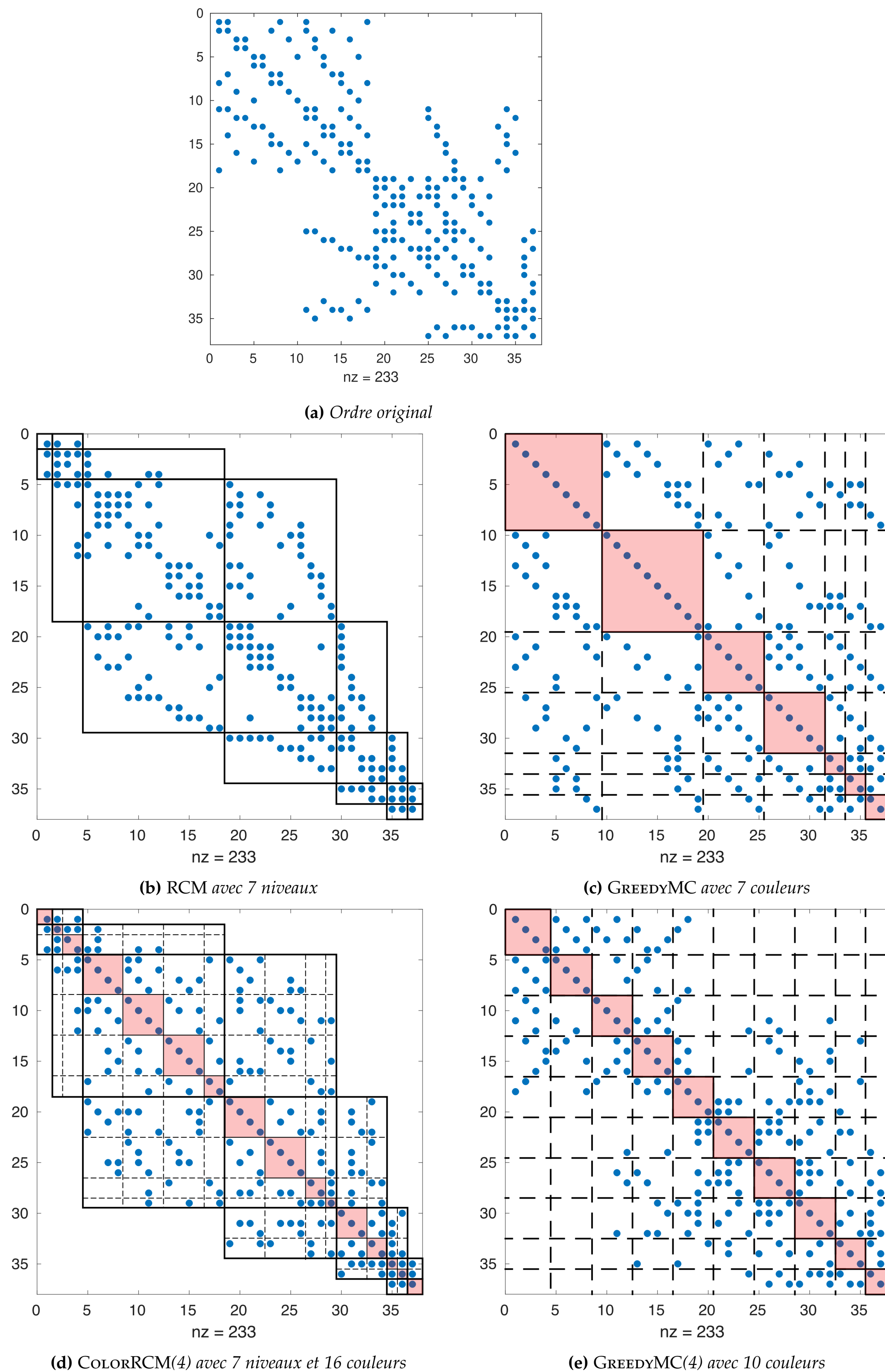
Algorithm 5 MULTI-COLORIAGE GLOUTON avec une limite de la taille des couleurs

1: **procedure** GREEDY_GRAPH_COLOR(G, π, p)
2: **Inputs :** Graphe : $G = \{\mathcal{V}, \mathcal{E}\}$; Vecteur de permutation : π ; Entier : p
3: Vecteur d'entier de taille $|\mathcal{V}|$: **NumberOfVertexInColor**
4: Initialiser **NumberOfVertexInColor**$[:] = 0$
5: **for** $u \in \mathcal{V}$ dans l'ordre apporter par π **do**
6: **for** v voisin de u **do**
7: $color(u) = \min(c \geq 0 \mid c \neq color(v))$ et **NumberOfVertexInColor**$[c] <= p$
8: **end for**
9: **end for**
10: **end procedure**

Algorithm 6 MULTI-COLORIAGE suivant l'ordre définit par RCM.

1: **procedure** COLORRCM(G, p)
2: **Inputs :** Graphe : $G = \{\mathcal{V}, \mathcal{E}\}$; Entier : p
3: Calculer du sommet initial u ▷ Prendre le sommet de degré minimal
4: RCM (G, u) ▷ Stocker la nouvelle numérotation calculer par RCM dans G.perm
5: GREEDY_GRAPH_COLOR($G, G.perm, p$)
6: **end procedure**

En général, le sommet de départ de RCM est choisi comme le sommet pseudo-périphérique. Cela a été montré dans [35] que le sommet pseudo-périphérique augmente l'efficacité de RCM en matière de minimisation de la largeur de bande. Cependant, ce n'est pas cette propriété qui nous intéresse ici. Nous avons remarqué lors de nos tests que l'utilisation du sommet de degré minimal comme sommet de départ pour RCM donne sensiblement les mêmes résultats que le sommet pseudo-périphérique. Cette information est acquise lors de la construction du graphe et nous comptabilisons l'étape 3 de l'Algorithme 6 comme une opération d'écriture. La complexité algorithmique de COLORRCM est dominée par la complexité de RCM, c'est-à-dire en $O(M^2)$ dans le pire des cas, mais il y a une possibilité d'amélioration en $O(M \log_2(M) + 2N)$.



(a) *Ordre original*

(b) RCM *avec 7 niveaux*

(c) GreedyMC *avec 7 couleurs*

(d) ColorRCM*(4) avec 7 niveaux et 16 couleurs*

(e) GreedyMC*(4) avec 10 couleurs*

Figure 2.5 – *Représentation du profil creux de la matrice cage5. Les points bleus sont les éléments non nuls. Les lignes foncées continues définissent les niveaux de* RCM*, les blocs rouges illustrent les classes de couleur et les lignes pointillées définissent les blocs de couleur.*

2.4 Expérimentations numériques

Dans cette section, nous allons étudier la convergence numérique du solveur BiCGStab préconditionné avec ILU(0). Le solveur et les différentes fonctions présentées précédemment ont été développés en C++ et compilés avec GCC 11.1. Les matrices utilisées pour les tests sont des matrices structurellement symétriques issues de différentes applications. Une partie de notre jeu de matrices vient de la collection *Suite Sparse Matrix* au format Matrix Market [21]. La seconde partie est composée de matrices issues des simulateurs en milieux poreux de l'IFPEN.
Les tests ont été effectué avec un critère d'arrêt à 10^{-5} et un nombre maximal d'itération à 1000. Parmi les tests effectués, un certain nombre ont échoués soit parce qu'ils ont atteint le nombre maximal d'itérations fixé, soit pour cause de *breakdown* du solveur. Cela signifie que certaine quantités de l'Algorithme 1 prennent des valeurs infiniment grandes ou atteignent le zéro machine. Nous avons fourni en Annexe A.1 des tableaux détaillant les caractéristiques de chaque matrice.
La sensibilité de la qualité du préconditionneur par factorisation incomplète par rapport à la numérotation des équations du système est un sujet connu et documenté depuis les années 80. Certaines renumérotations sont connues pour avoir généralement un impact positif alors que d'autres ont souvent démontré une dégradation de la convergence du solveur de Krylov préconditionné. On peut énumérer deux avantages recherchés lors de l'utilisation des méthodes de renumérotation en pré-traitement d'une méthode de Krylov.

1. Introduire du parallélisme dans la construction et l'application du préconditionneur.
2. Améliorer l'efficacité du préconditionneur.

De plus, toutes les permutations seront symétriques pour préserver la structure symétrique des matrices. Le premier point a été discuté dans la Section 3.4. Le deuxième point sera l'objet de cette section. L'efficacité du préconditionneur ILU(0) sera analysée indirectement à travers le nombre d'itérations du solveur. Bien qu'il ait été montré dans l'article [18] que la mauvaise convergence du solveur de Krylov préconditionné avec une factorisation incomplète pouvait être due à deux facteurs : le mauvais conditionnement de la matrice et les petits pivots. La combinaison de ces deux phénomènes peut entraîner une mauvaise factorisation incomplète (trop imprécise) ou l'instabilité de la résolution des systèmes triangulaires. Les auteurs de [18] ont également montré que la mesure de la norme $\|\mathbf{I} - \mathbf{A}(\tilde{\mathbf{L}}\tilde{\mathbf{U}})^{-1}\|$ pouvait être un bon indicateur de la stabilité des facteurs dans le cas de matrices non symétriques définies positives. Cependant, ce calcul est très coûteux en temps, c'est pourquoi nous nous limiterons uniquement à la comparaison des itérations entre les différentes méthodes. Nous avons testé 11 configurations différentes sur 232 matrices.

— NO-ILU(0) : le solveur préconditionné sans méthode de renumérotation en pré-traitement.
— RCM, ND, MD, GreedyMC, ColorRCM(p), ColorMD(p), GreedyMC(p) : le solveur préconditionné avec respectivement Reverse Cuthill-McKee, Nested Dissection, Minimum Degree, Multi-Coloriage, ColorRCM, ColorMD et GreedyMC avec une taille de couleur de p.

La Table 2.1 renseigne sur le nombre de tests qui ont été effectués et la part des tests qui ont convergé ou non. Parmi les tests qui n'ont pas convergé, une partie concerne les tests qui ont atteint le nombre maximum d'itérations fixé à 1000 (549 tests) et l'autre partie est due à une divergence du solveur (164 tests).

Les matrices appartenant aux groupes "Symétrique" et "Str. Symétrique" proviennent de la collection *Suite Sparse Matrix*. Ces matrices proviennent de diverses applications. Le groupe SPE10

Matrix Type	Nb Test Converge	Nb Test Non-Converge (Max iter)	Nb Test Non-Converge (*Breakdown*)	**TOTAL**
Symétrique	184	333	143	**660**
Str. Symétrique	341	89	21	**451**
SPE10	1314	127	0	**1441**
TOTAL	**1839** (72%)	**549** (22%)	**164** (6%)	**2552**

Table 2.1 – *Comptabilisation de tous les tests effectués avec* ILU(0)-BiCGStab *toutes configurations confondues. Les résultats sont classés par type de matrice avec la part de test qui ont convergé et le nombre de ceux ayant échoués. Ce dernier cas est divisé en deux groupes, ceux ayant atteint le nombre maximum d'itérations (1000) et ceux ayant causé un arrêt du solveur par breakdown. Les totaux de chaque groupe sont représentés sur la dernière ligne avec la part en pourcentage qu'il représentent par rapport au 2552 tests effectués.*

contient des matrices issues de simulateurs de l'IFPEN sur le benchmark SPE10 [73]. Lors de la simulation du benchmark SPE10, à chaque pas de temps du simulateur, la matrice issue de l'itération de Newton est stockée. Plus la simulation avance dans le temps et plus le système est difficile à résoudre. Le groupe de SPE10 est un cas très intéressant, car toutes les matrices ont le même graphe et seul le conditionnement de la matrice change.

2.4.1 Impact de la renumérotation

	ILU(0)	*RCM*	*MD*	*ND*	*GreedyMC*	*ColorRCM(4)*	*ColorMD(4)*	*GreedyMC(4)*	*ColorRCM(8)*	*ColorMD(8)*	*GreedyMC(8)*	***TOTAL***
Max iter	36	35	41	99	107	38	42	34	39	39	39	**549**
Breakdown	16	14	10	16	20	13	10	21	14	13	17	**164**
TOTAL	**52 (7.3%)**	**49 (6.9%)**	**51 (7.2%)**	**115 (16.1%)**	**127 (17.8%)**	**51 (7.2%)**	**52 (7.3%)**	**55 (7.7%)**	**53 (7.4%)**	**52 (7.3%)**	**56 (7.7%)**	**713**

Table 2.2 – *Nombre d'échecs de* ILU(0)-BiCGStab *pour chaque renumérotation sur tout le jeu de matrice. Le total de 173 tests représente l'ensemble des test échoués pour cause de breakdown du solveur ou d'atteinte du nombre maximal d'itération fixé à 1000. La part (en pourcentage) de chaque renumérotation par rapport au 713 échecs est notée entre parenthèse sur la ligne des totaux.*

La Table 2.2 recense pour chaque renumérotation, la part de non-convergence due à une divergence du solveur ou à une convergence trop lente atteignant le nombre maximal d'itérations fixé à 1000. On remarque que ND et GreedyMC comptabilisent un nombre de tests échoués deux fois plus élevé que les autres renumérotations. Nous pouvons voir ici le comportement instable que donnent les méthodes massivement parallèles sur la convergence de ILU(0)-BiCGStab. Cependant, l'approche qui limite la taille des couleurs dans les méthodes de coloriage a un impact significativement positif sur l'efficacité du préconditionneur. Ce tableau nous permet d'avoir une idée générale du comportement de ces renumérotations.

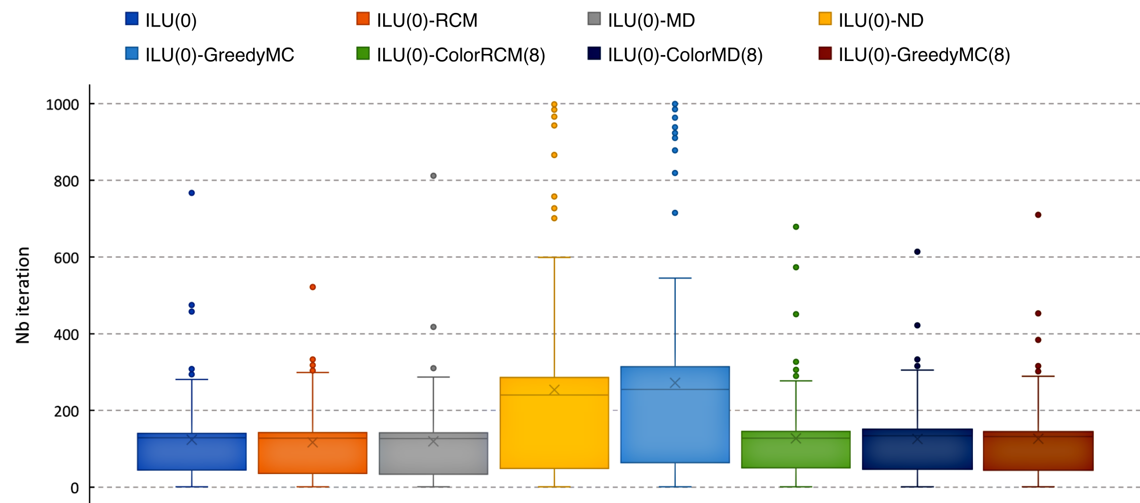


Figure 2.6 – *Boîte à moustache des nombres d'itérations des différentes renumérotations. La croix symbolise la moyenne et la ligne horizontale représente la médiane des itérations.*

La Figure 2.6 synthétise l'ensemble des résultats des tests au travers de boîtes à moustaches. La boîte encadre les itérations se situant entre le premier et le troisième quartile, ce qui signifie que la boîte représente 50 % des valeurs des itérations. La ligne horizontale qui sépare la boîte en deux représente la médiane, et la petite croix est la moyenne des itérations. Les traits limitant les moustaches de la boîte correspondent à 1,5 fois le premier ou le troisième quartile. Les points hors des moustaches sont considérés comme des valeurs aberrantes, car ils sont supérieurs à 1,5 fois le troisième quartile. Dans la Figure 2.6 sont représentées uniquement les itérations des tests qui ont convergé, soit 72 % des tests. Cette figure nous montre que le pré-traitement avec le Multi-Coloriage classique et Nested Dissection dégrade fortement la convergence du solveur préconditionné ILU(0)-BiCGStab par rapport à la configuration du solveur sans pré-traitement. Elle confirme aussi l'hypothèse selon laquelle la limitation de la taille des couleurs améliore la convergence du solveur préconditionné. La Figure 2.6 ne nous permet pas de nous rendre compte précisément du comportement du solveur avec chaque méthode de renumérotation. Pour la suite, nous allons utiliser l'outil *performance profil* pour obtenir un classement des différentes configurations du solveur préconditionné.

Performances numériques du solveur

Pour avoir une comparaison des différentes méthodes, nous avons la méthode *performance profil* décrite dans [23]. L'idée est la suivante : À partir d'une base de données contenant les résultats des tests pour chaque algorithme, le programme va calculer un ratio de performance relatif à chaque algorithme. Pour chaque test, le ratio de performance d'un algorithme correspond au rapport entre le résultat minimal et le résultat de l'algorithme. Dans notre cas, nous avons un jeu de matrices tests sur lesquelles nous allons tester différentes configurations du solveur BiCGStab préconditionné. Chaque configuration correspond à un algorithme de renumérotation. Dans cette partie, nous comparons les itérations du solveur pour les différentes configurations du solveur. L'outil *performance profil* va tracer les courbes des ratios de performances en fonction de la

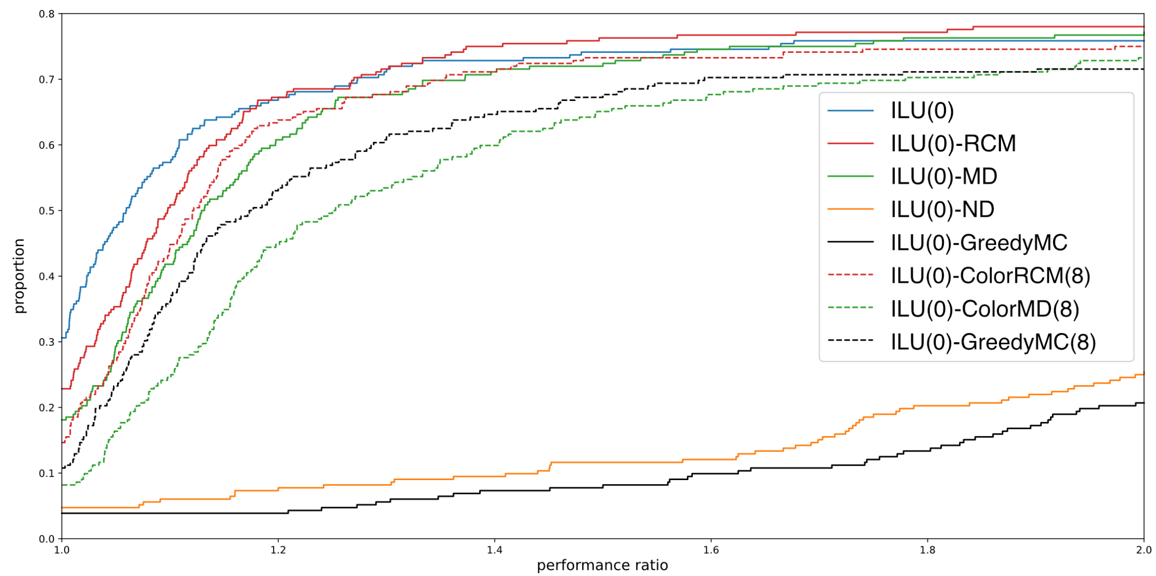


Figure 2.7 – *Représentations des performance profils des différentes renumérotations par rapport à leur nombre d'itérations. Le nombre de matrices total est de 232 matrices avec 11 configurations soit 2552 tests. Les courbes en pointillés sont les coloriages avec limitation de la taille des couleurs* (ColorRCM*(8)*, ColorMD*(8) et* GreedyMC*(8))*.

proportion de tests qui ont atteint ce ratio. Pour un test donné, une configuration avec un ratio de performance de 1 indique que cette configuration a fourni la plus petite itération par rapport aux autres configurations. La proportion nous indique le pourcentage de tests pour lesquels un ratio de performance donné a été atteint. Par exemple, pour la configuration ILU(0) du solveur (configuration sans renumérotation), le point (1.2 ; 0.67) de la Figure 2.7 indique que dans 67 % des tests, avec cette configuration, les itérations ont été au plus multipliées par 1.2 par rapport au minimum des itérations atteint. Une remarque, nous avons affiché des ratios de performances allant jusqu'à 2 et la proportion de tests jusqu'à 80 %. Nous avons limité le ratio de performance à 2 parce qu'au-delà, les courbes croissent faiblement. De plus, comme les tests qui n'ont pas convergé (environ 27 %) seront ignorés par *performance profil*, c'est la raison pour laquelle l'axe des ordonnées a été limité à 80 %.

Dans la Figure 2.7, nous avons comparé certaines renumérotations de la bibliographie, à savoir Reverse Cuthill-McKee, Nested Dissection, Minimum Degree, Multi-Coloriage et la version du solveur ILU(0)-BiCGStab sans renumérotation sur l'ensemble des tests (232 matrices). La première observation est que la version du solveur sans renumérotation semble meilleure que les autres renumérotations sur 65% des tests. Néanmoins, au-delà d'un certain seuil de dégradation de la convergence, par exemple 1,4 fois par rapport au min, RCM semble meilleur sur plus de 75% des tests. Comme nous l'avons observé dans la Figure 2.6, GreedyMC et ND sont les moins performants du groupe. Nous avons aussi comparé les renumérotations issues de la littérature avec les renumérotations de coloriage de graphe mises en oeuvre durant nos travaux dans la Figure 2.7 : GreedyMC(8), MD(8), ColorRCM(8). La première remarque concerne GreedyMC(8) qui est significativement plus performant que Multi-Coloriage classique. La courbe de GreedyMC(8) est au-dessus de celle de MD(8) sur la grande majorité des tests. Donc, le fait de limiter la taille des couleurs a un impact significativement positif sur les performances du préconditionneur.

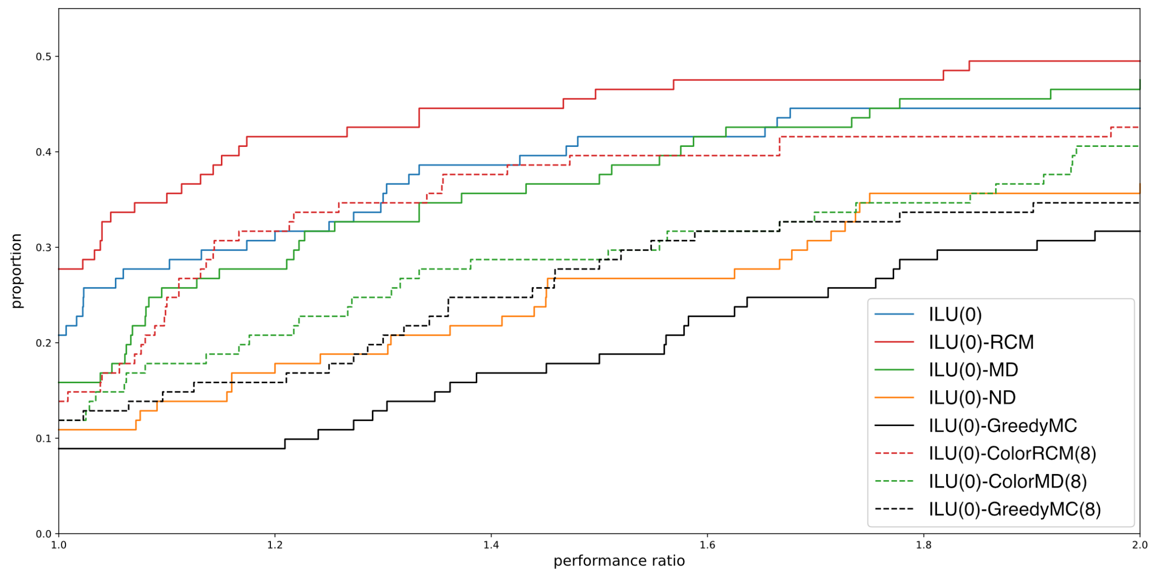


Figure 2.8 – *Représentations des performance profils des matrices issue de la collection Suite Sparse Matrix pour les différentes renumérotations par rapport à leur nombre d'itérations. Le nombre de matrices total est de 101 matrices avec 11 configurations soit 1117 tests. Les courbes en pointillés sont les coloriages avec limitation de la taille des couleurs (*COLORRCM*(8),* COLORMD*(8) et* GREEDYMC*(8)).*

Le coloriage piloté par MD perturbe l'efficacité du préconditionneur. Initialement, la renumérotation MD a été introduite pour minimiser le remplissage lorsqu'on accepte du remplissage dans la construction du préconditionneur. Même si dans notre cas nous n'acceptons pas de remplissage, on peut voir que la renumérotation MD aide le solveur préconditionné à converger rapidement. Cependant, MD(8) ne témoigne pas du même comportement que MD. Dans ce cas, l'approche coloriage classique avec limitation de la taille des couleurs à 8 noeuds (GREEDYMC(8)) est plus performante que MD(8).
L'approche COLORRCM, combinant une coloration du graphe pilotée par la méthode RCM, démontre de meilleures performances que toutes les autres méthodes de coloriage. COLORRCM a un comportement similaire à RCM avec une légère pénalité due au coloriage même si elle est moins forte que la pénalité de MD(8) par rapport à MD. Par exemple, pour un ratio d'itération de 1.4 fois par rapport au min, COLORRCM(8) atteint ce seuil dans 70 % des tests alors que GREEDYMC(8) l'atteint dans environ 61 % des cas et 59 % pour MD(8).
Il convient de relever que sur les 232 matrices, 56 % d'entre elles viennent du jeu de matrices SPE10 et les 44 % restants sont issus de la collection *Suite Sparse Matrix* (les matrices de type symétrique et structurellement symétrique de la Table 2.1). Et sur ces 44 %, seulement 29 % ont été prises en compte dans la représentation de *performance profil*, car les autres tests ont divergé ou ont atteint le nombre maximum d'itérations. Donc, le comportement des renumérotations observé ici est largement dominé par les résultats du groupe de matrices SPE10.

Tests des matrices *Suite Sparse Matrix* **:** La Figure 2.8 représente les performances du solveur avec les différentes renumérotations sur les 101 matrices issues de la collection *Suite Sparse Matrix*. Ces matrices proviennent de différentes applications. Dans ce cas, RCM présente les meilleures

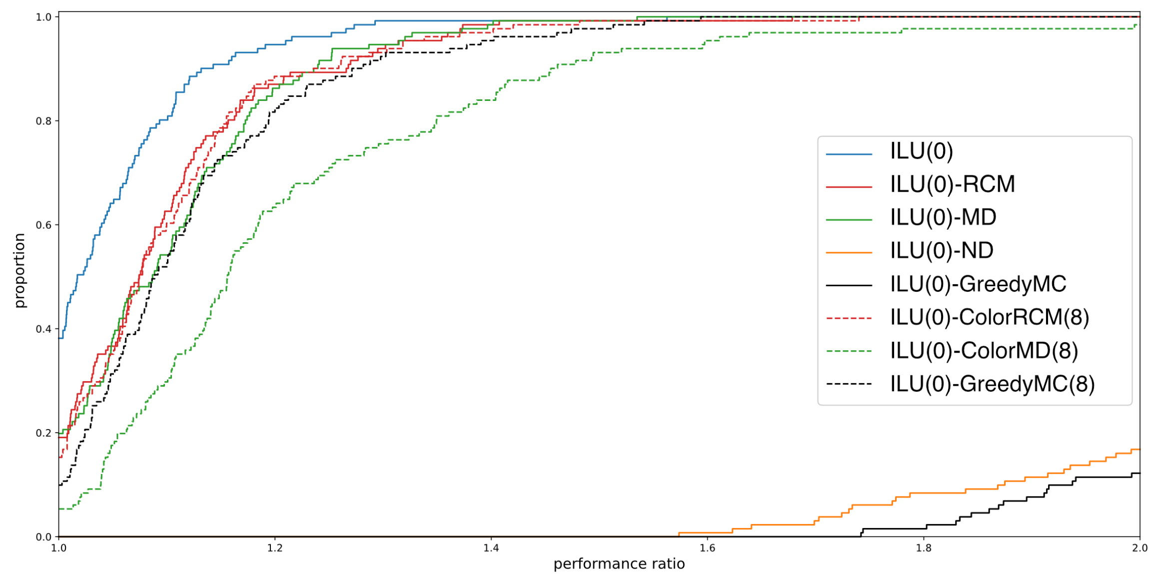


Figure 2.9 – *Représentations des performance profils des matrices SPE10 pour les différentes renumérotations par rapport à leur nombre d'itérations. Le nombre de matrices total est de 131 matrices avec 11 configurations soit 1441 tests. Les courbes en pointillés sont les coloriages avec limitation de la taille des couleurs (*ColorRCM*(8),* ColorMD*(8) et* GreedyMC*(8)).*

performances pour toutes les matrices. Et MD a des performances quasi similaires à ILU(0) (la configuration sans renumérotation). Ici aussi, ColorRCM a des performances nettement supérieures aux autres algorithmes de coloriage de graphe. Ces résultats semblent indiquer que dans le cas où l'on n'a aucune information a priori sur la matrice, RCM semble être une renumérotation efficace pour accélérer la convergence de ILU(0)-BiCGStab. De plus, si l'on cherche une parallélisation fine des calculs dans le solveur, ColorRCM est le coloriage qui dégrade le moins la convergence du solveur préconditionné.

Tests des matrices SPE10 : La Figure 2.9 représente les tests du groupe de matrices SPE10. Le nombre total de tests qui ont convergé étant majoritairement des matrices SPE10 (voir Table 2.1), on peut observer que ce sont ces tests qui ont largement influencé les représentations de la Figure 2.7. Nous pouvons remarquer que les renumérotations RCM, MD, ColorRCM(8) et GreedyMC(8) présentent quasiment les mêmes performances. Cependant, aucune d'entre elles n'arrive à faire mieux que la configuration sans renumérotation. Les matrices SPE10 ont été générées à partir de la résolution du benchmark SPE10 dans les simulateurs IFPEN. Il semble que les matrices issues du simulateur ont été normalisées et stockées dans un certain ordre préférentiel. Raison pour laquelle nos renumérotations, modifiant ainsi cet ordre initial, ne font que dégrader la convergence du solveur. La différence entre GreedyMC et GreedyMC(8) est encore plus importante dans ce groupe de matrices de test. Le fait de chercher à former des couleurs de petites tailles, le profil de la matrice permutée reste relativement proche de la matrice de départ comparativement au coloriage sans limite de taille. On peut voir un exemple de cet effet avec la Figure 2.5e par rapport à la Figure 2.5a. On suppose que dans le cas des matrices SPE10 dont le préconditionneur est très sensible à l'ordre, GreedyMC(8) a un ordre plus proche de l'ordre original que GreedyMC classique.

2.5 Discussion

Notre objectif était d'améliorer voire de limiter la dégradation de la convergence du solveur lors de l'utilisation de méthodes de renumérotation qui permettent de paralléliser la factorisation incomplète. Nous avons constaté que GreedyMC et ND, qui sont des méthodes permettant d'obtenir massivement du parallélisme dans les calculs, ont un impact négatif sur la convergence. Pour les méthodes de coloriages, nous avons limité la taille des couleurs et orienté le coloriage du graphe selon un ordre prédéfini (RCM pour ColorRCM, MD pour ColorMD et l'ordre initial de la matrice pour GreedyMC). Nous avons vu que cela améliore significativement la convergence du solveur par rapport au GreedyMC classique ou ND. Dans [18], une observation avait été faite sur l'amélioration de la convergence du solveur itératif pré-traité par une méthode de coloriage lorsqu'on augmente le nombre de couleurs. Par exemple, pour GreedyMC, la méthode cherche à créer de gros ensembles de noeuds indépendants entre eux et en minimisant le nombre de ces ensembles. Même si les groupes de couleurs sont dépendants entre eux, en cherchant à minimiser leur nombre, on minimise aussi la dépendance globale des équations du système. En revanche, en limitant la taille des couleurs, nous augmentons significativement le nombre de couleurs, surtout pour des petites tailles comme 4 ou 8. Mais cela améliore grandement la convergence du solveur, comme nous pouvons le voir sur la Figure 2.7 entre GreedyMC et GreedyMC(8). Finalement, ColorRCM(8) se positionne comme la méthode de coloriage la plus performante parmi celles testées. Elle démontre un comportement similaire à RCM et fournit un préconditionneur performant (voir le nombre de *breakdown* dans la Table 2.2). Sur les 2552 tests que nous avons réalisés, ColorRCM s'avère être un bon compromis pour apporter un parallélisme à grain fin dans les calculs tout en apportant une certaine efficacité du préconditionneur dans la convergence du solveur. Pour la suite, nous allons nous intéresser aux performances parallèles offertes par ces renuméroations lors de la résolution triangulaire des facteurs.

3

Application du préconditionneur ILU(0) : Résolution de systèmes triangulaires

Contents

3.1 Introduction 39
3.2 Formats de stockage creux 41
3.3 État de l'art pour la résolution de systèmes triangulaires creux 44
3.3.1 Level Scheduling 45
3.3.2 Coloriage de graphe 47
3.4 Parallélisation SIMD 49
3.4.1 Multiplication Matrice-Vecteur 49
3.4.2 Résolutions parallèle de systèmes triangulaires 51
3.5 Expérimentations numériques 53
3.5.1 Multiplication Matrice-Vecteur 53
3.5.2 Résolution des systèmes triangulaires 55
3.5.3 Performance du solveur de Krylov 56
3.6 Discussion 57

3.1 Introduction

Pour pouvoir simuler des phénomènes importants et complexes, nous avons besoin d'une grande puissance de calcul. Cette puissance est principalement liée au nombre d'unités de calcul qui peuvent travailler en même temps et à la fréquence à laquelle elles sont cadencées. Depuis les années 90, avec l'avènement des supercalculateurs dotés de processeurs vectoriels, les techniques avancées de vectorisation des calculs, comme celles utilisées dans l'élimination de Gauss, ont fait l'objet d'études intensives [24]. En raison des dépendances des données dans les calculs, les méthodes basées sur l'approche de l'élimination de Gauss sont difficiles à vectoriser sans avoir

une connaissance préalable du problème à résoudre. Ce problème est toujours d'actualité, car les processeurs modernes deviennent de plus en plus sophistiqués avec des tailles de registre de plus en plus grandes. La tendance de l'évolution du hardware des processeurs montre une meilleure prise en charge du parallélisme à grain fin par les jeux d'instructions vectorielles SIMD et le threading au niveau matériel. Nous sommes passés de petits vecteurs SIMD (x86 SSE2) à des vecteurs beaucoup plus grands (x86 AVX512). Une bonne stratégie pour exploiter le parallélisme à grain fin dans notre cas consiste à regrouper les équations sans relation directe entre elles et à les ordonner en premier pour obtenir des blocs d'équations indépendantes. Cette approche peut être transformée en un problème de coloration de graphe, que l'on rencontre dans de nombreuses applications de calcul scientifique. La première étape est de représenter la matrice du système en un graphe. Puis de chercher une renumérotation des nœuds du graphe qui permettra d'exploiter un certain parallélisme dans les calculs. C'est l'objectif du chapitre 2 de ce manuscrit. Dans ce chapitre, nous allons utiliser la structure mise en valeur par la méthode de coloriage de graphe pour paralléliser les calculs et mesurer les performances. Dans la table 3.1, nous avons recensé le pourcentage de temps de calcul des différentes opérations intervenant à chaque itération du solveur en séquentiel.

Matrix (n, NNZ)	Temps séquentiel/nbiter de ILU(0)-BiCGStab		
	2 SpMV	**4** SpTRSV	(4 dot + 6 AXPY)
PqDM1_N19_I1_F1 (2660, 17086)	33%	**59%**	8%
Canta_Frs200l_N32_I1_F1 (8016, 65202)	36%	**55%**	9%
IvaskBO_N1_I1_F1 (49572, 522498)	33%	**59%**	8%
GCS_400p_N1_I1_F2 (185498, 1279882)	30%	**59%**	11%
GCS2K_N1_I1_F1 (370982, 2928696)	31%	**60%**	9%

Table 3.1 – *Pourcentage des temps moyens par itération des différentes opérations dans* BiCGStab *préconditionné en séquentiel. Ces matrices sont structurellement symétriques et sont issues d'un jeu de test IFPEN. Elles sont rangées dans l'ordre croissant par rapport à leur nombre de lignes.*

Le coût moyen de la résolution de systèmes triangulaires par itération représente la majorité du temps de calcul par itération. La parallélisation de la résolution des systèmes triangulaires creux est essentielle pour améliorer les performances du solveur. À chaque itération du solveur, nous avons quatre résolutions de systèmes triangulaires de la forme suivante :

$$\mathbf{LUw} = \mathbf{f} \tag{3.1}$$

Où $(\mathbf{L}, \mathbf{U}) \in \mathbb{R}^{n,n}$ sont des matrices creuses triangulaire inférieure à diagonale unité et triangulaire supérieure, $(\mathbf{f}, \mathbf{w}) \in \mathbb{R}^n$ sont les vecteurs denses du second membre et du résultat. Pour résoudre (3.1), nous allons effectuer une Descente et une Remontée, telles que :

$$\begin{cases} \mathbf{Ly} = \mathbf{f} & (3.2a) \\ \mathbf{Uw} = \mathbf{y} & (3.2b) \end{cases}$$

Alors,

$$\begin{cases} \forall i \in \{0, ..., n-1\}, \quad y_i = \left(f_i - \sum_{j=0}^{i-1} l_{i,j} y_j \right) & (3.3a) \\ \forall i \in \{n-1, ..., 0\}, \quad w_i = \frac{1}{u_{i,i}} \left(y_i - \sum_{j=i+1}^{n-1} u_{i,j} w_j \right) & (3.3b) \end{cases}$$

Avec la convention que $\sum_{j=0}^{i-1} l_{i,j} y_j = 0$, si $i - 1 \leq 0$ et $\sum_{j=i+1}^{n-1} u_{i,j} w_j = 0$, si $n - 1 \leq i + 1$.
Les opérations de SpTRSV dans le solveur BiCGStab sont dues à l'application du préconditionneur ILU(0). Cela implique que les facteurs triangulaires **L**, **U** bénéficient de la même structure que la partie triangulaire inférieure et la partie triangulaire supérieure de la matrice **A**. L'objectif de ce chapitre est d'accélérer la résolution de ces systèmes triangulaires avec une parallélisation SIMD des calculs. Pour ce faire, nous allons exploiter la structure particulière de la matrice permutée par ColorRCM. Pour cela, le type de format de stockage utilisé pour les matrices **L**, **U** est fondamental. Nous allons passer en revue quelques formats de stockage de l'état de l'art.

3.2 Formats de stockage creux

En prenant en compte le profil creux de **A**, nous pouvons diminuer le stockage de **A** ainsi que le nombre d'opérations (complexité) des méthodes de résolution. Il existe différentes configurations de stockage qui visent à réduire l'empreinte mémoire de la matrice creuse en ne stockant que les éléments non nuls de la matrice.

le format COO (Coordinate List) :

Le format COO va stocker l'ensemble des informations caractérisant les éléments de la matrice dans trois tableaux. Deux tableaux permettent de stocker les indices de ligne et colonne des éléments non nuls de la matrice. Le troisième tableau contient les valeurs de ces éléments non nuls. Les trois tableaux ont la même taille (**nnz**). Avec ce format, l'ajout d'éléments se fait facilement, car il n'y a pas d'ordre particulier de stockage des éléments.

le format CSR (*Compressed Sparse Row*) :

Avec ce format, les éléments de la matrice seront stockés par ligne à l'aide de trois tableaux (ptr, col, val). Les valeurs non nulles seront stockées dans *val* en parcourant la matrice **A** ligne par ligne. Le tableau *col* contient les indices des colonnes de **A** tels que si $val[k] = a_{i,j}$ alors $col[k] = j$. Le vecteur *ptr* est un tableau qui contient les indices des premiers éléments non nuls des lignes de **A** tels que si $val[k] = a_{i,j}$ alors $ptr[i] \leq k < ptr[i+1]$. La taille des tableaux *val* et *col* est de **nnz** et $\mathbf{nnz} + 1$ est égale à la taille de *ptr* avec la convention que $ptr[\mathbf{nnz} + 1] = \mathbf{nnz}$. Il existe d'autres variantes du CSR telles que le CSC (Compressed Sparse Column) qui parcourt la matrice colonne par colonne pour stocker les valeurs non nulles de **A** dans le tableau *val* et les indices des lignes

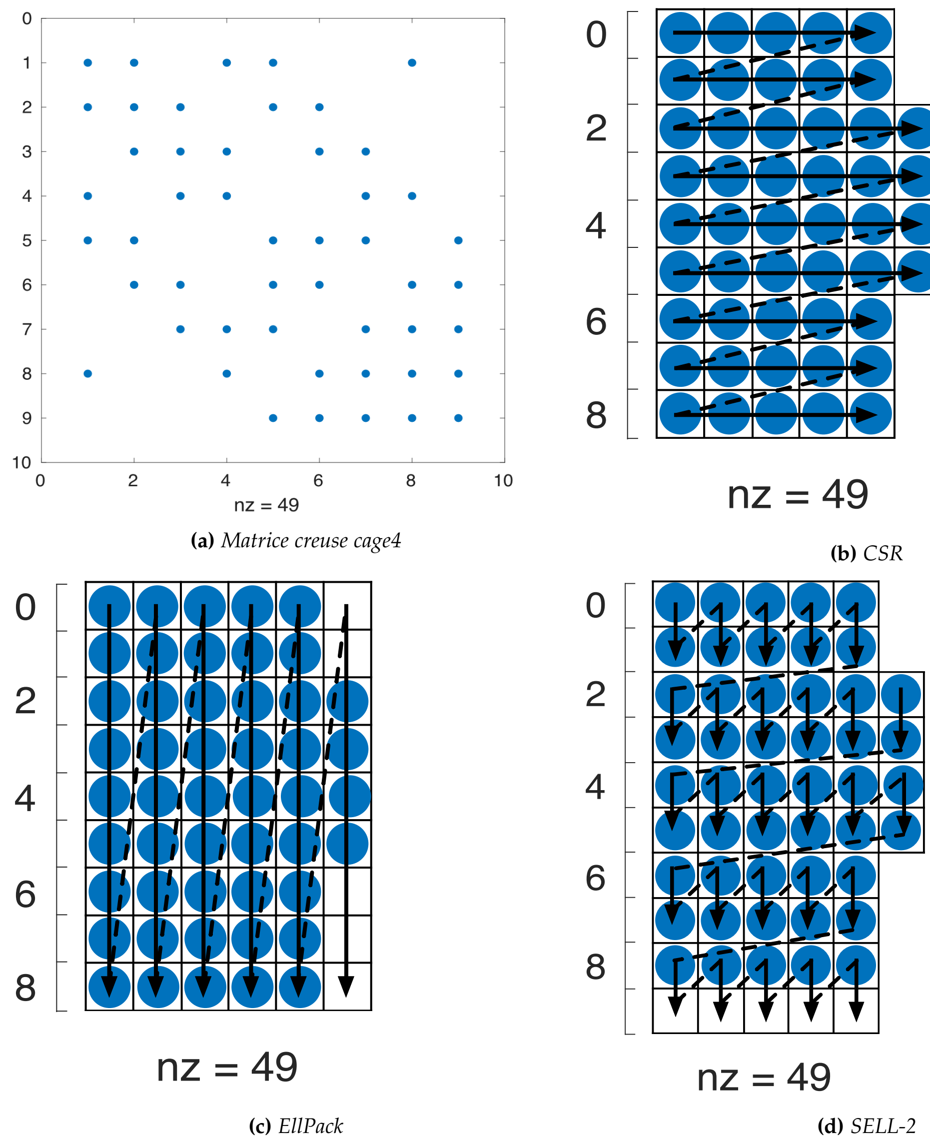
dans *col*. Le Block CSR ou BCSR (Figure 3.1b) stocke la **A** sous forme de blocs denses dans *val*. Les dimensions de *val* sont ainsi de $\mathbf{ssb} \times (\mathbf{nbb} * \mathbf{ssb})$ où **ssb** est la taille des blocs carrés et **nbb** est le nombre de blocs. Le format BCSR est réputé pour être efficace pour des opérations de type SpMV sur CPU. Mais pour le passage à des machines hautement parallèles comme les GPU, [4] souligne le manque d'accès coalescents à la mémoire.

le format EllPack :

Ce format allège un peu les contraintes sur le type d'éléments stockés en acceptant des zéros dans le tableau de valeurs *val* de manière à ce que toutes les lignes aient une taille uniforme. La largeur des lignes correspond à la taille de la plus grande ligne sans le rembourrage. Maintenant, que toutes les lignes font la même taille, il n'est plus nécessaire d'avoir un tableau de pointeur des premiers éléments de chaque ligne. Ainsi, le format EllPack stocke les valeurs de la matrice creuse à l'aide d'un tableau de valeurs *val* et d'un tableau d'indices des colonnes *col*. Contrairement au format CSR ou BCSR, EllPack permet des accès mémoire coalescent.Dans ce contexte, le surcoût mémoire causé par le rembourrage des lignes par des zéros est largement compensé par la rapidité des calculs sur GPU. Cependant, pour une parallélisation sur CPU, il est souhaitable de chercher à minimiser ce rembourrage.

le format SELL-C-σ :

Le format SELL-P reprend l'idée du format EllPack en cherchant à réduire le surcoût mémoire du rembourrage de zéros. Pour cela, la matrice va être découpée en paquet de ligne et chaque tranche de la matrice sera stockée au format EllPack. Ainsi, le rembourrage des lignes sera en fonction de la plus grande ligne du paquet. Originellement, ce format fut introduit dans [52], notée SELL-C-σ. SELL-C-σ indique que la matrice sera stockée par paquet de C lignes et que les σ lignes consécutive seront rangées par ordre décroissant de leur taille. Cette approche réduit efficacement le rembourrage nécessaire des lignes et les auteurs ont montré un gain significatif par rapport au format CSR pour une opération SpMV sur CPU et GPU. Dans [4], les auteurs ont proposé une légère modification du format SELL-P pour que les tailles des paquets soient divisibles par le nombre de threads affecté à la ligne. Puis ils ont utilisé le format pour le calcul d'opérations SpMV en multithread. Les résultats montrent la supériorité de ce format pour des matrices non structurée par rapport aux formats CSR ou EllPack. Les auteurs ont précisé que leur format SELL-P n'utilise pas de tri préalable des lignes comme pour SELL-C-σ. Ce qui correspond à une configuration SELL-C-σ avec $\sigma = 1$. Pour la suite de notre travail, le format SELL-P sera utilisé pour une vectorisation SIMD du solver BiCGStab préconditionné. Ce format offre un bon compromis entre CSR et EllPack. En effet, un stockage SELL-P P=1 correspond exactement à un format CSR alors qu'avec P=n cela correspond à un format EllPack. Dans [66], l'auteur a montré l'intérêt d'utiliser ce format pour des processeurs embarquant beaucoup de coeurs de calculs comme le KNL.



(a) *Matrice creuse cage4*

(b) *CSR*

(c) *EllPack*

(d) *SELL-2*

Figure 3.1 – *Des exemples de représentation des formats de stockage CSR (3.1b), EllPack (3.1c) et SELL-2 (3.1d) de la matrices creuses cage4.*

Stockage creux SELL-P **avec la renumérotation** ColorRCM

Le découpage du format SELL-P correspondra à un nombre de lignes égal à un paquet SIMD. Pour que les blocs de couleur de la matrice permutée par ColorRCM soient correctement stockés avec le format SELL-P, il faut que la taille maximale des couleurs soit divisible par le nombre de paquets de lignes. Cependant, il n'y a aucune garantie que les couleurs soient exactement d'un nombre de sommets égal à la taille maximale des couleurs fixée par l'utilisateur. Pour corriger cela, nous allons augmenter artificiellement les couleurs qui ne sont pas remplies au maximum. Cela correspond à l'ajout de 1 sur la diagonale de la matrice au niveau des blocs de couleur

correspondants. Ainsi, le nombre total de lignes de la matrice permutée et étendue sera divisible par la taille maximale de couleur fixée par l'utilisateur. La Figure 3.2 illustre le profil obtenu après permutation à l'aide de ColorRCM(4) et remplissage des couleurs jusqu'au maximum par l'ajout d'éléments diagonaux. Ce rembourrage engendre un faible surcoût de stockage et dépendra du taux de remplissage des couleurs à leur maximum. Le Tableau 3.2 renseigne sur la structure des matrices tests issues du jeu de matrices de l'IFPEN et le nombre de couleurs formées avec ColorRCM(8). Le remplissage des couleurs au maximum dépendra de la connectivité des sous-graphes et du nombre de noeuds par niveaux de RCM. Cependant, l'ajout d'éléments pour compléter les couleurs est très minime par rapport au nombre d'éléments totaux de la matrices. Par exemple, pour la matrice `PqDM1_N19_I1_F1` qui a un taux de couleurs remplies avec 8 noeuds égal à 87 %, cela représente une augmentation d'environ 0.97 %. Pour la matrice `GCS2K_N1_I1_F1` le rembourrage de la matrice est égale à 0.04 % d'augmentation.

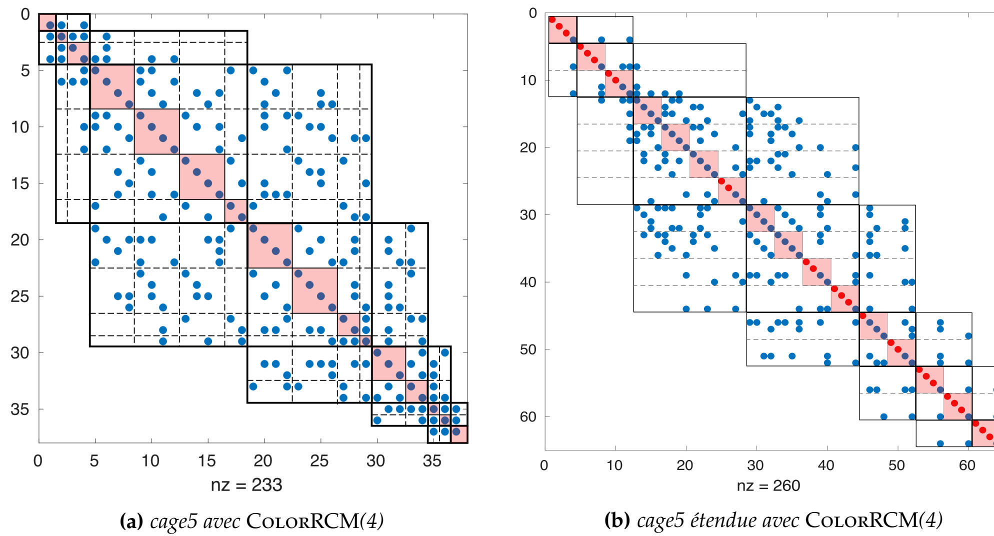
Name_Matrix	Nrows	NNZ	NNZ par ligne (min, avg, max)	NColors (taille max 8)	Nelem ajouté
poisson3Da	13 514	352 762	(6, 26, 110)	1 742 (94%)	422
bcsstk39	46 772	2 089 294	(12, 44, 63)	6 367 (72%)	4 164
CurlCurl_1	226 451	2 472 071	(4, 10, 13)	28 448 (99%)	1 133
ecology1	1 000 000	4 996 000	(3, 4, 5)	125 875 (98%)	7 000
cage15	5 154 859	99 199 551	(3, 19, 47)	64 4436 (99%)	629
PqDM1_N19_I1_F1	2 660	17 086	(4, 6, 7)	354 (87%)	165
Canta_Frs200l_N32_I1_F1	8 016	65 202	(2, 8, 33)	1 050 (91%)	379
IvaskBO_N1_I1_F1	49 572	522 498	(5, 10, 13)	6 306 (97%)	869
GCS_400p_N1_I1_F2	18 5498	1 279 882	(3, 6, 26)	23 441 (98%)	2 023
GCS2K_N1_I1_F1	370 982	2 928 696	(4, 7, 27)	46 513 (99%)	1 115

Table 3.2 – *Informations sur la structures creux des matrices et le pourcentage du nombre de couleurs de taille maximum (i.e. les couleurs ayants un nombre de nœuds égale à 8). Ces matrices sont structurellement symétriques et sont issues d'un jeu de test IFPEN. La ligne pointillé sépare les matrices issue de la collection Suite Sparse Matrix. Dans chaque partie, les matrices sont rangées dans l'ordre croissant par rapport à leur nombre de lignes.*

3.3 État de l'art pour la résolution de systèmes triangulaires creux

La résolution de systèmes triangulaires (SpTRSV) dans le solveur ILU(0)-BiCGStab est très coûteuse (voir Table 3.1) et peut être un goulot d'étranglement pour les performances parallèles du solveur s'il n'est pas parallélisé efficacement. Or, le calcul de la solution par une Descente (3.3b) et/ou une Remontée (3.3b) sont des algorithmes intrinsèquement séquentiels. En effet, on peut voir que le calcul d'un y_i dépend de y_j, $0 \leq j \leq i$ *i.e.* il dépend des solutions précédentes. Supposons que les matrices triangulaires soient stockées au format CSR. Par exemple, la résolution de $\mathbf{Ly} = \mathbf{f}$ peut être décrite par l'algorithme suivant :

La boucle interne sur j de l'Algorithme 7 permet, pour une inconnue y_i, de balayer les éléments $l_{i,j}$ et les inconnues y_j qui ont précédé y_i. Une première approche de parallélisation de cet algorithme serait de scinder cette boucle interne en plusieurs blocs pour que chaque bloc soit calculé en



(a) *cage5 avec* ColorRCM*(4)* **(b)** *cage5 étendue avec* ColorRCM*(4)*

Figure 3.2 – *Représentation du profile de la matrice cage5 permutée par* ColorRCM*(4) dans la Figure 3.2a et le profile de la matrice étendue à partir de* ColorRCM*(4) dans la Figure 3.2b. Les billes bleues symbolisent les éléments non nuls et les billes rouges désignent les éléments non nul ajoutées.*

Algorithm 7 Descente pour SpTRSV

Inputs : Matrice creuse triangulaire inférieure à diagonale unité $\mathbf{L} \in \mathbb{R}^{n,n}$ et vecteur second membre $\mathbf{f} \in \mathbb{R}^n$.

Outputs : Vecteur solution $\mathbf{y} \in \mathbb{R}^n$.

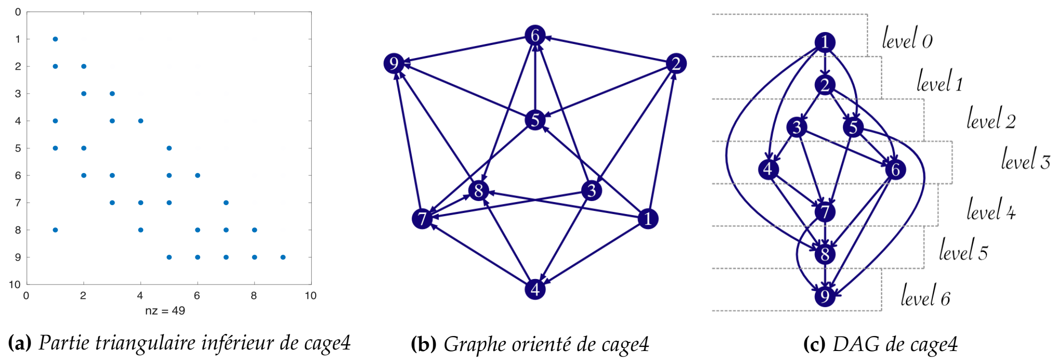
1: **for** $(i = 0\,;\, i < n\,;\, i = i + 1)$ **do**
2:     **for** $(j = IA(i)\,;\, j < IA(i+1) - 1\,;\, j = j + 1)$ **do**
3:         $y(i) = f(i) - VA(j) \times y(JA(j))$
4:     **end for** j
5: **end for** i

---

parallèle. Une fois ces calculs effectués, il faudra réduire les résultats en une valeur scalaire qui sera la solution finale de y_i. Cette approche de parallélisation multithread ne sera pas efficace pour le type de matrice que nous manipulons, à savoir des matrices très creuses. La Table 3.2 indique qu'en moyenne, nous avons entre 7 et 27 éléments par ligne de la matrices, ce qui signifie que la partie inférieure stricte de nos matrices a entre 3 et 13 éléments par ligne. Ce qui est trop peu pour une approche de parallélisation multithread par colonnes. Sans compter le coût de synchronisation des threads à chaque étape, qui sera extrêmement élevé. Une autre approche serait de paralléliser les calculs au niveau de la boucle externe sur i. Le peu d'éléments par ligne peut cacher un grand nombre d'inconnues indépendantes. L'ensemble de ces inconnues pourra être calculé en parallèle. Différentes méthodes existent pour dévoiler du parallélisme à grain fin lors de la résolution de systèmes triangulaires. Nous allons discuter de deux d'entre elles.

3.3.1 Level Scheduling

La méthode du Level Scheduling (LS) consiste à analyser la structure creuse de la matrice et les dépendances des inconnues pour classer les inconnues par niveau. Les niveaux correspondent



(a) *Partie triangulaire inférieur de cage4* **(b)** *Graphe orienté de cage4* **(c)** *DAG de cage4*

Figure 3.3 – *(3.3a) représente le profile de la partie triangulaire inférieur de la matrice cage4 et (3.3b) est une représentation du graphe non-orienté de (3.3a). La Figure 3.3c est une représentation du graphe des dépendances de tâches pour la résolution du système triangulaires.*

aux ensembles des inconnues qui pourront être calculées en parallèle. On entend par dépendance d'une inconnue, le fait que pour calculer une inconnue y_i, on aura besoin de la valeur de y_j précédente. On dit que y_i dépend de y_j. Les dépendances des inconnues peuvent être identifiées en analysant le graphe de dépendance de la matrice **L** avec le *Direct Acyclic Graph* (DAG). Le DAG, comme son nom l'indique, est un graphe orienté qui indique une relation de dépendance entre un nœud v et un nœud u par une arête orientée de v à u (voir la Figure 3.3). L'idée est donc de résoudre le système niveau par niveau. Le niveau de l'inconnue y_i sera déterminé par $lev(i) = 1 + max(lev(j))$, $\forall j$ tel que $l_{i,j} \neq 0$. En pratique, le calcul des niveaux correspond à un parcours en largeur du graphe. Donc on peut utiliser une version modifiée de BFS (une description de l'algorithme BFS est faite dans le chapitre 2). La Figure 3.3c montre une représentation du DAG de la résolution du système triangulaire avec la partie triangulaire inférieure de la matrice cage4. Par exemple, au niveau 3, on pourra calculer les inconnues y_4 et y_6 en simultané après avoir calculé les inconnues y_3, y_5 du niveau 2 et y_2 du niveau 1 et y_1 du niveau 0. Alors l'Algorithme 7 de Descente peut être réécrit comme suit :

Algorithm 8 Descente avec Level Scheduling pour SpTRSV

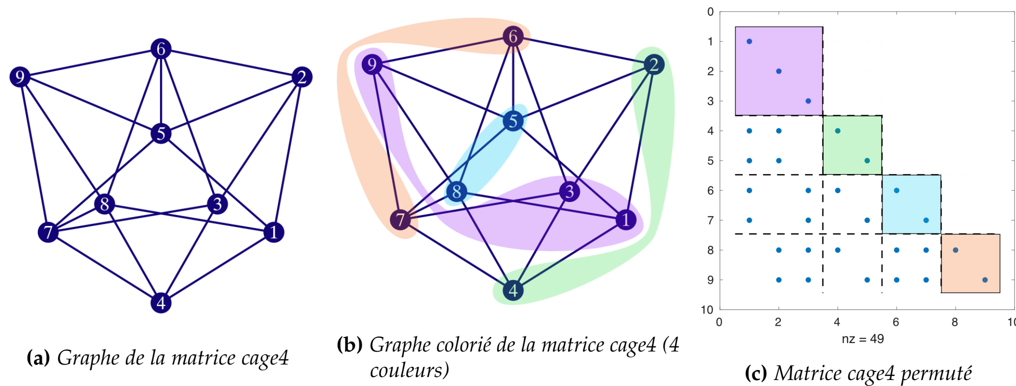
Inputs : Matrice creuse triangulaire inférieure à diagonale unité $\mathbf{L} \in \mathbb{R}^{n,n}$, et Vecteur second membre $\mathbf{f} \in \mathbb{R}^n$.
Inputs : Vecteur **jlev** $\in \mathbb{R}^{n_{lev}}$ contenant les nœuds par ordre croissant de leur niveau.
Inputs : Vecteur **ilev** $\in \mathbb{R}^{n_{lev}}$ contenant les pointeur de debut chaque niveau **jlev**.
Outputs : Vecteur solution $\mathbf{y} \in \mathbb{R}^n$.

1: **for** $(l = 0\,;\, l < n_{lev}\,;\, l = l + 1)$ **do**
2:     **for** $(k = ilev(l)\,;\, k < ilev(l+1) - 1\,;\, k = k + 1)$ **do**
3:         $i = jlev(k)$
4:         **for** $(j = IA(i)\,;\, j < IA(i+1) - 1\,;\, j = j + 1)$ **do**
5:             $y(i) = f(i) - VA(j) \times y(JA(j))$
6:         **end for** j
7:     **end for** k
8: **end for** l

Avec cette approche, plus le nombre de niveaux n_{lev} se rapprochera de 1 et plus le degré de parallélisme sera élevé. Par exemple, si $n_{lev} = 1$, toutes les inconnues pourront être calculées en parallèle. C'est le cas d'une matrice diagonale. La boucle interne sur k pourra totalement être

parallélisée. Cependant, avant de continuer au niveau suivant (à la fin de la boucle sur k), il faudra s'assurer que tous les processus ont fini leur travail. C'est-à-dire qu'il faudra une barrière de synchronisation avant le début de chaque niveau. Cette dernière, si elle n'est pas performante, peut être un goulot d'étranglement des performances parallèles car certains algorithmes de synchronisation ont des complexités linéaires par rapport au nombre de coeurs à synchroniser (voir le Chapitre 4 pour plus de détails).

3.3.2 Coloriage de graphe



(a) *Graphe de la matrice cage4*

(b) *Graphe colorié de la matrice cage4 (4 couleurs)*

(c) *Matrice cage4 permuté*

Figure 3.4 – *(3.4a) représente le graphe de la matrice cage4 (le profile de la matrice est représenté par la Figure 3.1a) et (3.4b) est une représentation de la coloration du graphe (3.4a). La Figure 3.4c est une représentation de la partie triangulaire inférieur de la matrice cage4 permutée avec un coloriage de graphe classique (Algorithme 5).*

Un autre cas de figure qui permet d'avoir massivement du parallélisme à grain fin pour le calcul des inconnues est la coloration de graphe. Le principe est le suivant : parcourir le graphe d'adjacence de la matrice **L** et attribuer une couleur à chaque nœud de manière à ce que deux nœuds voisins ne puissent pas avoir la même couleur. On a une illustration de la méthode avec la partie triangulaire inférieur de la matrice cage4 sur la Figure 3.4. L'effet de cette méthode sur la matrice est l'apparition de blocs diagonaux qui sont des sous-matrices diagonales. Par la suite, la résolution du système triangulaire peut se faire en plusieurs étapes. En réalisant une Descente sur la partie triangulaire inférieure de la matrice cage4 permutée par coloriage de graphe (Figure 3.4c), on aura :

$$\left(\begin{array}{c|c|c|c} \mathbf{L}_{D_0} & 0 & 0 & 0 \\ \hline \mathbf{L}_{1,0} & \mathbf{L}_{D_1} & 0 & 0 \\ \hline \mathbf{L}_{2,0} & \mathbf{L}_{2,1} & \mathbf{L}_{D_2} & 0 \\ \hline \mathbf{L}_{3,0} & \mathbf{L}_{3,1} & \mathbf{L}_{3,2} & \mathbf{L}_{D_3} \end{array}\right) \left(\begin{array}{c} \mathbf{y}_0 \\ \hline \mathbf{y}_1 \\ \hline \mathbf{y}_2 \\ \hline \mathbf{y}_3 \end{array}\right) = \left(\begin{array}{c} \mathbf{f}_0 \\ \hline \mathbf{f}_1 \\ \hline \mathbf{f}_2 \\ \hline \mathbf{f}_3 \end{array}\right)$$

Avec les $\mathbf{L}_{i,j}$ sont des blocs de la matrice cage4 de taille égale à la taille de la couleur et $\mathbf{y}_i$ et $\mathbf{f}_i$ sont des blocs de vecteurs $\mathbf{y}$ et $\mathbf{f}$. Donc, pour cet exemple, le calcul de la solution par une Descente va se faire en 4 étapes (le nombre d'étapes correspond au nombre de couleurs) :

$$\xrightarrow{\textbf{Step 0}} \boxed{\mathbf{y}_0 = \mathbf{L}_{D_0}^{-1}\mathbf{f}_0} \xrightarrow{\textbf{Step 1}} \boxed{\mathbf{y}_1 = \mathbf{L}_{D_1}^{-1}\mathbf{f}_1 - \mathbf{L}_{1,0}\mathbf{f}_0} \xrightarrow{\textbf{Step 2}} \boxed{\mathbf{y}_2 = \mathbf{L}_{D_2}^{-1}\mathbf{f}_2 - \sum_{j=0}^{1}\mathbf{L}_{2,j}\mathbf{y}_j} \xrightarrow{\textbf{Step 3}} \boxed{\mathbf{y}_3 = \mathbf{L}_{D_3}^{-1}\mathbf{f}_3 - \sum_{j=0}^{2}\mathbf{L}_{3,j}\mathbf{y}_j}$$

L'intérêt de cette méthode vient de son efficacité à trouver des inconnues indépendantes. En comparaison à la méthode LS décrite ci-avant, il y a moins d'étapes (4 au lieu de 7) et plus d'inconnues par étapes. Ici aussi, chaque étape peut être calculée totalement en parallèle. De plus, pour ce qui nous intéresse dans cette thèse, la matrice **L** est une matrice triangulaire à diagonale unité, c'est-à-dire que sa diagonale est égale à 1. Donc, les différentes étapes peuvent être simplifiées, car l'inversion des sous-matrices $\mathbf{L}_{D_i}$ et la multiplication avec les vecteurs $\mathbf{f_i}$ ne sont pas à faire. L'algorithme de Descente avec une méthode de coloriage de graphe est très similaire à l'Algorithme 8. La boucle sur les niveaux est remplacée par une boucle sur les couleurs, avec n_c qui est le nombre de couleurs. **jcolors** est un vecteur contenant les nœuds permutés dans l'ordre croissant des indices de leur couleur. Et **icolors** est le vecteur contenant les pointeurs vers les couleurs dans **jcolors**. Nous avons alors l'algorithme suivant :

Algorithm 9 Descente avec une renumérotation GreedyMC pour SpTRSV

Inputs : Matrice creuse triangulaire inférieure à diagonale unité $\mathbf{L} \in \mathbb{R}^{n,n}$, et Vecteur second membre $\mathbf{f} \in \mathbb{R}^n$.
Inputs : Vecteur **jcolors** $\in \mathbb{R}^{n_{col}}$ contenant les nœuds par ordre croissant de leur couleurs.
Inputs : Vecteur **icolors** $\in \mathbb{R}^{n_{col}}$ contenant les pointeurs de début chaque couleur **jcolors**.
Outputs : Vecteur solution $\mathbf{y} \in \mathbb{R}^n$.

1: **for** $(c = 0\,; c < n_{col}\,; c = c + 1)$ **do**
2:     **for** $(k = icolors(l)\,; k < icolors(l + 1) - 1\,; k = k + 1)$ **do**
3:         $i = jcolors(k)$
4:         **for** $(j = IA(i)\,; j < IA(i + 1) - 1\,; j = j + 1)$ **do**
5:             $y(i) = f(i) - VA(j) \times y(JA(j))$
6:         **end for** j
7:     **end for** k
8: **end for** c

Ici, nous aurons également besoin de synchroniser les threads pour le passage d'une couleur à une autre. En général, on essaie de minimiser le nombre de couleurs en utilisant cette méthode ce qui réduit aussi le nombre de synchronisations lors des calculs en parallèle. Cependant, dans notre contexte de travail, la résolution des systèmes triangulaires vient de la factorisation ILU(0) de la matrice pour la résolution du système linéaire par une méthode de Krylov préconditionnée. Nous avons vu dans le chapitre 2 que l'utilisation de cette méthode dégrade souvent la convergence du solveur et que l'augmentation du nombre de couleurs à un effet positif sur la convergence du solveur. Dans un premier temps, nous chercherons à vectoriser (en AVX2 ou AVX512) notre méthode itérative. On décide ainsi de travailler avec une taille de couleur de 8 inconnues. Cela correspond à la taille d'une ligne de cache pour des valeurs en double précision. Dans ce cas, l'algorithme s'apparente à une version bloc de l'algorithme 7.

3.4 Parallélisation SIMD

3.4.1 Multiplication Matrice-Vecteur

Parmi les différentes opérations qui interviennent à chaque itération du BiCGStab préconditionné, nous avons 4 résolutions de systèmes triangulaires (SpTRSV), 2 multiplications matrice-vecteur (SpMV), six combinaisons linéaires de type dAXPY et cinq produits scalaires. Dans cette section, nous allons nous intéresser à la mise en œuvre vectorielle des noyaux SpMV et SpTRSV. La Figure (3.1) nous illustre les différences qui existent entre le format CSR et le format SELL-P en matière d'ordre de stockage des valeurs de la matrice, ce qui a un impact sur la manière de vectoriser les calculs. Prenons l'exemple de la multiplication matrice-vecteur. Cette opération consiste à faire un produit scalaire de chaque ligne de la matrice avec le vecteur multiplicateur pour construire le vecteur résultat.

Algorithm 10 CSR - Multiplication Matrice-Vecteur

1: **procedure** SMPV($\mathbf{A}, \mathbf{z}, \mathbf{y}$)
2: **Inputs :** Matrice : $\mathbf{A}$; Vecteur multiplicateur : $\mathbf{z}$
3: **Outputs :** Vecteur résultat : $\mathbf{y}$
4: **for** $(i = 1\,; i < n\,; i = i + 1)$ **do**
5: $\mathbf{y}[i] = 0$
6: **for** $(j = ptr[i]\,; j < ptr[i+1]\,; j = j + 1)$ **do**
7: $\mathbf{y}[i] += val[j] * \mathbf{z}[col[j]]$
8: **end for**
9: **end for**
10: **end procedure**

L'Algorithme 10 décrit la multiplication matrice-vecteur avec un pseudo-code C++. La matrice $\mathbf{A}$ en entrée est stockée au format CSR. Le vecteur $val[:]$ contient les valeurs de la matrice et le vecteur $col[:]$ nous donne les indices des colonnes de la matrice.

Parallélisation par colonnes

Une approche de vectorisation de cette opération avec un format de stockage CSR est de dérouler la boucle interne sur les colonnes pour prendre en compte plusieurs colonnes de la même ligne. Dans notre exemple d'algorithme, nous avons déroulé les opérations de la première boucle interne qui seront en pratique réduites à une instruction SIMD. Le paquet de vecteurs SIMD correspond à un registre AVX2 avec 4 valeurs en double précision. La première remarque qu'il faut noter est que l'accès aux valeurs du vecteur multiplicateur dans la première boucle interne ne sera pas continu. En effet, comme la matrice $\mathbf{A}$ est creuse, les indices des colonnes des éléments non nuls d'une ligne donnée peuvent correspondre à des indices assez éloignés dans le vecteur multiplicateur. En conséquence, nous ne pourrions pas bénéficier de la propriété de mise en cache par effet de localité spatiale des données. Il faudra faire une opération de type *gather* pour récupérer les valeurs correspondantes. La seconde remarque concerne la réduction de valeurs qu'il faudra faire à la fin de la boucle interne pour avoir un résultat scalaire (Algorithme 11-12). Le coût de cette opération va croître avec la taille des paquets de colonnes que l'on souhaite manipuler. La

Algorithm 11 CSR - Multiplication Matrice-Vecteur

```
1:  procedure 4 UNROLLING-SPMV(A, z, y)
2:      Inputs : Matrice : A ; Vecteur multiplicateur : z
3:      Outputs : Vecteur résultat : y
4:      for (i = 1 ; i < n ; i = i + 1) do
5:          tmp0 = tmp1 = tmp2 = tmp3 = 0
6:          for (j = ptr[i] ; j < ptr[i + 1] ; j = j + 4) do
7:              tmp0+ = val[j + 0] * z[col[j + 0]]
8:              tmp1+ = val[j + 1] * z[col[j + 1]]
9:              tmp2+ = val[j + 2] * z[col[j + 2]]
10:             tmp3+ = val[j + 3] * z[col[j + 3]]
11:         end for
12:         y[i]+ = tmp0 + tmp1 + tmp2 + tmp3
13:         for (j = j − 4 ; j < ptr[i + 1] ; j = j + 1) do
14:             y[i]+ = val[j] * z[col[j]]
15:         end for
16:     end for
17: end procedure
```

troisième remarque que l'on peut observer concerne la seconde boucle interne (Algorithme 11-13). Elle est nécessaire lorsque le nombre d'éléments de la ligne n'est pas divisible par la taille du paquet SIMD. Or, lorsque l'on manipule des matrices très creuses, ce cas de figure peut devenir majoritaire. Le surcoût de la seconde boucle et de la réduction peut devenir plus important que celui de la boucle interne vectorisée. On perd ainsi l'intérêt de la vectorisation.

Parallélisation par lignes

Une autre approche consiste à vectoriser la multiplication matrice-vecteur par rapport aux lignes, c'est-à-dire la boucle externe. Cela élimine les problèmes de réduction et de la seconde boucle interne. En revanche, avec un format CSR, nous aurons des accès aux éléments de la matrice qui ne seront plus de manière contiguë en mémoire. Il nous faudra des opérations de type *gather* pour récupérer les valeurs qui nous intéressent. De plus, il nous faudra gérer des tailles de lignes non uniformes avec le format CSR. Cependant, avec le format Ellpack et SELL-P, la taille des lignes est uniforme. Dans le cas Ellpack, toutes les lignes de la matrice ont la même taille et dans le cas du format SELL-P ce sont les lignes d'un paquet donné qui ont une taille uniforme. Pour la suite, nos méthodes utiliseront le format SELL-P car celui-ci est parfaitement adapté à la parallélisation SIMD que nous souhaitons mettre en œuvre [4, 52].

Dans la Figure 3.5, nous avons une illustration de la multiplication matrice-vecteur avec plusieurs lignes pour un stockage SELL-P. Sur cette figure, les billes rouges représentent le rembourrage des blocs de couleurs pour avoir des paquets de lignes qui soient un multiple d'un paquet SIMD. Les éléments encadrés peuvent être chargés en même temps avec une instruction SIMD. Cependant, étant donné que nous avons rembourré la matrice pour qu'elle ait les dimensions souhaitées, les éléments symbolisés par des croix n'existent pas dans la matrice d'origine. Il nous faut faire attention à ne pas manipuler des éléments virtuels dans nos calculs.

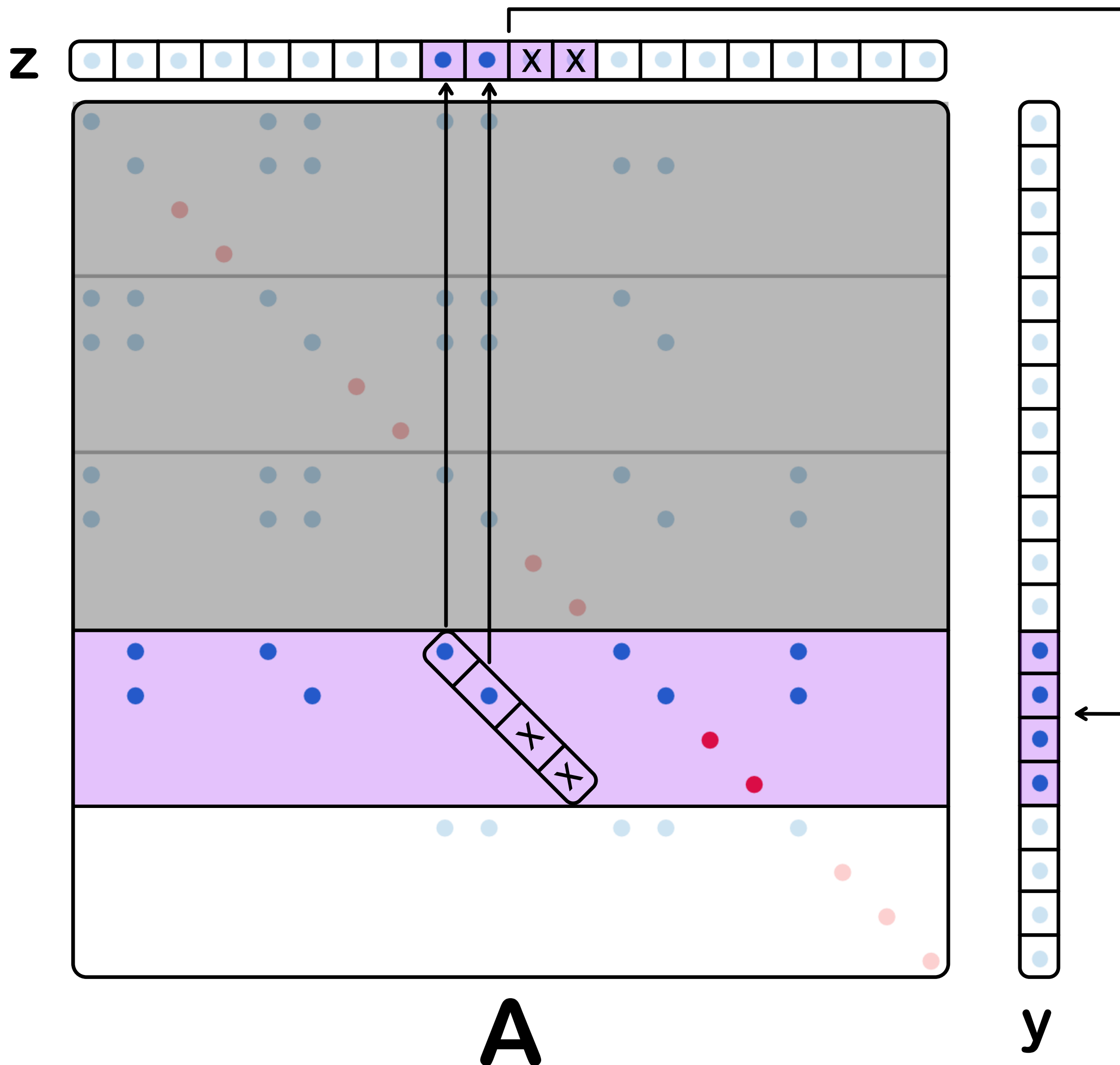


Figure 3.5 – *Illustration d'une multiplication matrice-vecteur* $\mathbf{A} \times \mathbf{z} = \mathbf{y}$ *avec* **A** *stocké au format* SELL-P.

3.4.2 Résolutions parallèle de systèmes triangulaires

Nous allons décrire la mise en œuvre vectorielle de la résolution de système triangulaire (SpTRSV) avec le format SELL-P. Les paquets d'éléments manipulés correspondent à une taille de registre AVX2. Les valeurs de la matrice sont en double précision et donc les registres AVX2 pourront contenir jusqu'à 4 valeurs. De même, la taille de nos paquets de lignes pour le format SELL-P sera égale à 4. La matrice creuse **A** sera stockée en trois parties : sa partie triangulaire inférieure stricte, sa partie triangulaire supérieure stricte et sa diagonale au format SELL-P. Nous utiliserons des notations du langage C++ pour manipuler les différents attributs de la matrice creuse. Notamment, $\mathbf{A}.strict_L_*$ et $\mathbf{A}.strict_U_*$ feront référence à la partie triangulaire inférieure stricte et à la partie triangulaire supérieure stricte. $\mathbf{A}.diag_*$ contient les différentes informations de la diagonale de la matrice. Comme nous l'avons illustré avec la multiplication matrice-vecteur, l'idée est de résoudre

plusieurs lignes d'équations simultanément.

Algorithm 12 Descente AVX2-SpTRSV en double précision

1: **procedure** AVX2-SPTRSV($\mathbf{A}, \mathbf{y}, \mathbf{f}$)
2: **Inputs :** Matrice : $\mathbf{A}$; Vecteur solution : $\mathbf{y}$; Vecteur second membre : $\mathbf{f}$
3: **Output :** Vecteur solution : $\mathbf{y}$
4: //La matrice $\mathbf{A}$ est séparée en une partie triangulaire inférieure stricte, triangulaire supérieure stricte et diagonale
5: dPaquet tmpVal ▷ un dPaquet est un conteneur de 4x64-bit
6: dPaquet tmpY
7: dPaquet tmpDiag
8: dPaquet sum
9: iPaquet *index* ▷ un iPaquet est un conteneur de 4x32-bit
10: **for** ($i = 0$; $i < nRowsPacket; i = i + RowsPacketSize$) **do**
11: sum = load(&$\mathbf{f}[i]$) ▷ Charger un paquet de valeur de 4x64-bit
12: **for** ($j = 0$; $j < \mathbf{A}.strict_L_ptr[i]$; $j = j + 1$) **do**
13: $index$ = $\mathbf{A}.strict_L_col[j].simdVec$ ▷ Charger le paquet d'indice des colones pointer par j
14: tmpY = gather(y,$index$) ▷ Récuperer les valeurs dans $\mathbf{y}$ correspondant aux indices dans $index$
15: tmpVal = $\mathbf{A}.strict_L_val[j].simdVec$ ▷ Copier le paquet d'élements pointer par j
16: sum = sum - tmpVal $\times$ tmpY ▷ Operation SIMD de type *fmadd*
17: **end for**
18:
19: tmpDiag = $\mathbf{A}.diag_val[i].simdVec$
20: sum = sum / tmpDiag
21: store(&$\mathbf{y}[i]$,sum)
22: **end for**
23: **end procedure**

L'Algorithme 12 décrit l'approche multi-lignes de la résolution d'un système triangulaire inférieur avec un stockage de la matrice au format SELL-P. Pour ne pas surcharger l'algorithme, nous avons décidé de simplifier certaines étapes. Nous omettons toutes les contraintes liées aux alignements des vecteurs sur des frontières de ligne de cache. Contrairement à la version CSR, maintenant, nous pouvons prendre plusieurs lignes consécutives, car l'ordre de stockage du format nous le facilite. En effet, comme illustré dans la Figure 3.1d, le format SELL-P va stocker d'abord les premiers éléments de chaque ligne, puis les seconds, etc. Cela a pour avantage qu'à un paquet de P lignes données et un pointeur j d'indice de colonne correspondant, les P éléments $val[j+0]$ à $val[j+P-1]$ seront stockés de manière contiguë en mémoire. Ainsi, nous pourrons charger ces éléments en une fois avec une instruction SIMD. C'est ce qui est fait à la ligne 15 de l'Algorithme 12. Par ailleurs, à l'étape 14, nous avons des opérations de type *gather*. C'est-à-dire que nous allons charger un ensemble de valeurs du vecteur $\mathbf{z}$ correspondant aux indices contenus dans *index*. En pratique, cette opération ne peut pas se faire directement. Les indices des colonnes de la matrice $\mathbf{A}$ chargés à l'étape 13 peuvent ne pas exister. En effet, avec le format SELL-P, un rembourrage des lignes est parfois nécessaire pour avoir une taille des lignes dans les paquets uniforme. De plus, nous avons rajouté artificiellement des éléments diagonaux (des 1 sur la diagonale) pour avoir un nombre de lignes total égal à multiple de paquet SIMD (4 en AVX2 et 8 en AVX512 pour des valeurs en double précision). Cela signifie que nous allons rajouter des indices de colonnes virtuelles (par exemple des indices de colonne à -1 ou illustrés dans la Figure 3.5 par des croix). Or, en chargeant notre paquet d'indices à l'étape 13, il se peut que nous ayons des indices à -1. Donc, pour faire l'opération de *gather* sans problème, nous allons utiliser des masques sur les vecteurs. Il

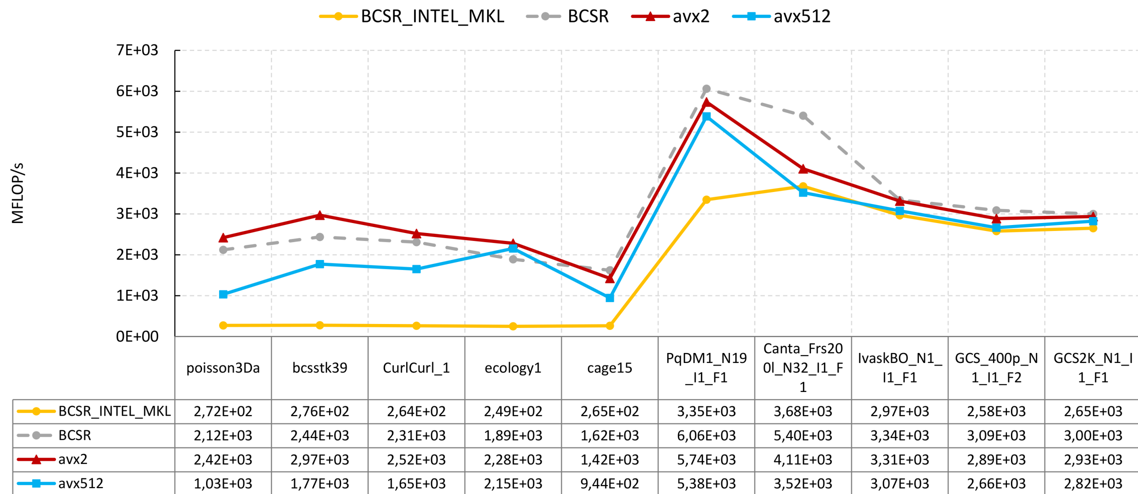
nous faudra construire un masque qui indiquera lors du *gather* les indices à ignorer. Ainsi, nous aurons un vecteur **tmpY** avec les bonnes valeurs et les valeurs masquées seront par défaut fixées à zéros. Pour les valeurs sur la diagonale, ce problème ne se pose pas. Enfin, l'opération à la ligne 16 est de type fmadd. C'est-à-dire une opération de multiplication de paquet de valeurs (**tmpVal** $\times$ **tmpY**) et d'additions du résultat avec les valeurs de **sum**. Le résultat final est stocké dans **sum**. À la fin du paquet de lignes, le résultat contenu dans **sum** est alors stocké dans $\mathbf{y}[i+0]$ à $\mathbf{y}[i+P-1]$. Il est évident que cette résolution de système triangulaire avec plusieurs lignes en simultané ne peut se faire sans s'assurer de l'indépendance des lignes de chaque paquet. C'est la raison pour laquelle en pré-traitement de la résolution, la matrice a été permutée avec la méthode de coloriage ColorRCM(p) qui va chercher à construire des blocs de p inconnues indépendantes entre elles. Certes, une méthode classique de coloriage glouton suffit à construire ces blocs d'éléments indépendants. Cependant, dans nos travaux, les résolutions de systèmes triangulaires interviennent lors de l'application du préconditionneur dans le solveur de Krylov. Dans ce contexte, la méthode de renumérotation utilisée en pré-traitement est importante pour l'efficacité du préconditionneur c'est-à-dire la dégradation ou pas de la convergence du solveur par rapport au solveur préconditionné sans renumérotation.

3.5 Expérimentations numériques

Dans cette section, nous allons nous intéresser aux performances parallèles de l'opération Matrice-Vecteur et de la résolution de systèmes triangulaires. Notre intérêt se porte sur les performances entre la version séquentielle format BCSR et la version SIMD au format SELL-P. Les tests ont été réalisés sur 10 matrices. Une partie est issue de la collection *Suite Sparse Matrix* et l'autre partie de problèmes d'écoulement en milieux poreux de l'IFPEN. Ces matrices sont structurellement symétriques et des informations sur leur structure creuse sont renseignées dans la Table 3.2. Chaque test a été lancé 10^3 fois et nous avons effectué une moyenne des temps de calcul par rapport aux nombres de lancés. Les performances des routines ont été mesurées en *floating-point operations per second* (FLOP/s) et leurs accélérations par rapport à `mkl_cspblas_dbsrgemv` (SpMV) et `mkl_cspblas_dbsrtrsv` (SpTRSV) de la librairie Intel MKL 21.4 sont renseignées dans la Table 3.3. Les tests ont été effectués sur le supercalculateur ENER440 de l'IFPEN. Il est constitué notamment de 2 processeurs Intel Xeon Gold 6140 par nœud de calcul, qui comportent 36 coeurs de calculs par socket et sont cadencés à 2.6 GHz. Le code est implémenté en C++ et est compilé avec le compilateur Intel 21.4.

3.5.1 Multiplication Matrice-Vecteur

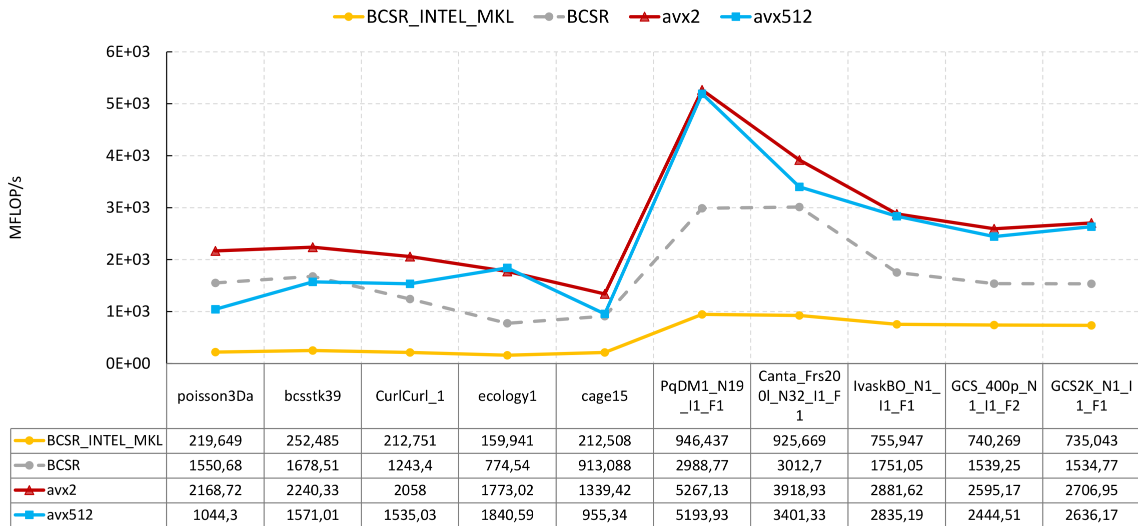
La Figure 3.6 représente les temps de calculs du produit matrice-vecteur. La première observation est qu'il y a une différence de performance entre la version en AVX2 et en AVX512. Ce dernier est nettement moins performant pour certaines matrices. Dans [54], l'auteur discute la baisse de performance de l'AVX512 par rapport à l'AVX2 qu'il a observée pour l'opération de produit matricielle de Intel MKL 19 sur un processeur Intel Xeon Gold 5118. Une réponse qui a été apportée suppose que ce problème est dû à l'architecture même des processeurs Intel Xeon Gold. En effet, certains processeurs Intel Xeon Gold (la série des 5000) possèdent 1 unité AVX512 de 512 bits et 2 unités AVX2 de 256 bits chacune avec en plus la technologie Turbo Boost. Cela permet



	poisson3Da	bcsstk39	CurlCurl_1	ecology1	cage15	PqDM1_N19_I1_F1	Canta_Frs20 0l_N32_I1_F1	IvaskBO_N1_I1_F1	GCS_400p_N1_I1_F2	GCS2K_N1_I1_F1
BCSR_INTEL_MKL	2,72E+02	2,76E+02	2,64E+02	2,49E+02	2,65E+02	3,35E+03	3,68E+03	2,97E+03	2,58E+03	2,65E+03
BCSR	2,12E+03	2,44E+03	2,31E+03	1,89E+03	1,62E+03	6,06E+03	5,40E+03	3,34E+03	3,09E+03	3,00E+03
avx2	2,42E+03	2,97E+03	2,52E+03	2,28E+03	1,42E+03	5,74E+03	4,11E+03	3,31E+03	2,89E+03	2,93E+03
avx512	1,03E+03	1,77E+03	1,65E+03	2,15E+03	9,44E+02	5,38E+03	3,52E+03	3,07E+03	2,66E+03	2,82E+03

Figure 3.6 – *Représentation les nombres d'opération flottante par seconde (en MFLOP/s) du produit Matrice-Vecteur de Intel MKL 21.4 (courbe jaune) et notre version au format BCSR (courbe en pointillé). Les résultats des versions AVX2 et AVX512 sont représentés par les courbes cyan et rouge.*

aux unités AVX2 d'augmenter automatiquement et temporairement la fréquence du processeur si les conditions de température le permettent. Donc cela semble donner un certain avantage à la performance de l'AVX2 qui pourra utiliser 2 instructions de 256 bits en parallèle et a une fréquence plus élevée que l'AVX512. Cependant, pour le processeur Intel Xeon Gold 6140 que nous avons utilisé pour nos tests, il est équipé de 2 unités AVX512 mais à cause de leur basse fréquence d'exécution par rapport aux unités AVX2, l'AVX512 ne semble pas non plus avoir un avantage majeur. Ce sont les conclusions évoquées dans [54]. La seconde observation est que la routine du produit matrice-vecteur de Intel MKL 21.4 avec les matrices de *Suite Sparse Matrix* a des performances assez basses. En moyenne, le produit matrice-vecteur de Intel MKL 21.4 est à 265 MFLOP/s pour les matrices de *Suite Sparse Matrix* alors que notre version AVX2 est à 2,3 GFLOP/s. Ce qui fait que, en moyenne, on a une accélération à 8. La version BCSR a des performances quasiment égales à la version AVX2 et est même plus performante que la version AVX152 pour les matrices de la collection *Suite Sparse Matrix*. La seconde partie de ce jeu de tests concerne les matrices de l'IFPEN. Dans ce cas, la majorité des tests ont une accélération légèrement moins importante. `mkl_cspblas_dbsrgemv` de Intel MKL 21.4 a des performances quasiment égales à AVX512-SpMV. Dans ce jeu de matrices, nous n'avons pas d'apport dans l'utilisation du SIMD. La version séquentielle avec le format BCSR est témoin de performances parfois nettement meilleures que les versions SIMD. Notre produit matrice-vecteur avec BCSR est optimisé pour tirer parti au maximum du calcul vectoriel pour des blocs supérieurs à 2x2. Les matrices de l'IFPEN sont stockées au format BCSR avec des blocs de 3X3. Pour les tests BCSR-SpMV, le code a été compilé avec le flag AVX2. Donc, le calcul des blocs 3x3 a été accéléré avec des instructions 256-bit SIMD. Cela explique les performances de BCSR-SpMV.



	poisson3Da	bcsstk39	CurlCurl_1	ecology1	cage15	PqDM1_N19_I1_F1	Canta_Frs200l_N32_I1_F1	IvaskBO_N1_I1_F1	GCS_400p_N1_I1_F2	GCS2K_N1_I1_F1
BCSR_INTEL_MKL	219,649	252,485	212,751	159,941	212,508	946,437	925,669	755,947	740,269	735,043
BCSR	1550,68	1678,51	1243,4	774,54	913,088	2988,77	3012,7	1751,05	1539,25	1534,77
avx2	2168,72	2240,33	2058	1773,02	1339,42	5267,13	3918,93	2881,62	2595,17	2706,95
avx512	1044,3	1571,01	1535,03	1840,59	955,34	5193,93	3401,33	2835,19	2444,51	2636,17

Figure 3.7 – *Représentation du nombre d'opérations flottante par seconde (en MFLOP/s) de la résolution des systèmes triangulaires de Intel MKL 21.4 (courbe jaune) et de nos versions SIMD* SpTRSV *(courbes cyan et rouge).*

Name_Matrix	Accél. AVX2/IMKL	Accél. AVX512/IMKL	Accél. AVX2/BCSR
poisson3Da	9,87	4,75	1,40
bcsstk39	8,87	6,22	1,33
CurlCurl_1	9,67	7,22	1,66
ecology1	11,09	11,51	2,29
cage15	6,30	4,50	1,47
PqDM1_N19_I1_F1	5,57	5,41	1,76
Canta_Frs200l_N32_I1_F1	4,23	3,86	1,30
IvaskBO_N1_I1_F1	3,81	3,72	1,65
GCS_400p_N1_I1_F2	3,51	3,27	1,69
GCS2K_N1_I1_F1	3,68	3,55	1,76

Table 3.3 – *Accélération de la version SIMD de la résolution de systèmes triangulaires par rapport à mkl_cspblas_dbsrtrsv de Intel MKL 21.4. La dernière colonne représente les accélérations BCSR-*SpTRSV *par rapport à AVX2-*SpTRSV*. La ligne pointillé sépare les matrices issues de la collection Suite Sparse Matrix aux matrice de l'IFPEN.*

3.5.2 Résolution des systèmes triangulaires

La Figure 3.7 représente les temps de calculs de la résolution de systèmes triangulaires pour la Descente et la Remontée. Concernant les performances du solveur en AVX512, il n'y a pas de gain par rapport à l'AVX2, voire même une sous-performance pour les matrices de la collection *Suite Sparse Matrix*. Avec la matrice `GCS_400p_N1_I1_F2`, nous avons en moyenne 2,7 GFLOP/s avec la version AVX2 de SpTRSV alors que la version de Intel MKL 21.4 est en moyenne à 735 MFLOP/s. Les accélérations que l'on observe de la version SIMD par rapport à Intel MKL 21.4 sont au moins trois fois plus rapides. Elles ont été reportées dans la Table 3.3. En moyenne, nous observons environ quatre fois plus d'accélération avec AVX2-SpTRSV par rapport à Intel MKL 21.4 pour les matrices de simulation d'écoulement en milieu poreux. Pour les matrices de la collection *Suite*

Sparse Matrix, la version AVX2 est neuf fois plus rapide que la résolution de systèmes triangulaires avec Intel MKL 21.4. Si on compare le nombre d'opérations flottantes par seconde de la version AVX2 par rapport au format BCSR, nous avons en moyenne des accélérations de 1,6. Nous avons reporté les accélérations pour chaque test dans la Table 3.3.

3.5.3 Performance du solveur de Krylov

Name_Matrix	Seq. temps iter./Niter (s)	AVX2 temps iter./Niter (s)	Accélération
poisson3Da	1.73E-03	1.20E-03	1.44
bcsstk39	9.96E-03	8.25E-03	1.21
CurlCurl_1	1.62E-02	1.21E-02	1.34
ecology1	5.05E-02	3.13E-02	1.61
cage15	7.39E-01	5.51E-01	1.34
PqDM1_N19_I1_F1	6.17E-04	2.58E-04	2.40
Canta_Frs200l_N32_I1_F1	1.67E-03	†	†
IvaskBO_N1_I1_F1	2.01E-02	1.48E-02	1.36
GCS_400p_N1_I1_F2	5.70E-02	4.02E-02	1.42
GCS2K_N1_I1_F1	1.31E-01	9.02E-02	1.46

Table 3.4 – *Temps de calcul moyen par itération du solveur entre la version du solveur* ILU(0)-BiCGStab *sans SIMD et le solveur* ILU(0)-BiCGStab *avec AVX2 pré-traiter par* ColorRCM*(4). La dernière colonne du tableau montre les accélérations des itérations du solveur SIMD par rapport aux itérations du* ILU(0)-BiCGStab *séquentiel. Le symbole † signifie que le solveur a divergé. La ligne pointillé sépare les matrices issues de la collection Suite Sparse Matrix aux matrice de l'IFPEN.*

Dans cette section, nous nous sommes intéressés à l'impact sur les performances du solveur de Krylov préconditionné. Nous avons comparé les temps de calculs moyen par itération entre la version AVX2 du solveur et la version sans SIMD. Pour la version en AVX2 du solveur, nous avons calculé une permutation de la matrice avec ColorRCM(4). Dans cette configuration du solveur, les opérations SpMV et les applications du préconditionneur lors des itérations ont été accélérées grâce au SIMD. Pour la configuration sans SIMD du solveur, nous n'avons pas utilisé de pré-traitement de la matrice. Lors de la résolution du système linéaire, l'application du préconditionneur est très coûteuse en temps. Le fait que nous ayons une forte accélération de la résolution triangulaire grâce au calcul SIMD se fait ressentir sur le temps de calcul moyen par itération. La version séquentielle du solveur utilise les routines SpMV et SpTRSV avec le format BCSR que nous avons présenté plus haut. Lors de l'analyse des performances du produit matrice-vecteur, nous n'avons pas observé de réelle accélération entre la version BCSR et la version AVX2. Donc, l'accélération des itérations du solveur observée reflète les accélérations des résolutions des systèmes triangulaires. Comme pour les performances des résolutions de systèmes triangulaires en AVX2 par rapport à la version avec le format BCSR, nous avons en moyenne une accélération de 1,5 des itérations du solveur. Cela traduit un gain de temps relatif de 50 % par itération du solveur (voir Table 3.4).

3.6 Discussion

Dans cette partie du manuscrit, le but était de tirer parti de la structure apportée par le coloriage de graphe pour le calcul vectoriel dans la phase de résolution des systèmes triangulaires. Nous avons effectué des mesures des temps de calculs de SpTRSV pour 10 matrices avec un nombre d'éléments non nuls allant de 17 086 pour la plus petite à plus de 99 millions pour la plus grosse. Le jeu de tests était composé de matrices issues de la collection *Suite Sparse Matrix* et de simulations d'écoulement en milieux poreux de l'IFPEN. Nos tests en AVX2 par rapport aux routines sans AVX2 de la librairie Intel MKL 21.4 ont montré de fortes accélérations. Le produit matrice-vecteur montre des performances moyennes de 2,3 GFLOP/s pour les matrices de la collection *Suite Sparse Matrix* et 3,7 GFLOP/s pour les matrices de l'IFPEN. Pour la résolution de systèmes triangulaires, nous avons au minimum une accélération de 3,5 en AVX2 et de 3,2 jusqu'à 5,4 en AVX512 par rapport à la routine `mkl_cspblas_dbsrtrsv` de Intel MKL 21.4. Et 1,3 d'accélération minimum par rapport à notre version au format BCSR. Comme nous l'avons vu en préambule du manuscrit, dans un solveur de Krylov préconditionné comme BiCGStab avec ILU(0), il y a 4 résolutions de systèmes triangulaires (SpTRSV), 2 multiplications matrice-vecteur (SpMV), 6 combinaisons linéaires de type dAXPY et 5 produits scalaires. Nous avons implémenté les versions SIMD de SpMV et SpTRSV. Nos tests préliminaires montraient qu'une majorité du temps de calcul à chaque itération du solveur ILU(0)-BiCGStab était consacrée à l'application du préconditionneur ILU(0). Nous nous sommes intéressés à l'accélération des itérations du solveur ILU(0)-BiCGStab dans la version AVX2 par rapport à la version séquentielle au format BCSR. Nos tests ont montré qu'en moyenne nous avons une accélération de 1,5, soit une accélération de 50 % par itération pour la version AVX2 du solveur (voir Table 3.4).

Perspectives

Comme travail à venir, nous avons remarqué que l'application de la permutation de coloriage qui est locale au préconditionneur pouvait être coûteuse en temps. En effet, pour le calcul du préconditionneur, nous faisons une copie de la matrice sur laquelle la permutation du coloriage est appliquée puis ILU(0) y est calculé. À chaque application du préconditionneur, nous avons besoin d'appliquer la permutation du coloriage au vecteur second membre puis la permutation inverse au vecteur résultat. Par exemple, ces permutations de vecteurs représentent 39 % du coût de l'application du préconditionneur pour les matrices SPE10 et 17 % pour `PqDM1_N19_I1_F1`. Une possibilité d'amélioration des performances du solveur pourrait être que la permutation soit globale à tout le solveur. Ainsi, il n'y aurait plus besoin de permuter le second membre et la solution à chaque application du préconditionneur.

Factorisation LU Incomplète Dans cette section, nous allons discuter des difficultés de la mise en œuvre vectorielle de la factorisation incomplète ILU(0). En effet, contrairement à la multiplication matrice-vecteur ou la résolution de système triangulaire, la factorisation ILU(0) implique trois boucles imbriquées.

1: **for** $(i = 2\,; i < n\,; i = i + 1)$ **do**
2:     **for** $(k = 1\,; k < i - 1\,; k = k + 1)$ **do**
3:         $a_{i,k} = \frac{a_{i,k}}{a_{k,k}}$; //Uniquement sur les éléments non nul

4: **for** ($j = k+1$; $j < n$; $j = j+1$) **do**
5: $a_{i,j} = a_{i,j} - a_{i,k} \times a_{k,j}$; //Uniquement sur les éléments non nul
6: **end for** j
7: **end for** k
8: **end for** i

Pour calculer les éléments $a_{i,j}$, il nous faut avoir les éléments $a_{i,k}$ et $a_{k,j}$. Dans une approche multiligne avec le format SELL-P, nous matérialisons les paquets d'indices par les lettres en majuscules. Ainsi, $a_{I,J}$ signifie que nous manipulons les éléments $a_{i+0,j}$, $a_{i+1,j}$, $a_{i+2,j}$, $a_{i+3,j}$ avec des paquets de 4 lignes par exemple. Comme nous sommes sur le paquet de lignes I, la manipulation des éléments $a_{I,K}$ se fait aisément. En revanche, les éléments $a_{K,J}$ se situent sur les lignes $K < I$. La première difficulté qui se présente à nous est due au fait que la ligne d'indice k dans K n'est pas nécessairement une frontière de paquet de lignes. C'est-à-dire que k peut être une ligne quelconque au milieu d'un paquet de lignes. La seconde difficulté est que les différents indices k ne sont pas nécessairement consécutifs. Par exemple, $k+0$ peut appartenir à un certain paquet de lignes et $k+1$ appartenir à un autre paquet de lignes et de même pour les autres indices. Cela implique qu'il nous faudra faire plusieurs opérations de type *gather* sur différents paquets de lignes pour constituer le vecteur $a_{K,J}$. Or, le format SELL-P n'a pas été conçu pour ce type d'utilisation. Nous avons illustré ce problème avec un exemple dans la Figure 3.8. Les lignes en gris sont celles dont la factorisation a été effectuée. Le paquet de lignes en violet est l'ensemble de celles en cours de calcul. La partie de la matrice non colorée est l'ensemble des lignes qui ne sont pas encore modifiées. L'Algorithme 2 de Factorisation LU Incomplète tel que décrit plus haut indique que la zone grise ne sera plus modifiée, mais nous aurons uniquement des opérations de lecture dans cette zone. Et les opérations d'écriture seront uniquement dans la zone violette et indiquées par les flèches. Dans cet exemple, nous voulons calculer le bloc d'éléments $a_{I,J} = [a_{12,8}, a_{13,9}, a_{14,-1}, a_{15,-1}]$. Nous avons donc besoin du bloc $a_{I,K} = [a_{12,4}, a_{13,5}, a_{14,-1}, a_{15,-1}]$ et du bloc $a_{K,J} = [a_{8,4}, a_{9,5}, a_{10,-1}, a_{11,-1}]$ Le chargement du vecteur $a_{I,K}$ peut se faire avec une instruction SIMD et un masque pour ne pas charger les colonnes inexistantes dans la matrice d'origine. Cependant, le chargement des éléments de $a_{K,J}$ va se faire à l'aide de deux opérations de *gather* pour récupérer l'élément $a_{8,4}$ et $a_{9,5}$. Ceci pourrait s'avérer coûteux en temps surtout lorsque le vecteur SIMD est plus conséquent, notamment pour l'AVX512. Il semblerait donc que ce format SELL-P ne soit pas adapté pour la parallélisation de l'algorithme de factorisation ILU(0) de manière performante. Cependant très récemment, dans [75], les auteurs ont proposé une méthode de Factorisation LU Incomplète par bloc avec la spécificité de n'accepter du remplissage que dans les blocs de taille préalablement fixée par l'utilisateur. Par la suite, ils sont proposés une parallélisation de leur factorisation avec une vectorisation des calculs lors de la factorisation en utilisant au préalable une renumérotation des lignes avec la méthode de coloriage ABMC (*Algebraic Bloc Multi-Coloring*) présenté dans [47]. Le format de stockage des données utilisé est le format BCSR. Il serait intéressant de voir les performances que l'on pourrait obtenir avec une vectorisation de la factorisation ILU(0) renumérotée au préalable par ColorRCM avec le format de stockage des données BCSR.

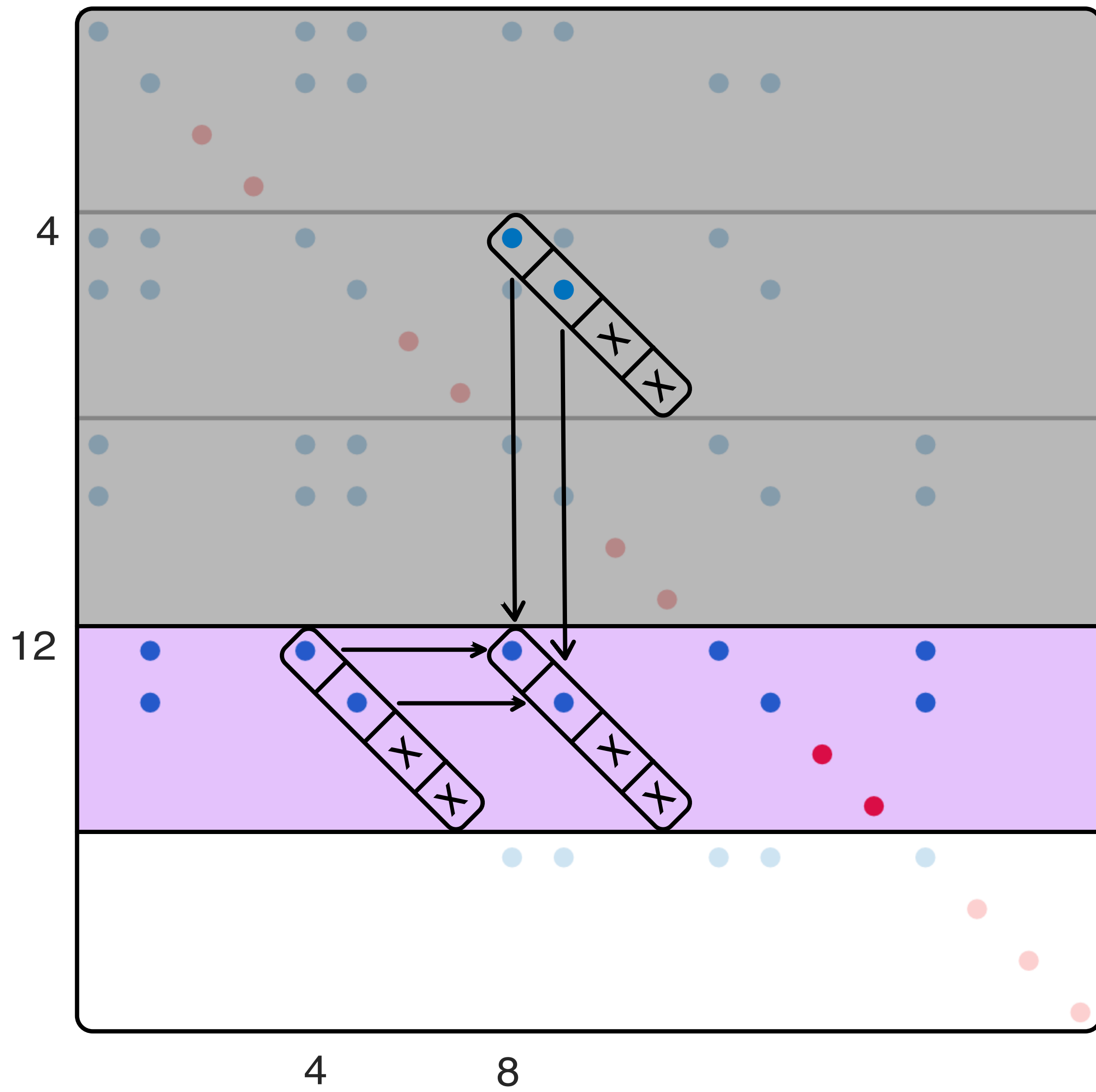


Figure 3.8 – *Illustration des échanges de valeurs pour la factorisation* ILU(0) *avec* **A** *stocké au format* SELL-P

4

Réduction combinée avec une barrière de synchronisation pour des processeurs multi-coeurs

Contents

4.1 Introduction **62**

4.2 État de l'art des barrières de synchronisation **65**

4.2.1 La barrière *Sense-reversing Centralized* 65

4.2.2 La barrière *Linear Centralized* 65

4.2.3 La barrière *Software Combining Tree* 66

4.2.4 La barrière *Tournament* 67

4.2.5 La barrière *Butterfly* 68

4.2.6 La barrière de *Dissemination* 69

4.3 La barrière ***Extended Butterfly*** **70**

4.3.1 False sharing 72

4.4 Réduction combinée avec une barrière **73**

4.4.1 *Padding* des flags de synchronisation 74

4.4.2 Écriture SIMD atomique 75

4.4.3 Effet des communications redondantes lors de la réduction 76

4.4.4 Mise en œuvre de la réduction *Extended Butterfly* 78

4.5 Résultats expérimentaux **80**

4.5.1 Environnement test 80

4.5.2 Résultats sur les barrières et réductions 81

4.5.3 Benchmark EPCC 85

4.6 Discussion **86**

4.1 Introduction

Le processeur KNL (Knight Landing) d'Intel, doté d'une architecture *Many Integrated Core* (MIC) couplée à l'hyperthreading, possède jusqu'à 72 cœurs physiques et peut avoir jusqu'à 4 threads par cœur. Ce processeur est l'un des premiers processeurs many-core et représente la tendance actuelle de la forte croissance du nombre de cœurs dans les microprocesseurs. L'ensemble de ces cœurs ont besoin, au cours d'une exécution, de points de synchronisation parfois récurrents pour le bon fonctionnement des algorithmes parallèles. Pour éviter que ces points de synchronisation ne deviennent des goulots d'étranglement des performances parallèles des applications, il est nécessaire de mettre en œuvre des méthodes de synchronisation robustes et scalables. Pour atteindre ces niveaux de performance, il est primordial de prendre en compte l'architecture des microprocesseurs et leur spécificité. Par exemple, l'utilisation d'instructions SIMD dans les barrières de synchronisation et les réductions des valeurs. Notre intérêt se porte sur la réduction et plus particulièrement sur la réduction combinée à une barrière de synchronisation. Il est donc nécessaire de s'intéresser à l'accélération de ces dernières avant de chercher à optimiser le calcul de réduction. Une réduction combinée à une barrière consiste à profiter de la synchronisation des threads dans la barrière pour effectuer l'opération de réduction sur les valeurs à réduire. De nombreuses études proposent de nouveaux algorithmes de synchronisation prenant en compte les caractéristiques particulières des architectures de microprocesseurs. Parmi les mécanismes de synchronisation considérés, les barrières et les réductions sont certainement les plus utilisés. Les barrières sont des parties du code où les threads s'attendent mutuellement. Les réductions sont des mécanismes où les threads vont échanger des valeurs et appliquer des opérations sur ces dernières pour obtenir la même valeur scalaire à la fin du traitement. Habituellement, pour effectuer une opération de réduction, nous avons obligatoirement besoin d'une barrière avant et/ou après la réduction même si cela est coûteux en temps. Une solution alternative consiste à utiliser les threads au cours de leurs synchronisations dans la barrière pour effectuer l'opération de réduction. Dans la suite de ce chapitre, nous allons nous concentrer sur cette deuxième approche. Comme mentionné dans [44], notre barrière passera par trois phases : l'initialisation, la phase entrante et la phase sortante, dont nous parlerons plus loin. Nous commencerons par discuter des barrières proposées dans la littérature, puis nous traiterons des méthodes de réduction. Du point de vue des barrières de synchronisation, la plus connue et peut-être la plus ancienne est la barrière centralisée. Son mode de fonctionnement consiste à ce que les threads incrémentent un compteur global à chaque entrée et se mettent en attente-active sur un flag global jusqu'à ce que le dernier thread vienne modifier l'état du flag. Le flag désigne la variable de synchronisation qui indique aux threads s'ils peuvent sortir de leur état d'attente-active ou pas. Ce flag prendra les valeurs `WAIT` ou `Go` (0 ou 1). Dans la suite, par abus de langage, nous parlerons de flag de synchronisation pour désigner la valeur du flag de synchronisation. La méthode de la barrière centralisée crée de nombreux points chauds lors de l'exécution du code, provoquant une sérieuse dégradation des performances ainsi que de la contention mémoire à mesure que le nombre de threads augmente, [63]. Les points chauds se produisent lorsqu'un grand nombre de processeurs essaient d'accéder simultanément à la même variable globale. À la fin des années 80, les algorithmes *Butterfly* [12] et *Software Combining Tree* [80] ont été publiés et ont significativement réduit les conflits mémoire durant l'exécution de la barrière. Dans sa version originale, l'algorithme *Butterfly* effectue une synchronisation par paire à chaque étape en utilisant un tableau de flags partagé entre les threads. Il est connu pour être très efficace et est couramment utilisé lorsque le nombre de threads est une puissance de deux, même

si une version non-puissance de deux a également été proposée dans [12]. L'algorithme *Software Combining Tree*, quant à lui, est basé sur un arbre binaire, où le dernier thread atteignant son nœud décrémente le compteur global du nœud. Cette méthode réduit considérablement le nombre d'accès simultanés à un compteur global, contrairement à la barrière centralisée, en utilisant des compteurs locaux. Mais les threads doivent descendre dans l'arbre jusqu'à atteindre la racine, puis remonter vers les feuilles, voir la Figure 4.2. La barrière *Software Combining Tree* fonctionne bien lorsque le nombre de threads augmente et pour un nombre quelconque de threads. Basée sur [12, 40], les barrières de *Dissemination* et *Tournament* ont été présentées dans [42]. La barrière de *Dissemination* est une amélioration de la barrière *Butterfly*, elle a les mêmes performances que la barrière *Butterfly* pour un nombre de threads égal à une puissance de deux et passe très bien à l'échelle pour les autres puissances de deux. La barrière *Tournament* est aussi une amélioration de la barrière *Software Combining Tree*. Cette approche élimine le besoin de compteur dans chaque nœud et donc le besoin d'opération atomique pour synchroniser les threads au sein du nœud. Par la suite, dans [58], sont passées en revue les cinq barrières de [12, 42, 80] et est proposée une nouvelle barrière basée sur un arbre. De plus, les auteurs ont proposé une amélioration de ces différents algorithmes en utilisant des flags locaux pour les threads en attente-active. Leurs résultats expérimentaux ont montré que la barrière *Dissemination* était la plus efficace de l'état de l'art. Depuis, d'autres études et algorithmes sont apparus pour améliorer et gagner quelques pourcentages supplémentaires [2, 38, 44, 61]. Néanmoins, une avancée majeure dans le domaine, pour les synchronisations en mémoire partagée, a été proposée par [15] avec la barrière en arbre *Multi-Degree SIMD Combining Tree*. Cette barrière est une *Software Combining Tree* barrière qui gère plus de deux threads par nœud et peut changer la taille du groupe de threads suivant le niveau dans l'arbre. Cette approche est compétitive, car elle utilise des instructions SIMD pour réveiller des groupes de threads à chaque niveau. Elle montre également une accélération allant jusqu'à 2,84 fois par rapport à la barrière OpenMP avec le compilateur Intel 2013 dans le micro-benchmark EPCC [13]. La démocratisation de la technologie SMT (Simultaneous Multi-Threading) dans les architectures de processeurs modernes oblige les développeurs à adapter leurs algorithmes pour prendre en compte l'évolution apportée du côté matériel. Dans [65], un nouvel algorithme nommé *Hybrid* est introduit et consiste à : synchroniser d'abord tous les threads s'exécutant dans le même cœur physique avec une barrière centralisée, puis utiliser une barrière de *Dissemination* pour la synchronisation interne au noyau. Cet algorithme a montré 3 fois moins de surcoûts que la barrière OpenMP avec le compilateur Intel 2014. Une autre approche intéressante consiste à synchroniser les threads en fonction de leur emplacement dans un nœud NUMA (Non-Uniform Access Memory). Elle consiste à :

1. Synchroniser les threads sur le même nœud NUMA à l'aide d'une barrière en arbre. Dans chaque groupe, le dernier thread atteignant la racine sera alors désigné pour continuer.

2. Synchroniser ces derniers threads entre eux.

Cette méthode a été développée dans [81]. Nous constatons donc que les méthodes récentes qui montrent de bonnes performances sur un grand nombre de threads procèdent en deux étapes en mélangeant deux méthodes de synchronisation différentes. Nous adoptons la même approche pour concevoir la barrière *Extended Butterfly*. En utilisant cette idée de synchronisation par groupe de threads, nous proposons une barrière hybride comme [65] mais avec une barrière *Butterfly* pour la synchronisation inter-groupe et une barrière centralisée pour la synchronisation intra-groupe. Nous avons choisi l'algorithme *Butterfly* car ses schémas de communication sont adaptés pour la mise en œuvre de l'opération de réduction. Pour rappel, l'opération de réduction consiste à

appliquer une opération à un ensemble de valeurs locales pour obtenir une valeur scalaire ou un tableau. Le cas le plus simple est de réduire les valeurs en utilisant des opérations telles que la somme ou la multiplication, le maximum ou le minimum. Dans les grands systèmes avec NUMA, ces opérations de réduction sont critiques, car la communication entre les différents modules de mémoire peut être très coûteuse. Dans [14], les auteurs ont proposé de combiner une opération de réduction avec une barrière *Multi-Degree SIMD Combining Tree*. Elle apporte une accélération jusqu'à 1,56 fois par rapport à la réduction de ICC 2013 OpenMP 4.0. Cependant, malgré les efforts pour optimiser la communication et exploiter les spécificités des différents processeurs, la majorité des travaux n'abordent pas le sujet des espaces inutilisés dans les lignes de cache (payload). Lorsque d'une réduction combinée avec une barrière de synchronisation en mémoire partagée, la valeur de synchronisation est souvent alignée à la frontière d'une ligne de cache et laisse inutilisé l'espace restant de la ligne de cache. Cette charge utile est nécessaire pour éviter un faux partage (*false sharing*) au cours de la barrière. Une opération de réduction combinée à une barrière de synchronisation proposée dans [74], utilise cette charge utile pour transporter les valeurs de réduction et le flag de synchronisation sur la même ligne de cache. Cela a comme avantage que les valeurs de réduction sont disponibles en même temps que le flag de synchronisation. Cet article est le premier à proposer de manipuler le *payload* engendré par la barrière de synchronisation lors de l'opération de réduction combinée à la barrière *Tournament*. Cette nouvelle méthode montre un gain de vitesse allant jusqu'à 1,53 fois par rapport à la réduction OpenMP par défaut avec le compilateur GCC 4.5 dans le benchmark 312.swim_m SPEC OMP2001. Nous appliquons la même approche à notre réduction. De plus, nous utilisons des instructions SIMD de lecture/écriture pour travailler avec toute la ligne de cache.
Dans ce travail, un algorithme de réduction combinée à une barrière pour les processeurs avec un grand nombre de threads est proposé. La solution adoptée propose une amélioration de la barrière *Butterfly* appelée *Extended Butterfly*. Cette amélioration réside dans le fait de créer des groupes de threads pour une synchronisation à deux niveaux : intra-groupe et inter-groupe. De plus lors de la réduction, nous proposons une méthode pour transporter les valeurs de réduction avec le flag de synchronisation, qui peut gérer jusqu'à 7 valeurs en double précision à réduire par thread. Cette méthode est simplifiée et optimisée en utilisant des instructions SIMD pour écrire les 8 doubles (flag de synchronisation et valeurs de réduction) de manière atomique.
La suite de la rédaction est structurée comme suite : nous présentons l'état de l'art des algorithmes de barrière dans la Section 4.2. Dans cette section, ça sera l'occasion de détailler les principales barrières issues de l'état de l'art. Nous illustrons le fonctionnement de chaque barrière et analyserons certaines propriétés qui nous intéressent pour la suite de notre travail. Notre nouvelle barrière sera décrite dans la Section 4.3. Comme pour les autres barrières, celle-ci sera illustrée et nous discuterons ses propriétés. Ensuite, nous analyserons les méthodes de réduction dans la Section 4.4 et les spécificités de la réduction *Extended Butterfly*. Ils seront détaillés dans cette section, la mise en œuvre de la réduction combinée avec la barrière *Extended Butterfly* et de la manière dont nous manipulons les valeurs à réduire au cours de l'exécution. Enfin, dans la Section 4.5, les résultats expérimentaux obtenus sur différents processeurs multi-cœurs (Intel Skylake X (SKL), AMD Milan (MILAN) et Intel Knights Landing (KNL)) seront également présentés.

4.2 État de l'art des barrières de synchronisation

Dans cette section, nous présentons plusieurs algorithmes de barrière connus. Nous allons décrire et discuter en détail du fonctionnement et de certaines propriétés des barrières. Notre intérêt se porte essentiellement sur deux points : le comportement théorique de la barrière par rapport à un nombre croissant de threads et la possibilité ou non de la coupler à une réduction. Le premier point reflète la scalabilité des méthodes et le second point concerne les échanges de notifications des threads dans la barrière. Nous allons au préalable définir quelques notions qui nous serons utiles par la suite.

Définition 4.1 (Arbre binaire)**.** Un arbre binaire est un graphe connexe acyclique (en un morceau et sans cycle), tel que le degré de chaque sommet (nœud) soit au plus égale à 3. Un arbre peut se caractériser par une structure hiérarchique débutant par une racine et terminant par les feuilles.
Une racine est le seul nœud de l'arbre qui n'a pas de nœud lui précédant (nœud père).
Une feuille est le seul nœud de l'arbre qui n'a pas de nœud lui succédant (nœud fils).

4.2.1 La barrière *Sense-reversing Centralized*

Un algorithme naïf de barrière centralisée utilise un flag de synchronisation global et un compteur global. Les premiers $\mathbf{P} - 1$ threads atteignant la barrière incrémenteront le compteur et passeront en mode "attente-active". Le dernier thread les libère en inversant la valeur du flag global. Chaque thread décrémentera le compteur et le dernier thread réinitialisera le flag. La méthode de *sense-reversing* améliore l'approche naïve en utilisant des flags de synchronisation locaux, et chaque thread va tourner sur son propre flag local. L'algorithme peut être décrit par les étapes suivantes :

1. Les premiers $\mathbf{P} - 1$ threads vont diminuer le compteur et se mettre en attente-active jusqu'à ce que leur flag local soit égal au flag global.
2. Le dernier thread atteignant la barrière remet le compteur global à $\mathbf{P}$ et le flag global à la valeur du flag local. Cela libère tous les threads en attente active et prépare la barrière pour sa prochaine utilisation.

Cette approche proposée dans [58, Algorithme 8] réduit le nombre d'opérations d'incrémentations atomique par rapport à la barrière centralisée naïve [58, Algorithme 6]. Ici, nous avons un compteur global qui ne pourra être décrémenté que de manière atomique, c'est-à-dire par un thread à la fois. Ce qui sérialise ces opérations de décrémentation. Ainsi, cela rend la scalabilité de cet algorithme intrinsèquement linéaire en fonction du nombre de threads. Cependant, l'échange de notifications dans la barrière entre les threads est réduit au strict minimum à savoir une notification. En effet, dans cette variante de [58], tous les threads dans la barrière attendent la notification du dernier thread entrant dans la barrière.

4.2.2 La barrière *Linear Centralized*

Les threads qui arrivent dans la barrière signalent leur présence en inversant leur flag de synchronisation, puis passent en mode attente active. Pendant la phase d'arrivée, le thread maître vérifie si tous les autres threads sont arrivés dans la barrière. Les flags sont ensuite mis à jour

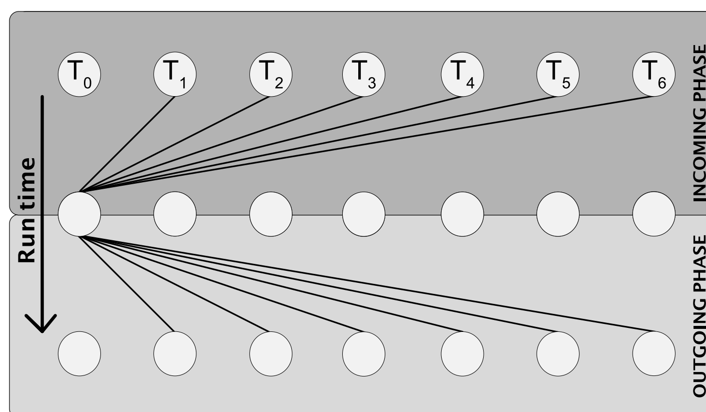


Figure 4.1 – *Exemple de schéma d'échanges de notifications dans la barrière Linear Centralized avec 7 threads.*

par le maître pendant la phase sortante pour libérer les élèves en attente active. La Figure 4.1 illustre la manière dont les informations sont traitées par le thread maître. Cette barrière contraste avec la barrière *Sense-reversing Centralized* car elle choisit un thread pour vérifier à la fois la phase entrante et la phase sortante de la barrière. De cette façon, les opérations atomiques ne sont pas nécessaires.Comme les entrées et sorties des threads dans la barrière sont gérées par le thread maître. Pour la phase d'entrée, le thread maître doit vérifier un par un que les threads sont bien arrivés dans la barrière. Ce qui fait que le nombre d'étapes nécessaire pour finaliser la phase d'entrée pour **P** threads sera de **P**-1. Ce qui implique une scalabilité linéaire par rapport au nombre de threads. Durant la phase de sortie, il y a autant de notification envoyé par le thread maître qu'il y a de thread en attente-active dans la barrière. Donc chaque thread recevra une notification du thread maître.

4.2.3 La barrière *Software Combining Tree*

"Diviser pour régner" est le principe appliqué dans ce cas. Les threads sont rassemblés en groupes d'au plus k membres. Nous allons considérer le cas où k est au plus égal à 2. Dans chaque nœud de l'arbre, une barrière centralisée est utilisée, et le dernier thread à atteindre le nœud (coloré dans la Figure 4.2) pourra continuer vers le nœud suivant pendant que l'autre attendra. Ainsi de suite jusqu'à ce que le dernier thread atteigne le nœud racine et remette le compteur à zéro. Ensuite, les threads se réveilleront de la racine vers les feuilles (voir la phase sortante de la Figure 4.2). L'algorithme est présenté dans [58, Algorithme 9]. Cette approche réduit l'effet de sérialisation de la barrière *Sense-reversing Centralized* en distribuant les opérations atomiques sur les nœuds. Plus récemment, une nouvelle version de l'Algorithme *Software Combining Tree* dans [15] permet de gérer des tailles de groupe différentes d'un niveau de l'arbre à l'autre et utilise une instruction SIMD pour libérer le groupe dans chaque nœud. Cette barrière est nommée *Multi-Degree SIMD Combining Tree*.Cependant dans [58, Algorithme 9], les threads ont besoin de $\lceil log_2(\text{nombre de groupes})\rceil$ étapes pour atteindre la racine et finaliser la première phase de la barrière. Nous estimons que la scalabilité de cette barrière sera logarithmique en fonction du nombre de groupes de thread. Concernant les échanges de notifications, celles-ci se font lors de la second phase de la barrière. Les premiers threads de chaque nœud de l'arbre vont attendre que

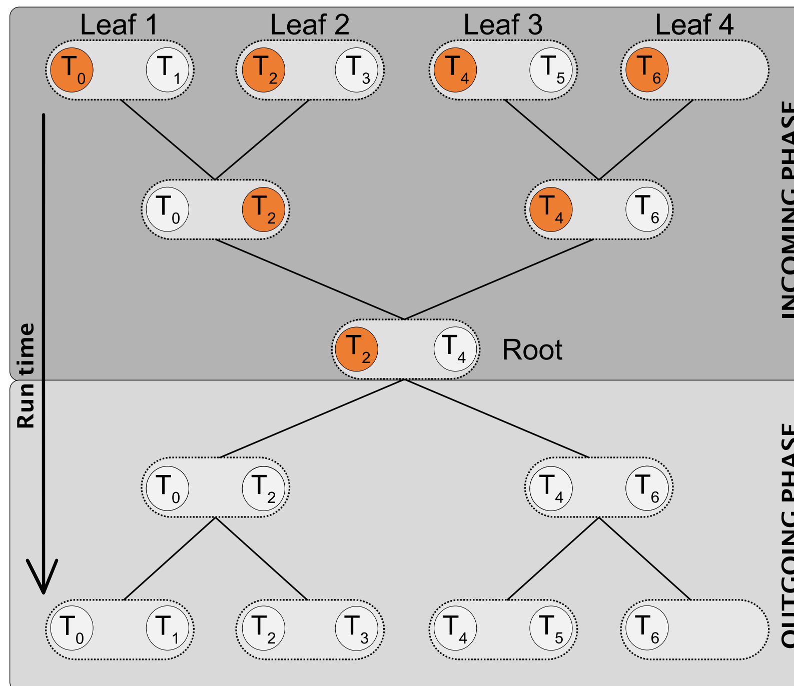


Figure 4.2 – *Exemple de schéma d’échanges de notifications dans une barrière Software Combining Tree avec 7 threads. Le thread colorié est le dernier à atteindre le nœud et celui qui continuera vers le nœud suivant.*

leur partenaire leur notifie la sortie de la barrière. Ainsi, nous avons au plus une notification par nœud. Dans l’exemple illustré par la Figure 4.2, nous avons 7 threads participants à la barrière. Nous avons 7 nœuds dans l’arbre en incluant le nœud racine et les feuilles de l’arbre. Ce qui nous donne au plus 7 notifications dans la phase de sortie de la barrière en incluant la racine. Cependant la feuille 4 ne contient que le thread 6 et donc il n’y aura pas de notification au sein de la feuille. Donc il y a 6 notifications au total. Ce qui correspond au nombre de voisins de chaque thread. Ce raisonnement peut-être généralisé à **P** threads. Dans ce cas, il y aura **P**-1 notifications nécessaire pour sortir de la barrière.

4.2.4 La barrière *Tournament*

La barrière *Tournament* est réalisée dans une structure hiérarchique en arbre binaire comme la barrière *Software Combining Tree*. Donc elle peut s’illustrer de la même manière (Figure 4.2). La différence, ici, est que les deux threads du même nœud vont s’affronter pour déterminer un gagnant et un perdant. Le gagnant continuera dans l’arbre pendant que le perdant attendra la fin de la synchronisation pour sortir de l’arbre. Le thread qui gagnera le tournoi à la racine de l’arbre sera désigné champion du tournoi et il pourra commencer le réveil de l’ensemble des threads en attente-active dans l’arbre. La barrière *Tournament* nécessite $\lceil \log_2(\mathbf{P}) \rceil$ étapes, où **P** est le nombre de threads. Durant la phase d’initialisation, les gagnants, perdants et le champion du tournoi sont choisis. À l’étape r, le thread $i = 2^r$ désigné gagnant va attendre l’arrivée du thread $j = i - 2^r$ perdant du tournoi [42, 58]. Une fois le threads j aura signalé son arrivé, il se mettra en attente-active jusqu’à la fin de synchronisation pendant que le thread i continuera dans l’arbre.

Contrairement à la barrière *Software Combining Tree*, il n'y a pas besoin de compteur pour vérifier l'arrivée des threads dans chaque nœud. Donc il n'y a pas d'utilisation d'opération atomique d'incrémentations ou décrémentation. Ainsi, le coût de la synchronisation sera significativement réduit par rapport à la barrière *Software Combining Tree*. Par ailleurs, comme pour la barrière *Software Combining Tree*, la scalabilité de cette synchronisation est logarithmique. De même, il y aura **P**-1 notifications nécessaire pour sortir de la barrière.

4.2.5 La barrière *Butterfly*

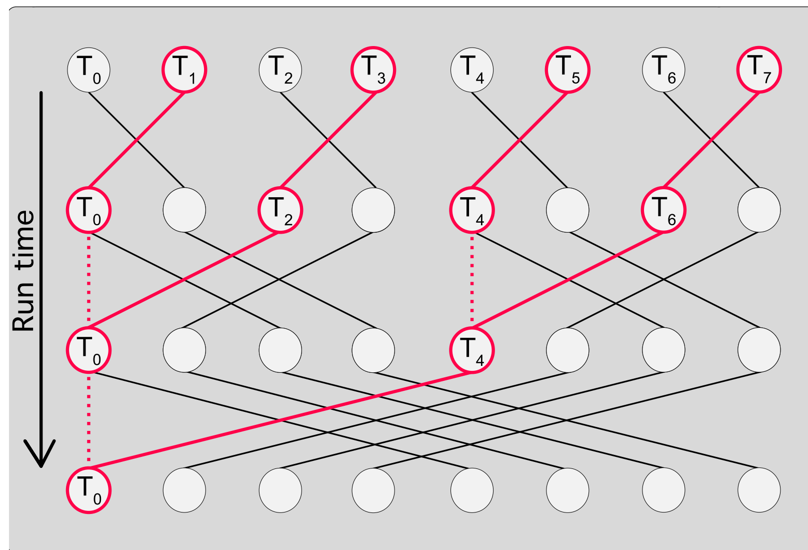


Figure 4.3 – *Exemple de schéma d'échanges de notifications dans la barrière Butterfly avec 8 threads.*

Cette barrière a été largement commentée dans la littérature [12, 44, 77]. La communication entre les threads se fait deux par deux à chaque étape. Le thread i à l'étape r notifie son arrivée au thread $\left((i + 2^{r-1}) \mod 2^r\right) + s$, $i \in [\![0, \mathbf{P}-1]\!]$, $r \in [\![0, \log_2(\mathbf{P})]\!]$ et $s = i - (i \mod 2^{r-1})$ si $i \geqslant 2^r$ ou $s = 0$ sinon. Ce type de barrière ne nécessite que $\log_2(\mathbf{P})$ étapes pour finaliser la synchronisation. Sa principale contrainte est qu'elle n'est efficace qu'avec une puissance de deux threads. Une version pour un nombre quelconque de threads est proposée dans [12]. Pour **P** threads, **P** pas une puissance de deux et si **T** est la prochaine puissance de deux supérieure à P, la méthode consiste à simuler la présence des **T**-**P** threads manquants par les threads participant à la barrière. Cette approche n'est pas efficace pour un grand nombre de threads. Dans la suite du manuscrit, la barrière *Butterfly* ne fera référence qu'à la version avec un nombre de threads égal à une puissance de deux. Comme pour la barrière *Software Combining Tree*, la scalabilité de la barrière *Butterfly* est logarithmique du fait qu'il y a $\log_2(\mathbf{P})$ étapes pour finaliser la synchronisation. Pour ce type de barrière, nous avons une seule phase. Contrairement aux barrières linéaires ou *Software Combining Tree*, il n'y a qu'une seule phase dans cette barrière.Dans l'exemple de la Figure 4.3, nous avons 8 threads participant et nous avons 3 étapes pour finaliser la synchronisation. Suivons les échanges faits par le thread T_0. Remarquons que tous les échanges sont symétriques donc quand T_i attend une notification de T_j, celui-ci attend aussi la notification de T_i pour avancer. À l'étape 0, T_0 attend de recevoir une notification de T_1 pour passer à l'étape suivante. À l'étape 1, T_0 attend la

notification de T_2. Celui-ci ne pourra arriver à l'étape 1 qu'après réception de la notification de T_3. Lorsque T_0 reçoit la notification de T_2, il sait implicitement que T_3 aussi est dans la barrière. En résumé, à cette étape, le thread T_0 a connaissance de l'arrivée de T_1, T_2 et T_3 dans la barrière. À l'étape 2, T_0 attend la notification de T_4. Or, celui-ci ne pourra atteindre cette dernière étape qu'après avoir réception de la notification de T_5 puis de T_6 qui lui même aura eu au préalable la notification de T_7. Finalement, la réception de la notification de T_4 par T_0 implique la présence dans la barrière des threads T_5, T_6 et T_7. Ainsi, pour le thread T_0, il aura reçu à la fin de la barrière les 7 notifications de ses voisins. Le schéma des notifications des threads avec T_0 décrit un arbre binaire avec la racine en T_0 et les feuilles représentées par T_1, T_2, T_3, T_5, T_6 et T_7. Le même raisonnement peut-être appliqué aux autres threads participants. Ici aussi, le nombre de notification reçu pour un thread dans la barrière est de $2^{log_2(\mathbf{P})} - 1 = \mathbf{P} - 1$ pour $\mathbf{P}$ threads.

4.2.6 La barrière de *Dissemination*

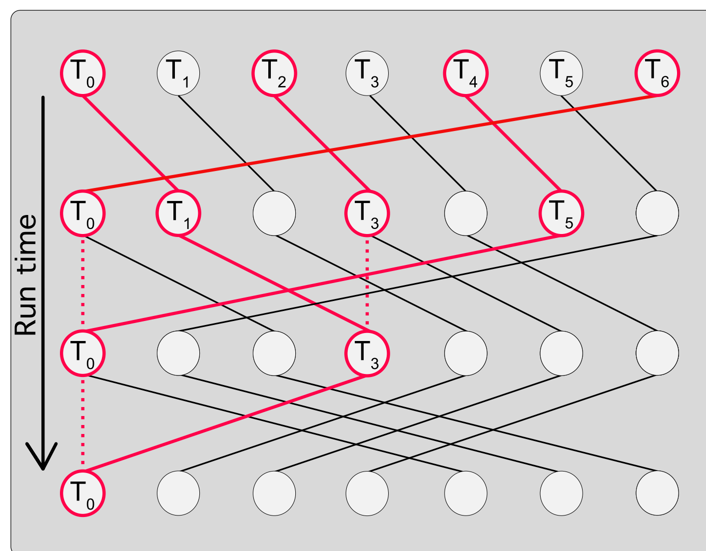


Figure 4.4 – *Exemple de schéma d'échanges de notifications dans la barrière Dissemination avec 7 threads.*

Les threads ont besoin de $\lceil log_2(\mathbf{P}) \rceil$ étapes pour finaliser la synchronisation. À l'étape de barrière r, le thread i notifiera son partenaire de thread $(i + 2^r) \mod \mathbf{P}$ avec $i \in [\![0, \mathbf{P} - 1]\!]$, $r \in [\![0, \lceil log_2(\mathbf{P}) \rceil]\!]$. Elle a été étudiée dans [44, 58, 65]. Cette barrière fonctionne très bien pour tout nombre de threads. Cependant, la barrière *Dissemination* doit créer des chemins de communication redondants si le nombre de threads est non-puissance deux afin de s'assurer que tous les threads ont atteint l'étape r avant de passer à l'étape suivante. Comme la version de la barrière *Butterfly* pour un nombre de threads différent d'une puissance de deux, l'Algorithme *Dissemination* simule la présence des **T**-**P** threads manquants par les threads participant à la barrière. Toutefois, cette barrière est l'une des plus efficaces dans notre état de l'art. Avec un nombre d'étapes de $\lceil log_2(\mathbf{P}) \rceil$, elle a une scalabilité logarithmique comme la barrière *Butterfly*.Pour le nombre de notifications dans la barrière, regardons par l'exemple de la Figure 4.4. Avec 7 threads, il faudra 3 étapes pour finaliser la synchronisation. Comme nous l'avons fait pour l'exemple de la barrière *Butterfly*, suivons les notifications reçues par le thread T_0. À la première étape 0, T_0 attend la notification du T_6. À

l'étape 1, T_0 attend la réception de la notification de T_5 qui lui-même a reçu celle de T_4 à l'étape 0. À l'étape 2, T_0 attend la notification de T_3. Pour que T_3 arrive à l'étape 2, il lui faudra la notification de T_2 à l'étape 0 et de T_1 à l'étape 1 qui lui-même aura reçu au préalable la notification de T_0 à l'étape 0. Donc la notification de T_3 implique l'arrivée dans la barrière de T_0, T_1 et T_2. Au final à cette dernière étape, dès réception de la notification de T_3, T_0 saura que T_0, T_1, T_2, T_3, T_4, T_5 et T_6 sont dans la barrière. Il y a ainsi 7 notifications faites à T_0 alors qu'il n'y a que 6 threads voisins à T_0. Donc il y a une redondance des notifications dans cette barrière le thread 0. Avec la même raison, nous observerons le même résultat pour les autres threads. En général, pour **P** threads, avec **P** est diffèrent d'une puissance de deux, alors le nombre de notifications nécessaire pour finaliser la synchronisation pour un thread sera égale à $2^{\lceil log_2(\mathbf{P}) \rceil} - 1$. Cela signifie que pour un nombre de thread égale à une puissance de 2, le nombre de notifications sera égale à $2^{\lceil log_2(\mathbf{P}) \rceil} - 1 = \mathbf{P} - 1$. Dans ce cas, il n'y aura pas de notifications redondantes dans la barrière.

4.3 La barrière *Extended Butterfly*

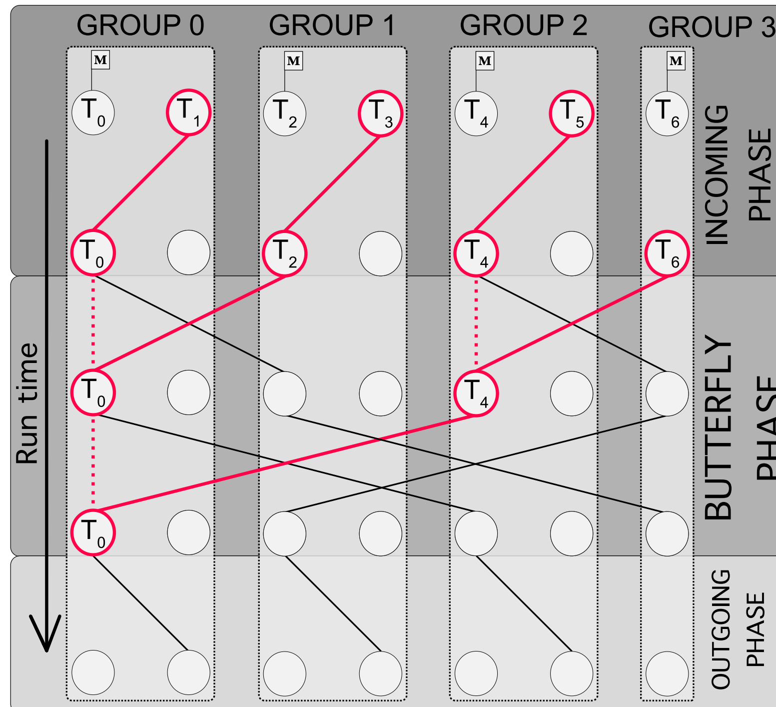


Figure 4.5 – *Exemple de schéma d'échanges de notifications dans la barrière Extended Butterfly avec 7 threads.*

Dans cette partie, nous allons décrire notre barrière. La Figure4.5 montre les schémas de communication de la barrière *Extended Butterfly* pour **P** = 7. Lors de l'initialisation, les threads sont statiquement assignés à un groupe dirigé par un maître. Le thread maître est le thread désigné pour effectuer la phase *Butterfly*. Il est également chargé de réveiller les threads du groupe durant la troisième phase. Donc la synchronisation se passera aux deux niveaux : une synchronisation intra-groupe puis inter-groupe. Pour minimiser le coût de la synchronisation dans les groupes, nous avons décidé de limiter la taille des groupes à au plus 2 threads. Lors de la phase entrante

de la barrière *Extended Butterfly*, le thread maître du groupe va attendre la notification de son partenaire. Une fois cette phase entrante terminée, les maîtres vont alors initier une synchronisation *Butterfly*. Le fait de réunir les threads par groupe, nous laisse le choix du nombre de groupes. La phase 2 de la barrière *Extended Butterfly* étant une synchronisation *Butterfly*, le nombre total de groupes de thread doit être égale à une puissance de deux. Nous avons donc fixé le nombre de groupes égal à la première puissance de deux inférieures à **P** c'est-à-dire $numGroup = 2^{\lfloor \log_2 \mathbf{P} \rfloor}$. Ainsi, la phase 2 de notre barrière nécessitera $\lfloor \log_2 \mathbf{P} \rfloor$ étapes pour être finalisé. À la fin de cette deuxième phase, chaque maître sera en mesure de libérer son partenaire en attente active. En supposant que les phases entrantes et sortantes comptent pour deux étapes dans la barrière, nous aurons exactement $\lfloor \log_2(\mathbf{P}) \rfloor + 2$ étapes dans la barrière. Si le nombre de threads participants est égal à une puissance de deux, la création de groupes est inutile. Dans ce cas, seul une synchronisation de type *Butterfly* sera nécessaire, ce qui nécessitera $\log_2(\mathbf{P})$ étapes. Ainsi, nous aurons une scalabilité logarithmique avec cette barrière. Analysons les échanges de notifications dans la barrière comme nous l'avons fait pour les autres barrières. Nous allons nous intéresser à T_0 pour l'exemple, mais le même raisonnement est applicable aux autres threads. À l'étape 0, dans la phase d'entrée, T_0 étant le maître du groupe, il va attendre la notification de son partenaire T_1. Puis T_0 va entamer la phase de 2 de la barrière et à l'étape 1, il va attendre la notification de T_2. Cette notification implique que T_3 est aussi dans la barrière. À l'étape 2, T_0 attend de recevoir la notification de T_4. Ce qui indiquera à T_0 que T_5 et T_6 sont aussi dans la barrière. À l'issue de cette dernière étape, le thread T_0 aura reçu 6 notifications issues de ses voisins T_1, T_2, T_3, T_4, T_5 et T_6. Nous avons 6 notifications échangées pour 7 threads participants à la barrière. En généralisant ce raisonnement pour **P** threads, il y aura **P**-1 notifications reçu par thread. Contrairement à *Dissemination*, il n'y a pas de notifications redondantes avec cette barrière.

Barrière	Scalabilité	Notification
Sense-reversing Centralized	Linéaire	Non-redondant
Linear Centralized	Linéaire	Non-redondant
Software Combining Tree	Logarithmique	Non-redondant
Tournament	Logarithmique	Non-redondant
Butterfly	Logarithmique	Non-redondant
Dissemination	Logarithmique	Redondant
Extended Butterfly	Logarithmique	Non-Redondant

Table 4.1 – *Résumé des propriétés des barrières*

Nous avons synthétisé les deux propriétés mises en évidence pour chaque barrière dans la Table 4.1. Le type de scalabilité des barrières nous renseigne sur leur comportement lorsque le nombre de threads va croître. Pour rappel, nous cherchons à mettre en place une réduction combinée à une barrière performante pour un grand nombre de threads. Par ailleurs, comme l'opération de réduction est appliquée durant la synchronisation, nous voulons que cette opération soit le moins intrusif possible. Pour cela, nous allons échanger les valeurs à réduire en même temps que la valeur du flag de synchronisation. C'est la raison pour laquelle nous avons cherché à identifier le nombre d'échanges de notifications lors de la synchronisation. Une barrière qui présente des notifications redondantes ne sera pas adapter pour le type de réduction que nous voulons mettre en place. Ce qui écarte d'office la barrière *Dissemination*. Finalement lors de notre état de l'art, la barrière *Butterfly* se présentait comme le meilleur choix pour une réduction combinée. Cependant, pour un nombre de threads différent d'une puissance de deux, elle n'était pas fonctionnelle. Nous

avons ainsi proposé la barrière *Extended Butterfly* pour combler les lacunes de *Butterfly* pour un nombre de threads différent d'une puissance deux.

4.3.1 False sharing

Un phénomène qui doit être pris en compte pour assurer l'efficacité des barrières est le *false sharing* des flags de synchronisation. Ce problème peut se produire lorsqu'une donnée partagée est modifiée et mise à jour fréquemment par plusieurs threads. Pour respecter la propriété de localité spatiale, cette donnée sera chargée sur la même ligne de cache. Le *false sharing* se cache derrière les deux principes de localité des données (spatiale et temporelle).

— La localité spatiale : les données proches de la donnée référencée par le processeur pourront potentiellement être utilisées dans un futur proche. Ils seront donc chargés dans le cache du processeur.

— La localité temporelle : la donnée qui vient d'être référencée par le processeur pourra potentiellement être réutilisée dans un futur proche. Elle sera donc gardée dans le cache du processeur.

Ces données chargées en cache grâce à leur localité se trouveront sur la même ligne de cache. Par exemple, un calcul censé s'exécutait en parallèle va avoir de faibles performances dues au fait que les threads passeront la majorité du temps dans des accès mémoire pour charger leurs lignes de cache modifiée au préalable par un autre thread. Lorsqu'un thread T_1 modifie une donnée D appartenant à un autre thread T_2, alors avant d'utiliser D, T_2 doit mettre à jour toute sa ligne de cache. C'est l'un des principes du protocole MESI (*Modified, Exclusive, Shared, Invalid*) de mise en cohérence du cache [45]. Le protocole MESI (ou une des autres variantes) est un mécanisme de mise en cohérence du cache utilisé par les processeurs multi-cœurs. Il y a quatre états différents caractérisant une ligne de cache avec MESI. La Table 4.2 définit ces différents états et décrit pour chaque état les changements d'état possible.

État	Définition	Changement d'état possible (opérations d'écriture et lecture)
M	*Modified*. La ligne de cache est présent uniquement dans le cache du thread courant mais elle est differente de la mémoire principale.	**M** $\longmapsto$ **M**, **I**, **S**
E	*Exclusive*. La ligne est présente uniquement dans le cache du thread courant et elle est cohérente avec la mémoire principale	**E** $\longmapsto$ **M**, **I**
S	*Shared*. La ligne est partagée avec d'autre threads et elle est cohérente avec la mémoire principale.	**S** $\longmapsto$ **I**, **E**
I	*Invalid*. La ligne de cache n'est plus valide dans aucun cache (L1 ou L2).	**I** $\longmapsto$ **S**, **E**

Table 4.2 – *Description des états du protocole MESI*

Finalement, si l'ensemble des flags de synchronisation sont contiguës en mémoire. Le principe de localité spatiale des données, nous indique que certaines flags peuvent être chargé sur des lignes de caches d'autres threads. Par conséquent, certains threads vont voir leur ligne de cache invalidée

par d'autres simplement parce qu'ils possèdent le flag de synchronisation d'un autre thread à côté de leur propre flag de synchronisation. Ils peuvent passer la majorité de leur temps à charger leur ligne de cache depuis la mémoire avant de pouvoir notifier un partenaire de leur arrivée. C'est le phénomène de *false sharing*. Pour éviter ce *false sharing*, plusieurs techniques existent. En ce qui concerne les variables de synchronisation, elles doivent être alignées sur une frontière d'une ligne de cache. Ce qui signifie qu'il faut les espacer d'une longueur de ligne de cache (64 octets). Cela garantit que deux variables de synchronisation ne partageront pas la même ligne de cache. En revanche, sur la ligne de cache où est stocké la valeur de la variable de synchronisation, il y aura un remplissage de zéro qui sera inutilisé, Figure 4.6. C'est cet espace inutilisé (ou *padding*) que nous allons exploiter lors de la réduction.

State	Adress	Data cache							
...	...	...	...	...	...	...	...	...	...
E	0x...	flag sync	...	...	...	...	...	...	...
...	...	...	...	...	...	...	...	...	...

Figure 4.6 – *Exemple de ligne de cache avec une charge utile non utilisée.*

4.4 Réduction combinée avec une barrière

Algorithm 13 Réduction centralisée

Require: Tableau global : data_thread, global_red ;
Require: Valeur privée : val_red ;
1: data_thread[tid]=val_red ;
2: barrière() ;
3: **if** tid==0 **then**
4: global_red=0 ;
5: **for** i=1 ; i<NTHREAD ; ++i **do**
6: global_red +=data_thread[i] ;
7: **end for**
8: **end if**
9: barrière() ;

Une opération de réduction consiste à appliquer une opération sur la valeur privée de chaque thread pour obtenir un résultat qui sera connu de tous. Prenons le cas d'une opération de réduction avec une somme, comme le montre l'Algorithme 13. Chaque thread stocke sa valeur dans le vecteur `data_thread` et attend que l'ensemble du groupe ait terminé cette étape. Ensuite, le thread maître effectue la somme sur les valeurs du tableau et le résultat est stocké dans la variable globale `global_red`. Cette réduction naïve nécessite deux barrières de synchronisation. Notez que l'Algorithme 13 peut être amélioré si chaque thread réduit les valeurs dans une variable locale, puis le thread maître met à jour la variable globale. Mais ceci va probablement générer un point chaud lorsque chacun souhaitera charger les valeurs stockées dans le vecteur globales.

Cependant, cela nous permet d'éliminer la deuxième barrière (ligne 13) car tous les threads auront le résultat de l'opération de réduction dans une variable locale s'ils ont besoin de l'utiliser immédiatement, sinon le résultat sera stocké dans la variable globale par le thread maître pour une utilisation ultérieure. Dans l'approche que nous avons décidé d'adopter dans ce travail, l'opération de réduction combinée à une barrière est effectuée lorsque les threads traversent la barrière. L'avantage est qu'une seule barrière est nécessaire pour synchroniser les threads et réduire les valeurs. Aussi en fonction de la méthode utilisée cela permet de réduire les point chaud mémoire. Par exemple, [14] utilise la barrière *Software Combining Tree* pour calculer la réduction, et les auteurs de [74] utilisent la barrière *Tournament*. Cependant, un point sur lequel il nous faut être attentif pendant la phase de synchronisation des threads est la communication entre ces derniers et plus précisément le partage de variables. Nous parlons ici de communication, car les threads vont échanger, en plus de la notification de synchronisation, les valeurs à réduire.

4.4.1 *Padding* des flags de synchronisation

Le *padding* des flags de synchronisation désigne l'espace inutilisé de la ligne de cache du flag de synchronisation. Cette charge utile entraîne une légère pénalité de mémoire, car la majeure partie de la ligne de cache ne sera pas utilisée (voir la Figure 4.6) mais cette pénalité est négligeable face à la pénalité induite par le *false sharing*. L'un des principaux objectifs de ce travail est d'utiliser cet espace disponible sur les lignes de cache pour stocker la réduction partielle à chaque étape de la synchronisation. La Figure 4.7 illustre cette idée de transport des valeurs de réduction et du flag de synchronisation sur la même ligne de cache.

State	Adress	Data cache							
...	...	...	...	...	...	...	...	...	...
E	0x...	flag sync	red val 1	red val 2	red val 3	red val 4	red val 5	red val 6	red val 7
...	...	...	...	...	...	...	...	...	...

Figure 4.7 – *Exemple de ligne de cache stockant le flag de synchronisation et des valeurs de réduction (7 valeurs en double précision).*

Dans la littérature, seul [74] propose cette approche : transporter les valeurs de réduction sur la même ligne de cache que le flag de synchronisation. Leur méthode consiste à utiliser un "conteneur" dans chaque nœud de la barrière *Tournament* pour stocker à la fois le flag et les valeurs à réduire. Comme la taille du conteneur correspond exactement à celle d'une ligne de cache, le flag occupe un bit et l'espace restant est utilisé pour les valeurs de réduction. Lorsque le flag et les valeurs de réduction correspondent à la taille du conteneur, le chemin dit "rapide" est choisi. Si la taille du type de données de réduction dépasse la capacité du conteneur, une nouvelle variable sera créée pour stocker les valeurs de réduction. Cette voie est appelée "lente". Cependant, le problème de cette méthode est que si le flag et la valeur de réduction partielle dépassent la taille du conteneur de 2 bits, alors la solution trouvée pour utiliser la "voie rapide" est d'exclure les 2 bits d'exposant de la valeur de réduction partielle. Une fonction d'empaquetage vérifie s'il est possible de négliger les bits d'exposant et d'utiliser ensuite le "chemin rapide" sinon, le "chemin lent" sera utilisé pour réduire les valeurs. La différence entre le chemin rapide et le chemin lent est le nombre barrière mémoire. En effet, dans le cas du chemin lent, la valeur de réduction est stockée

dans un conteneur auxiliaire avant de mettre à jour à flag de synchronisation. Pour s'assurer la valeur de réduction soit bien mise à jour avant le flag de synchronisation, il faut recourir à une barrière mémoire entre ces deux étapes. Donc il y aura besoin d'autant de barrières mémoire qu'il y aura de nœud dans la barrière. De plus, le type du "conteneur" est un entier de 64 bits. Les auteurs ont une contrainte assez forte à savoir compacter la valeur de réduction et flag sur 64 bits. Pour plus de détails, nous redirigeons le lecteur vers [74]. Notre approche se propose d'être moins contraignante. Nous voulons travailler sur toute la ligne de cache et avoir la possibilité de manipuler plusieurs valeurs réelles en double précision. Pour cela, nous nous appuyons sur les opérations de lecture/écriture SIMD pour échanger toutes les informations liées à la réduction (flag de synchronisation et valeurs de réduction).

1. Pendant l'opération de réduction, nous n'utilisons pas de variable globale pour partager les valeurs réduites.
2. Réduction de plusieurs valeurs jusqu'à 7 double.
3. Mises à jour des valeurs au fur et à mesure de la progression de la synchronisation.
4. Manipulation de toute la ligne de cache à l'aide d'instructions SIMD.

De plus, nous profitons du protocole de cohérence du cache MESI. En théorie, lorsque les threads communiquent dans la barrière, ils peuvent échanger des flags de synchronisation et des valeurs de réduction sans coût supplémentaire. Mais il faut garantir que la lecture/écriture de l'ensemble de la ligne de cache soient effectuées de manière atomique.

4.4.2 Écriture SIMD atomique

Dans les rapports d'Intel et d'AMD sur l'architecture de leurs processeurs respectifs, l'atomicité des instructions SIMD n'est pas claire. Dans le rapport technique d'AMD [6], il est expliqué que les instructions de *load/store* de plus de 8 bytes ne sont pas garanties comme étant atomiques. De même, le rapport technique d'Intel [7] informe les utilisateurs qu'à l'exception de la liste d'instructions détaillées dans le [46], "AVX and FMA instructions do not introduce any new guaranteed atomic memory operations". Pour garantir que les threads puissent voir le changement du flag et la mise à jour des valeurs à réduire, nous avons besoin d'opérations d'écriture SIMD atomiques. Par atomique, nous entendons qu'une écriture SIMD est effectuée en une seule étape non interrompue. Ainsi, les threads qui remarquent un changement dans le vecteur ont la garantie que le vecteur entier est aussi mise à jour. Cette atomicité n'est généralement pas garantie au-delà de 64 bits d'opérations d'écriture sur les processeurs x86. Néanmoins, nous avons mis en place un test simple pour vérifier si nous avons un comportement atomique pour les écritures SIMD. Ce test consiste en un thread maître et en un thread partenaire. Le thread maître effectuant une écriture SIMD avec un flag de synchronisation au début et quelques valeurs spécifiques. Le thread partenaire attend le changement d'état du flag et vérifie ensuite si les valeurs spécifiques du vecteur sont correctes. Si le partenaire ne voit pas les bonnes valeurs dans le vecteur après le changement d'état du flag, le test est marqué comme échoué. Les threads sont liés à deux cœurs différents du même socket, et le test est effectué plusieurs fois pour différentes lignes de cache. Nous montrons dans le Tableau 4.3 le taux d'échec pour 10^6 tests sur 1000 lignes de cache sur différents processeurs x86. Nous voyons que tous les processeurs semblent avoir des écritures SIMD atomiques au moins jusqu'à 128 bits. Le processeur Ivy Bridge n'a pas d'écritures SIMD

atomiques sur 256 bits, contrairement aux autres processeurs. Les processeurs Skylake-X et KNL semblent avoir une atomicité d'écriture SIMD jusqu'à 512 bits, ce qui correspond à une ligne de cache entière.

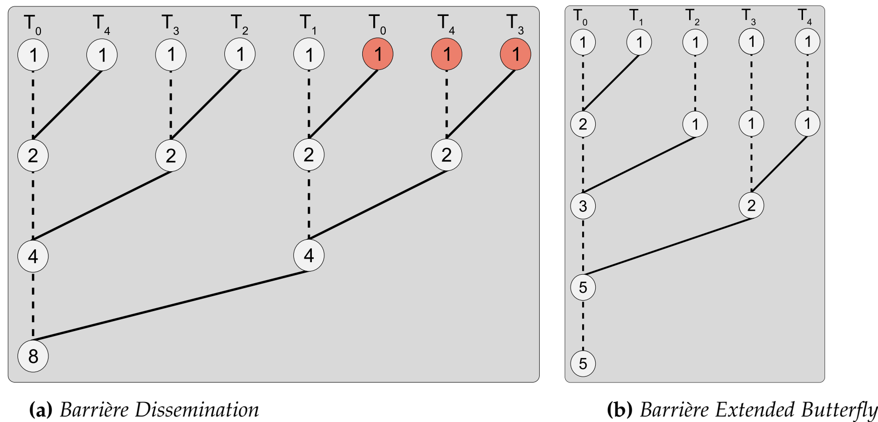
Taille des données	128 bits	256 bits	512 bits
Ivy-Bridge (E5-2680 v2)	0.0	3E-9	Non pris en charge
Rome (Epyc 7H12)	0.0	0.0	Non pris en charge
Milan (Epyc 7763)	0.0	0.0	Non pris en charge
SkyLake-X (Xeon Gold 6140)	0.0	0.0	0.0
KNL (Xeon Phi 7290)	0.0	0.0	0.0

Table 4.3 – *Le taux d'échec pour million de tests sur 1000 lignes de cache entre 2 threads sur 2 cœurs d'un même socket.*

Ces résultats montrent un niveau d'atomicité des instructions SIMD de *load/store* qui n'est pas explicitement garanti dans la documentation des fabricants, par exemple sur AMD MILAN ou Intel SKL. Le niveau d'atomicité limite le nombre de valeurs de réduction que nous pouvons transporter en même temps avec le flag de synchronisation. Sur Ivy-Brige, nous pouvons gérer jusqu'à 3 floats (3×32 bits + flag) alors que nous ne pouvons transporter que 1 double (1×64 bits + flag). Ces observations sont similaires à celles obtenues par [64].

4.4.3 Effet des communications redondantes lors de la réduction

Nous avons introduit dans les Sections 4.3 et 4.2 la propriété de non-redondance. Prenons un exemple (Figure 4.8) pour montrer pourquoi les communications redondantes sont un problème lorsqu'une opération de réduction et une barrière sont utilisées ensemble.



(a) *Barrière Dissemination* **(b)** *Barrière Extended Butterfly*

Figure 4.8 – *Exemple de communication lors d'une réduction avec une somme impliquant 5 threads. On représente les ancêtres de T_0 dans la barrière Dissemination (4.8a) et la barrière Extended Butterfly (4.8b). Les lignes pleines indiquent la communication entre les threads partenaires et les lignes pointillées symbolisent l'attente des threads.*

Dans la Figure 4.8, nous avons représenté les communications avec T_0 par un arbre binaire. Ces illustrations s'obtiennent en remontant les échanges à l'envers (de la dernière étape vers la première). Les feuilles de l'arbre représentent l'ensemble des threads qui ont contribué aux

échanges. Ils sont désignés comme les ancêtres du thread T_0 pour cet exemple. En raison de la redondance des communications, la valeur de T_0 à la fin de l'opération de réduction combinée à la barrière *Dissemination* est incorrecte (Figure 4.8a). Le résultat de l'opération de réduction devrait être égale à 5 (chaque thread ayant une valeur initiale de 1). Or, le résultat est 8 en raison du phénomène de redondance des communications. La Figure 4.8a montre que T_0, T_3 et T_4 apparaissent deux fois parmi les ancêtres de T_0. Cela ne respecte pas la définition de la non-redondance des communications entre les threads introduite précédemment. En revanche, les ancêtres de T_0 provenant de la barrière *Extended Butterfly* (4.8b) correspondent à l'ensemble des threads participant à la barrière. La propriété de non-redondance est obligatoire pour une opération de réduction combinée à une barrière. Cependant, pour des opérations telles que min/max, cette propriété n'est pas nécessaire. En effet, la recherche de max ou min ne tient pas compte de la fréquence à laquelle la valeur du thread est rencontrée lors de la synchronisation. Ainsi, une opération de réduction combinée à la barrière de *Dissemination* pour la recherche de min/max est possible. Néanmoins, notre objectif est de proposer une opération de réduction qui puisse fonctionner avec toutes les opérations supportées par la réduction OpenMP.

4.4.4 Mise en œuvre de la réduction *Extended Butterfly*

Algorithm 14 Réduction Extended Butterfly avec une somme pour AVX512.

```
1: Type enum {WAIT,GO}
2: Type union {
3: volatile avx512dVec simdVec
4: volatile double val[8]
5: volatile int64 flag[8]
6: } CacheLine byte_aligned(64)
7:
8: Type union {
9: avx512dVec simdVec
10: double val[8]
11: int64 flag[8]
12: } NvCacheLine
13:
14: Type struct {
15: Boolean isMaster ;
16: int idGroup                                          ▷ Identifiant du groupe
17: int numGroup                                         ▷ Nombre total de groupes
18: int sizeGroupe
19: CacheLine flag[2] [⌊log2(P)⌋]
20: } GroupThread
21:
22: procedure REDUCE_EXTENDED_BUTTERFLY(N,val)
23:      Inputs : Vecteur des valeurs à réduire : val ; Entier : N ▷ Nombre de valeurs à réduire (au
     plus égal à 7 en double précision)
24:      Inputs : Vecteur des valeurs à réduire : val
25:     // Phase d'initialisation
26:     GroupThread groupOfThread // Vecteur de taille égale à P
27:     groupOfThread[ :].flag[0][ :].flag[0] = 1
28:     groupOfThread[ :].flag[1][ :].flag[0] = 0
29:     CacheLine flag // Vecteur de taille égale à P
30:     flag[ :].flag[0] = GO
31:     int nStep = log2(groupOfThread[tid].numGroup)    ▷ Nombre d'étapes pour la barrière
     Butterfly
32:     thread private : int sense = WAIT
33:     thread private : int parity = 1
34:     thread private : int tid = omp_get_thread_num()
35:
36:     // Phase entrante
37:     call REDUCE_LINEAR_CENTRALIZED(N,val,tid)
38:     if groupOfThread[tid].isMaster then
39:         call REDUCE_BUTTERFLY(N,val,tid,sense)
40:     end if
41:     // Phase sortante
42:     call BCAST_LINEAR_CENTRALIZED(N,val,tid)
43:     if parity == 1 then                 ▷ Préparer la barrière pour la prochaine utilisation
44:         sense = !sense
45:     end if
46:     parity = 1-parity
47: end procedure
```

L'Algorithme 14 décrit chaque étape de la réduction avec une somme en AVX512. Les fonctions appelées dans chaque phase de la réduction *Extended Butterfly* sont détaillées en Annexe A.2. Pour un nombre de threads égale à une puissance de 2, tous les threads effectuent une opération de réduction combinée avec la barrière *Butterfly* (Algorithme 18) automatiquement. Dans le cas où l'AVX512 serait activé sur un processeur x86, nous stockons le flag et les valeurs de réduction dans un conteneur de 512 bits, puis nous utilisons l'instruction *_mm512_store* pour écrire de manière atomique le conteneur entier dans le vecteur du thread partenaire actuel. Pour l'AVX2 ou non-AVX, le vecteur du thread partenaire sera mis à jour en 2 paquets de 256 bits ou 4 paquets de 128 bits. En commençant par stocker les paquets contenant les valeurs de réduction, puis en terminant par le paquet contenant le flag de synchronisation, le thread partenaire verra le changement de son flag de synchronisation et quittera son mode d'attente actif avec les valeurs de réduction mises à jour.

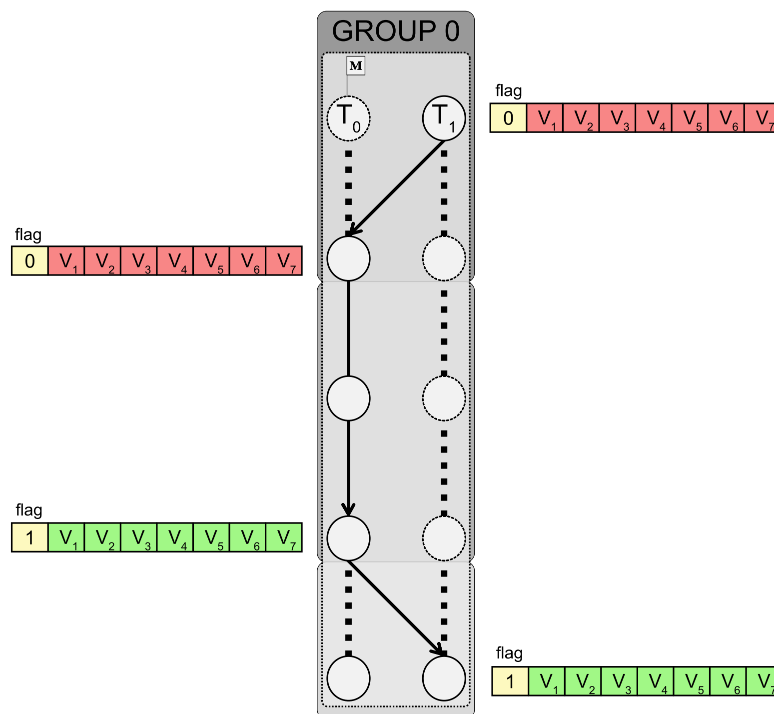


Figure 4.9 – *Exemple de communications entre le thread T_0 et T_1 du groupe 0 de la réduction Extended Butterfly*

L'exemple de la Figure 4.9 illustre les communications au sein du groupe 0 de la réduction *Extended Butterfly*.

1. *Incomming phase* : T_1 envoie le conteneur avec les valeurs de réduction et le flag de synchronisation à son partenaire T_0.

   Si l'AVX512 est activé, alors l'instruction *_mm512_store* est utilisée.

   Sinon, si l'AVX2 est activé, alors l'instruction *_mm256_store* pour envoyer deux paquets de 256 bits, et le dernier paquet contiendra le flag de synchronisation.

   Si l'AVX n'est pas activé, l'instruction *_mm128_store* est utilisée pour envoyer quatre paquets de 128 bits, et le dernier paquet contiendra le flag de synchronisation.

2. *Butterfly phase* : T_0 effectue une réduction *Butterfly* avec les autres threads maîtres pour compléter le résultat.
3. *Outgoing phase* : T_0 envoie de le résultat de la réduction à T_1 avec la notification qui lui permettra de sortir de son attente-active.

Dans la section suivante, nous allons comparer les performances des algorithmes de réduction à la réduction de OpenMP 4.5.

4.5 Résultats expérimentaux

Dans cette section, nous évaluons le comportement de nos algorithmes. Nous présentons et discutons des résultats des opérations de réduction. Nous évaluons d'abord nos méthodes à l'aide de tests synthétiques où nous mesurons le nombre de cycles d'horloge moyen passé dans une opération de barrière et de réduction. Dans la deuxième partie, nous comparons les surcoûts de la réduction *Extended Butterfly* par rapport à la réduction d'OpenMP en utilisant la suite de micro-benchmarks EPCC [13].

4.5.1 Environnement test

Supercomputer	IFPEN (ENER)	IFPEN	TGCC * (TOPAZE)
Processor type	Intel Xeon Gold 6140 (SKL)	Intel Xeon Phi 7290 (KNL)	AMD EPYC Milan 7763 (MILAN)
# processor / node	2	1	2
# core / processor	18	72	64
Freq (GHz)	2.6	1.5	2.45
Compiler	Intel 2021.2	Intel 2021.2	GCC 11.1
OpenMP version	4.5	4.5	4.5

Table 4.4 – *Les caractéristiques des processeurs.*

Les tests ont été effectués sur trois processeurs différents : Intel Xeon Gold 6140 (SKL), Intel Xeon Phi 7290 (KNL) et AMD EPYC Milan 7763 (MILAN). Le Tableau 4.4 détaille quelques caractéristiques des processeurs. Les codes ont été compilés avec GCC 11.1 et Intel 21.4 sauf sur le processeur MILAN sur lequel seul GCC 11.1 a été utilisé. Le but est d'observer le comportement de notre Réduction Extended Butterfly par rapport aux autres réductions de l'état de l'art et sur des processeurs embarquant un grand nombre de cœurs physiques. Pour les expériences, l'hyperthreading n'a pas été activé et les tests ont été effectués sur un seul socket du nœud de calcul. Le processeur KNL est configuré en mode **All2All** et **flat memory**. Concernant le l'environnement d'exécution des binaires du compilateur GCC 11.1, la variable d'environnement `OMP_PROC_BIND` est fixée à `True` et la `OMP_WAIT_POLICY` est fixée à `ACTIVE`. Pour les binaires du compilateur Intel 21.4, la variable `KMP_AFFINITY` a été définie sur `compact` avec `granularity=fine` et les variables `KMP_PLAIN_*_PATTERN` ont été définies sur `hyper`. La configuration `hyper` d'Intel fournit les meilleures performances pour la réduction et la barrière OpenMP d'Intel.

4.5.2 Résultats sur les barrières et réductions

Dans cette section, nous présentons les résultats du test synthétique. L'Algorithme 15 décrit comment calculer les surcoûts des opérations de barrières et de réduction. Les fonctions sont appelées un nombre prédéfini de fois dans une boucle, et le nombre global de cycles est mesuré. Ensuite, une moyenne est calculée.

Algorithm 15 Test synthétique pour les opérations de réductions

```
1:  NTEST                                                    ▷ Fixer le nombre de tests
2:  start = getclock();
3:  #pragma omp parallel
4:  {
5:  for (j = 0; j < NTEST; j=j+1) do
6:      #pragma omp for reduction(+ :val_red) schedule(static,1)
7:      for (i = 0; i < nthreads; i=i+1) do
8:          val_red += 1;
9:      end for
10: end for
11: }
12: times = (getclock() - start);                            ▷ Calculer le nombre de cycles
13: times /= (double) NTEST;
```

Chaque algorithme (barrière ou réduction) est appelé 10^4 fois pour calculer le nombre moyen de cycles par rapport au nombre de tests (`NTEST = ` 10^4).

Barrières de synchronisation

Les Figures 4.10 et 4.11 comparent les différentes barrières de synchronisation introduites précédemment sur AMD MILAN et Intel KNL. Plus les valeurs sont faibles, plus les performances sont bonnes. Nous pouvons constater qu'en fonction des processeurs les barrière *Software Combining Tree* (en violet) et *Linear Centralized* (en vert) ont des comportements significativement différents. La croissance linéaire de la barrière *Linear Centralized* est plus forte sur KNL que sur MILAN alors que la croissance logarithmique de la barrière *Software Combining Tree* est plus stable sur KNL que sur MILAN. Néanmoins, deux barrières se distinguent : la barrière de *Dissemination* (en marron)et la barrière *Extended Butterfly* (en bleu).

La première observation que nous pouvons faire est que sur MILAN avec le compilateur GCC 11.1, la barrière *Extended Butterfly* est largement plus performante que la barrière par défaut d'OpenMP 4.5. Celle-ci semble être basée sur un modèle de barrière centralisée. Ce type de barrière a une croissance linéaire par rapport au nombre de threads qui n'est pas idéal pour un grand nombre de threads. Cependant, la barrière *Linear Centralized* a une croissance linéaire très faible par rapport à la barrière *Sense-reversing Centralized* et la barrière d'OpenMP 4.5. Mais l'écart finit par se creuser avec les barrières *Extended Butterfly* et *Dissemination*. La barrière *Extended Butterfly* est celle qui démontre les meilleures performances.

La Figure 4.11 nous montre un autre comportement de la barrière *Linear Centralized* sur KNL. Elle a une forte croissance avec ce type d'architecture la classant comme la moins bonne barrière sur KNL.

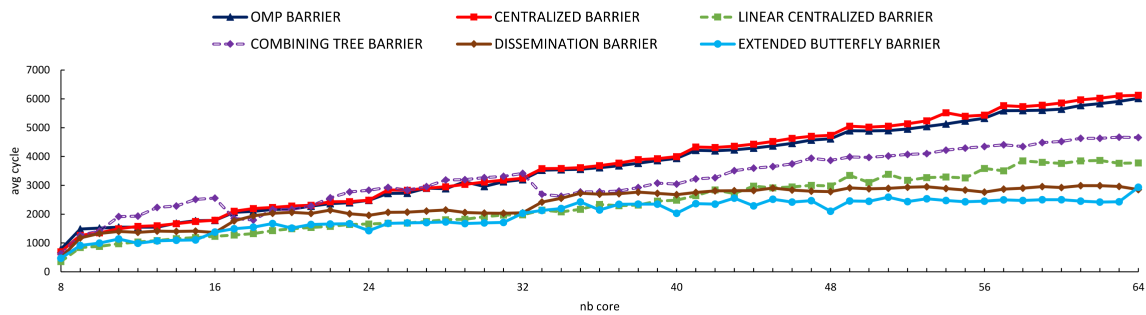


Figure 4.10 – *Résultat comparatif des barrières avec le test synthétique utilisant le compilateur GCC 11.1 sur le processeur* MILAN.

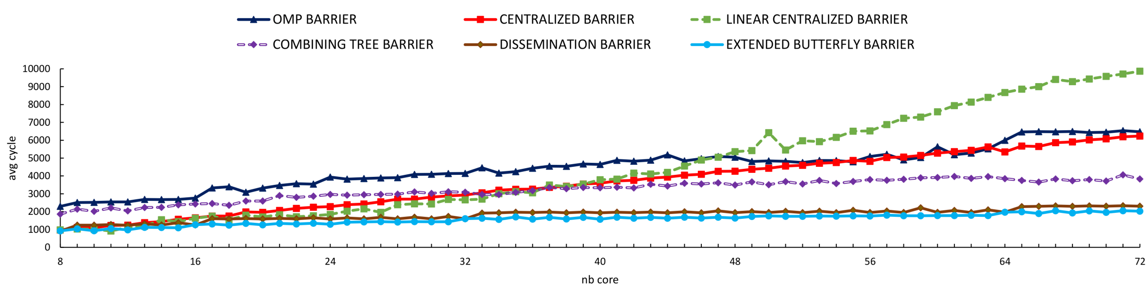


Figure 4.11 – *Résultat comparatif des barrières avec le test synthétique utilisant le compilateur Intel 21.4 sur le processeur* KNL.

Alors que la barrière d'OpenMP avec Intel 21.4 a une meilleur performance que la barrière *Linear Centralized* avec une croissance logarithmique la rendant plus intéressante sur KNL. Cependant, les barrières *Extended Butterfly* et *Dissemination* dominent totalement les autres barrières et restent en dessous des 2500 cycles en moyenne pour un nombre de threads au delà des 60 threads (voir Figures 4.10, 4.11). Lorsque le nombre de threads est égale à une puissance de deux, la barrière *Extended Butterfly* est équivalente à la barrière de *Dissemination*. Comme nous l'avons expliqué dans la Section 4.2 et 4.3, la barrière de *Dissemination* est équivalente à une barrière *Butterfly* lorsque le nombre de threads participant à la barrière est une puissance de deux. Et notre barrière *Extended Butterfly* devient une barrière *Butterfly* pour un nombre de threads égale une puissance de deux (voir Section 4.3). Ce qui explique cette égalité parfaite à chaque nombre de threads égale à une puissance de deux. De plus, on observe dans les Figures 4.10 et 4.11, qu'entre deux puissances de deux, notre barrière est légèrement plus rapide que la barrière de *Dissemination*. Elle a une croissance logarithmique par rapport au nombre de threads avec un coût moyen entre deux puissances de deux qui est légèrement plus faible que la barrière de *Dissemination*.
Dans la Section 4.4.3, nous avons vu que la barrière de *Dissemination* ne peut pas être utilisée pour effectuer une opération de réduction avec la méthode proposée ici. En effet, les communications redondantes de certains threads à certaines étapes vont fausser le résultat final, et il est assez difficile d'identifier ces communications a priori pour espérer corriger le résultat final de la réduction. Donc la barrière *Dissemination* ne sera pas combinée à une réduction. Cependant, nous avons implémenté une réduction combinée avec la barrière *Linear Centralized*. Pour la suite, des tests comparatif seront faits uniquement pour la réduction d'OpenMP 4.5, les réductions combinées à la barrière *Linear Centralized* et *Extended Butterfly*.

Réductions d'une valeur en double précision

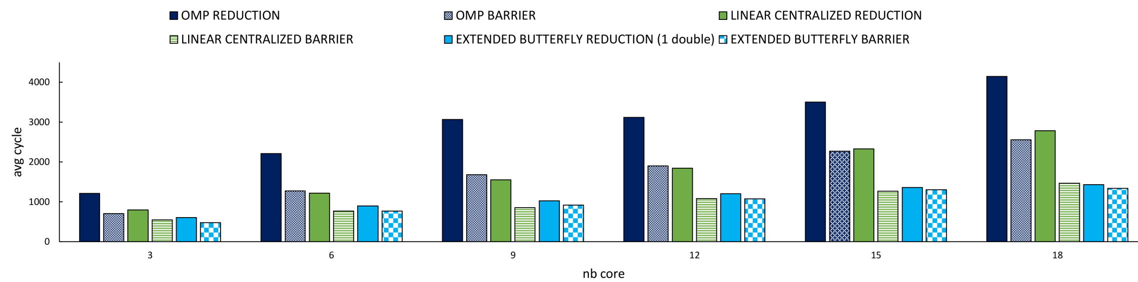


Figure 4.12 – *Résultat comparatif du test synthétique des réductions avec le compilateur Intel 21.4 sur le processeur* SKL. *La partie hachurée représente la consommation de la barrière et la partie foncée est le surcoût de l'opération de réduction.*

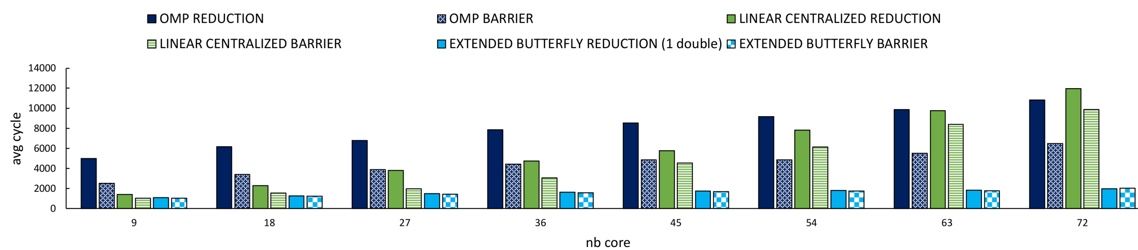


Figure 4.13 – *Résultat comparatif du test synthétique des réductions avec le compilateur Intel 21.4 sur le processeur* KNL. *La partie hachurée représente la consommation de la barrière et la partie foncée est le surcoût de l'opération de réduction.*

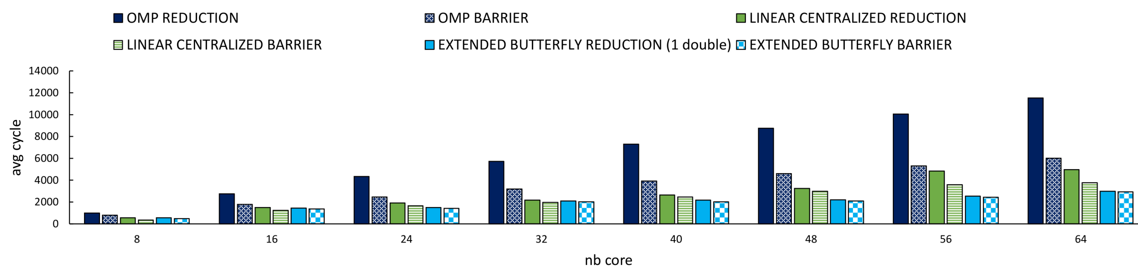


Figure 4.14 – *Résultat comparatif du test synthétique des réductions avec le compilateur GCC 11.1 sur le processeur* MILAN. *La partie hachurée représente la consommation de la barrière et la partie foncée est le surcoût de l'opération de réduction.*

Les Figures (4.12,4.13,4.14) représentent le nombre de cycles moyens des barrières et réductions d'OpenMP d'Intel 21.4 et GCC 11.1, *Linear Centralized* et *Extended Butterfly*. Les histogrammes hachurés qui correspondent à la consommation des barrières et les histogrammes foncées correspondent à la consommation des réductions. La première remarque est la différence significative entre le coût de la barrière et le coût de la réduction pour OpenMP sur les trois processeurs. Cette différence peut venir de différents facteurs, notamment la manière de faire la réduction. Pour la barrière d'OpenMP avec GCC 11.1, nous avons vu dans la Figure 4.10 qu'elle semble être basée sur une barrière centralisée. Si la réduction d'OpenMP est combinée à cette barrière, cela explique

en partie la croissance linéaire observée dans la Figure 4.14. Néanmoins, pour les tests avec le compilateur Intel 21.4, nous avons configuré les variables d'environnement `KMP_PLAIN_*_PATTERN` à `hyper`. Ce qui active leur barrière de type hiérarchique qui a une croissance logarithmique comme nous pouvons le voir dans la Figure 4.13. Leur réduction bénéficie du même comportement que la barrière et finit (au-delà de 60 threads) par être plus rapide que la réduction *Linear Centralized* sur KNL. La seconde remarque concerne la réduction *Linear Centralized*. Comme elle est combinée à la barrière *Linear Centralized*, nous nous attendons à ce qu'elle laisse transparaître le caractère linéaire de sa barrière. Cependant, il y a une très forte différence entre le coût de la barrière et le coût de la réduction. Sur SKL, la barrière *Linear Centralized* a un coût compétitif à la barrière *Extended Butterfly* (Histogramme hachuré de la Figure 4.12). En revanche, le surcoût de la réduction par rapport à la barrière est relativement important à mesure que le nombre de threads augmente. Cela vient en partie de la manière de faire la réduction *Linear Centralized*. En effet, dans *Linear Centralized*, l'opération de réduction est gérée uniquement par le thread maître. Donc plus le nombre de threads augmente et plus l'opération de réduction prendra du temps (Histogrammes foncés des Figures 4.13, 4.14). Cette manière de faire pénalise grandement la réduction *Linear Centralized* en plus du coût linéaire de sa barrière. Sur KNL, la réduction *Linear Centralized* devient plus lente que la réduction d'OpenMP (au-delà de 60 threads). La troisième remarque concerne la réduction *Extended Butterfly*. Elle est significativement plus rapide que les autres réductions et sur les trois processeurs. En revanche, pour un nombre moyen de threads (moins d'une trentaine), la barrière *Extended Butterfly* ne montre pas de gain significatif par rapport à la barrière *Linear Centralized* et ceux sur les trois processeurs. Cependant, les différences se creusent pour un grand nombre de threads. La barrière *Extended Butterfly* est nettement plus rapide que la barrière *Linear Centralized*. Pour la réduction, la différence est encore plus marquée dû à la manière de réduire les valeurs. Dans la réduction *Extended Butterfly*, les opérations sont distribuées dans les groupes alors qu'elles sont séquentielles pour la réduction *Linear Centralized*. C'est la raison pour laquelle le surcoût de l'opération de réduction est significativement réduit par rapport au coût de sa barrière. Au final, la réduction *Extended Butterfly* met en moyenne moins de 2500 cycles pour réduire une valeur en double précision avec 64 ou 72 threads pour KNL (Histogramme foncé de la Figure 4.13) ou MILAN (Histogramme foncé de la Figure 4.14).

Réduction de plusieurs valeurs en double précision

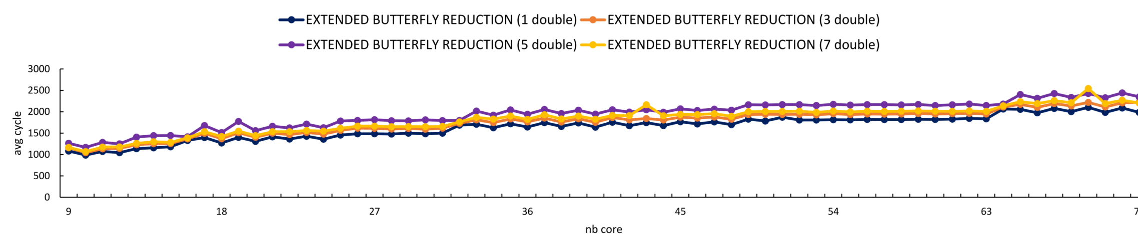


Figure 4.15 – *Coût moyen de la réductions de 1, 3 et 7 valeurs sur le processeur* KNL *avec le compilateur Intel 21.4.*

De la même manière que pour la mesure du coût de la réduction avec une valeur, nous allons nous intéresser maintenant au coût de réduction de plusieurs valeurs avec la Réduction Extended Butterfly. Dans ce cas, nous pouvons réduire jusqu'à 7 valeurs en double précision (voir Figure 4.7). Le nombre de cycles moyen consommé pour 1, 3 et 7 doubles est indiqué sur la Figure 4.15. Un

coût supplémentaire mineur apparaît entre AVX, AVX2 et AVX512 (voir Figure 4.15). Cependant, le coût de la réduction reste inférieur à 2500 cycles en moyenne sur le processeur KNL, ce qui est 5,5 fois moins que le coût de la réduction d'OpenMP avec Intel 21.4 (voir la Figure 4.13). Pour la suite, nous avons décidé de tester nos fonctions sur le microbenchmark EPCC.

4.5.3 Benchmark EPCC

L'idée du benchmark EPCC est d'exécuter en boucles, un grand nombre de fois, les directives OpenMP 4.5 afin de rendre le temps de création de ces directives significativement mesurable. L'Algorithme 16 décrit comment le benchmark met en évidence la réduction dans la section parallèle.

Algorithm 16 Calcul du coût de la réduction avec le benchmark EPCC

```
1:  for k=0; k<=OUTERLOOPS; ++k do
2:      start = getclock();
3:      for j = 0; j < INNERLOOPS; ++j do
4:          #pragma omp parallel reduction(+ :aaaa)
5:          {
6:          delay(delaylength);
7:          aaaa+=1;
8:          }
9:      end for
10:     times[k]=(getclock()-start)*1.0e6;
11:     times[k]/=(double) INNERLOOPS;
12: end for
```

L'idée étant de rendre le coût de création de la section parallèle négligeable par rapport au coût de la réduction. Pour ce faire, dans la boucle interne, le benchmark simule le temps de calcul par la fonction `delay()` et mesure le temps moyen total pour effectuer la boucle interne. Puis cette boucle interne est répétée `OUTERLOOPS` fois. À la fin, une moyenne des temps de la boucle interne est calculée. La mesure du surcoût de la réduction est obtenue en examinant la différence entre le temps moyen d'exécution parallèle et un temps d'exécution de référence [13]. Le calcul des surcoûts de barrière et de réduction est assez similaire à nos tests synthétiques. Cependant, le benchmark impose un délai entre chaque itération de la boucle interne pour simuler un calcul avant la réduction. Ce qui est plus réaliste que notre test synthétique.

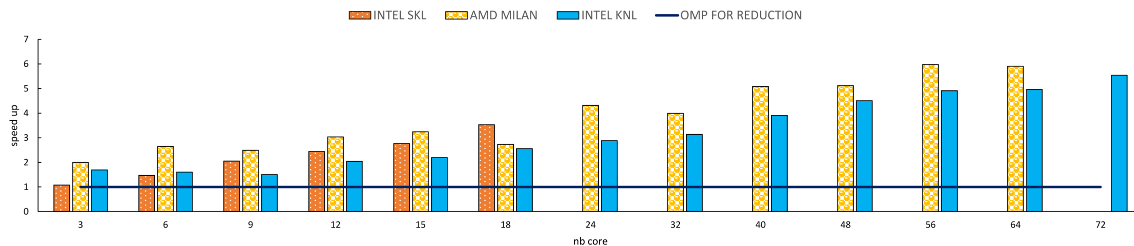


Figure 4.16 – *Représentation des coûts de la Réduction Extended Butterfly par rapport à la réduction OpenMP 4.5 sur* SKL, KNL *et* MILAN. *Les coûts sont calculés avec le benchmark EPCC.*

La Figure 4.16 montre l'évolution, en fonction du nombre de threads, du rapport entre la Réduction Extended Butterfly et l'opération de réduction OpenMP fournis par les compilateurs GCC 11.1 et Intel 21.4 respectivement sur MILAN et SKL, KNL. De sorte que plus les accélérations sont élevées, plus le coût est faible par rapport à la réduction d'OpenMP. Le coût de la Réduction Extended Butterfly est environ 4 fois moins élevé que celui la réduction d'OpenMP avec GCC 11.1. De plus, la Réduction Extended Butterfly présente environ 2 à 3,5 fois moins de surcoût que la réduction d'OpenMP avec Intel 21.4 sur les processeurs SKL et KNL.

4.6 Discussion

Notre objectif était d'avoir une opération de réduction combinée à une barrière qui soit performante pour un grand nombre de threads. Pour atteindre cet objectif, nous avions besoin d'une barrière qui soit efficace pour un grand nombre de threads. Nous avons d'abord étudié et implémenté différentes barrières de la littérature. Nous avons ensuite proposé une nouvelle barrière (la barrière *Extended Butterfly*) basée sur la barrière *Butterfly* et efficace pour un nombre quelconque de threads. Nos tests synthétiques ont montré que la barrière *Extended Butterfly* est la plus performante pour un grand nombre de threads. Elle surpasse de manière significative les barrières d'OpenMP avec les compilateurs Intel 21.4 et GCC 11.1. Nous avons ensuite proposé une méthode de réduction des valeurs combinée à une barrière en transportant les valeurs à réduire avec le drapeau de synchronisation. Ainsi, les valeurs de réduction sont directement disponibles dans la ligne de cache du thread et sont mises à jour à chaque étape de la barrière. Cette méthode a été utilisée dans la réduction combinée avec la barrière *Linear Centralized* et la barrière *Extended Butterfly*. Nous avons optimisé l'opération de réduction en utilisant des instructions SIMD pour lire et écrire les lignes de cache contenant les valeurs de réduction et le drapeau de synchronisation pour chaque thread de manière atomique. L'atomicité de l'écriture SIMD est primordiale pour la réduction. Nous avons remarqué que l'écriture n'était pas atomique pour les Intel Ivy-Bridge. Nous avons ainsi mené nos tests comparatifs sur trois processeurs différents (SKL, KNL et MILAN) et avec deux versions de compilateurs (Intel 21.4 et GCC 11.1). La Figure 4.16 montre que la réduction *Extended Butterfly* a en moyenne 2 fois et 3,5 fois moins de surcoûts que la réduction d'OpenMP avec Intel 21.4 sur SKL et KNL. Par exemple, avec 72 cœurs sur KNL, la réduction *Extended Butterfly* est 5,4 fois plus rapides que la réduction d'OpenMP avec Intel 21.4 et avec 64 cœurs sur MILAN, la réduction *Extended Butterfly* est 5,9 fois plus rapides que la réduction d'OpenMP avec GCC 11.1.
Comme travail futur, nous continuerons les tests sur des benchmarks applicatifs basés sur des solveurs de Krylov tels que CG ou BiCGSTAB. Il serait intéressant de configurer la réduction *Extended Butterfly* pour gérer l'hyperthreading. De plus, les processeurs utilisent des topologies d'interconnexion des cœurs différentes (KNL et SKL sont similaires, mais pas MILAN), il pourrait donc être intéressant d'étudier l'impact sur les barrières et les opérations de réduction. Nous prévoyons qu'avec l'augmentation du nombre de cœurs, l'impact de la topologie peut être significatif.

Conclusions générales et perspectives

Conclusions générales

Performance du préconditionneur ILU(0) **pour le solveur** BiCGStab. Dans le premier Partie de ce manuscrit, nous avons vu que la résolution du système linéaire 1.1 se déroulait en 3 étapes :

1. Renumérotation des équations du système
2. Préconditionnement du système avec ILU(0)
3. Résolution du système préconditionné avec BiCGStab

Au cours de la troisième étape, il est nécessaire de résoudre quatre systèmes triangulaires à chaque itération du solveur. Le coût des remontées et descentes de ces systèmes triangulaires représente en moyenne la majeure partie du temps de calcul par itération. Notre but était donc de paralléliser la résolution de ces systèmes triangulaires. Plus précisément, nous cherchons tout d'abord à mettre en œuvre une version SIMD de notre solveur. Cependant, la plupart des méthodes permettant une parallélisation grain fin des calculs à cette troisième étape ont un impact négatif sur la convergence du solveur. Nous avons constaté qu'une parallélisation grain fin des calculs en mémoire partagée entraîne souvent une augmentation significative du nombre d'itérations du solveur préconditionné. Des articles qui ont analysé ce problème ont conclu que cela était dû à une précision insuffisante de la factorisation incomplète ou à une instabilité des facteurs triangulaires issus de la factorisation incomplète. Enfin, notre objectif dans ce chapitre était double : travailler sur une méthode permettant d'obtenir la parallélisation grain fin souhaitée tout en minimisant l'impact négatif sur la convergence du solveur. Notre approche pour résoudre ce problème consiste à combiner deux méthodes de renumérotation des équations du système. La plupart des articles qui traitent de la convergence d'un solveur de Krylov préconditionné indiquent que RCM est souvent l'une des méthodes permettant d'obtenir une bonne convergence. D'un autre côté, les méthodes de coloriage sont considérées comme permettant d'obtenir un fort degré de parallélisme à grain fin. La renumérotation avec RCM, construit des ensembles de nœuds à chaque niveau qui sont ensuite classés par ordre croissant de leur degré. L'idée est de

chercher une coloration des nœuds suivant cet ordre et par niveau. Nous avons nommé cette méthode ColorRCM. De plus, nous avons remarqué qu'augmenter le nombre de couleurs dans les méthodes de coloriage améliore significativement la convergence du solveur mais au détriment du parallélisme. Donc, la taille des couleurs de nos méthodes sera limitée à la taille d'un paquet SIMD, par exemple 4 en AVX2 ou 8 en AVX512 (pour la double précision). Une première étape était de comparer l'impact des méthodes de renumérotation sur la convergence du solveur. On observe que ColorRCM(8) se place parmi les meilleures méthodes de renumérotation, proche de RCM et MD. Parmi les méthodes de coloriage, ColorRCM est celle qui dégrade le moins la convergence du solveur comparé à un GreedyMC classique. De plus, il semble que ColorRCMpermet d'avoir un préconditionneur ILU(0) plus efficace qu'avec GreedyMC(k).

Par la suite, nous avons tiré parti de la structure particulière apportée par ColorRCM pour mettre en œuvre une version SIMD du solveur. Nous avons stocké le préconditionneur au format SELL-P. Ce format a été introduit pour favoriser le calcul vectoriel et réduire le surcoût de remplissage qui qu'implique le format EllPack. Nous avons commencé par mesurer les performances synthétiques qu'apporte la vectorisation de la résolution de systèmes triangulaires permutés avec ColorRCM. Nous observons une accélération minimum de 3,5 et jusqu'à 11 pour l'AVX2 par rapport à `mkl_cspblas_dbsrtrsv` de Intel MKL 21.4. Aussi nous avons mesuré les temps moyens par itération du solveur BiCGStab préconditionné. En comparant la version SIMD AVX2 à la version séquentielle, nous obtenons un gain de temps moyen d'au moins 50%. Cela se traduit par une accélération de 1,5 par itération.

Mécanismes de synchronisation. La seconde partie de notre manuscrit porte sur les mécanismes de synchronisations des threads. Dans un contexte de parallélisation massive en mémoire partagée, ces mécanismes peuvent parfois devenir des goulots d'étranglement des performances globales d'un solveur de Krylov. Par exemple, nous avons, à chaque itération du BiCGStab préconditionné cinq produits scalaires qui ont besoin de réduction et de synchronisation des threads. Lors de tests préliminaires, nous avons remarqué que le coût de la réduction de valeurs dans une zone parallèle en mémoire partagée croît linéairement avec le compilateur GCC 11.1 et a une croissance logarithmique avec le compilateur Intel 21.4.

Notre but dans cette deuxième partie était de proposer un algorithme de réduction combinée à une barrière pour des processeurs multi-cœurs comme skl, knl ou milan. Nous avons étudié des barrières issues de l'état de l'art. La comparaison de ces barrières par rapport aux barrières OpenMP des compilateurs Intel 21.4 et GCC 11.1 a conduit à la proposition d'une nouvelle barrière *Extended Butterfly* qui a une croissance logarithmique et sans redondance. Elle est donc adaptée pour la combinaison avec une réduction.

Nous avons mis en œuvre la réduction combinée avec la barrière *Extended Butterfly* en exploitant l'atomicité des écritures SIMD qui permet d'effectuer la réduction avec peut de surcoût par rapport à la barrière seule. Ce qui n'est pas le cas avec les réductions OpenMP testées.

Perspectives

Dans les processeurs modernes, nous avons vu qu'il y a 3 niveaux de parallélisme :

1. un parallélisme multi-threads au niveau des coeurs ;

2. un parallélisme SIMD au niveau des données ;
3. un parallélisme au niveau des flots d'instructions dans les unités de calculs (hyperthreading).

Point 1

Dans ce manuscrit, nous n'avons pas abordé la parallélisation multithread du solveur. Les résultats des tests de ND et GreedyMC classiques, ont mis en évidence les contrecoups élevés de ces méthodes sur la convergence du solveur de Krylov préconditionné avec ILU(0). La méthode ColorRCM n'est pas adapté à la création de couleurs très grandes, même si en théorie l'utilisateur peut définir la taille de couleurs qu'il souhaite. Cependant, cela a pour contrepartie l'impossibilité de remplir suffisamment les couleurs et le surcoût de stockage causé par le rembourrage des couleurs, i.e., l'augmentation artificielle de la taille des couleurs.

Coloriage hiérarchique Nous avons repéré une approche intéressante pour le traitement des calculs en multithread : l'approche de coloriage hiérarchique présentée dans [48]. Le principe consiste à stocker la matrice sous forme de blocs de taille choisie par l'utilisateur, puis à appliquer un coloriage en partant de ces blocs. Le second niveau de coloriage vise à trouver une coloration à l'intérieur des blocs. Le premier niveau permet de paralléliser le traitement multithread, tandis que le second niveau permet de paralléliser le traitement des calculs avec du SIMD. Une solution possible pourrait être d'utiliser ColorRCM pour créer des couleurs de taille égale à un paquet SIMD. Ensuite, dans chaque niveau de ColorRCM, nous pourrions chercher une coloration des blocs de couleurs. Grâce à cette approche, nous conservons la structure globale de RCM tout en profitant de la parallélisation multithread dans les couleurs, qui elles-mêmes seront composées de blocs permettant de vectoriser les calculs à leur niveau. Cette méthode présente néanmoins plusieurs contraintes. La performance de la parallélisation multithread dépend en partie de la charge de travail des threads, qui doit être suffisante et équilibrée entre les threads. Cette charge de travail correspond à la taille des couleurs créées lors de la seconde phase de la méthode hiérarchique présentée ci-dessus. Or, la taille de ces couleurs dépend fortement de la taille des niveaux dans RCM. Un autre point à prendre en compte est l'équilibre des charges de travail entre les threads. Enfin, il faudra être vigilant quant à une possible dégradation de l'efficacité du préconditionneur.

ColorRCM Nous pensons qu'il y a un aspect de la méthode qui peut être optimisé : la parallélisation de ColorRCM. Dans la Section 2.3, nous avons décrit la conception de la méthode. Bien que le calcul des niveaux avec RCM soit intrinsèquement séquentiel, la partie de coloriage peut être effectuée de manière parallèle. En effet, le coloriage d'un niveau de RCM est indépendant des autres niveaux.

Point 2

Le point 2 a été le sujet principale de la thèse avec les algorithmes de coloriage.

ILU(0) La parallélisation SIMD de ILU(0) reste a développer. Comme nous l'avons vu, l'approche SIMD avec le format SELL-P ne nous semble pas préférable. Une première approche sera la mise en oeuvre d'une version multi-ligne avec un format CSR. D'autre part, nous savons que la disponibilité des données en caches est très important pour assurer de bonne performances lors des calculs. Dans nos travaux, nous n'avons pas exploiter la structure apporter par RCM. En effet, RCM nous donne une structure tri-diagonale par blocs. Il y a un travail à faire sur la mise en cache de certaines données se situant dans le niveau de RCM juste au dessus du niveau courant.

Point 3

Dans le Chapitre 4 sur les mécanismes de synchronisation, nous avons mis en oeuvre une méthode dans un contexte de parallélisation multithreads. La synchronisation des threads dans un contexte d'hyperthreading serait un point intéressant à étudier. Une première approche serait de s'assurer que les groupes de threads soient constitués des hyperthreads d'un même coeurs. L'idée fut exploitée dans [65, 81] avec les coeurs d'un même noeud NUMA et ils ont démontré de meilleur performances que les barrières d'OpenMP.

A ANNEXE

Contents

A.1 **Résultats des tests avec le solveur** ILU(0)-BiCGStab 91
A.2 **Algorithmes pour la réduction des valeurs avec** *Extended Butterfly* 98

A.1 Résultats des tests avec le solveur ILU(0)-BiCGStab

Les résultats des tests qui sont présentés dans les Tables A.1 et A.2 ont été effectué avec un critère d'arrêt à 10^{-5} et un nombre maximal d'itérations à 1000. Les valeurs *NaN* symbolise un *breakdown* du solveur. Ce *breakdown* survient lorsque certaines quantités de l'algorithme BiCGStab 1 prennent des valeurs infiniment grandes ou atteignent le zéros machine. Les valeurs *inf* indique que le nombre maximal d'itérations a été atteint.

Table A.1 – *Résultats des tests* ILU(0)-BiCGStab *avec les matrices de la collection Suite Sparse Matrix.*

	Name_Matrix	Nrows	NNZ	*NColors*	*% of size colors 8*	*Nelem add*	*ILU(0)*	*ColorRCM(8)*	*GREEDYMC*
0	ABACUS_shell_ud	23412	218484	3150	86,0635%	1788	101	NaN	511
1	ACTIVSg70K	69999	238627	8834	98,2567%	673	NaN	inf	NaN
2	CO	221119	7666060	27674	99,7904%	273	243	167	228
3	CurlCurl_0	11083	113343	1451	91,3853%	525	20	16	29
4	CurlCurl_1	226451	2472070	28448	99,0333%	1133	46	679	79
5	CurlCurl_2	806529	8921790	101001	99,6554%	1479	51	NaN	74
6	CurlCurl_3	1219570	13544600	152637	99,7687%	1522	56	NaN	73
7	CurlCurl_4	2380520	26515900	297797	99,8536%	1861	62	NaN	80
8	F1	343791	26837100	43694	97,1689%	5761	NaN	609	852
9	F2	71505	5294280	9394	91,5797%	3647	NaN	NaN	NaN
10	FEM_3D_thermal1	17880	430740	2425	84,8247%	1520	3	5	9
11	FEM_3D_thermal2	147900	3489300	18760	97,3294%	2180	3	5	7
12	Ga10As10H30	113081	6115630	14405	97,6328%	2159	NaN	NaN	NaN
13	Ga19As19H42	133123	8884840	17128	96,2226%	3901	NaN	NaN	NaN
14	Ga3As3H12	61349	5970950	8539	85,373%	6963	366	NaN	341
15	Ga41As41H72	268096	18488500	33851	98,8006%	2712	NaN	NaN	NaN
16	GaAsH6	61349	3381810	8778	85,3725%	8875	139	113	143
17	Ge87H76	112985	7892200	14152	99,6326%	231	NaN	NaN	NaN
18	Ge99H100	112985	8451400	14288	98,4113%	1319	NaN	NaN	NaN
19	H2O	67024	2216740	8416	98,9306%	304	45	50	61
20	Lin	256000	1766400	32090	99,4391%	720	310	303	982
21	af_shell1	504855	17588900	65095	94,1086%	15905	767	573	826
22	af_shell10	1508060	52672300	192050	96,1599%	28335	34	10	16
23	af_shell2	504855	17588900	65095	94,1086%	15905	NaN	NaN	NaN
24	af_shell5	504855	17588900	65095	94,1086%	15905	475	451	545
25	af_shell6	504855	17588900	65095	94,1086%	15905	NaN	NaN	NaN
26	af_shell9	504855	17588900	65095	94,1086%	15905	183	209	262
27	appu	14000	1853100	1768	98,19%	144	14	11	14
28	atmosmodd	1270430	8814880	158974	99,7861%	1360	60	61	101
29	atmosmodj	1270430	8814880	158974	99,7861%	1360	52	54	89
30	atmosmodl	1489750	10319800	186432	99,7726%	1704	16	16	26
31	barrier2-1	113076	3805070	14248	98,5191%	908	NaN	294	NaN
32	barrier2-2	113076	3805070	14248	98,5191%	908	NaN	NaN	NaN
33	barrier2-3	113076	3805070	14248	98,5191%	908	NaN	NaN	NaN
34	barrier2-4	113076	3805070	14248	98,5191%	908	NaN	NaN	NaN
35	bcircuit	68902	375558	8720	97,8326%	858	NaN	NaN	NaN
36	bcsstk35	30237	1450160	4142	84,3554%	2899	NaN	NaN	NaN
37	bcsstk37	25503	1140980	3497	84,2436%	2473	NaN	NaN	NaN
38	bcsstk39	46772	2089290	6367	72,2475%	4164	NaN	NaN	NaN
39	bmw3_2	227362	11288600	29216	95,1088%	6366	NaN	NaN	NaN
40	c-42	10471	110285	1313	99,543%	33	inf	NaN	inf

	Name_Matrix	Nrows	NNZ	NColors	% of size colors 8	Nelem add	ILU(0)	ColorRCM(8)	GREEDYMC
41	c-44	10728	85000	1351	98,8897%	80	inf	NaN	NaN
42	c-45	13206	174452	1655	99,6375%	34	inf	NaN	inf
43	c-46	14913	130397	1868	99,6788%	31	NaN	inf	inf
44	c-47	15343	211401	1924	99,4802%	49	inf	inf	inf
45	c-48	18354	166080	2301	99,5219%	54	NaN	inf	inf
46	c-55	32780	403450	4103	99,7319%	44	NaN	inf	NaN
47	c-59	41282	480536	5167	99,7678%	54	NaN	inf	inf
48	c-61	43618	310016	5462	99,6705%	78	NaN	inf	NaN
49	c-63	44234	434704	5537	99,7833%	62	inf	inf	NaN
50	c-64	51035	717841	6384	99,8904%	37	inf	NaN	inf
51	c-64b	51035	717841	6384	99,8904%	37	inf	NaN	inf
52	c-70	68924	658986	8624	99,8493%	68	inf	inf	NaN
53	c-73	169422	1279270	21185	99,9481%	58	inf	inf	inf
54	c-73b	169422	1279270	21185	99,9481%	58	inf	NaN	inf
55	cage10	11397	150645	1449	97,1014%	195	3	3	3
56	cage11	39082	559722	4927	98,5184%	334	3	3	3
57	cage12	130228	2032540	16319	99,5649%	324	3	3	3
58	cage13	445315	7479340	55718	99,8205%	429	3	3	3
59	cage14	1505780	27130300	188272	99,9511%	391	3	3	3
60	cage15	5154860	99199600	644436	99,9786%	629	3	3	3
61	chem_master1	40401	201201	5226	93,2836%	1407	61	69	138
62	chipcool0	20082	281150	2590	94,5946%	638	25	26	31
63	chipcool1	20082	281150	2590	94,5946%	638	27	28	31
64	copter2	55476	759952	7032	97,5114%	780	NaN	inf	NaN
65	crystk02	13965	968583	1994	79,1374%	1987	NaN	NaN	NaN
66	crystk03	24696	1751180	3394	82,7637%	2456	NaN	NaN	inf
67	cvxqp3	17500	122462	2202	98,8647%	116	NaN	NaN	NaN
68	dielFilterV2real	1157460	48539000	145388	99,1705%	5648	804	948	NaN
69	dielFilterV3real	1102820	89306000	138986	98,5718%	9064	NaN	inf	inf
70	dixmaanl	60000	299998	20003	0%	100024	inf	NaN	64
71	ecology1	1000000	4996000	125875	98,6097%	7000	NaN	NaN	NaN
72	ex11	16614	1096950	2290	83,4934%	1706	NaN	NaN	NaN
73	filter3D	106437	2707180	13602	96,177%	2379	inf	inf	NaN
74	garon2	13535	390607	2178	60,1469%	3889	147	NaN	inf
75	gas_sensor	66917	1703360	8477	97,4047%	899	44	233	923
76	gsm_106857	589446	21758900	74218	98,7631%	4298	inf	inf	inf
77	helm2d03	392257	2741940	49666	97,3342%	5071	924	NaN	NaN
78	helm3d01	32226	428444	4082	97,5257%	430	inf	NaN	NaN
79	igbt3	10938	234006	1997	47,2709%	5038	458	63	180
80	k3plates	11107	378927	1894	51,7951%	4045	106	58	22
81	largebasis	440020	5560100	80005	49,9994%	200020	NaN	8	NaN
82	linverse	11999	95977	11999	0%	83993	1	1	NaN
83	nlpkkt120	3542400	96845800	442846	99,9727%	368	NaN	NaN	NaN
84	nlpkkt160	8345600	229518000	1043261	99,9846%	488	NaN	NaN	NaN
85	nmos3	18588	386594	2590	80,695%	2132	20	41	67

	Name_Matrix	Nrows	NNZ	NColors	% of size colors 8	Nelem add	ILU(0)	ColorRCM(8)	GREEDYMC
86	ohne2	181343	11063500	22895	98,183%	1817	inf	367	inf
87	para-4	153226	5326230	19284	98,7814%	1046	inf	119	NaN
88	poisson3Da	13514	352762	1742	94,7187%	422	13	11	15
89	poisson3Db	85623	2374950	10777	98,7566%	593	37	27	39
90	raefsky3	21200	1488770	3128	71,867%	3824	1	1	1
91	sme3Da	12504	874887	1821	74,5744%	2064	653	434	inf
92	sme3Db	29067	2081060	3985	85,1694%	2813	912	660	inf
93	sme3Dc	42930	3148660	5792	87,3446%	3406	NaN	753	inf
94	thermomech_dK	204316	2846230	27036	89,6878%	11972	NaN	NaN	NaN
95	tmt_unsym	917825	4584800	115406	98,838%	5423	901	796	NaN
96	venkat01	62424	1717790	8216	90,0682%	3304	9	10	21
97	venkat25	62424	1717790	8216	90,0682%	3304	123	164	inf
98	venkat50	62424	1717790	8216	90,0682%	3304	213	296	inf
99	wang3	26064	177168	3295	97,7542%	296	24	28	47
100	wang4	26068	177196	3296	97,7852%	300	21	24	40

La Table A.2 présente les résultats des tests ILU(0)-BiCGStab avec les matrices SPE10. Toutes les matrices SPE10 ont la même structure : Nrows = 1 094 420 et NNZ = 7 515 590. Avec l'algorithme de coloriage ColorRCM(8) on obtient 137 015 couleurs dont 99,7248% sont composés de 8 nœuds ce qui implique un ajout de 1699 éléments diagonaux dans le format étendu. L'ensemble des tests ont été effectué avec un critère d'arrêt à 10^{-5} et un nombre maximal d'itérations à 1000. Les valeurs *inf* indique que le nombre maximal d'itérations a été atteint.

Table A.2 – *Résultats des tests* ILU(0)-BiCGStab *avec les matrices SPE10.*

	Name_Matrix	ILU0	CB8ILU0RCM	GREEDYMC
101	spe10_Frs750b_N110_I1_F111	346	396	inf
102	spe10_Frs750b_N111_I1_F112	356	354	inf
103	spe10_Frs750b_N112_I1_F113	349	396	inf
104	spe10_Frs750b_N113_I1_F114	373	389	inf
105	spe10_Frs750b_N114_I1_F115	406	334	inf
106	spe10_Frs750b_N115_I1_F116	387	468	inf
107	spe10_Frs750b_N116_I1_F117	328	368	inf
108	spe10_Frs750b_N117_I1_F118	263	301	inf
109	spe10_Frs750b_N118_I1_F119	341	478	inf
110	spe10_Frs750b_N119_I1_F120	226	285	inf
111	spe10_Frs750b_N11_I1_F12	131	124	235
112	spe10_Frs750b_N120_I1_F121	370	355	inf
113	spe10_Frs750b_N121_I1_F122	274	265	inf
114	spe10_Frs750b_N122_I1_F123	356	489	inf
115	spe10_Frs750b_N123_I1_F124	251	264	inf
116	spe10_Frs750b_N124_I1_F125	310	343	inf
117	spe10_Frs750b_N125_I1_F126	343	366	inf
118	spe10_Frs750b_N126_I1_F127	356	464	inf
119	spe10_Frs750b_N127_I1_F128	257	276	819
120	spe10_Frs750b_N128_I1_F129	342	400	inf
121	spe10_Frs750b_N129_I1_F130	486	406	inf
122	spe10_Frs750b_N12_I1_F13	133	136	243
123	spe10_Frs750b_N130_I1_F131	411	388	inf
124	spe10_Frs750b_N131_I1_F132	261	265	933
125	spe10_Frs750b_N132_I1_F133	360	405	inf
126	spe10_Frs750b_N133_I1_F134	344	425	inf
127	spe10_Frs750b_N134_I1_F135	351	386	inf
128	spe10_Frs750b_N135_I1_F136	350	609	inf
129	spe10_Frs750b_N136_I1_F137	353	344	inf
130	spe10_Frs750b_N137_I1_F138	346	384	inf
131	spe10_Frs750b_N138_I1_F139	349	402	inf
132	spe10_Frs750b_N139_I1_F140	432	490	inf
133	spe10_Frs750b_N13_I1_F14	137	124	237
134	spe10_Frs750b_N140_I1_F141	336	403	inf
135	spe10_Frs750b_N141_I1_F142	332	392	inf
136	spe10_Frs750b_N142_I1_F143	321	347	inf
137	spe10_Frs750b_N14_I1_F15	138	130	240
138	spe10_Frs750b_N15_I1_F16	121	130	211
139	spe10_Frs750b_N16_I1_F17	121	128	261
140	spe10_Frs750b_N17_I1_F18	124	118	255
141	spe10_Frs750b_N18_I1_F19	131	130	259
142	spe10_Frs750b_N19_I1_F20	140	147	275
143	spe10_Frs750b_N1_I1_F2	31	33	54
144	spe10_Frs750b_N20_I1_F21	141	144	276
145	spe10_Frs750b_N21_I1_F22	142	158	260
146	spe10_Frs750b_N22_I1_F23	139	148	314
147	spe10_Frs750b_N23_I1_F24	145	147	303
148	spe10_Frs750b_N24_I1_F25	133	139	268
149	spe10_Frs750b_N25_I1_F26	129	140	316
150	spe10_Frs750b_N26_I1_F27	149	136	303
151	spe10_Frs750b_N27_I1_F28	135	133	275
152	spe10_Frs750b_N28_I1_F29	130	144	316

	Name_Matrix	ILU0	CB8ILU0RCM	GREEDYMC
153	spe10_Frs750b_N29_I1_F30	142	143	337
154	spe10_Frs750b_N2_I1_F3	48	51	88
155	spe10_Frs750b_N30_I1_F31	136	154	314
156	spe10_Frs750b_N31_I1_F32	129	130	327
157	spe10_Frs750b_N32_I1_F33	121	142	243
158	spe10_Frs750b_N33_I1_F34	136	137	266
159	spe10_Frs750b_N34_I1_F35	131	127	284
160	spe10_Frs750b_N35_I1_F36	129	125	304
161	spe10_Frs750b_N36_I1_F37	134	128	265
162	spe10_Frs750b_N37_I1_F38	139	131	252
163	spe10_Frs750b_N38_I1_F39	125	123	325
164	spe10_Frs750b_N39_I1_F40	129	137	344
165	spe10_Frs750b_N3_I1_F4	53	53	86
166	spe10_Frs750b_N40_I1_F41	131	132	288
167	spe10_Frs750b_N41_I1_F42	125	137	296
168	spe10_Frs750b_N42_I1_F43	145	125	335
169	spe10_Frs750b_N43_I1_F44	129	133	263
170	spe10_Frs750b_N44_I1_F45	143	130	281
171	spe10_Frs750b_N45_I1_F46	141	149	287
172	spe10_Frs750b_N46_I1_F47	136	128	298
173	spe10_Frs750b_N47_I1_F48	124	126	279
174	spe10_Frs750b_N48_I1_F49	144	126	290
175	spe10_Frs750b_N49_I1_F50	134	136	406
176	spe10_Frs750b_N4_I1_F5	59	67	111
177	spe10_Frs750b_N50_I1_F51	127	147	409
178	spe10_Frs750b_N51_I1_F52	139	175	429
179	spe10_Frs750b_N52_I1_F53	168	182	715
180	spe10_Frs750b_N53_I1_F54	240	269	911
181	spe10_Frs750b_N54_I1_F55	240	255	924
182	spe10_Frs750b_N55_I1_F56	294	253	inf
183	spe10_Frs750b_N56_I1_F57	236	253	inf
184	spe10_Frs750b_N57_I1_F58	280	277	999
185	spe10_Frs750b_N58_I1_F59	259	260	inf
186	spe10_Frs750b_N59_I1_F60	256	266	902
187	spe10_Frs750b_N5_I1_F6	64	73	124
188	spe10_Frs750b_N60_I1_F61	249	283	inf
189	spe10_Frs750b_N61_I1_F62	286	273	inf
190	spe10_Frs750b_N62_I1_F63	280	302	inf
191	spe10_Frs750b_N63_I1_F64	274	287	inf
192	spe10_Frs750b_N64_I1_F65	291	282	inf
193	spe10_Frs750b_N65_I1_F66	312	293	inf
194	spe10_Frs750b_N66_I1_F67	285	321	inf
195	spe10_Frs750b_N67_I1_F68	294	290	963
196	spe10_Frs750b_N68_I1_F69	269	290	878
197	spe10_Frs750b_N69_I1_F70	281	331	938
198	spe10_Frs750b_N6_I1_F7	78	80	137
199	spe10_Frs750b_N70_I1_F71	244	314	848
200	spe10_Frs750b_N71_I1_F72	308	306	910
201	spe10_Frs750b_N72_I1_F73	318	301	inf
202	spe10_Frs750b_N73_I1_F74	299	327	985
203	spe10_Frs750b_N74_I1_F75	300	316	inf
204	spe10_Frs750b_N75_I1_F76	306	332	inf
205	spe10_Frs750b_N76_I1_F77	301	312	inf
206	spe10_Frs750b_N77_I1_F78	287	314	inf

	Name_Matrix	ILU0	CB8ILU0RCM	GREEDYMC
207	spe10_Frs750b_N78_I1_F79	318	313	inf
208	spe10_Frs750b_N79_I1_F80	309	336	inf
209	spe10_Frs750b_N7_I1_F8	92	93	159
210	spe10_Frs750b_N80_I1_F81	412	488	inf
211	spe10_Frs750b_N81_I1_F82	355	468	inf
212	spe10_Frs750b_N82_I1_F83	466	366	inf
213	spe10_Frs750b_N83_I1_F84	355	475	inf
214	spe10_Frs750b_N84_I1_F85	272	290	inf
215	spe10_Frs750b_N85_I1_F86	374	393	inf
216	spe10_Frs750b_N86_I1_F87	394	353	inf
217	spe10_Frs750b_N87_I1_F88	457	398	inf
218	spe10_Frs750b_N88_I1_F89	296	229	inf
219	spe10_Frs750b_N89_I1_F90	403	414	inf
220	spe10_Frs750b_N8_I1_F9	101	101	196
221	spe10_Frs750b_N90_I1_F91	480	457	inf
222	spe10_Frs750b_N91_I1_F92	440	441	inf
223	spe10_Frs750b_N92_I1_F93	396	402	inf
224	spe10_Frs750b_N93_I1_F94	386	411	inf
225	spe10_Frs750b_N94_I1_F95	376	448	inf
226	spe10_Frs750b_N95_I1_F96	395	376	inf
227	spe10_Frs750b_N96_I1_F97	375	430	inf
228	spe10_Frs750b_N97_I1_F98	379	351	inf
229	spe10_Frs750b_N98_I1_F99	550	353	inf
230	spe10_Frs750b_N99_I1_F100	348	402	inf
231	spe10_Frs750b_N9_I1_F10	114	114	193

A.2 Algorithmes pour la réduction des valeurs avec *Extended Butterfly*

Algorithm 17 Phase entrante dans la barrière *Extended Butterfly*

```
1: procedure REDUCE_LINEAR_CENTRALIZED(N,val,tid)
2:     Input : Vecteur des valeurs de réduction : val ; Entier : N, tid
3:     Output : Vecteur des valeurs de réduction : val
4:    if groupOfThread[tid].isMaster then
5:        for i parmi les membres du groupe do
6:            Repeat until (flag[i].flag[0]==WAIT)
7:            for j dans intervalle (0,N-1) do  ▷ Déroulement d'une opération SIMD de type add
8:                val[j]+=flag[i].val[j+1]
9:            end for
10:       end for
11:   else
12:       NvCacheLine localFlag ;
13:       localFlag.flag[0]=WAIT
14:       for j dans l'intervalle (1,N) do          ▷ Déroulement d'une écriture SIMD
15:           localFlag.val[j] = val[j-1]
16:       end for
17:       flag[tid].simdVec = localFlag.simdVec
18:   end if
19: end procedure
```

Algorithm 18 Phase *Butterfly* dans la barrière *Extended Butterfly*

```
1: procedure REDUCE_BUTTERFLY(N,val,tid,sense)
2:     Input : Vecteur des valeurs de réduction : val ; Entier : N, tid ; Entier 0 ou 1 : sense
3:     Output : Vecteur des valeurs de réduction : val
4:    NvCacheLine localFlag
5:    for step dans l'intervalle (0,nStep-1) do
6:        int partner = getpartner(groupOfThread[tid].idGroup,step)  ▷ Calculer l'identifiant du
   partenaire (voir la Section 4.2.5)
7:        localFlag.flag[0] = sense
8:        for j dans l'intervalle (1,N) do          ▷ Déroulement d'une écriture SIMD
9:            localFlag.val[j] = val[j-1]
10:       end for
11:       groupOfThread[partner].flag[parity][step].simdVec = localFlag.simdVec
12:       Répéter jusqu'à (groupOfThread[tid].flag[parity][step].flag[0]==sense)
13:       for j dans l'intervalle (0,N-1) do     ▷ Déroulement d'une opération SIMD de type add
14:           val[j]+=groupOfThread[tid].flag[parity][step].val[j+1] ;
15:       end for
16:   end for
17: end procedure
```

Algorithm 19 Phase sortante dans la barrière *Extended Butterfly*

```
1: procedure BCAST_LINEAR_CENTRALIZED(N,val,tid)
2:     Input : Vecteur des valeurs de réduction : val ; Entier : N, tid
3:     Output : Vecteur des valeurs de réduction : val
4:     if groupOfThread[tid].isMaster then
5:         NvCacheLine localFlag
6:         localFlag.flag[0] = GO
7:         for j dans l'intervalle (1,N) do          ▷ Déroulement d'une écriture SIMD
8:             localFlag.val[j] = val[j-1]
9:         end for
10:        for i parmi les membres du groupe do
11:            flag[i].simdVec = localFlag.simdVec
12:        end for
13:    else
14:        Répéter jusqu'à (flag[tid].flag[0]==GO)
15:        for j dans l'intervalle (0,N-1) do        ▷ Déroulement d'une écriture SIMD
16:            val[j] = flag[tid].val[j+1]
17:        end for
18:    end if
19: end procedure
```

Bibliographie

[1] Technologie Intel® Hyper-Threading. URL https://www.intel.com/content/www/fr/fr/architecture-and-technology/hyper-threading/hyper-threading-technology.html.

[2] A. A. Barrier Synchronization : Simplified, Generalized, and Solved Without Mutual Exclusion. In *2018 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 773–782, Los Alamitos, CA, USA, may 2018. IEEE Computer Society. doi : 10.1109/IPDPSW.2018.00124. URL https://doi.ieeecomputersociety.org/10.1109/IPDPSW.2018.00124.

[3] G. Allaire and S. M. Kaber. *Numerical Linear Algebra*. Texts in Applied Mathematics. Springer-Verlag, New York, 2008. ISBN 978-0-387-34159-0. doi : 10.1007/978-0-387-68918-0. URL https://www.springer.com/gp/book/9780387341590.

[4] H. Anzt, S. Tomov, and J. Dongarra. Implementing a sparse matrix vector product for the sell-c/sell-c-sigma formats on nvidia gpus. Technical Report UT-EECS-14-727, 2014-04 2014. URL https://www.icl.utk.edu/files/publications/2014/icl-utk-772-2014.pdf.

[5] H. Anzt, E. Chow, and J. Dongarra. ParILUT—A New Parallel Threshold ILU Factorization. *SIAM Journal on Scientific Computing*, 40(4) :C503–C519, Jan. 2018. ISSN 1064-8275. doi : 10.1137/16M1079506. URL https://epubs.siam.org/doi/abs/10.1137/16M1079506. Publisher : Society for Industrial and Applied Mathematics.

[6] A. Architecture. AMD64 Architecture Programmer's Manual, Volumes 1-5, 40332, 24592, 24593, 24594, 26568, 26569. Technical Report Volumes 1-5, 2020. URL https://www.amd.com/system/files/TechDocs/40332.pdf.

[7] I. Architecture. Intel® Architecture Instruction Set Extensions Programming Reference. Technical Report 319433-023, Aug. 2015. URL https://software.intel.com/sites/default/files/managed/07/b7/319433-023.pdf.

[8] M. Benzi. Preconditioning Techniques for Large Linear Systems : A Survey. *Journal of*

Computational Physics, 182(2) :418–477, Nov. 2002. ISSN 0021-9991. doi : 10.1006/jcph.2002.7176. URL http://www.sciencedirect.com/science/article/pii/S0021999102971767.

[9] M. Benzi, W. Joubert, and G. Mateescu. Numerical experiments with parallel orderings for ILU preconditioners. *Electronic Transactions on Numerical Analysis*, Volume 8 :88–114, 1999. URL http://etna.mcs.kent.edu/volumes/1993-2000/vol8/abstract.php?vol=8&pages=88-114.

[10] M. Benzi, D. B. Szyld, and A. van Duin. Orderings for Incomplete Factorization Preconditioning of Nonsymmetric Problems. *SIAM Journal on Scientific Computing*, 20(5) : 1652–1670, Jan. 1999. ISSN 1064-8275. doi : 10.1137/S1064827597326845. URL https://epubs.siam.org/doi/abs/10.1137/S1064827597326845. Publisher : Society for Industrial and Applied Mathematics.

[11] E. Bodewig. *Matrix Calculus*. North Holland, Sept. 2014. ISBN 978-1-4832-5539-2. doi : 10.1016/C2013-0-12461-0. URL https://linkinghub.elsevier.com/retrieve/pii/C20130124610.

[12] E. D. Brooks. The butterfly barrier. *International Journal of Parallel Programming*, 15(4) :295–307, 1986. doi : 10.1007/BF01407877. URL https://doi.org/10.1007/BF01407877.

[13] J. M. Bull. Measuring Synchronisation and Scheduling Overheads in OpenMP. October 1999. URL http://www.epcc.ed.ac.uk/sites/default/files/PDF/ewomp99paper.pdf.

[14] D. Caballero. SIMD@OpenMP : a programming model approach to leverage SIMD features, 2015. URL http://hdl.handle.net/2117/96011.

[15] D. Caballero, A. Duran, and X. Martorell. An OpenMP* Barrier Using SIMD Instructions for Intel® Xeon Phi™ Coprocessor. In A. P. Rendell, B. M. Chapman, and M. S. Müller, editors, *OpenMP in the Era of Low Power Devices and Accelerators*, pages 99–113, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-40698-0.

[16] E. Chow and A. Patel. Fine-Grained Parallel Incomplete LU Factorization. *SIAM Journal on Scientific Computing*, 37(2) :C169–C193, Jan. 2015. ISSN 1064-8275. doi : 10.1137/140968896. URL https://epubs.siam.org/doi/abs/10.1137/140968896. Publisher : Society for Industrial and Applied Mathematics.

[17] E. Chow and Y. Saad. Ilus : An incomplete lu preconditioner in sparse skyline format. *International Journal for Numerical Methods in Fluids*, 25(7) :739–748, 1997. doi : https://doi.org/10.1002/(SICI)1097-0363(19971015)25:7<739::AID-FLD581>3.0.CO;2-Y. URL https://onlinelibrary.wiley.com/doi/abs/10.1002.

[18] E. Chow and Y. Saad. Experimental study of ILU preconditioners for indefinite matrices. *Journal of Computational and Applied Mathematics*, 86(2) :387–414, Dec. 1997. ISSN 0377-0427. doi : 10.1016/S0377-0427(97)00171-4. URL http://www.sciencedirect.com/science/article/pii/S0377042797001714.

[19] P. Concus, G. H. Golub, and D. P. O'Leary. A GENERALIZED CONJUGATE GRADIENT METHOD FOR THE NUMERICAL SOLUTION OF ELLIPTIC PARTIAL DIFFERENTIAL EQUATIONS. In J. R. Bunch and D. J. Rose, editors, *Sparse Matrix Computations*, pages 309–332. Academic Press, Jan. 1976. ISBN 978-0-12-141050-6. doi : 10.1016/B978-0-12-141050-6.50023-4. URL http://www.sciencedirect.com/science/article/pii/B9780121410506500234.

[20] E. Cuthill and J. McKee. Reducing the bandwidth of sparse symmetric matrices. In *Proceedings of the 1969 24th national conference*, ACM '69, pages 157–172, New York, NY, USA, Aug. 1969. Association for Computing Machinery. ISBN 978-1-4503-7493-4. doi : 10.1145/800195.805928. URL https://doi.org/10.1145/800195.805928.

[21] T. A. Davis and Y. Hu. The university of florida sparse matrix collection. *ACM Trans. Math. Softw.*, 38(1), dec 2011. ISSN 0098-3500. doi : 10.1145/2049662.2049663. URL https://doi.org/10.1145/2049662.2049663.

[22] S. Doi. On parallelism and convergence of incomplete lu factorizations. *Applied Numerical Mathematics*, 7(5) :417–436, 1991. ISSN 0168-9274. doi : https://doi.org/10.1016/0168-9274(91)90011-N. URL https://www.sciencedirect.com/science/article/pii/016892749190011N.

[23] E. D. Dolan and J. J. Moré. Benchmarking optimization software with performance profiles. *Math. Program.*, 2001. doi : 10.1007/s101070100263. URL https://doi.org/10.1007/s101070100263.

[24] J. J. Dongarra, L. S. Duff, D. C. Sorensen, and H. A. V. Vorst. *Numerical Linear Algebra for High Performance Computers*. SIAM, USA, 1998. ISBN 0898714281.

[25] P. F. Dubois, A. Greenbaum, and G. H. Rodrigue. Approximating the inverse of a matrix for use in iterative algorithms on vector processors. *Computing*, 22(3) :257–268, Sept. 1979. ISSN 1436-5057. doi : 10.1007/BF02243566. URL https://doi.org/10.1007/BF02243566.

[26] I. S. Duff and G. A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT Numerical Mathematics*, 29(4) :635–657, Dec. 1989. ISSN 1572-9125. doi : 10.1007/BF01932738. URL https://doi.org/10.1007/BF01932738.

[27] I. S. Duff and H. A. van der Vorst. Developments and trends in the parallel solution of linear systems. *Parallel Computing*, 25(13) :1931–1970, 1999. ISSN 0167-8191. doi : https://doi.org/10.1016/S0167-8191(99)00077-0. URL https://www.sciencedirect.com/science/article/pii/S0167819199000770.

[28] I. S. Duff, A. M. Erisman, C. W. Gear, and J. K. Reid. Sparsity structure and Gaussian elimination. *ACM SIGNUM Newsletter*, 23(2) :2–8, Apr. 1988. ISSN 0163-5778. doi : 10.1145/47917.47918. URL https://doi.org/10.1145/47917.47918.

[29] E. F. D'Azevedo, P. A. Forsyth, and W.-P. Tang. Ordering Methods for Preconditioned Conjugate Gradient Methods Applied to Unstructured Grid Problems. *SIAM Journal on Matrix Analysis and Applications*, 13(3) :944–961, July 1992. ISSN 0895-4798. doi : 10.1137/0613057. URL https://epubs.siam.org/doi/abs/10.1137/0613057. Publisher : Society for Industrial and Applied Mathematics.

[30] R. Fletcher. Conjugate gradient methods for indefinite systems. In G. A. Watson, editor, *Numerical Analysis*, Lecture Notes in Mathematics, pages 73–89, Berlin, Heidelberg, 1976. Springer. ISBN 978-3-540-38129-7. doi : 10.1007/BFb0080116.

[31] M. J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers*, C-21(9) :948–960, Sept. 1972. ISSN 1557-9956. doi : 10.1109/TC.1972.5009071. Conference Name : IEEE Transactions on Computers.

[32] A. George. Nested Dissection of a Regular Finite Element Mesh. *SIAM Journal on Numerical Analysis*, 10(2) :345–363, Apr. 1973. ISSN 0036-1429. doi : 10.1137/0710032. URL https://epubs.siam.org/doi/10.1137/0710032. Publisher : Society for Industrial and Applied Mathematics.

[33] A. George, M. Heath, J. Liu, and E. Ng. Solution of sparse positive definite systems on a hypercube. *Journal of Computational and Applied Mathematics*, 27(1) :129–156, Sept. 1989. ISSN 0377-0427. doi : 10.1016/0377-0427(89)90364-6. URL http://www.sciencedirect.com/science/article/pii/0377042789903646.

[34] J. A. George. Computer Implementation of the Finite Element Method. Technical report, STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, Feb. 1971. URL https://apps.dtic.mil/sti/citations/AD0726171. Section : Technical Reports.

[35] N. E. Gibbs, J. Poole, William G., and P. K. Stockmeyer. An Algorithm for Reducing the Bandwidth and Profile of a Sparse Matrix. *SIAM Journal on Numerical Analysis*, 13(2) :236–250, Apr. 1976. ISSN 0036-1429. doi : 10.1137/0713023. URL https://epubs.siam.org/doi/abs/10.1137/0713023. Publisher : Society for Industrial and Applied Mathematics.

[36] P. Graves-Morris. The breakdowns of bicgstab. *Numerical Algorithms*, 29, 2002. doi : 10.1023/A:1014864007293. URL https://doi.org/10.1023/A:1014864007293.

[37] M. J. Grote and T. Huckle. Parallel Preconditioning with Sparse Approximate Inverses. *SIAM Journal on Scientific Computing*, 18(3) :838–853, May 1997. ISSN 1064-8275. doi : 10.1137/S1064827594276552. URL https://epubs.siam.org/doi/abs/10.1137/S1064827594276552. Publisher : Society for Industrial and Applied Mathematics.

[38] D. Grunwald and S. Vajracharya. Efficient barriers for distributed shared memory computers. pages 604 – 608, 05 1994. ISBN 0-8186-5602-6. doi : 10.1109/IPPS.1994.288242.

[39] I. Gustafsson. A class of first order factorization methods. *BIT Numerical Mathematics*, 18 (2) :142–156, June 1978. ISSN 1572-9125. doi : 10.1007/BF01931691. URL https://doi.org/10.1007/BF01931691.

[40] Y. Han and R. Finkel. An optimal scheme for disseminating information. *Proceedings of 1988 International Conference on Parallel Processing, Chicago*, pages 198–203, 1988. URL http://h.web.umkc.edu/hanyij/html/research/icpp88.pdf.

[41] J. L. Hennessy and D. A. Patterson. *Computer Architecture : A Quantitative Approach*. Elsevier, Oct. 2011. ISBN 978-0-12-383873-5.

[42] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1) :1–17, 1988. doi : 10.1007/BF01379320. URL https://doi.org/10.1007/BF01379320.

[43] M. R. Hestenes and E. Stiefel. Methods of conjugate gradients for solving linear systems. 1952. doi : 10.6028/JRES.049.044.

[44] T. Hoefler, T. Mehlan, F. Mietke, and W. Rehm. A Survey of Barrier Algorithms for Coarse Grained Supercomputers. Chemnitzer Informatik Berichte. 2004. URL http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.83.5369.

[45] Intel. Intel® Xeon Phi™ Coprocessor system software developers guide. (1), 2014. URL https://www.intel.com/content/dam/www/public/us/en/documents/product-briefs/xeon-phi-coprocessor-system-software-developers-guide.pdf.

[46] Intel. Intel® 64 and IA-32 Architectures Software Developer's Manual. Technical report, 2021. URL https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html.

[47] T. Iwashita, H. Nakashima, and Y. Takahashi. Algebraic block multi-color ordering method for parallel multi-threaded sparse triangular solver in iccg method. *2012 IEEE 26th International Parallel and Distributed Processing Symposium*, pages 474–483, 2012.

[48] T. Iwashita, S. Li, and T. Fukaya. Hierarchical block multi-color ordering : A new parallel ordering method for vectorization and parallelization of the sparse triangular solver in the ICCG method. *CoRR*, abs/1908.00741, 2019. URL http://arxiv.org/abs/1908.00741.

[49] E. E. Jarlebring, G. Mele, E. Ringh, D. Ek, F. Izzo, P. Upadhyaya, and E. Jarlebring. *Preconditioning for linear systems*. Independently published, May 2020. ISBN 9798646306334.

[50] M. T. Jones and P. E. Plassmann. The Efficient Parallel Iterative Solution of Large Sparse Linear Systems. In A. George, J. R. Gilbert, and J. W. H. Liu, editors, *Graph Theory and Sparse Matrix Computation*, pages 229–245, New York, NY, 1993. Springer New York. ISBN 978-1-4613-8369-7.

[51] L. Y. Kolotilina and A. Y. Yeremin. Factorized Sparse Approximate Inverse Preconditionings I. Theory. *SIAM Journal on Matrix Analysis and Applications*, 14(1) :45–58, Jan. 1993. ISSN 0895-4798. doi : 10.1137/0614004. URL https://epubs.siam.org/doi/abs/10.1137/0614004. Publisher : Society for Industrial and Applied Mathematics.

[52] M. Kreutzer, G. Hager, G. Wellein, H. Fehske, and A. R. Bishop. A unified sparse matrix data format for efficient general sparse matrix-vector multiplication on modern processors with wide simd units. *SIAM Journal on Scientific Computing*, 36(5) :C401–C423, 2014. doi : 10.1137/130930352. URL https://doi.org/10.1137/130930352.

[53] C. Lanczos. Solution of systems of linear equations by minimized iterations. *Journal of Research of the National Bureau of Standards*, 49(1) :33, July 1952. ISSN 0091-0635. doi : 10.6028/jres.049.006. URL https://nvlpubs.nist.gov/nistpubs/jres/049/jresv49n1p33_A1b.pdf.

[54] V. Lee. Avx512 slower than avx2 with intel mkl dgemm on intel gold 5118, 07 2019. URL https://community.intel.com/t5/Software-Tuning-Performance/AVX512-slower-than-AVX2-with-Intel-MKL-dgemm-on-Intel-Gold-5118/td-p/1135951.

[55] K.-W. Lih. *Equitable Coloring of Graphs*, pages 1199–1248. Springer New York, New York, NY, 2013. ISBN 978-1-4419-7997-1. doi : 10.1007/978-1-4419-7997-1_25. URL https://doi.org/10.1007/978-1-4419-7997-1_25.

[56] L. O. Mafteiu-Scai. The bandwidths of a matrix. a survey of algorithms. *Annals of West University of Timisoara - Mathematics and Computer Science*, 52(2) :183–223, 2015. doi : doi:10.2478/awutm-2014-0019. URL https://doi.org/10.2478/awutm-2014-0019.

[57] J. A. Meijerink and H. A. van der Vorst. An Iterative Solution Method for Linear Systems of Which the Coefficient Matrix is a Symmetric M-Matrix. *Mathematics of Computation*, 31

(137) :148–162, 1977. ISSN 0025-5718. doi : 10.2307/2005786. URL http://www.jstor.org/stable/2005786. Publisher : American Mathematical Society.

[58] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. *ACM Trans. Comput. Syst.*, 9(1) :21–65, Feb. 1991. ISSN 0734-2071. doi : 10.1145/103727.103729. URL https://doi.org/10.1145/103727.103729.

[59] W. Meyer. Equitable coloring. *The American Mathematical Monthly*, 80(8) :920–922, 1973. doi : 10.1080/00029890.1973.11993408. URL https://doi.org/10.1080/00029890.1973.11993408.

[60] E. Moore. *The Shortest Path Through a Maze*. Bell Telephone System. Technical publications. monograph. Bell Telephone System., 1959. URL https://books.google.fr/books?id=IVZBHAAACAAJ.

[61] R. Nanjegowda, O. Hernandez, B. Chapman, and H. H. Jin. Scalability Evaluation of Barrier Algorithms for OpenMP. In M. S. Müller, B. R. de Supinski, and B. M. Chapman, editors, *Evolving OpenMP in an age of extreme parallelism*, pages 42–52, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg. ISBN 978-3-642-02303-3.

[62] NVIDIA. NVIDIA CUDA Compute Unified Device Architecture. Technical Report Version 1.0, June 2007. URL http://developer.download.nvidia.com/compute/cuda/1.0/NVIDIA_CUDA_Programming_Guide_1.0.pdf.

[63] G. F. Pfister and V. Norton. "hot-spot" contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, 34(10) :943–948, oct 1985. ISSN 1557-9956. doi : 10.1109/TC.1985.6312198.

[64] E. Rigtorp. Aligned AVX loads and stores are atomic, June 2020. URL https://rigtorp.se/isatomic/.

[65] A. Rodchenko, A. Nisbet, A. Pop, and M. Luján. Effective barrier synchronization on intel xeon phi coprocessor. In J. L. Träff, S. Hunold, and F. Versaci, editors, *Euro-Par 2015 : Parallel Processing*, pages 588–600, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg. ISBN 978-3-662-48096-0.

[66] A. Roussel. *Parallelization of iterative methods to solve sparse linear systems using task based runtime systems on multi and many-core architectures : application to Multi-Level Domain Decomposition methods*. Theses, Université Grenoble Alpes, Feb. 2018. URL https://tel.archives-ouvertes.fr/tel-01753992.

[67] Y. Saad. Highly Parallel Preconditioners for General Sparse Matrices. In G. Golub, M. Luskin, and A. Greenbaum, editors, *Recent Advances in Iterative Methods*, The IMA Volumes in Mathematics and its Applications, pages 165–199, New York, NY, 1994. Springer. ISBN 978-1-4613-9353-5. doi : 10.1007/978-1-4613-9353-5_11.

[68] Y. Saad. *Iterative Methods for Sparse Linear Systems : Second Edition*. SIAM, Apr. 2003. ISBN 978-0-89871-534-7. Google-Books-ID : qtzmkzzqFmcC.

[69] Y. Saad and M. H. Schultz. GMRES : A Generalized Minimal Residual Algorithm for Solving Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 7(3) : 856–869, July 1986. ISSN 0196-5204. doi : 10.1137/0907058. URL https://epubs.siam.org/doi/abs/10.1137/0907058. Publisher : Society for Industrial and Applied Mathematics.

[70] Y. Saad and H. A. van der Vorst. Iterative solution of linear systems in the 20th century. *Journal of Computational and Applied Mathematics*, 123(1) :1–33, Nov. 2000. ISSN 0377-0427. doi : 10.1016/S0377-0427(00)00412-X. URL http://www.sciencedirect.com/science/article/pii/S037704270000412X.

[71] S. S. Skiena. *Sorting and Searching*, pages 103–144. Springer London, London, 2008. ISBN 978-1-84800-070-4. doi : 10.1007/978-1-84800-070-4_4. URL https://doi.org/10.1007/978-1-84800-070-4_4.

[72] S. W. Sloan. An algorithm for profile and wavefront reduction of sparse matrices. *International Journal for Numerical Methods in Engineering*, 23(2) :239–251, 1986. doi : https://doi.org/10.1002/nme.1620230208. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/nme.1620230208.

[73] SPE. The 10th spe comparative solution project, 2000. URL http://www.spe.org/web/csp/datasets/set02.htm.

[74] E. Speziale, A. di Biagio, and G. Agosta. An optimized reduction design to minimize atomic operations in shared memory multiprocessors. In *2011 IEEE International Symposium on Parallel and Distributed Processing Workshops and Phd Forum*, pages 1300–1309, 2011. doi : 10.1109/IPDPS.2011.271.

[75] K. Suzuki, T. Fukaya, and T. Iwashita. A novel ilu preconditioning method with a block structure suitable for simd vectorization. *Journal of Computational and Applied Mathematics*, 419 :114687, 2023. ISSN 0377-0427. doi : https://doi.org/10.1016/j.cam.2022.114687. URL https://www.sciencedirect.com/science/article/pii/S0377042722003478.

[76] H. A. van der Vorst. Bi-CGSTAB : A Fast and Smoothly Converging Variant of Bi-CG for the Solution of Nonsymmetric Linear Systems. *SIAM Journal on Scientific and Statistical Computing*, 13(2) :631–644, Mar. 1992. ISSN 0196-5204. doi : 10.1137/0913035. URL https://epubs.siam.org/doi/10.1137/0913035. Publisher : Society for Industrial and Applied Mathematics.

[77] O. Villa, G. Palermo, and C. Silvano. Efficiency and Scalability of Barrier Synchronization on NoC Based Many-Core Architectures. In *Proceedings of the 2008 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, CASES'08, page 81–90, New York, NY, USA, 2008. Association for Computing Machinery. ISBN 9781605584690. doi : 10.1145/1450095.1450110. URL https://doi.org/10.1145/1450095.1450110.

[78] P. K. W. Vinsome. Orthomin, an Iterative Method for Solving Sparse Sets of Simultaneous Linear Equations. Society of Petroleum Engineers, Jan. 1976. ISBN 978-1-55563-751-4. doi : 10.2118/5729-MS. URL https://www.onepetro.org/conference-paper/SPE-5729-MS.

[79] C. Vuik. Krylov Subspace Solvers and Preconditioners. *ESAIM : Proceedings and Surveys*, 63 : 1–43, 2018. ISSN 2267-3059. doi : 10.1051/proc/201863001. URL https://www.esaim-proc.org/articles/proc/abs/2018/03/proc186301/proc186301.html. Publisher : EDP Sciences.

[80] P.-C. Yew, N.-F. Tzeng, and D. H. Lawrie. *Distributing Hot-Spot Addressing in Large-Scale Multiprocessors*, page 302–309. IEEE Computer Society Press, Washington, DC, USA, 1994. ISBN 0818661976. doi : 10.5555/201173.201223. URL https://dl.acm.org/doi/abs/10.5555/201173.201223.

[81] Z. Yi, F. Chen, and Y. Yao. A barrier optimization framework for NUMA multi-core system. *Concurrency and Computation : Practice and Experience*, 32(5) :e5527, 2020. doi : https://doi.org/10.1002/cpe.5527. URL https://onlinelibrary.wiley.com/doi/abs/10.1002/cpe.5527. e5527 cpe.5527.

[82] D. P. Young, R. G. Melvin, F. T. Johnson, J. E. Bussoletti, L. B. Wigton, and S. S. Samant. Application of Sparse Matrix Solvers as Effective Preconditioners. *SIAM Journal on Scientific and Statistical Computing*, 10(6) :1186–1199, Nov. 1989. ISSN 0196-5204. doi : 10.1137/0910072. URL https://epubs.siam.org/doi/abs/10.1137/0910072. Publisher : Society for Industrial and Applied Mathematics.