# Un modèle de programmation à grain fin pour la parallélisation de solveurs linéaires creux

Corentin Rossignon

université de BORDEAUX

# THÈSE

présentée pour obtenir le grade de

DOCTEUR DE L'UNIVERSITÉ DE BORDEAUX

École doctorale de Mathématiques et Informatique de Bordeaux
Spécialité : Informatique

présentée par

**Corentin ROSSIGNON**

---

# Un modèle de programmation à grain fin pour la parallélisation de solveurs linéaires creux

---

Directeur de thèse : Raymond NAMYST
Co-direction : Olivier AUMAGE et Samuel THIBAULT
Encadrant dans l'entreprise (Total S.A.) : Pascal HÉNON

Soutenue le 17 juillet 2015 avec pour jury :

| | | |
|---|---|---|
| M. Raymond NAMYST | Professeur des universités, LaBRI Bordeaux | Directeur |
| M. Olivier AUMAGE | Chargé de recherche Inria, LaBRI Bordeaux | Co-Directeur |
| M. Pascal HÉNON | Chercheur, Total Pau | Co-Directeur |
| M. Jean-François MÉHAUT | Professeur des universités, LIG Grenoble | Rapporteur |
| M. Jean-Yves L'EXCELLENT | Chargé de recherche Inria, LIP Lyon | Rapporteur |
| M. François BODIN | Professeur des universités, Irisa Rennes | Examinateur |
| M. George BOSILCA | Directeur de recherche, ICL Knoxville (USA) | Examinateur |

# TABLE DES MATIÈRES

# RÉSUMÉ

## Un modèle de programmation à grain fin pour la parallélisation de solveurs linéaires creux

**Mots-clés :** parallélisme, graphe de tâches, supports d'exécution, NUMA, multi-coeurs, algèbre linéaire creuse

La résolution de grands systèmes linéaires creux est un élément essentiel des simulations numériques. Ces résolutions peuvent représenter jusqu'à 80% du temps de calcul des simulations.

Une parallélisation efficace des noyaux d'algèbre linéaire creuse conduira donc à obtenir de meilleures performances. En mémoire distribuée, la parallélisation de ces noyaux se fait le plus souvent en modifiant le schéma numérique. Par contre, en mémoire partagée, un parallélisme plus efficace peut être utilisé. Il est donc important d'utiliser deux niveaux de parallélisme, un premier niveau entre les noeuds d'une grappe de serveur et un deuxième niveau à l'intérieur du noeud. Lors de l'utilisation de méthodes itératives en mémoire partagée, les graphes de tâches permettent de décrire naturellement le parallélisme en prenant comme granularité le travail sur une ligne de la matrice. Malheureusement, cette granularité est trop fine et ne permet pas d'obtenir de bonnes performances à cause du surcoût de l'ordonnanceur de tâches.

Dans cette thèse, nous étudions le problème de la granularité pour la parallélisation par graphe de tâches. Nous proposons d'augmenter la granularité des tâches de calcul en créant des agrégats de tâches qui deviendront eux-mêmes des tâches. L'ensemble de ces agrégats et des nouvelles dépendances entre les agrégats forme un graphe de granularité plus grossière. Ce graphe est ensuite utilisé par un ordonnanceur de tâches pour obtenir de meilleurs résultats. Nous utilisons comme exemple la factorisation LU incomplète d'une matrice creuse et nous montrons les améliorations apportées par cette méthode. Puis, dans un second temps, nous nous concentrons sur les machines à architecture NUMA. Dans le cas de l'utilisation d'algorithmes limités par la bande passante mémoire, il est intéressant de réduire les effets NUMA liés à cette architecture en plaçant soi-même les données. Nous montrons comment prendre en compte ces effets dans un intergiciel à base de tâches pour ainsi améliorer les performances d'un programme parallèle.

# ABSTRACT


## A fine grain model programming for parallelization of sparse linear solver



**Keywords :** parallelism, task-based programming, runtime, NUMA, multicore, sparse linear algebra



Solving large sparse linear system is an essential part of numerical simulations. These resolve can take up to 80% of the total of the simulation time.

An efficient parallelization of sparse linear kernels leads to better performances. In distributed memory, parallelization of these kernels is often done by changing the numerical scheme. Contrariwise, in shared memory, a more efficient parallelism can be used. It's necessary to use two levels of parallelism, a first one between nodes of a cluster and a second inside a node.

When using iterative methods in shared memory, task-based programming enables the possibility to naturally describe the parallelism by using as granularity one line of the matrix for one task. Unfortunately, this granularity is too fine and doesn't allow to obtain good performance.

In this thesis, we study the granularity problem of the task-based parallelization. We offer to increase grain size of computational tasks by creating aggregates of tasks which will become tasks themself. The new coarser task graph is composed by the set of these aggregates and the new dependencies between aggregates. Then a task scheduler schedules this new graph to obtain better performance. We use as example the Incomplete LU factorization of a sparse matrix and we show some improvements made by this method. Then, we focus on NUMA architecture computer. When we use a memory bandwidth limited algorithm on this architecture, it is interesting to reduce NUMA effects. We show how to take into account these effects in a task-based runtime in order to improve performance of a parallel program.

# REMERCIEMENTS

Je souhaite remercier dans un premier temps mes encadrants de thèse, Pascal Hénon, Olivier Aumage, Samuel Thibault et Raymond Namyst, qui m'ont soutenu tout au long de ces trois années de thèse. Grace à eux, j'ai pu approfondir mes connaissances dans plusieurs domaines scientifiques. Je tiens aussi à remercier les rapporteurs de ce manuscrit, Jean-Yves L'excellent et Jean-François Méhaut, qui ont su fournir des remarques constructives sur le manuscrit et ainsi permettre d'améliorer la compréhension de celui-ci. J'aimerais aussi remercier les autres membres du jury de thèse, François Bodin et George Bosilca, qui ont accepté de faire le déplacement jusqu'à Bordeaux pour assister à ma soutenance de thèse.

Je remercie toutes les personnes avec qui j'ai travaillé durant mes trois ans dans l'équipe simulation de réservoir chez Total. J'ai vraiment apprécié les discussions de *geek* avec Pascal, il était aussi très plaisant de pouvoir parler informatique bas niveau avec Stephan et Éric. J'ai aussi eu la chance d'avoir Bernard et Gilles qui sont des managers hors pair. Les discussions endiablées du midi entre Romain et Léonardo me manquent déjà. Je remercie aussi mes deux co-bureau, Martin et Guillaume, qui m'ont soutenu durant cette dernière phase d'écriture en détendant l'atmosphère. Durant ces trois ans, j'ai pu rencontrer un certain nombre de personnes toutes aussi intéressantes les unes que les autres et la plupart sont même devenues des amis (Giulano, Raffik, Brice, Quentin, Yvann, Cyril, François, Sylvain, Nicolas et Malik), j'espère pouvoir tous les revoir. Je remercie aussi tous ceux que j'ai oublié dans ces remerciements (Rabeb, Philippe, Alexandre, Guilhem ...).

Ma famille aussi m'a beaucoup soutenu durant ces trois ans. Je remercie mes parents pour m'avoir permis de faire de longues études grâce à leurs encouragements du début jusqu'à la fin. Merci à mes soeurs (Amandine, Marine et Laura) ainsi qu'à ma nièce Louna pour avoir été là. Merci aussi à mes oncles, tantes, cousins et cousines d'être là dès que l'on a besoin d'eux.

Je souhaite aussi remercier tous les amis que je me suis fait sur Pau. Ces trois années sur Pau ont été inoubliables et j'ai eu la chance de rencontrer de vrais et nombreux amis. Je garde en tête les fameuses séances de *bodyweight* de Jeremy, avec Jonathan et Élodie dans lesquelles on travaillait plus les abdos que le reste à force de rire. Ainsi que les séances jeu de rôle avec nos maîtres du jeu favoris Nico et Guillaume, sans oublier les joueurs : Rémi, Sylvain, Emy, Lisa, Ambre et Gérald. L'équipe du "basket pour les nuls" composée en autres d'Élodie la tueuse de chevilles, Magic Benji dit la muraille et Maxence, qui permettait de bien s'amuser et se défouler. J'ai eu la chance de faire un magnifique voyage à la Réunion pour le premier de l'an avec Nathalie, Jonathan, Ingrid, Stéphanie et Josie. Je garde aussi un excellent souvenir de bon nombre de personnes que j'ai rencontrés durant ces trois ans (Kevin, Aglaé, Olivier, Julien ...).

# CHAPITRE 1

# INTRODUCTION

## 1.1 Contexte

Cette thèse a été effectuée dans le cadre d'un contrat CIFRE avec la société Total. Total est une société pétrolière dont l'activité est séparée en trois secteurs :

- le secteur *amont*, dont le but est d'extraire les hydrocarbures des ressources naturelles ;
- le secteur *raffinage-chimie* qui traitera les hydrocarbures pour les transformer en produits utilisables dans la vie courante ;
- le secteur *marketing et service* qui s'occupera de la vente de ces produits.

Tous ces secteurs utilisent des outils informatiques, mais le secteur *amont* est le secteur qui a le plus besoin de puissance de calcul. Cette puissance est utilisée pour modéliser le sous-sol à partir de données sismiques, puis à simuler l'extraction du pétrole. C'est pourquoi ce secteur renouvelle régulièrement ses supercalculateurs pour toujours être à la pointe de la technologie. Sans logiciels spécifiques, ces supercalculateurs ne permettent pas d'être utilisés au maximum de leurs capacités. Pour cela, il est aussi nécessaire de renouveler les logiciels utilisés ou au moins les modifier pour qu'ils puissent utiliser des nouvelles techniques de programmation qui pourront mieux exploiter ces supercalculateurs.

## 1.2 Objectifs

Le calcul haute performance est essentiel à Total pour pouvoir optimiser l'exploitation des champs pétrolifères. C'est dans ce cadre là que j'ai intégré l'équipe du projet R&D de la simulation de réservoir afin d'aider à développer de nouvelle techniques de programmation parallèle. Nous nous sommes concentrés sur la partie algèbre linéaire creuse du simulateur de réservoir. Ces nouvelles techniques de programmation permettront de paralléliser efficacement des noyaux d'algèbre linéaire creux. En effet, la partie résolution de systèmes linéaires creux représente une part importante du temps de calcul total du logiciel de simulation de réservoir. Un de nos objectifs est de pouvoir paralléliser efficacement cette partie du logiciel sans changer les méthodes numériques utilisées. Pour cela, nous allons nous consacrer à la parallélisation de certains noyaux d'algèbre linéaire creuse tels que la factorisation ILU ou bien le produit matrice vecteur creux. L'objectif principal de cette thèse est donc de réussir à exploiter au mieux toute la puissance d'une machine lors de l'utilisation de ces noyaux d'algèbre linéaire creuse. Pour cela, nous avons eu besoin d'améliorer les techniques de programmation parallèle existantes en utilisant de nouveaux outils que nous avons développés.

## 1.3 Problèmes

Les techniques de parallélisation utilisées actuellement dans les codes de calcul industriels ne permettent pas toujours d'utiliser pleinement les capacités des nouveaux supercalculateurs. Il existe des cas, comme l'algèbre linéaire creuse, où ces outils ne donnent pas de bonnes performances.

Les réseaux faible latence-haut débit, permettant de relier les noeuds d'une grappe de serveurs, sont en constante amélioration grâce à l'utilisation de nouvelles technologies comme la fibre optique par exemple. Malgrès ces évolutions, l'utilisation seule du paradigme de programmation par passage de messages n'est plus suffisante pour exploiter toutes les capacités des nouvelles machines. Il peut aussi être utilisé conjointement à un autre type de parallélisme permettant ainsi de bénéficier des avantages de chaque paradigme. Le paradigme de programmation parallèle à base de threads permet d'exploiter plus efficacement l'architecture des processeurs actuels. En effet, ceux-ci ont radicalement changés depuis le début des années 2000, avec la multiplication des coeurs de calcul sur la même puce. Ces coeurs doivent partager plusieurs ressources

matérielles, et parmi ces ressources, il y en a une qui est vraiment critique pour les performances, il s'agit de la mémoire. L'utilisation des threads permet de profiter de la mémoire partagée ainsi que des différents niveaux de cache partagés. Il est possible d'écrire de nouveaux algorithmes qui prendront en compte cette spécificité pour en tirer avantage. Dans le but d'exploiter au mieux la machine, il devient nécessaire de combiner un paradigme de parallélisation par passage de messages avec un paradigme de parallélisation multi-threads.

La programmation parallèle à l'aide de threads peut se faire à l'aide de différentes techniques. Chaque technique ayant ses qualités et ses défauts, son choix dépendra essentiellement de l'algorithme à paralléliser. Par exemple, la parallélisation à base de graphe de tâches sera très efficace pour décrire des problèmes avec des dépendances de donnés mais imposera au programmeur de définir une granularité correcte pour son problème. Alors qu'au contraire, un parallélisme de boucle utilise des mécanismes pour optimiser le grain de calcul mais ne permet de décrire des dépendances de données complexes.

Pour des raisons de passage à l'échelle, la mémoire est distribuée physiquement et les coeurs de calcul ne pourrons y accéder de manière uniforme. Les temps d'accès à une donnée dépendront de l'emplacement physique de la donnée et de celui qui souhaite y accéder. Ces machines sont appelées NUMA et permettent de conserver de bonnes performances sur les accès mémoires même avec un grand nombre de coeurs de calcul. Mais cette architecture impose au système d'exploitation de bien choisir le placement des données. Ce choix se fait à l'aide de politiques d'allocation mémoire, et un mauvais choix peut affecter significativement les performances.

## 1.4 Contributions

Durant cette thèse, nous avons développé un cadriciel, nommé Taggre, qui nous permet de résoudre l'un des problèmes majeurs de la parallélisation par graphe de tâches : la granularité des tâches. La gestion des tâches ayant un surcoût, il est nécessaire d'effectuer suffisamment de calcul dans une tâche pour pouvoir le négliger. Mais, il faut aussi faire attention au parallélisme en ayant assez de tâches pour occuper tous les coeurs de calcul. Taggre travaillera à transformer le graphe de tâches afin que ces deux critères soient respectés.

Nous avons aussi étudié le placement mémoire sur machines NUMA. Nous utilisons le graphe de tâches pour définir le placement des données sur les différents bancs mémoires. En connaissant les données utilisées par une tâche, nous utilisons une heuristique permettant d'équilibrer la charge des accès mémoire au cours du déroulement du graphe de tâches. Une fois les données correctement réparties, il faut encore que l'ordonnanceur de tâches utilise cette information lors de l'ordonnancement. C'est pourquoi nous proposons une méthode d'ordonnancement des tâches prenant en compte les effets NUMA. Et nous comparons cette méthode aux différentes politiques d'allocation mémoire existantes.

## 1.5 Plan du manuscrit

Ce manuscrit de thèse est composé de 4 chapitres. Dans le premier chapitre, nous introduisons le contexte dans lequel ce travail a été réalisé. Pour cela, nous présentons la simulation de réservoir, puis nous étudions un cas très simple de simulation physique. Ce qui nous permet de présenter l'algèbre linéaire et d'expliquer en quoi l'algèbre linéaire peut être utile lors des simulations physiques. A la suite de de cela, plusieurs méthodes de résolutions de système linéaire sont présentées. Le manuscrit s'oriente ensuite un peu plus vers l'informatique en décrivant l'évolution des architectures au fil du temps. Pour en arriver finalement au méthodes de programmation parallèle permettant d'utiliser les architectures modernes.

Le deuxième chapitre est consacré à la programmation à base de graphe de tâches. Nous prendrons comme cas d'étude la parallélisation d'une factorisation ILU [1] en mémoire partagée. Nous montrons les problèmes liés à la programmation à base de graphe de tâches dans le cas où les tâches sont trop fines. Nous proposerons ensuite une solution à ce problème. Cette solution est composée d'un cadriciel nommé Taggre et de plusieurs heuristiques permettant de modifier la granularité d'un graphe de tâches. Nous avons appliqué cette solution à la factorisation ILU et nous présenterons les résultats.

1. Incomplete LU

Le troisième chapitre se concentrera sur la gestion de la mémoire dans les machines NUMA. Nous montrerons les limites de notre problème en terme de bande passante mémoire, puis nous étudierons les différents mécanismes permettant de gérer la mémoire physique sur des machines NUMA. Nous appliquons ces différents mécanismes à notre problème et nous proposons une solution permettant d'obtenir de bons résultats.

Finalement, le dernier chapitre revient sur tous les résultats obtenus durant cette thèse. Puis nous suggérons des idées pour améliorer les solutions présentées.

# CHAPITRE 2

# CONTEXTE : PARALLÉLISATION DES NOYAUX DE CALCUL D'ALGÈBRE LINÉAIRE CREUSE

**Sommaire**



Ce chapitre a pour but d'expliquer des notions essentielles à la compréhension du manuscrit. De plus, cela permet de situer nos travaux par rapport à l'existant. Dans un premier temps, nous étudions la simulation de réservoir et à l'aide d'un exemple physique, nous regardons comment passer d'un modèle physique à un code de calcul. Cet exemple nous permet de commencer à nous intéresser à l'algèbre linéaire et plus particulièrement aux résolutions de systèmes linéaires creux en informatique. Puis nous passons en revu les évolutions des architectures des ordinateurs, pour finalement nous consacrer à l'étude des différents moyens de parallélisation existants. À la fin de ce chapitre, nous nous consacrons à l'étude des supports d'exécution pour la parallélisation par graphe de tâches.

## 2.1 La simulation de réservoir

Ce chapitre commence par une description succincte de la simulation de réservoir (section 2.1.1). Puis, à l'aide d'un exemple, nous montrons une approche permettant de transformer un problème physique, dont nous connaissons l'équation, en un système d'équations linéaires à résoudre (section 2.1.3).

### 2.1.1 Vue d'ensemble

Dans les profondeurs de la Terre, du pétrole et du gaz naturel se trouvent piégés. Ces sources d'énergie sont le résultat de la transformation de matière organique, provenant de végétaux et d'animaux morts, sous très fortes contraintes durant des millions d'années. Les compagnies pétrolières telles que Total ont pour but de localiser et d'optimiser l'extraction de ces ressources.

Plus généralement, nous appelons réservoir d'hydrocarbure, ou simplement réservoir, une concentration majeure de pétrole et/ou de gaz naturel sous le sol. La première étape pour trouver un réservoir consiste à analyser le sous-sol avec des ondes acoustiques. Ces ondes sont générées par des explosions pour les analyses sous la mer ou avec un camion sismique pour la surface de la Terre. Ces ondes sont ensuite analysées avec des logiciels de modélisation basés sur les équations des ondes, les compagnies pétrolières peuvent ainsi obtenir une bonne représentation du sous-sol. Après analyse de cette représentation, des puits d'explorations sont forés pour s'assurer de la présence de pétrole. Quand un réservoir est trouvé, l'une des premières questions est : "Est-il rentable d'exploiter ce réservoir ?". La simulation de réservoir est un des outils permettant de répondre à cette question. En faisant une simulation de fluides en milieu poreux, les compagnies pétrolières peuvent obtenir une approximation de la quantité d'huile pouvant être récupérée. Si l'exploitation du réservoir est rentable, les compagnies pétrolières commencent l'exploitation.

Mais il ne suffit pas de creuser et d'attendre que le pétrole jaillisse sous la forme d'un geyser comme nous pouvons le voir dans certains cartoons. Les compagnies pétrolières doivent installer des puits. Les puits peuvent être catégorisés en deux catégories majeures (Fig. 2.1) :

— les puits injecteurs, ce sont les puits qui augmentent la pression à l'intérieur du réservoir en injectant de la matière (eau, gaz($CO_2$), polymère ...) ;
— les puits producteurs, ce sont les puits qui récupèrent l'huile, ils sont aussi essentiels au contrôle de la pression en produisant plus ou moins d'huile et de gaz.

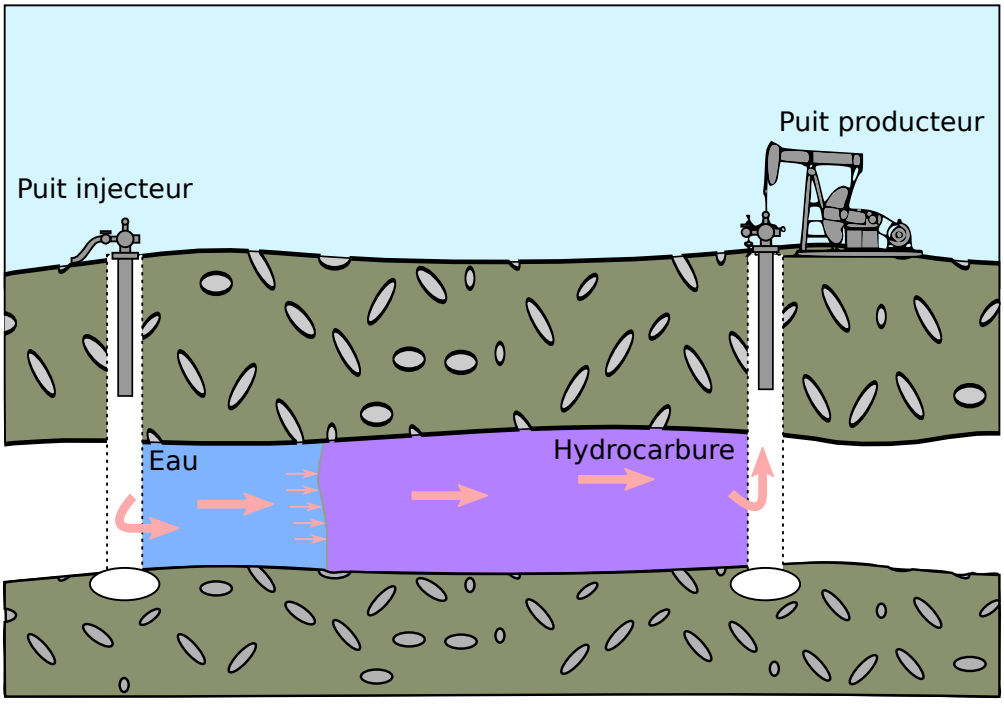



FIGURE 2.1 – Exemple d'un champ avec deux puits.

L'une des activités des ingénieurs réservoir consiste à trouver le nombre optimal de puits ainsi que

l'emplacement optimal de chacun pour exploiter le réservoir. Encore une fois, la simulation de réservoir leur permet de tester différentes configurations de placement des puits. Plus tard, quand la compagnie pétrolière commence à exploiter le champ, il est intéressant d'avoir des prévisions sur la production d'huile. Une fois de plus, ces résultats sont obtenus avec une simulation de réservoir (Fig. 2.2).

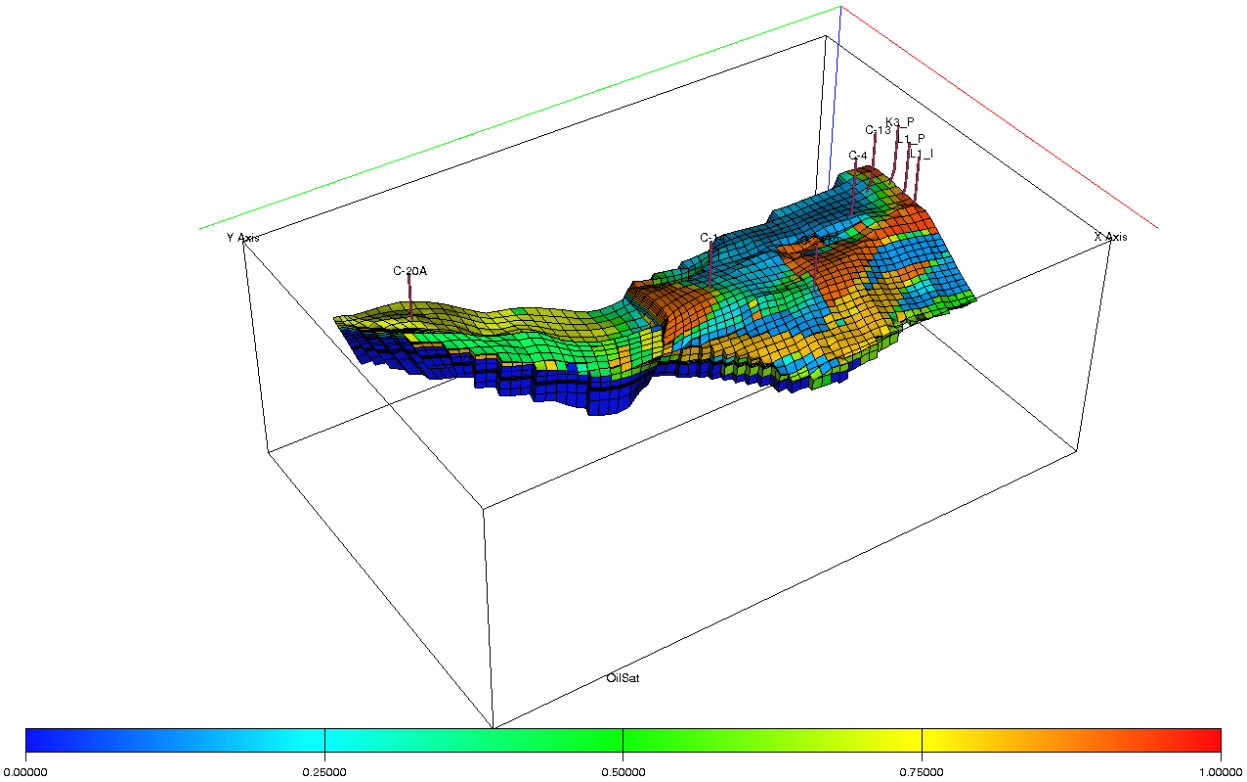



FIGURE 2.2 – Visualisation de la quantité d'huile dans le réservoir, l'image provient du logiciel floviz. L'échelle représente la part d'huile dans chaque cellule, cette valeur est normalisée entre 0 (0 %) et 1 (100 %).

Comme montré précédemment, la simulation de réservoir est une étape clé dans le processus de récupération d'hydrocarbures. Il est intéressant pour une compagnie pétrolière de simuler de plus en plus précisément l'intérieur d'un réservoir et bien sûr le simuler aussi vite que possible. Focalisons-nous sur la structure interne d'un simulateur de réservoir.

### 2.1.2 De la physique au calcul informatique

Pour pouvoir faire une simulation de réservoir, tout commence par un physicien qui modélise un écoulement de fluide en milieu poreux. La plupart de ces modèles sont basés sur trois équations physiques : la conservation de la masse, la loi des gaz parfaits et la loi de Darcy. Puis le réservoir est discrétisé en cellules en utilisant la méthode des volumes finis. Pour chaque cellule du réservoir, nous obtenons une équation non-linéaire par variable primaire (ex. : pression, saturation en huile ...). Pour résoudre le système d'équations non-linéaires nous utilisons la méthode de Newton aussi appelée la méthode de Newton-Raphson. Cette méthode est itérative, nous démarrons avec une valeur initiale $X_0$ suffisamment proche de la solution $X_n$ qui satisfait $F(X_n) = 0$. La méthode NewtonRaphson nous garantit que chaque itération nous rapproche du minimum local sous condition que le hessien soit défini positif.

L'exemple de la figure 2.3 n'est que dans une seule dimension, mais la même approche peut être utilisée quand on travaille avec un nombre arbitraire de dimensions. En simulation de réservoir, il y a autant de dimensions que de cellules dans le réservoir. Les équations linéaires de la simulation de réservoir peuvent être représentées sous la forme d'une matrice creuse très grande, nous reviendrons sur la définition de creuse plus loin dans ce manuscrit. Dans cette matrice, chaque ligne représente les interactions des éléments d'une cellule avec les éléments de son voisinage direct. Donc, si nous prenons un cube 3D régulier, les inconnues d'une cellule pourrons interagir avec les inconnues des 6 cellules voisines. Chaque interaction est représentée

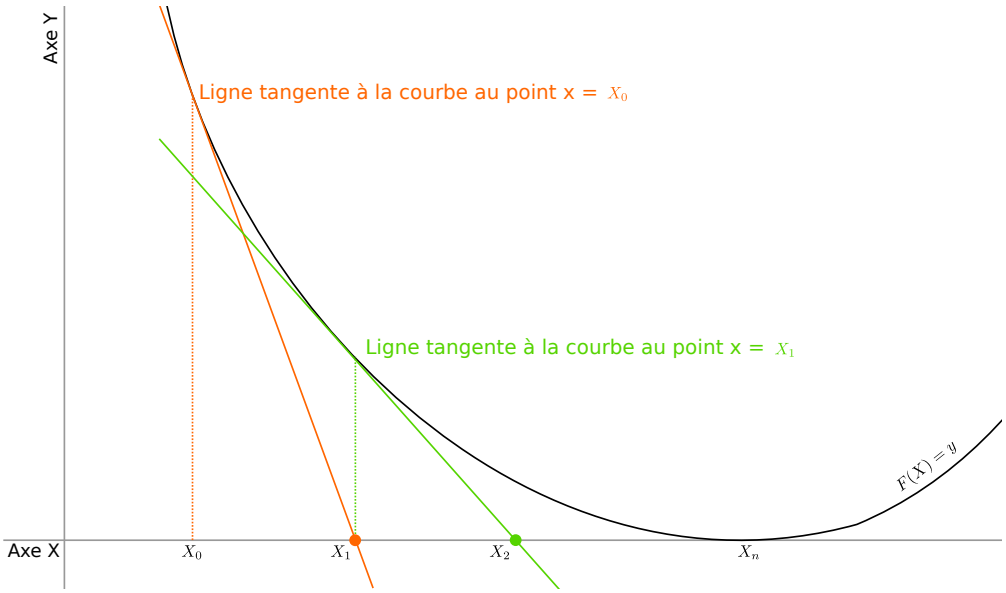



FIGURE 2.3 – Exemple de deux étapes de Newton en une dimension. Chaque tangente correspond à une équation linéaire à résoudre. Nous recherchons $X_n$ et nous démarrons au point $X_0$.

sous la forme d'une petite matrice dense dont la taille correspond au nombre de variables physiques simulées. Par la suite, un solveur de systèmes d'algèbre linéaire creux est utilisé.

### 2.1.3 Simulation d'un exemple physique simple

Prenons un exemple physique simple pour avoir une idée de comment l'algèbre linéaire peut être utilisée dans une simulation. L'exemple présenté est trivial et peut être résolu directement sans utiliser l'algèbre linéaire, mais il permet de comprendre facilement l'utilisation de l'algèbre linéaire dans un code de simulation numérique.

Nous souhaitons simuler une colonne remplie d'huile. Nous connaissons la densité de l'huile contenue dans la colonne : $\rho = 0.9192\ kg/m^3$. Nous connaissons aussi l'équation physique de la pression hydrostatique :

$$\frac{\mathrm{d}P}{\mathrm{d}z} = \rho g \tag{2.1}$$

Dans l'équation (2.1), $P$ désigne la pression, $z$ la profondeur et $g$ l'accélération gravitationnelle. En utilisant le théorème de Taylor au premier ordre, nous obtenons l'équation :

$$P(z_0 + h) = P(z_0) + h\frac{\mathrm{d}P}{\mathrm{d}z}(z_0) + o(h^2) \tag{2.2}$$

$$\frac{\mathrm{d}P}{\mathrm{d}z}(z_0) = \frac{P(z_0 + h) - P(z_0)}{h} + o(h^2) \tag{2.3}$$

On discrétise le problème en $n$ cellules avec la méthode des différences finies (Fig. 2.4) : considérons $Z_i$ l'approximation de $z$ sur la cellule $i$ en son centre, $i$ allant de 0 à $n-1$. Chaque centre est séparé du centre de la cellule voisine d'une distance $h$ appelée $\Delta z$ :

$$\frac{\mathrm{d}P}{\mathrm{d}z}(Z_i) \approx \frac{P(Z_i) - P(Z_{i-1})}{\Delta z} \tag{2.4}$$

L'injection de (2.4) dans (2.1) conduit à :

$$\frac{P(Z_i) - P(Z_{i-1})}{\Delta z} = \rho g \tag{2.5}$$

$$P(Z_i) - P(Z_{i-1}) = \rho g \Delta z \tag{2.6}$$

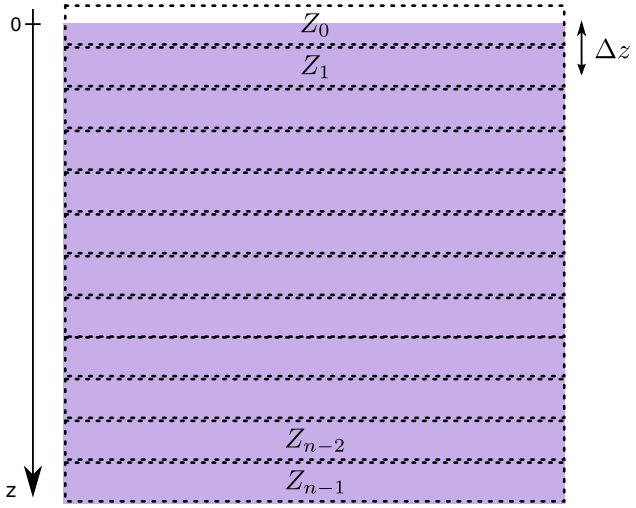



FIGURE 2.4 – Schéma de la colonne d'huile, il s'agit d'une discrétisation en $n$ cellules $Z_i$. Le centre de chaque cellule est séparé d'une distance $\Delta z$.

Nous avons aussi la condition limite qu'à la profondeur 0, correspondant au centre de la cellule $Z_0$, la pression est de 1000 hPa ou $10^5$ Pa :

$$P(Z_0) = 10^5 \tag{2.7}$$

Nous pouvons donc écrire le système entier sous la forme d'une matrice à $n$ lignes :

$$\begin{bmatrix} 1 & 0 & 0 & \cdots & \cdots & \cdots & \cdots & 0 \\ -1 & 1 & 0 & \ddots & & & & \vdots \\ 0 & -1 & 1 & 0 & \ddots & & & \vdots \\ \vdots & \ddots & \ddots & \ddots & \ddots & \ddots & & \vdots \\ \vdots & & \ddots & \ddots & \ddots & \ddots & \ddots & \vdots \\ \vdots & & & \ddots & -1 & 1 & 0 & 0 \\ \vdots & & & & \ddots & -1 & 1 & 0 \\ 0 & \cdots & \cdots & \cdots & \cdots & 0 & -1 & 1 \end{bmatrix} \begin{pmatrix} P(Z_0) \\ P(Z_1) \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ P(Z_{n-2}) \\ P(Z_{n-1}) \end{pmatrix} = \begin{pmatrix} 10^5 \\ \rho g \Delta z \\ \vdots \\ \vdots \\ \vdots \\ \vdots \\ \rho g \Delta z \\ \rho g \Delta z \end{pmatrix} \tag{2.8}$$

En multipliant chaque ligne de $A$ par le vecteur des $P(Z_i)$ pour $i$ allant de 0 à $n-1$, nous obtenons exactement le système d'équations (2.6). Ce système d'équations dont la forme générale est $A.x = b$ (où $A$ est une matrice inversible, $b$ un vecteur donné et $x$ le vecteur des inconnues que l'on cherche à déterminer), est la partie critique en temps CPU dans beaucoup de codes de simulation numérique.

## 2.2 Algèbre linéaire

### 2.2.1 Algèbre linéaire dense

Résoudre un système d'équations linéaires équivaut à résoudre un problème du type $Ax = b$ dans lequel $A$ est une matrice, $b$ est le vecteur second membre donné du système et $x$ est le vecteur que nous cherchons. Dans l'exemple de la simulation de colonne d'huile, nous avons une matrice triangulaire pour $A$, la solution peut donc être trouvée directement en résolvant chaque équation une à une en démarrant par $P(X_0) = 1000$. Mais ce n'est pas toujours aussi simple. Dans des cas plus difficiles, d'autres méthodes doivent être utilisées pour se ramener à la résolution de systèmes triangulaires comme l'élimination de Gauss-Jordan, l'élimination de variables ou bien la décomposition LU. Lorsque ces systèmes peuvent être résolus de manière approchée, on pourra aussi utiliser une méthode itérative plus économique en mémoire et en nombre d'opérations. Nous donnerons dans 2.3 plus de détail sur ces méthodes. Nous reviendrons plus tard sur l'une de ces méthodes :

la décomposition LU. Pour écrire le code de ces méthodes, nous avons besoin d'outils informatiques pour manipuler des matrices et des vecteurs.

En informatique, il existe plusieurs bibliothèques spécialisées dans les opérations d'algèbre linéaire dense. La plus connue est BLAS [1][LHKK79], c'est un ensemble d'opérations d'algèbre linéaire qui s'appliquent sur des vecteurs et des matrices. Ces opérations sont classées en 3 niveaux :

— Niveau 1 : ce sont les opérations sur les vecteurs (produit scalaire, addition de deux vecteurs ...) ;
— Niveau 2 : ce sont les opérations matrice-vecteur (multiplier une matrice par un vecteur, résoudre un système d'équations linéaires dont les coefficients sont dans une matrice triangulaire ...) ;
— Niveau 3 : ce sont les opérations matrice-matrice (multiplier une matrice par une autre matrice ...).

Le niveau d'un BLAS est directement lié à sa complexité en nombre d'opérations. Les BLAS de niveau 1 sont limités par la bande passante mémoire, il n'y a aucune réutilisation des données. Chaque donnée n'est utilisée qu'une seule fois, la seule optimisation possible se fait au niveau du prefetch mémoire. Les BLAS de niveau 2 peuvent réutiliser des données du vecteur, des optimisations peuvent être effectuées ici, par exemple il est possible de garder des parties du vecteur en cache. Les BLAS de niveau 3 ont une complexité plus grande ce qui permet d'avoir un plus grand nombre d'optimisations [WD98].

LAPACK [2][ABD+90] est une autre bibliothèque utilisée en algèbre linéaire, elle est construite par dessus BLAS. Les opérations faites dans BLAS et LAPACK sont bien optimisées, par exemple certaines implémentations utilisent la structuration en bloc qui permet d'optimiser la localité des données et de réduire les défauts de cache. D'autres implémentations utilisent les jeux d'instructions SIMD(SSE, AVX ...) des processeurs modernes [int09]. Des implémentations GPGPU [3] [nVi12] existent aussi, de même que des implémentations en mémoire distribuée [BBD+11]. La plupart de ces optimisations peuvent exister parce que le motif des accès mémoire des opérations du type BLAS est déterministe et que certaines opérations peuvent être réordonnées sans impacter le résultat final. Nous reviendrons sur certaines des notions utilisées ici dans les sections traitant de l'architecture des ordinateurs.

Retournons à la matrice de l'équation (2.8), nous pouvons voir que cette matrice contient un grand nombre de valeurs nulles et que ces valeurs n'ont aucun impact sur le calcul. Nous pouvons différencier les matrices composées en majorité de valeurs nulles, des matrices, qui au contraire, sont composées quasiment intégralement de valeurs non nulles. Une matrice peut être considérée comme creuse quand son nombre de valeurs non nulles est de l'ordre de la dimension de la matrice. Cette notion fait opposition aux matrices denses que nous avons l'habitude de manipuler. Les méthodes utilisées pour résoudre les systèmes linéaires creux sont différentes de celles utilisées pour les systèmes linéaires denses.

### 2.2.2 Algèbre linéaire creuse

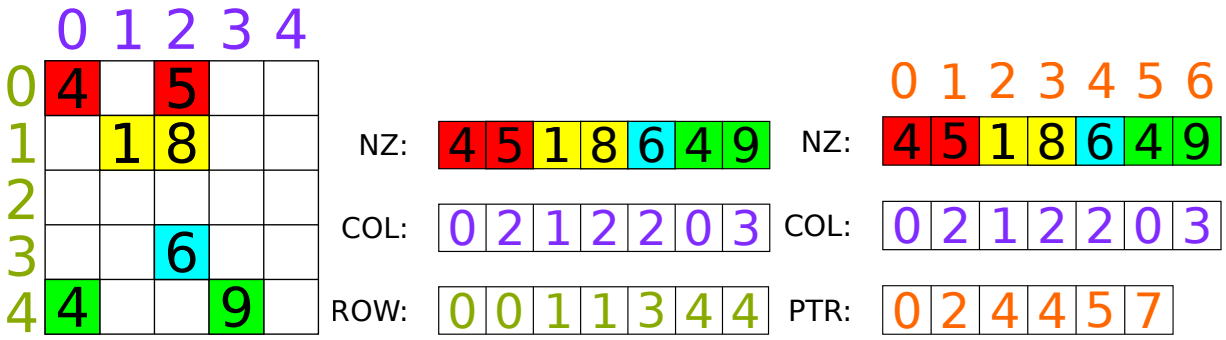
À la différence de l'algèbre linéaire dense, la majorité des calculs faits en creux sont irréguliers. C'est en partie dû à la façon de stocker la matrice creuse. En effet, pour avoir un stockage efficace, seuls les coefficients non nuls de la matrice creuse sont stockés. Le motif des valeurs non nulles de la matrice est défini par le problème que nous souhaitons résoudre.

Le format le plus générique pour stocker des matrices creuses s'appelle COO [4]. Dans ce format, chaque valeur non nulle est stockée avec ses coordonnées 2D dans la matrice. Ce format est souvent implémenté sous la forme de trois tableaux (Fig. 2.5(b)). Chaque tableau s'occupera d'une propriété des éléments non nuls (valeur, colonne et ligne). Toutes les matrices peuvent donc être représentées dans ce format. Il peut être utile d'utiliser ce format lorsque nous souhaitons utiliser des algorithmes qui changent des valeurs nulles de la matrice en valeurs non nulles. Mais ce format à l'inconvénient de ne pas imposer d'ordre sur les éléments de la matrice. Il est donc difficile d'écrire un programme optimisé utilisant ce format.

À la place, un autre format est souvent utilisé, il s'agit du format CSR [5]. Ce format permet lui aussi de stocker toutes sortes de matrices. Il impose que les éléments non nuls soient rangés par ligne. Il est aussi composé de trois tableaux et reprend deux des trois tableaux du format COO (valeur et colonne). Le troisième tableau correspond aux indices de début et de fin de chaque ligne dans les deux autres tableaux

1. Basic Linear Algebra Subprograms
2. Linear Algebra PACKage
3. Genenal Purpose Graphical Processing Unit
4. COOrdinate list
5. Compress Sparse Row

(Fig. 2.5(c)). Ce troisième tableau peut être vu comme une compression du tableau indiçant les lignes de la matrice. D'autres formats moins génériques existent, mais nous n'en parlerons pas ici.




(a) Exemple de matrice creuse (b) Stockage COO (c) Stockage CSR

FIGURE 2.5 – Comparaison entre les formats de stockage de matrices creuses COO et CSR.

Le choix du format de stockage va avoir beaucoup d'effet sur les performances d'une application. Avec la plupart des formats, nous aurons au moins deux accès mémoire pour obtenir les coordonnées 2D d'un coefficient non nul alors qu'avec l'algèbre linéaire dense nous pouvons calculer ces coordonnées à partir de la position dans la matrice. Une partie non négligeable de la bande passante mémoire est donc utilisée juste pour la lecture des coordonnées 2D. Les propriétés creuse et irrégulière de ces matrices impliquent aussi une mauvaise efficacité mémoire des noyaux d'algèbre linéaire creuse à cause d'une mauvaise réutilisation du cache. La plupart des optimisations faites en algèbre linéaire dense ne peuvent pas être appliquées à l'algèbre linéaire creuse à cause de l'irrégularité dans l'ordre des calculs ainsi que dans les accès mémoire. Mais l'algèbre linéaire creuse nous permet de résoudre des problèmes bien plus grands que ceux qui utilisent l'algèbre linéaire dense. Ceci est dû au fait qu'à ordre de matrice équivalente, une matrice creuse utilise vraiment moins de mémoire qu'une matrice dense.

Pour la simulation de réservoir, chaque entrée de nos matrices est composée d'un petit bloc dense dont la taille dépend du nombre de variables primaires utilisées par la simulation. Nous reviendrons sur la définition de ces variables primaires plus loin dans ce document, disons qu'il s'agit du nombre de variables physiques simulées par cellule. Ces blocs permettent d'obtenir une meilleure réutilisation des caches lors des opérations élémentaires (addition, multiplication ...). Pour prendre en compte ces blocs et ainsi optimiser nos calculs, nous utilisons une version modifiée du CSR, le Block CSR ou BCSR. La différence avec le format CSR provient du tableau stockant les valeurs non nulles de la matrice. Au lieu de stocker des scalaires, nous stockons des blocs denses. Ces blocs sont eux-mêmes stockés au format *column major*, ce qui signifie qu'en mémoire deux éléments consécutifs appartiennent à la même colonne. Exception faite du dernier élément de la colonne dont l'élément consécutif est le premier élément de la colonne suivante.

Nous avons effectué une comparaison des différents formats de stockage des matrices appliqué au réservoir dans le cadre de l'utilisation d'un GPU, les résultats sont disponibles dans l'article [Ros13]. Le format BCSR est un bon compromis entre flexibilité et performance.

Résoudre des problèmes linéaires creux est aussi très différent de résoudre des problèmes denses. Nous ne pouvons pas utiliser une inversion directe de matrice, ou la technique de l'élimination de Gauss car la complexité en temps et en mémoire de ces méthodes est rédhibitoire pour nos applications. De plus, nos problèmes ne nécessitent qu'une résolution approchée de la solution à chaque itération de l'algorithme de Newton. Donc des méthodes différentes ont été inventées pour être capable de résoudre ces problèmes, beaucoup sont basées sur des méthodes itératives. Nous démarrons donc avec une solution, ensuite ces méthodes réduisent itérativement la différence entre notre solution approximée et la solution réelle. À la fin, nous obtenons une bonne approximation de la solution, ce qui est souvent suffisant pour être considéré comme la solution du problème.

## 2.3 Résoudre de grands systèmes linéaires creux

### 2.3.1 Méthode directe

Les méthodes de résolution directe permettent de résoudre exactement, dans les limites de la précision machine, l'équation $Ax = b$. Parmi ces méthodes, la décomposition LU est la plus utilisée. La factorisation LU correspond à la factorisation d'une matrice $A$ en deux matrices triangulaires $L$ et $U$ tel que $L.U = A$. Grâce à cette factorisation, résoudre l'équation $Ax = b$ est équivalent à résoudre successivement les équations $Ly = b$ (opération dite de descente) et $Ux = y$ (opération de remontée). Ces deux équations peuvent être résolues facilement parce que les matrices $L$ et $U$ sont triangulaires (Algo. 1).

**Algorithme 1 :** Résolutions triangulaires

**Données** : $L$ : Matrice triangulaire inférieure
$U$ : Matrice triangulaire supérieure
$x$ : Vecteur des inconnues
$y$ : Vecteur temporaire
$b$ : Vecteur donné
$n$ : Taille des matrices

```
pour i = 1 à n faire
    sum := 0
    pour j = 1 à i − 1 faire
        sum+ = L_ij ∗ y_j
    fin
    y_i := (b_i − sum)/L_ii
fin
pour i = n à 1 faire
    sum := 0
    pour j = n à i + 1 faire
        sum+ = U_ij ∗ x_j
    fin
    x_i := (y_i − sum)/U_ii
fin
```

La construction des matrices $L$ et $U$ à partir de $A$ est appelée factorisation LU. Nous pouvons construire ces matrices en réduisant itérativement l'ordre de la matrice $A$. Soit $A$ une matrice d'ordre $n$, la première ligne nous donne les valeurs $U$. Les valeurs de $L$ sont obtenues en divisant la première colonne par la première valeur de la diagonale. Puis nous soustrayons le produit de la première colonne et de la première ligne à la sous matrice d'ordre $n-1$ Et nous recommençons avec la matrice d'ordre $n-1$ jusqu'à $n = 1$ pour obtenir la colonne et la ligne suivante (Fig. 2.6).

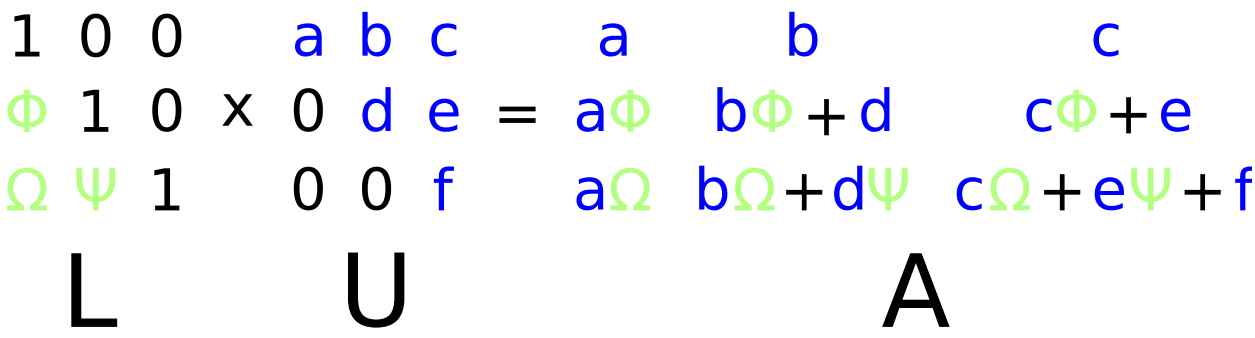



FIGURE 2.6 – Exemple d'égalité entre une matrice $A$ et le produit de deux matrices triangulaire $L$ et $U$.

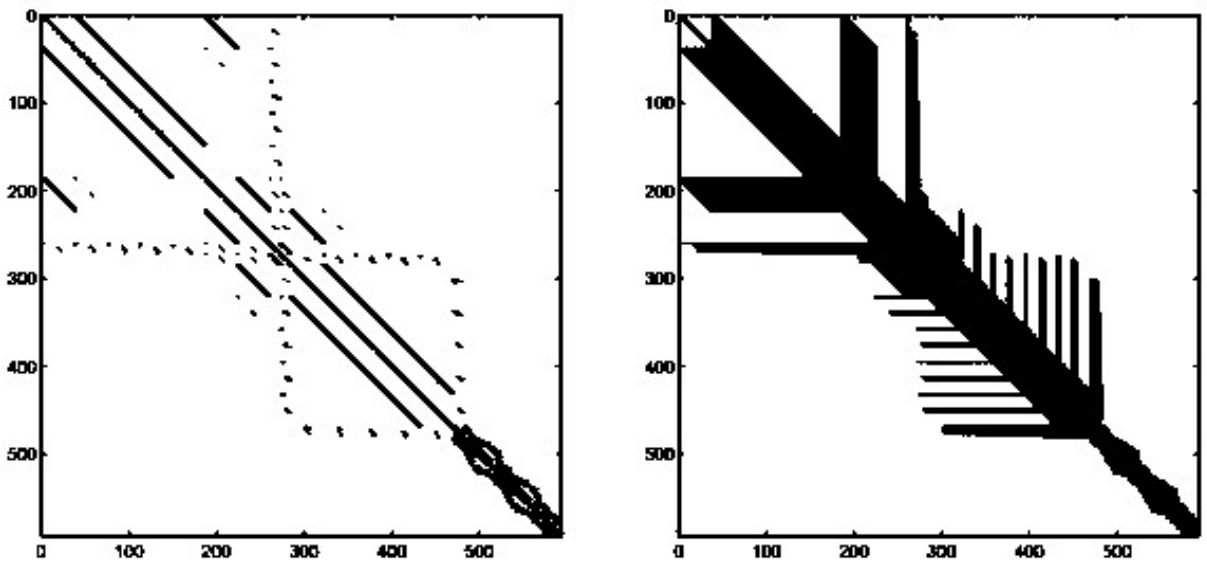
L'algorithme 2 représente la construction des matrices triangulaires $L$ et $U$ à partir de $A$. La factorisation est faite sur place, c'est à dire que les coefficients de $A$ sont remplacés au fur et à mesure par les coefficients

de $L$ et de $U$. En effet, $L$ et $U$ étant deux matrices triangulaires, respectivement inférieure et supérieure, nous pouvons les stocker sous la forme d'une matrice carrée à condition de ne pas stocker explicitement la diagonale de $L$. Celle-ci n'étant composée que de valeur unitaire, nous connaissons toujours ses valeurs. Par contre, en algèbre linéaire creuse, la factorisation LU présente un problème de remplissage c'est-à-dire que des éléments nuls deviennent non nuls. Avec une factorisation LU directe, les matrices $L$ et $U$ contiennent beaucoup plus d'éléments non nuls que la matrice $A$. Cette augmentation du nombre d'éléments peut conduire à ne plus pouvoir stocker les matrices $L$ et $U$ à cause du manque d'espace mémoire. Il existe des techniques de renumérotation permettant de limiter le remplissage des matrices[PR97, KK98].

**Algorithme 2 :** Factorisation LU sur place.

**Données** : $A$ : Matrice à factoriser
$n$ : Taille de la matrice

```
1 pour k = 1 à n − 1 faire
2     pour i = k + 1 à n faire
3         A_ik /= A_kk
4         pour j = k + 1 à n faire
5             A_ij −= A_ik ∗ A_kj
6         fin
7     fin
8 fin
```




(a) Matrice originale, nombre de non-zéros : 5104.

(b) Matrice factorisée, nombre de non-zéros : 58202.

FIGURE 2.7 – Exemple de factorisation LU d'une matrice creuse avec remplissage. Il s'agit de la donnée DWT 592 référencée sur *matrix market* [BPR+97].

Les méthodes directes ont l'avantage de fournir une solution précise au problème $Ax = b$. Par contre, ces méthodes sont coûteuses aussi bien en mémoire qu'en calcul. Pour un cas 3D à $n$ inconnues, nous aurons un stockage en $O(n^{4/3})$[LRT79] et une complexité en nombre d'opérations en $O(n^2)$. En simulation de réservoir, la valeur $n$ peut être grande (supérieur à $10^6$ et jusqu'à $10^9$). Il n'est donc pas envisageable d'utiliser ce genre de méthodes.

### 2.3.2 Méthode itérative

Une autre approche souvent utilisée pour résoudre de grands systèmes d'équations linéaires creux consiste à utiliser des méthodes de résolution itérative [Saa96]. On appelle méthode itérative une méthode qui permet de résoudre un problème en partant d'une solution initiale $x_0$ et qui à chaque itération donne une nouvelle solution $x_i$. Cette nouvelle solution $x_i$ étant plus proche de la solution exacte du problème que la solution précédente $x_{i-1}$. La méthode s'arrête lorsque $x_i$ est suffisamment proche de la solution exacte selon un critère de convergence choisi. Parmi ces méthodes nous pouvons citer la méthode Jacobi, Gauss-Seidel ou encore SOR, ce sont des méthodes itératives dites stationnaires. Mais ces méthodes ne sont pas génériques, leur convergence dépend de certaines propriétés de la matrice. Dans de nombreux cas concrets, ces méthodes ne convergent pas rapidement si elles sont utilisées telles quelles.

La méthode du gradient conjugué est une méthode qui s'applique seulement à des matrices carrées symétriques définies positives. Cette méthode permet en théorie de converger en au plus $n$ itérations avec $n$ la dimension de la matrice mais en pratique seules quelques itérations sont nécessaires. Mais avec un bon préconditionnement, on obtient rapidement une solution très proche de la solution exacte. Puis cette méthode a été étendue aux matrices non-symétriques sous le nom du gradient biconjugué. Le gradient biconjugué est une méthode par projection dans un espace de Krylov. Le GMRES est aussi une méthode de Krylov et il fonctionne avec n'importe quelle matrice inversible. Chaque itération du GMRES (Algo. 3) est composée d'un produit matrice vecteur creux (SpMV [1]) ainsi que de plusieurs produits scalaires (DOT). Pour la simulation de réservoir, nous utilisons la variante flexible du GMRES (FGMRES[Saa96]) ainsi que le préconditionneur CPR[WKL+85].

**Algorithme 3 :** GMRES avec une orthogonalisation Householder, nous avons surligné le produit matrice-vecteur creux ainsi que les produits scalaires. En pratique, le résidu $||b - Ax_j||$ n'est pas calculé explicitement[Saa96].

**Données** : $max$ : Le nombre maximum d'itérations du GMRES
$tolerance$ : valeur maximale de la norme du résidu

$r_0 := b - Ax_0$
$\beta := ||r_0||_2$
$v_1 := r_0/\beta$
Définir la $(max+1)$x$max$ matrice $\bar{H}_{max} = \{h_{ij}\}_{1 \leq i \leq max+1, 1 \leq j \leq max}$.
$\bar{H}_{max} = 0$
**pour** $j = 1$ **à** $max$ **faire**
    $w_j := A * b$
    **pour** $i = 1$ **à** $j$ **faire**
        $h_{ij} := (w_j, v_i)$
        $w_j := w_j - h_{ij} v_i$
    **fin**
    $h_{j+1,j} := ||w_j||_2$
    Calculer $y_j$ le minimiseur de $||\beta e_1 - \bar{H}_j y||_2$
    $x_j := x_0 + V_j y_j$
    **si** $||b - Ax_j|| < tolerance$ **alors**
        $x_{solution} := x_j$
        **break**
    **fin**
    $v_{j+1} := w_j / h_{j+1,j}$
**fin**

La convergence (i.e. nombre d'itérations pour atteindre une précision donnée) des méthodes itératives

1. Sparse Matrix Vector multiply

dépend du nombre de conditionnement de la matrice $||A||_p.||A^{-1}||_p$ où $||A||_p$ est une norme matricielle ($p = 1, ..., \infty$). Plus ce nombre est grand, plus le nombre d'itérations est grand. Comme les matrices utilisées dans la simulation de réservoir ne sont pas bien conditionnées, l'algorithme du GMRES ne converge qu'après beaucoup d'itérations. Dans ce cas, nous pouvons préconditionner la matrice pour faire en sorte que le GMRES converge avec moins d'itérations. Il faut choisir une matrice $M^{-1}$ tel que $M^{-1}A$ soit mieux conditionnées que $A$ (i.e. $||M^{-1}A||_p.||A^{-1}M||_p < ||A||_p.||A^{-1}||_p$). Un cas idéal serait d'avoir $M = A$, dans ce cas-là on obtient la matrice identité qui se trouve être très bien conditionnée. Or, calculer $A^{-1}$ est très coûteux, à la fois en termes de calcul que de mémoire.

La factorisation LU permet de calculer directement $x = U^{-1}.L^{-1}.b = A^{-1}b$ mais comme nous l'avons vu, cette opération est très chère en calculs et en mémoire. Néanmoins, une factorisation approchée de $A$, telle que $||A - LU||$ est relativement petite, peut jouer le rôle de préconditionneur dans une méthode de Krylov. Ainsi, les factorisations ILU [1][Saa96] sont des choix très populaires de préconditionneurs algébriques. Les principales techniques de factorisation ILU visent à calculer une factorisation LU qui ne tient pas compte de certains coefficients de remplissage dans la matrice afin de limiter le coût tout en essayant de minimiser $||A - LU||$. En effet, au cours de la factorisation, de nombreux termes de remplissage apparaîtront et l'espace mémoire nécessaire au stockage de ces nouveaux termes deviendrait gigantesque. Mais nous ne sommes pas obligés de connaître parfaitement $A^{-1}$ pour obtenir un bon préconditionnement. La méthode par précondionnement ILU est composée de deux opérations, la première correspond à la *factorisation* de la matrice en deux sous matrices, elle est faite juste avant le GMRES. La deuxième correspond à la *résolution triangulaire* effectuée avec les deux sous matrices, elle est effectuée à chaque itération du GMRES. Pour maintenir un espace mémoire raisonnable, on peut choisir de limiter le niveau de remplissage, les deux algorithmes les plus fréquemment utilisés sont obtenus :

— Soit en fixant une valeur seuil à partir de laquelle le remplissage est accepté, il s'agit de l'algorithme ILUt ;
— Soit en choisissant un niveau d'interaction maximal entre les lignes de la matrice, il s'agit de l'algorithme ILU(k) (Fig. 2.8).

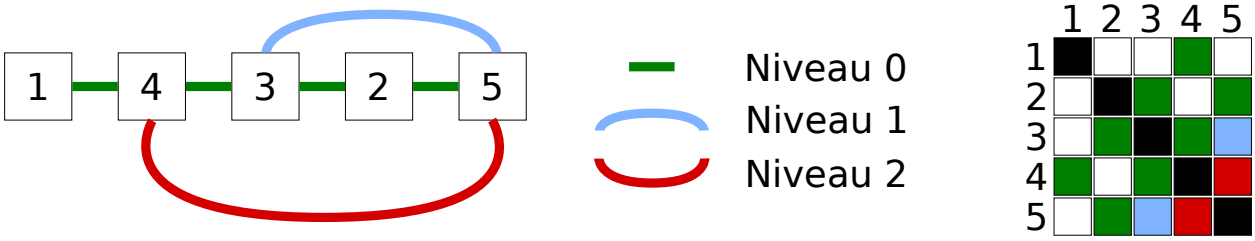



FIGURE 2.8 – Remplissage d'une matrice lors d'une factorisation ILU(k). Avec $k = 2$ nous obtenons déjà le plus haut niveau de remplissage.

La factorisation ILU(k) peut être faite en deux étapes :

— une factorisation symbolique pour connaître le motif de non-zéros de la matrice ;
— puis la factorisation réelle en utilisant ce motif.

La factorisation symbolique (Algo. 4) va utiliser le graphe d'adjacence de la matrice. S'il existe un chemin entre deux noeuds d'indices $i$ et $j$ tel que tous les indices du chemin soient inférieurs au minimum entre $i$ et $j$, alors une arête entre les noeuds $i$ et $j$ apparaîtra. La longueur de ce chemin déterminera le niveau de remplissage. Par exemple, sur la figure 2.8, le noeud d'indice "5" aura une relation de niveau 2 avec le noeud d'indice "4" car le chemin 4-3-2-5 existe et les indices "3" et "2" sont inférieurs à "4" et "5".

L'algorithme ILUt a l'inconvénient d'avoir un motif de remplissage qui dépend des coefficients de la matrice. Nous ne pouvons donc pas prévoir à l'avance l'espace nécessaire au stockage des matrices L et U et le stockage de ces matrices devient complexe. Par contre, l'algorithme ILU(k) permet de connaître en avance le motif de remplissage de la matrice (Fig. 2.8). Il est donc possible de connaître la structure des matrices L et U à l'avance. Pour la simulation de réservoir, nous utilisons l'algorithme ILU(k). L'algorithme 5 représente un GMRES préconditionné avec une méthode ILU.

1. Incomplete LU

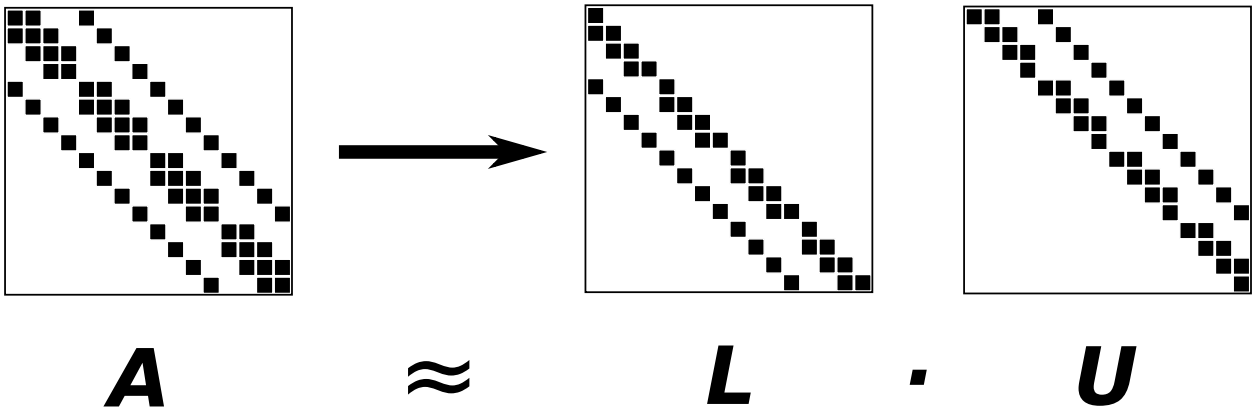



FIGURE 2.9 – Exemple de motif obtenu par une factorisation ILU(0), il n'y a pas de remplissage.

### 2.3.3 Parallélisation des méthodes itératives

La partie résolution de système linéaire creux représente souvent la partie qui consomme le plus de temps dans une simulation numérique, par exemple dans la simulation de réservoir cela peut représenter jusqu'à 80 % du temps de simulation. Il est donc essentiel de paralléliser cette partie du code. Les noyaux de calcul les plus coûteux utilisés dans les méthodes de Krylov préconditionnées par une méthode ILU sont :

— la factorisation ILU ;
— les résolutions triangulaires ;
— le produit matrice vecteur creux ;
— le produit scalaire.

La parallélisation des méthodes de Krylov revient à paralléliser les noyaux de calcul cités précédemment. L'une des méthodes de précondionnement parallèles les plus courantes est la méthode de Jacobi par blocs. Dans cette méthode, nous découpons des blocs dans la matrice et nous appliquons notre préconditionneur sur chaque bloc. Ces blocs proviennent d'un partitionnement du graphe d'adjacence de la matrice à l'aide d'un logiciel de partitionnement, tel que Scotch[CP08] ou Metis[Kar03], avec pour objectif un bon équilibrage de charge (Fig. 2.10(a)). La matrice est ensuite réordonnée pour que la numérotation des cellules à l'intérieur d'un domaine soit contiguë. Ces domaines formeront des blocs dans la matrice sur lesquels nous appliquerons notre préconditionneur ILU. Avec autant de blocs que de coeurs de calcul, nous pouvons factoriser tous les blocs parallèlement. Cette méthode ignore donc les interactions entre domaines et fournit donc un préconditionneur naturellement parallélisable. Sur la figure 2.10(b), les entrées de la matrice en dehors des blocs ne seront pas utilisées par la factorisation ILU ni par les résolutions triangulaires.

Les deux autres noyaux de calcul (le produit matrice vecteur creux et le produit scalaire) peuvent être parallélisés facilement. Dans le cas du produit matrice vecteur creux, chaque coeur de calcul s'occupera de multiplier un ensemble de lignes de la matrice. Cet ensemble peut provenir de la décomposition de domaine faite pour la méthode Jacobi par bloc, mais ce n'est pas obligatoire. Pour le produit scalaire, chaque coeur de calcul s'occupera de multiplier un ensemble d'éléments des deux vecteurs.

La méthode de Jacobi par blocs n'affecte donc que la partie préconditionneur des méthodes itératives. Le nombre de blocs aura un impact sur la convergence de la méthode itérative utilisée. Cet impact n'est pas facilement prédictible et dépendra du problème étudié. Il est donc quasiment impossible de connaître le nombre optimal de blocs nécessaire pour avoir un bon rapport parallélisme sur nombre d'itérations. C'est pourquoi nous allons essayer de réduire au minimum le nombre de blocs pour toujours obtenir les meilleures performances. Pour cela, nous allons devoir utiliser une autre forme de parallélisme qui s'appliquera à l'intérieur d'un bloc. Nous aurons donc du parallélisme sur deux niveaux :

— sur les blocs de la matrice avec la méthode Jacobi par blocs ;
— à l'intérieur des blocs en parallélisant la factorisation ILU.

Notre but étant d'affecter le moins possible la qualité du préconditionneur. D'une manière plus générale, le travail présenté dans le reste du document est destiné à pouvoir exploiter du parallélisme à grain fin dans les routines d'algèbre linéaire fondamentales pour l'implémentation des solveurs linéaires creux. Ainsi nous nous intéressons plus particulièrement aux factorisations ILU et aussi aux opérations usuelles trouvées dans

**Algorithme 4 :** Factorisation symbolique ILU(k)

**Données** : $A$ : La matrice à factoriser.
$k$ : le niveau maximal de remplissage.

```
pour chaque entrée A_ij de A faire
    si A_ij == 0 alors
        lev(A_ij) := k+1
    fin
    si A_ij != 0 alors
        lev(A_ij) := 0
    fin
fin
pour k = 1 à n-1 faire
    pour i = k+1 à n faire
        si lev(A_ik) <= k alors
            pour j = k+1 à n faire
                lev(A_ij) = min(lev(A_ij), lev(A_ik)+lev(A_kj)+1)
                si lev(A_ij) <= k alors
                    on ajoute A_ij au motif de non-zéro de la matrice.
                fin
            fin
        fin
    fin
fin
```

les méthodes itératives et les préconditionneurs (produit matrice vecteur, produit scalaire, résolution d'un système linéaire triangulaire creux ...). Le but de ce travail n'est donc pas de proposer une nouvelle méthode de résolution des systèmes linéaire creux mais de permettre à un développeur d'implémenter des méthodes en exploitant au mieux les architectures multicoeurs. Le bon compromis entre le parallélisme lié aux paramètres de la méthode (par exemple le nombre de domaines dans la méthode Jacobi par blocs) et le parallélisme direct des routines d'algèbres linéaires utilisées par la méthode restera toujours dépendant de l'application et de la méthode utilisée.

### 2.3.4 Cas d'étude

Pour être en mesure de tester notre méthode de parallélisation, nous utilisons un code de solveur linéaire développé chez Total. Nous allons essayer de paralléliser la partie GMRES préconditionné du code. Dans le but d'évaluer le gain de performance, nous avons choisi des systèmes linéaires à résoudre avec le solveur linéaire.

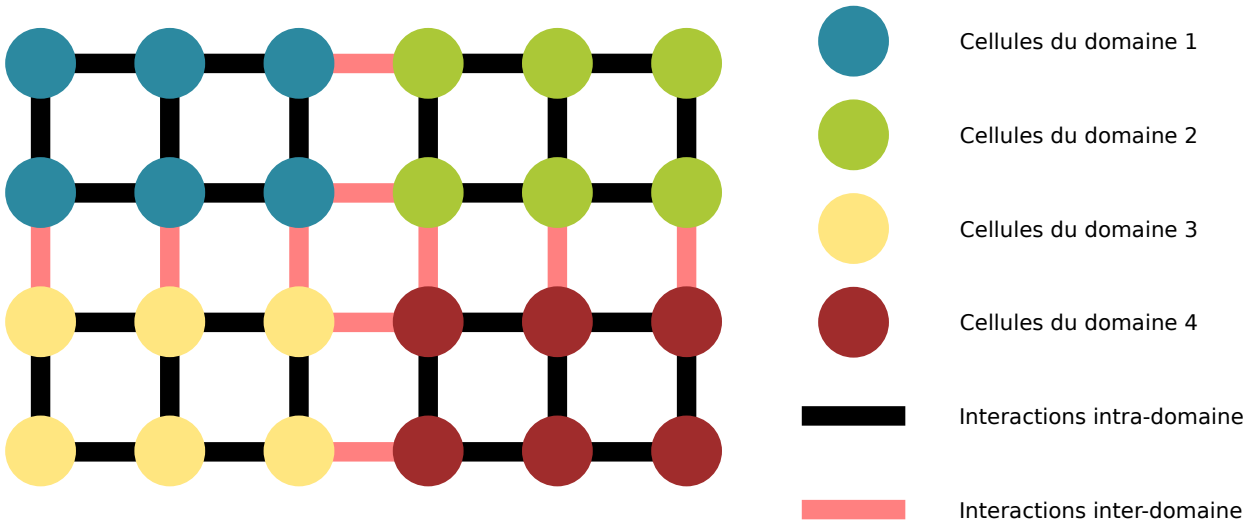
Ces systèmes linéaires sont représentés sous la forme d'une matrice et d'un vecteur second membre. La structure des matrices est dépendante du problème simulé. Nous utilisons un maillage structuré avec un schéma de discrétisation en 7 points (e.g., volume fini). En gardant une numérotation naturelle, nos matrices auront donc une structure composée de sept diagonales. En fonction de ce que nous voulons simuler, les entrées de la matrice pourront être scalaires ou composées de petits blocs. Si nous utilisons un schéma IMPES (implicite en pression et explicite en saturation), nous aurons des entrées scalaires. Les simulations de type *black-oil* sont les plus utilisées en simulation de réservoir. Il s'agit de simuler 3 variables primaires, la concentration en huile, en gaz et en eau de chaque cellule du réservoir. Il arrive aussi que l'on souhaite simuler plus précisément les différents types d'huiles contenues dans les réservoirs. Dans ce cas, nous utiliserons un modèle compositionnel dans lequel chaque variable primaire correspondra à la saturation d'un type d'hydrocarbure. Pour les cas black-oil et les cas compositionnels, les entrées de la matrice seront de petits blocs denses de taille $npri * npri$ où $npri$ est le nombre de variables primaires.

Pour évaluer notre code, nous allons utiliser le cas test SPE10 qui est basé sur les données prises du second modèle du 10ème cas test SPE[CB01]. C'est un réservoir de 1 122 000 de cellules, organisées dans une grille

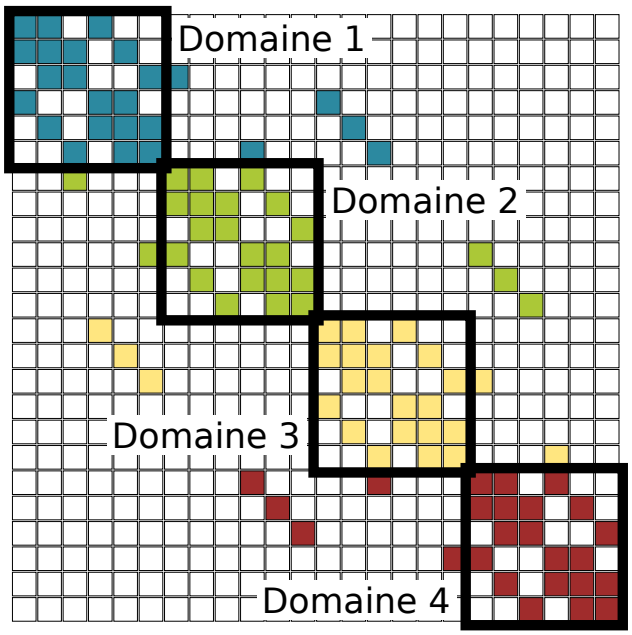
**Algorithme 5 :** GMRES préconditionné à droite par une méthode ILU. Les parties surlignées correspondent au préconditionnement. $X_{solution}$ est une solution approché de $Ax = b$.

**Données** : $max$ : Le nombre maximum d'itérations du GMRES
$tolerance$ : valeur maximale de la norme du résidu

1 $M =$ Factorisation_ILU($A$)
2 $r_0 := b - Ax_0$
3 $\beta := ||r_0||_2$
4 $v_1 := r_0/\beta$
5 Définir la $(max+1)$x$max$ matrice $\bar{H}_{max} = \{h_{ij}\}_{1 \leq i \leq max+1, 1 \leq j \leq max}$.
6 $\bar{H}_{max} = 0$
7 **pour** $j = 1$ **à** $max$ **faire**
8 $temp :=$ Résolutions_Triangulaires($M, v_j$)
9 $w_j := A * b$
10 **pour** $i = 1$ **à** $j$ **faire**
11 $h_{ij} := (w_j, v_i)$
12 $w_j := w_j - h_{ij} v_i$
13 **fin**
14 $h_{j+1,j} := ||w_j||_2$
15 Calculer $y_j$ le minimiseur de $||\beta e_1 - \bar{H}_j y||_2$
16 $x_j := x_0 + V_j y_j$
17 **si** $||b - Ax_j|| < tolerance$ **alors**
18 $x_{solution} := x_j$
19 **break**
20 **fin**
21 $v_{j+1} := w_j / h_{j+1,j}$
22 **fin**




(a) Exemple d'une décomposition en quatre domaines.




(b) Une matrice ordonnée par domaine. Les entrées en dehors des blocs seront ignorées lors de la factorisation ILU.

FIGURE 2.10 – Exemple d'une décomposition de domaine appliquée à une méthode de Jacobi par blocs.

3D cartésienne de taille 60 x 220 x 85. Il s'agit d'un modèle black-oil, donc à 3 variables primaires, et c'est un problème de référence dans l'industrie pétrolière. Les autres cas tests seront générés par un programme développé en interne chez Total, nous utiliserons un cas pression, un cas black-oil et un cas compositionnel à 8 composants. Ce programme génère des cubes 3D cartésiens de taille arbitraire. Ces cas générés nous permettent de tester différentes combinaisons de tailles dans le but d'évaluer le passage à l'échelle de nos algorithmes.

La partie GMRES du code que nous souhaitons paralléliser est composée de plusieurs noyaux d'algèbre linéaire creuse. Il y a la factorisation ILU et les résolutions triangulaires, le produit matrice vecteur creux et le produit scalaire. Le parallélisme exploitable dans ces noyaux est différent, il peut être plus ou moins difficile à exploiter. Dans le cas du produit matrice vecteur creux, la multiplication de chaque ligne de la matrice est indépendante, le parallélisme s'exploite facilement. De même pour le produit scalaire, chaque élément du vecteur peut être traité indépendamment. Pour la factorisation ILU c'est différent, certaines lignes doivent être factorisées avant d'autres, le parallélisme est donc plus dur à exploiter. Les résolutions triangulaires se

parallélisent de la même façon que la factorisation ILU. Dans la suite de la thèse, nous expliquerons comment exploiter efficacement le parallélisme dans ces quatre cas.

## 2.4 Évolution des architectures

### 2.4.1 Processeurs monocoeur

Pour être capables de simuler de grands problèmes physiques, nous avons besoin de beaucoup de puissance de calcul. Cette puissance se mesure en FLOPS [1], il s'agit du nombre d'opérations par seconde qu'un ordinateur peut effectuer sur des nombres à virgules flottantes. Elle dépendra de différents critères comme de la fréquence d'horloge du processeur, du jeu d'instructions utilisé ou du nombre d'unités de calcul.

Même les processeurs monocoeur, c'est à dire composés d'un seul coeur de calcul, peuvent faire des opérations en parallèle. Le parallélisme au niveau des instructions en est un bon exemple, les pipelines d'instructions permettent de paralléliser les différentes étapes liées au traitement d'une instruction. Ces étapes diffèrent selon l'architecture du processeur. En simplifiant, ces étapes peuvent être :

— *Fetch*, récupération de l'instruction depuis la mémoire ;
— *Decode*, décodage de l'instruction pour choisir les unités arithmétiques et les registres à utiliser ;
— *Execute*, effectue l'opération désignée par l'instruction ;
— *Write-back*, écriture du résultat en mémoire si besoin est.

Avec ce modèle simplifié, le processeur peut exécuter jusqu'à 4 instructions simultanément (Fig. 2.11). Deux problèmes se posent : dans le cas où l'instruction est un saut conditionnel, l'étape fetch ne peut être connu qu'au moment du write-back, le pipeline est donc vidé. Pour limiter l'impact de problème, le processeur pourra supposer un branchement et vider le pipeline seulement s'il a fait une erreur. Le deuxième problème est lié à l'étape execute, certaines opérations peuvent durer plus d'un cycle (multiplication, division, calcul flottant ...). Dans ce cas le pipeline est figé le temps que l'étape execute finisse, les instructions sont donc exécutées dans l'ordre (*in-order*). Les processeurs modernes résolvent ce problème en autorisant l'exécution des instructions dans le désordre (*out-of-order*). Si deux instructions consécutives n'utilisent pas les mêmes unités arithmétiques, alors il est possible de les exécuter simultanément. Comme par exemple dans le cas d'une multiplication flottante suivi d'une addition entière. Dans l'idéal, le processeur utilisant un pipeline d'instruction pourra exécuter une opération par cycle, donc un processeur à 4 GHz aura une puissance de calcul de 4 GFLOPS si les opérations flottantes sont faites en 1 cycle.

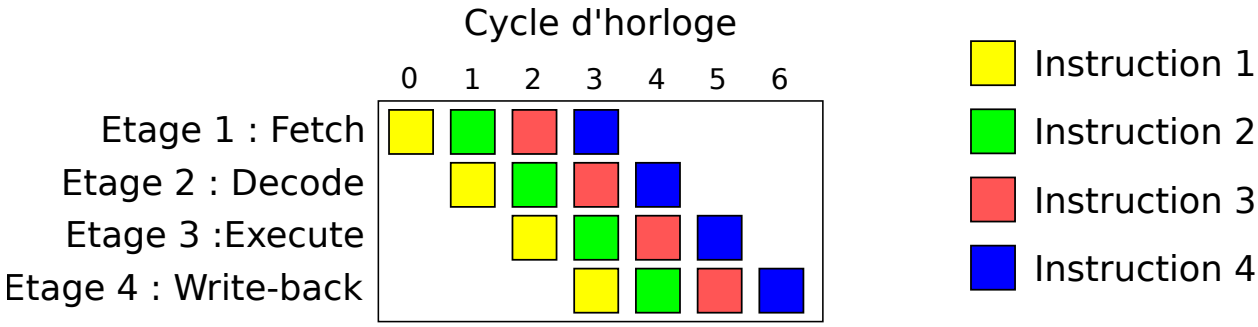



FIGURE 2.11 – Exemple d'un pipeline mettant 6 cycles d'horloges à exécuter 4 instructions

Par la suite, les processeurs ont gagné des instructions permettant d'effectuer une même opération sur des vecteurs de données, aussi appelées instructions SIMD dans la taxonomie de Flynn. Ces processeurs, dits vectoriels, peuvent donc avoir une puissance de calcul supérieure, si une instruction est capable d'effectuer 4 opérations à la fois et qu'il tourne à 4 GHz, alors il aura une puissance de calcul de 16 GFLOPS. Il s'agit ici d'une puissance théorique, tous les codes de calculs n'ont pas la possibilité d'exploiter les instructions vectorielles. Ces instructions sont souvent utilisées dans les noyaux de calculs d'algèbre linéaire dense. En simulation de réservoir, nous utilisons ces noyaux de calculs sur les petits blocs denses correspondants aux entrées de nos matrices.

1. FLoating-point Operations Per Second

### 2.4.2 Processeurs multicoeurs

Pour obtenir encore plus de parallélisme, il est possible de multiplier les unités de calcul au sein d'un processeur. Ces unités de calcul, aussi appelées coeurs de calcul, peuvent être considérées comme des processeurs. Chaque coeur a son propre pipeline d'instructions, ses registres et ses unités arithmétiques. Les coeurs partagent un ou plusieurs niveaux de cache entre eux ainsi que le bus d'accès à la mémoire. Ces caches sont le plus souvent cohérents entre eux, c'est à dire que pour chaque coeur de calcul l'accès à une variable en mémoire retournera toujours le dernier résultat connu par tous les coeurs de calcul. La cohérence est assurée par un protocole de cohérence de type MOESI [1]. Les caches partagés entre différents coeurs permettront réduire la complexité du mécanisme de cohérence des caches et aussi de bénéficier des données déjà pré-chargées par un autre coeur de calcul. Par contre, la taille du cache est donc partagée entre plusieurs coeurs de calcul et si un coeur demande souvent de nouvelles lignes de cache, le cache deviendra inutilisable et le deuxième coeur aura des problèmes de latence mémoire. Les caches de plus bas niveau (L1 et parfois L2) sont souvent dédiés à un coeur de calcul, l'espace mémoire n'est donc plus partagé. Mais il peut y avoir des effets négatifs, si deux coeurs de calcul écrivent souvent dans la même ligne de cache, il y a un problème de faux partage et la ligne de cache fera souvent des aller-retour entre les caches dédiés.

La puissance de calcul d'un processeur à 4 coeurs composés d'unités vectorielles capables d'effectuer 4 opérations par cycle et chaque coeur tournant à 4 GHz est de 64 GFLOPS. Encore une fois, cette puissance de calcul est théorique, il faut que le programme puisse utiliser tous les coeurs du processeur ainsi que les instructions vectorielles. Pour pouvoir utiliser tous les coeurs, il faut utiliser du parallélisme multicoeurs.

### 2.4.3 SMP

Pour encore gagner de la puissance, nous pouvons connecter plusieurs processeurs ensemble. Ces processeurs se partagent les ressources disponibles sur la carte mère, cela inclut les entrées/sorties et la mémoire. La façon d'inter-connecter tous les processeurs avec la carte mère peut différer entre différentes architectures. Avec l'architecture SMP, tous les processeurs sont connectés à un bus de données, ou à tout autre système de communication partagé tel que le crossbar par exemple. Un arbitre sera en charge de choisir quel processeur pourra utiliser le bus à un instant donné (Fig. 2.12). Cette conception ne passe pas à l'échelle au niveau des performances quand le nombre de processeurs grandit. La bande passante est partagée par tous les processeurs et l'arbitre du bus devient un goulot d'étranglement. Pire, la latence d'un accès mémoire va dépendre de la congestion du bus mémoire. L'utilisation de 4 processeurs comme décrite précédemment donne une puissance de calcul de 256 GLOPS. Cette puissance de calcul ne prend pas en compte les limitations mémoires. Pour pouvoir atteindre cette puissance, il faut limiter les accès à la mémoire partagée et privilégier les accès à la mémoire cache.

### 2.4.4 NUMA

Pour dépasser les limites de l'architecture SMP, la mémoire peut être physiquement distribuée entre chaque processeur (Fig. 2.13). Avec l'architecture NUMA, la latence et la bande passante de chaque accès mémoire dépendent de la distance entre le processeur qui fait la demande et la position physique de la mémoire. Il existe différents moyens d'inter-connecter les processeurs, on peut connecter tous les processeurs un à un pour faire en sorte que la latence soit la plus faible possible, mais tout comme l'architecture SMP cette méthode ne passe pas à l'échelle. Il est aussi possible de limiter le nombre de connexions par processeur tout en optimisant le nombre maximum de sauts, comme fait dans les grappes de calcul.

Chaque saut aura pour effet d'augmenter le temps de latence de l'accès mémoire. La distance entre chaque banc NUMA est fournie par le constructeur de la machine sous la forme d'une matrice de distance. Il existe souvent une correspondance entre la distance fournie par le constructeur et la latence mesurée. Nous pouvons donc utiliser cette matrice pour retrouver la topologie des noeuds NUMA de la machine. La puissance théorique est la même que pour une machine SMP, mais en pratique nous obtenons de meilleures performances grâce à la distribution des bancs mémoires. En contrepartie, pour obtenir ces performances, il est nécessaire d'avoir une bonne gestion de la mémoire. Cette gestion peut-être faite par le noyau du système

1. MOESI est l'acronyme de Modified, Owned, Exclusive, Shared et Invalid qui correspond aux différents états d'une ligne de cache.

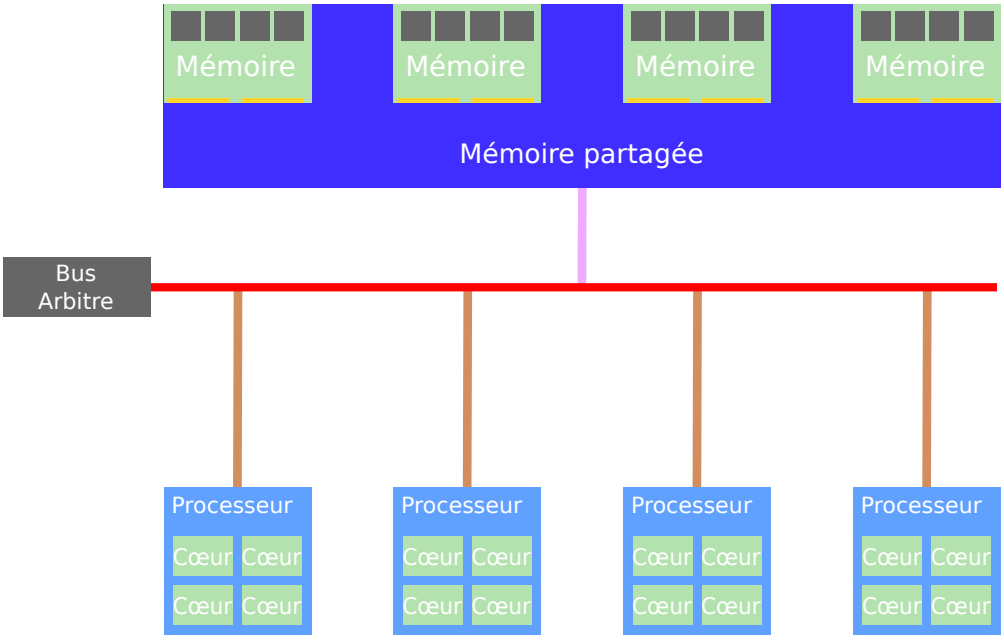



FIGURE 2.12 – Vue d'ensemble d'une architecture à accès mémoire uniforme (SMP).

d'exploitation dans un premier temps puis affinée par le programme lui-même. La plupart des machines NUMA sont ccNUMA [1], c'est-à-dire que les caches de données sont cohérents entre chaque processeur. Les machines que nous allons utiliser sont toutes ccNUMA, cela à pour conséquence qu'une écriture sur un banc NUMA distant coûtera plus cher qu'une lecture.

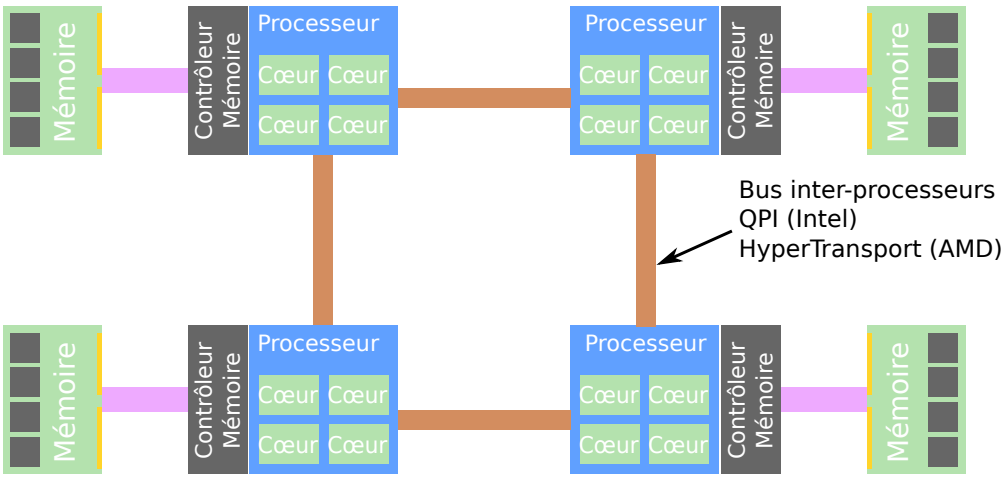



FIGURE 2.13 – Vue d'ensemble d'une architecture NUMA.

### 2.4.5 Grappe de serveurs

Finalement, il est possible de connecter plusieurs ordinateurs entre eux pour obtenir une machine encore plus puissante. Dans une grappe de serveurs, chaque ordinateur est appelé noeud de calcul, il a sa propre mémoire, fait tourner son propre système d'exploitation et peut être considéré comme une machine isolée. Les noeuds sont reliés entre eux par un réseau à faible latence/haut débit, tel que Infiniband ou Myrinet. Le principal avantage de cette solution est le passage à l'échelle. Il est possible de construire des machines de très grandes tailles et très puissantes.

Parmi les 500 machines les plus puissantes du monde au moment de l'écriture de cette thèse, 429 sont des grappes de serveurs. La machine la plus puissante est la *TIANHE-2* avec une puissance crête d'environ

1. cache coherent NUMA

55 PFLOPS. Mais l'utilisation de ces machines pose un sérieux problème, elles ne sont pas vraiment faciles à programmer. Il faut prendre en compte que la mémoire n'est pas globale, chaque noeud ne voit que sa mémoire locale.

### 2.4.6 Nos machines

Dans le but d'étudier différents problèmes liés à la programmation par graphe de tâches, nous avons sélectionné deux machines avec des architectures différentes.

#### 2.4.6.1 Rostand

Rostand est une grappe de serveurs appartenant à la compagnie Total. Elle est composée de 640 noeuds de calcul interconnectés avec un réseau Infiniband. Chaque noeud est lui-même composé de 2 bancs NUMA avec un processeur Intel Xeon X5660 et 24 Go de mémoire par banc NUMA (Fig. 2.14). Les processeurs ont 6 coeurs, soit un total de 12 coeurs par noeud de calcul et 7680 coeurs pour l'ensemble de la grappe. La puissance crête théorique de cette machine est de 122,88 TFlops.

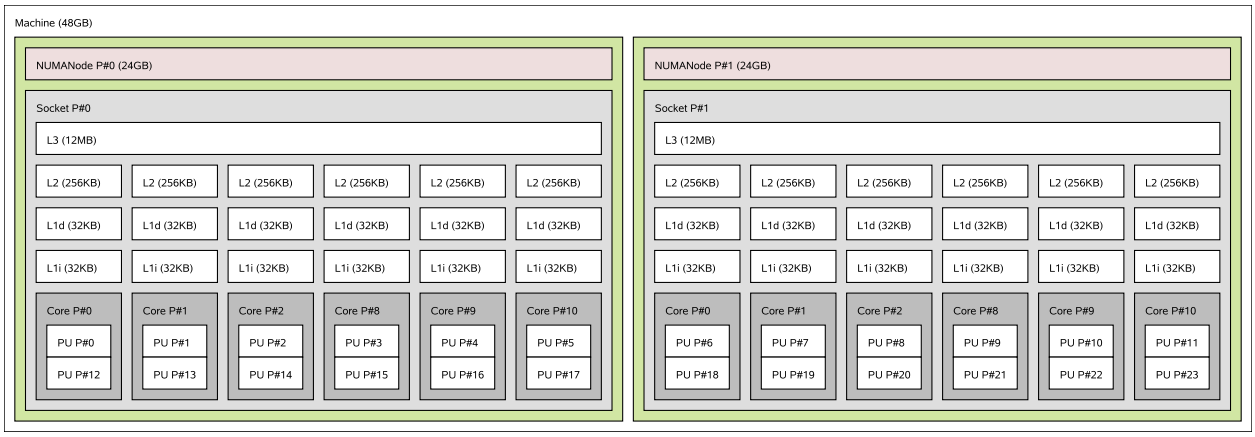



FIGURE 2.14 – Topologie d'un noeud de calcul de Rostand. Le schéma a été obtenu avec le logiciel hwloc.

Avec cette machine, nous pouvons tester deux paradigmes de programmation parallèle. Dans le cadre de nos travaux, nous utiliserons du parallélisme intra-noeud. Mais l'application finale qui bénéficiera des avancées liées à nos travaux pourra améliorer son parallélisme inter-noeud.

La matrice des distances (Fig. 2.15) nous indique la distance entre deux bancs NUMA donnée par le constructeur de la machine. Cette distance est adimensionnée, la distance entre un banc NUMA et lui-même est égale à 10, les autres distances sont mises à l'échelle par rapport à 10 comme spécifié dans la norme ACPI [MVM09]. Ces distances sont obtenues avec la commande *"numactl --hardware"* sous Linux. Étant donné que cette valeur ne peut servir qu'à connaître la topologie des bancs NUMA, nous avons décidé de mesurer la latence mémoire entre chaque banc. Pour cela, nous avons utilisé l'outil *lmbench* couplé à *numactl* pour définir les bancs NUMA à utiliser. Nous savons donc qu'un accès à un banc NUMA distant a un temps de latence 57 % plus grand qu'un accès au banc NUMA local.

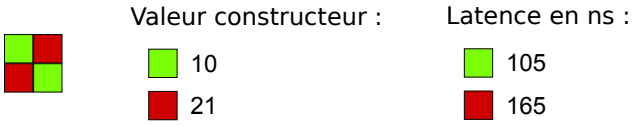
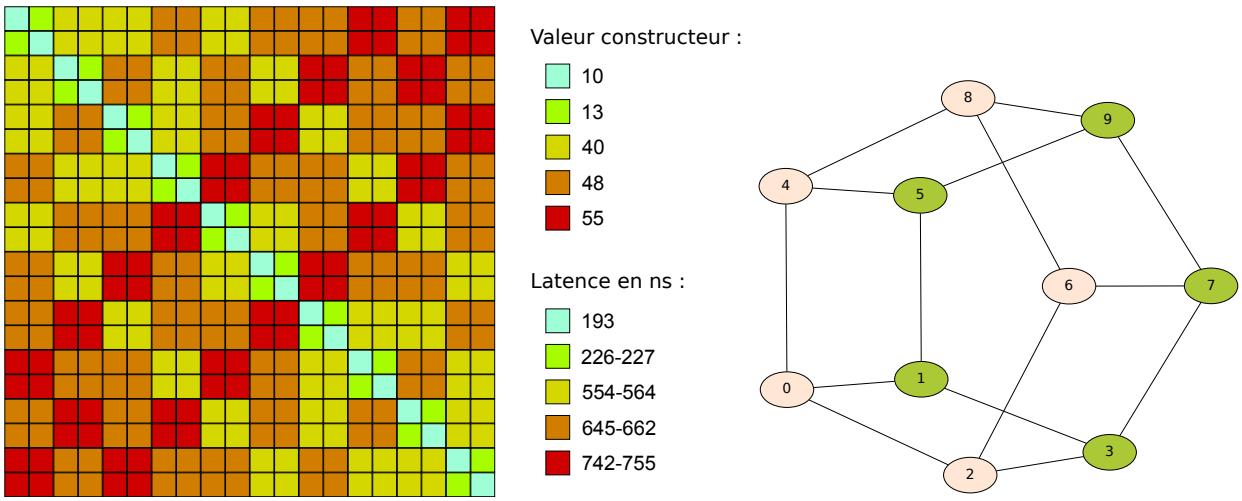



FIGURE 2.15 – Matrice des distances entre chaque banc NUMA de Rostand.

#### 2.4.6.2 Manumanu

Manumanu est une machine Altix UV100, cet ordinateur est composé de 20 bancs NUMA. Chaque banc NUMA est composé d'un processeur Intel Xeon E7-8837 ainsi que de 32 Go de mémoire. Les processeurs

ont chacun 8 coeurs de calcul, pour un total de 160 coeurs et 640 Go de mémoire partagée. Cette machine est donc intéressante pour évaluer les effets NUMA. À partir de la matrice des distances (Fig. 2.16(a)), nous




(a) Matrice des distances entre chaque banc NUMA.

(b) Topologie déduite de la matrice des distances. Chaque noeud représente un groupe de deux bancs NUMA.

FIGURE 2.16 – Architecture de Manumanu.

pouvons déduire la topologie de la machine. Les bancs NUMA sont regroupés deux par deux et chaque groupe est connecté à trois autres groupes. Ce regroupement permet de limiter la distance maximale entre deux bancs NUMA, il y aura au maximum 3 sauts. Le temps de latence entre deux bancs NUMA d'un même groupe n'est supérieur que de 17% par rapport à un accès local. Par contre, les temps de latence entre deux groupes sont entre 3 et 4 fois plus longs qu'un temps de latence local.

## 2.5 Parallélisme multi-coeurs

Lorsque l'on souhaite paralléliser un code de calcul, on doit choisir parmi plusieurs paradigmes de parallélisation. Le choix de ce paradigme est une étape importante, il déterminera les algorithmes à utiliser et donc aussi les performances du programme. En effet, un même problème ne se résoudra pas de la même façon en fonction du paradigme choisi. Mais le choix du paradigme est aussi déterminé par l'architecture de la machine cible. Dans le cas d'une grappe de serveurs, les noeuds de calculs ne peuvent communiquer que par un réseau. On préféra donc utiliser un paradigme par passage de messages ce qui nous obligera à utiliser des algorithmes distribués. Alors que dans le cas d'une machine à mémoire partagée nous aurons recours à l'utilisation de processus légers, aussi appelés *threads*. Plusieurs paradigmes peuvent être utilisés ensemble, nous pouvons ainsi tirer parti des avantages de chacun tout en limitant leurs inconvénients. Nous allons maintenant détailler les différents paradigmes de parallélisation que nous avons utilisés.

### 2.5.1 Parallélisme de boucle

Le parallélisme de boucle est un paradigme qui peut être utilisé sur des machines à mémoire partagée. Il s'agit de traiter en parallèle toutes les itérations d'une boucle en les distribuant équitablement sur tous les coeurs de calcul disponibles. Il faut bien sûr que ces itérations soient indépendantes, c'est-à-dire qu'avec un nombre infini de coeurs de calcul, on puisse traiter une itération par coeur simultanément. Ce paradigme fonctionne de la manière suivante : un thread est créé par coeur de calcul et chaque thread doit s'occuper de traiter une partie des itérations de la boucle. Puis tous les threads se synchronisent à la fin de la boucle sur une barrière implicite. Cette méthode est aussi appelée *fork and join*.

L'interface de programmation qui a le plus démocratisé ce paradigme est OpenMP. Cette interface utilise les directives de compilation en C ou en Fortran qui ont l'avantage d'être simples à utiliser et qui peuvent

être facilement désactivées pour retrouver un code séquentiel. En ajoutant la fameuse directive “#pragma omp parallel for” juste au-dessus d’une boucle for, on obtient facilement un programme multi-threadé avec une description du parallélisme très simple. Les performances obtenues avec ce paradigme sont souvent suffisantes pour un grand nombre de logiciels et le ratio entre le temps de développement et le gain en temps d’exécution est imbattable. Pour notre cas nous pouvons l’utiliser pour le produit matrice vecteur et pour le produit scalaire.

Malheureusement, il peut arriver qu’il y ait des dépendances de données entre deux itérations. Dans ce cas, ce paradigme de parallélisation ne fonctionne plus et donnera un résultat faux si nous l’utilisons. Il faut donc utiliser un autre paradigme de parallélisation. C’est ce qui arrive dans le cas d’une factorisation ILU. La version séquentielle du code est composée d’une boucle for sur les lignes de la matrice et chaque itération factorise une ligne dans l’ordre ascendant. Mais ces itérations ne sont pas indépendantes, il est nécessaire de factoriser certaines lignes avant d’autres.

### 2.5.2 Passage de messages

Certaines machines ne fonctionnent pas avec une mémoire globale, mais avec une mémoire distribuée. Chaque noeud de calcul a une mémoire locale et ne peut pas accéder directement à la mémoire des autres noeuds distants. Avec le paradigme de passage de messages, chaque processus a son propre espace mémoire virtuel et communique avec les autres processus par le biais d’envoi/réception de messages. Ces communications se font à l’aide d’une interface de programmation qui fournit des fonctions permettant l’échange de messages point-à-point. L’interface la plus connue et la plus utilisée actuellement est MPI [1]. Elle permet de faire communiquer deux processus ensemble sans se soucier du réseau utilisé ni même de la différence d’encodage des entiers (*little endian*/*big endian*) entre deux architectures différentes.

L’un des avantages majeurs de ce paradigme est qu’il permet d’utiliser un ensemble très varié de machines. Il fonctionne aussi bien en mémoire partagée qu’en mémoire distribuée. Son utilisation en mémoire partagée permet de n’utiliser qu’un seul type de parallélisme dans un programme. Un programme pur MPI peut donc utiliser tous les coeurs d’un noeud de calcul avec le même code source qui permet d’utiliser des grappes de serveurs. Mais il ne s’agit pas toujours de l’implémentation la plus efficace pour paralléliser un code, il est souvent plus performant d’utiliser une parallélisation hybride MPI+Threads[RHJ09]. De plus, certains algorithmes ne peuvent pas être écrits efficacement avec ce paradigme. Par exemple, dans notre cas la factorisation ILU(0) d’une matrice creuse se parallélise très mal en mémoire distribuée. Nous ne pouvons extraire du parallélisme qu’entre les factorisations de lignes de la matrice. Or ce niveau de granularité du calcul ne donne pas de bonnes performances avec un paradigme par passage de messages. Nous sommes donc obligés de modifier les méthodes de factorisation pour être capables d’obtenir de la performance. La méthode de Jacobi par blocs permet d’effectuer en parallèle une factorisation sur chaque bloc au détriment de la convergence de la méthode itérative. C’est donc cette méthode qui est utilisée pour une parallélisation par passage de messages. Cette méthode a l’inconvénient d’ignorer de nombreuses connexions entre les cellules du réservoir et fournit donc un préconditionnement de moins bonne qualité qu’une factorisation ILU sur la matrice complète.

### 2.5.3 Parallélisme à base de tâches

La programmation à base de tâches consiste à diviser un problème en plusieurs morceaux. Ces morceaux sont appelés tâches et le calcul de tous ces morceaux doit donner le même résultat que la résolution du problème non divisé. Ces tâches sont le plus souvent dépendantes les unes des autres, le résultat du calcul d’une tâche $T1$ peut être nécessaire au calcul d’une tâche $T2$. Dans ce cas, $T2$ dépend de $T1$, $T2$ est donc un successeur de $T1$ et $T1$ est un prédécesseur de $T2$. Une tâche ayant des prédécesseurs ne pourra commencer son calcul que lorsque toutes les tâches prédécesseuses ont complètement terminé leurs calculs. Une fois toutes ces dépendances explicitées, nous obtenons un graphe de tâches. Ce graphe doit être orienté et acyclique. La propriété directe donnera l’ordre d’exécution des tâches. La propriété acyclique est nécessaire pour éviter des inter-blocages, si une tâche $T1$ dépend d’une tâche $T2$ alors indirectement elle dépend aussi des tâches dont $T2$ dépend. Or, dans un cycle, cela signifie que $T1$ dépendra indirectement d’elle-même.

1. Message Passing Interface

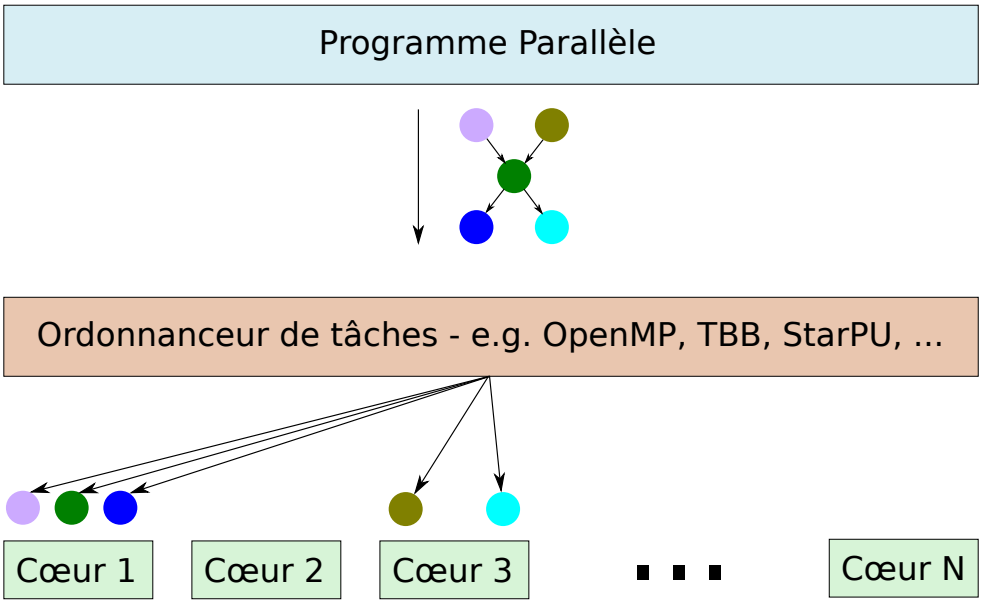



FIGURE 2.17 – Le programme parallèle fournit un graphe de tâches à l'ordonnanceur de tâches. Les tâches sont ensuite distribuées sur les coeurs de calcul disponibles.

Ainsi, nous pouvons représenter le parallélisme sous une forme abstraite indépendamment des ressources matérielles disponibles. Cette méthode de parallélisation a été appliquée à de nombreux travaux et permet ainsi de paralléliser efficacement des codes de calcul [BBAC14, LSAT13, LY14, ABGL13].

Les tâches de calcul peuvent être vues comme des fonctions avec des données en entrées et des données en sortie. Le processus de distribution des tâches sur les différents coeurs de calcul est appelé ordonnancement. Le degré de division du problème initial en tâche est appelé granularité. Le rapport entre le coût d'ordonnancement d'une tâche et le temps de calcul de la tâche définira si nous avons une granularité fine ou grossière. La démocratisation des processeurs multi-coeurs a engendré l'apparition de cadriciels à base de tâches [PBF10] qui intègre un moteur d'exécution capable d'abstraire la machine. comme illustré par des outils tels que Intel TBB [Rei07]. Dans ces modèles, la granularité correspond au nombre d'instructions processeur contenu dans la tâche. Cette granularité dépend de l'algorithme à paralléliser ainsi que de l'ordonnanceur de tâches.

## 2.6 Runtime

Un moteur d'exécution, ou *runtime*, est un morceau de logiciel utilisé par d'autres logiciels pour abstraire des parties du système. L'idée principale est *compiler une fois, exécuter partout*. Ils sont présents un peu partout et peuvent avoir différentes fonctions. Certains langages dits de haut niveau utilisent un runtime, par exemple Java a un runtime pour gérer son ramasse-miettes. Toutes les implémentations de MPI ont un runtime. Les cadriciels de programmation à base de tâches tendent à utiliser un runtime. Les parties suivantes de ce chapitre se concentreront sur le support des runtimes pour la programmation à base de tâches.

Les runtimes utilisés pour la programmation à base de tâches doivent en premier lieu être capables d'ordonnancer le traitement des tâches tout en respectant l'ordre des dépendances entre les tâches (Fig. 2.17). Ces runtimes doivent aussi fournir un équilibrage de charge entre toutes les ressources matérielles disponibles (potentiellement hétérogènes) dans le but de minimiser le temps de calcul. Certains runtimes s'occupent de transférer des données entre deux ressources potentiellement hétérogènes, comme par exemple entre la mémoire principale et la mémoire d'une carte graphique ou plus simplement entre deux processus. Ces transferts peuvent être implicites, le runtime a connaissance des données qui sont manipulées, ou ils peuvent être explicites avec l'utilisation d'une tâche spéciale qui s'occupera de faire les échanges de données.

Pour améliorer l'équilibrage de charge, les runtimes ont des politiques d'ordonnancement, la plupart de ces politiques sont dynamiques et peuvent s'adapter à la charge courante de la machine. D'autres politiques d'ordonnancement, dites statiques, permettent de réduire le coût d'ordonnancement. L'ordonnanceur parfait

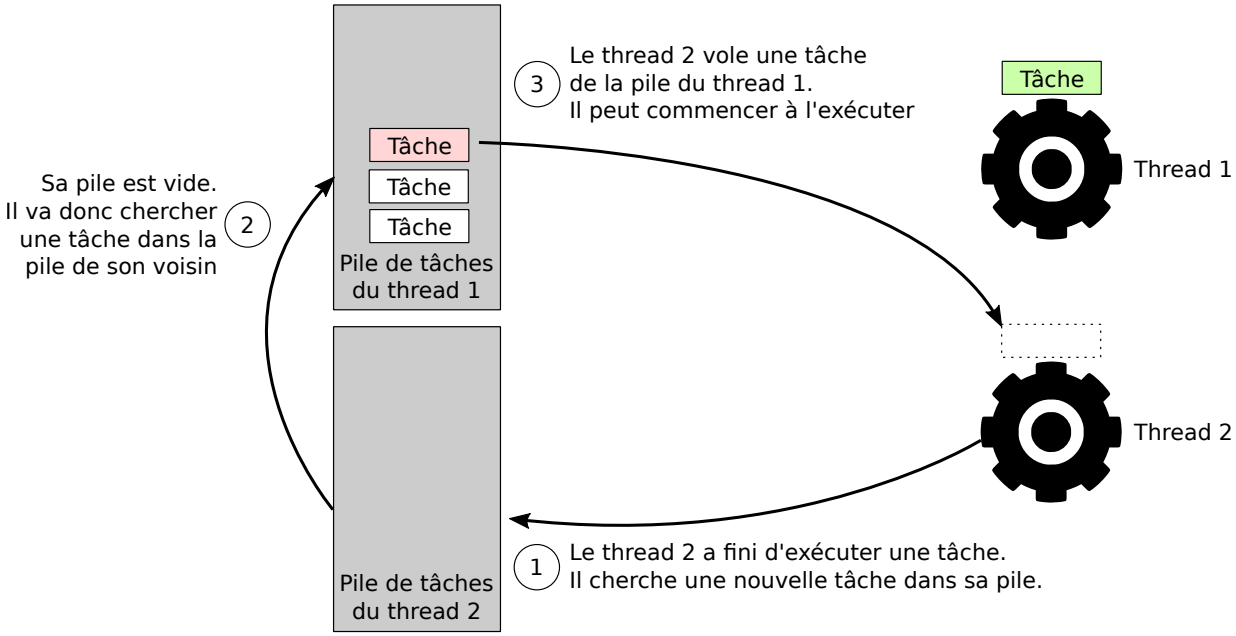



FIGURE 2.18 – Vol de tâche du thread 2 dans la pile du thread 1.

n'existe pas et n'existera sûrement jamais. En effet, trouver le meilleur ordonnancement d'un ensemble de tâches avec un nombre limité de ressources de calcul est un problème NP-complet [1] Il existe des heuristiques d'ordonnancement qui donnent de bons résultats dans la majorité des cas, nous pouvons citer l'algorithme HEFT[THW02]. Si le modèle de programmation le permet, des informations additionnelles peuvent être attribuées aux tâches, comme par exemple une estimation du temps de calcul, ces informations sont ensuite utilisées par l'ordonnanceur pour améliorer le placement des tâches.

L'apparition des premières machines parallèles à mémoire partagée a conduit à la recherche de nouvelles méthodes pour les programmer. Il faut donc distribuer une charge de travail sur plusieurs unités de calcul. Malheureusement, il arrive que cette charge de travail ne soit pas connue l'avance. Une idée est alors apparue pour rendre cette distribution plus flexible : le vol de travail (ou *work-stealing* en anglais). Dès qu'une ressource de calcul n'a plus de travail, elle essaye de voler du travail à une autre ressource. Le langage Cilk [BJK+95], apparu en 1994 et toujours développé sous le nom Intel Cilk Plus [Lei09], permet de faire du vol de travail. Les tâches sont décrites par le programmeur avec des mots clés additionnels au langage C, par exemple le mot-clé *spawn* placé avant l'appel d'une fonction permet à Cilk de comprendre qu'il doit créer une nouvelle tâche et qu'il doit l'ordonnancer. Ces tâches sont empilées sur une pile spécifique à chaque thread. Le vol de tâche se fait par le biais de cette pile de tâches (Fig. 2.18).

Peu de temps après, en 1997, une interface de programmation parallèle voit le jour, il s'agit d'OpenMP [Ope08]. Les premières versions d'OpenMP se concentrent sur le parallélisme de boucle. En ajoutant des annotations autour d'une boucle for, les itérations de la boucle seront distribuées sur les coeurs de calcul. Il existe aussi des annotations permettant d'effectuer des réductions à la fin de la boucle. Ce n'est qu'en 2008 que le support des tâches est ajouté à OpenMP dans sa version 3.0 [ACD+09]. Le programmeur peut créer des tâches parallèles, mais il n'y a aucun moyen de spécifier les dépendances entre les tâches. Les graphes de tâches ne peuvent donc pas être directement utilisés dans OpenMP 3. OpenMP est donc un runtime très complet pour faire du parallélisme de boucle, mais son modèle de tâches se rapproche du modèle de Cilk. La version 4.0 d'OpenMP ajoute le support des dépendances de données dans les tâches. Mais le support d'OpenMP 4.0 dans la plupart des compilateurs est apparut trop tard pour pris en compte dans cette thèse.

Intel TBB [Rei07], dont le développement a démarré en 2006, permet aussi de faire du parallélisme à base de tâches. La gestion des dépendances entre les tâches est à la charge du programmeur, mais TBB

1. Un problème est NP-complet s'il est dans NP et qu'il est au moins aussi difficile que tout problème de NP. La classe NP correspond à l'ensemble des problèmes algorithmiques tels que la complexité de résolution est polynomiale sur une machine de Turing non déterministe et si une solution est possible, cette solution peut être vérifiée en un temps polynomial par une machine de Turing déterministe.

fournit les méthodes nécessaires telles que l'incrémentation et la décrémentation atomique du compteur de référence d'une tâche. Le modèle d'abstraction du matériel dans TBB ne permet pas de connaître le nombre de threads utilisés. Ce choix est fait parce que pour Intel le programmeur n'a pas à se soucier des problèmes d'ordonnancement.

Plus récemment, à la fin des années 2000, la révolution du GPGPU donne lieu à l'apparition de nouveaux runtimes. Les premières méthodes permettant d'utiliser un GPU pour du calcul étaient rudimentaires. Il s'agissait de détourner l'utilisation des shaders programmables des interfaces de programmation graphiques comme par exemple OpenGL. Puis des langages spécifiques ont vu le jour, parmi ceux-ci, les plus populaires sont CUDA et OpenCL. CUDA est développé par NVidia et ne permet de programmer que des GPU NVidia. OpenCL est une spécification du Khronos Group et a pour but de fournir une interface de programmation standard pour programmer toutes sortes d'accélérateurs. Ces accélérateurs peuvent être des GPUs, mais de manière générale il s'agit de co-processeurs déportés. Même si un calcul peut être fait plus rapidement sur le GPU que sur le CPU, il n'est pas toujours plus performant de le faire sur le GPU. En effet, il faut aussi prendre en compte le surcoût des transferts mémoires.

StarPU [ATNW11], développé chez Inria et le LaBRI permet de décrire plusieurs versions d'une routine à la fois pour le CPU et le GPU. Ces morceaux de codes spécifiques à une architecture sont appelés *codelets*. Puis les stratégies d'ordonnancement intégrées à StarPU choisiront la codelet qui permettra d'obtenir le meilleur temps de calcul. Ce choix prend aussi en compte le temps de transfert mémoire entre la mémoire centrale et la mémoire du GPU. Pour avoir une gestion efficace de ces transferts mémoires, StarPU implémente un gestionnaire mémoire. Ce gestionnaire est capable d'effectuer des transferts entre toutes les zones mémoires de la machine (mémoire centrale, mémoire GPU, disques ...) et de maintenir la cohérence des données. Par exemple, si une donnée A est en mémoire centrale et qu'une codelet doit l'utiliser sur le GPU, il y aura d'abord une copie A vers la mémoire du GPU. Ensuite tant que cette donnée n'est accédée qu'en lecture, il y aura deux copies valides, une en mémoire centrale et une en mémoire GPU. Dès qu'une codelet accède à la donnée en écriture, toutes les autres copies sont invalidées et un transfert mémoire sera nécessaire pour les mettre à jour. StarPU intègre aussi plusieurs politiques d'ordonnancement permettant de s'adapter à plusieurs codes de calcul. L'équipe de développement met aussi en avant la possibilité d'écrire son propre ordonnanceur et de l'intégrer à StarPU.

X-KAAPI [GLFR12] est aussi développé chez Inria et permet de programmer des applications qui auront du code qui sera exécuté à la fois sur CPU et sur GPU. Il se différencie de StarPU par son système de vol de tâches dynamique. X-KAAPI va partitionner le graphe de tâches et distribuer les partitions aux threads. Chaque thread a ainsi une première approximation de l'ordonnancement du graphe. Le surcoût de gestion des dépendances est ainsi limité aux frontières entre les partitions. Lors de l'exécution du code, si un thread se retrouve à court de tâches, il va essayer d'en voler à un autre thread que l'on appellera *victime*. Mais il ne va pas la voler complètement, il la laisse dans la pile de la victime en ajoutant comme information que la tâche a été volée. Ainsi lorsque la victime découvrira qu'une tâche a été volée, elle devra s'occuper des dépendances de cette tâche. Ce système permet de réduire le surcoût d'ordonnancement, avec un bon partitionnement, le vol de tâche est plus rare et beaucoup de temps a été économisé sur la gestion des dépendances.

OmpSs [BMD+11] est un runtime qui permet, tout comme StarPU et X-KAAPI, d'écrire du code à la fois pour le CPU et pour le GPU puis de laisser le runtime choisir parmi toutes les versions d'une fonction. La différence entre ces deux runtimes provient surtout de la description du parallélisme. OmpSs propose une approche à base d'annotation de code en étendant la spécification OpenMP version 3. Cette extension permet au mot clé *task* d'être accompagné d'informations complémentaires sur l'utilisation des paramètres en entrée. OmpSs pourra ensuite déduire les dépendances entre les tâches depuis les informations sur les paramètres en entrée. Il n'y a pas non plus d'écriture automatisée de code, le programmeur doit toujours écrire le code spécifique à chaque architecture précédé d'informations concernant la fonction implémentée ainsi que l'architecture cible.

HMPP [DBB07] est un runtime, développé par CAPS entreprise, adressant le problème de la programmation hybride CPU/GPU. Il s'utilise avec des annotations de code de la même manière qu'OpenMP et qu'OmpSs. Puis le code annoté est ensuite transformé par un compilateur source-to-source vers un autre langage spécifique à l'architecture cible, comme le CUDA par exemple. Les transferts mémoires entre la mémoire centrale et la mémoire du GPU peuvent être faits de deux façons :

— implicitement au moment de l'appel de la codelet, mais cette méthode ne permet pas de recouvrir la

communication par du calcul ;
— explicitement avec l'ajout d'annotation, il est donc à la charge du programmeur de choisir le bon moment pour transférer les données.

OpenACC [Ope13] est un standard de programmation développé par un consortium de société dans le but de simplifier la programmation parallèle hybride CPU/GPU. Les spécificités de ce standard ressemblent en de nombreux points à HMPP (annotations, gestion mémoire ...). Son principal avantage est qu'il est soutenu par plusieurs sociétés là où HMPP n'est plus supporté. OpenMP ajoute dans version 4 le support de la programmation hybride, son fonctionnement est identique à OpenACC, seuls les mots-clé changent.

PaRSEC [BBD+13] est un runtime développé à l'ICL permettant de travailler directement en mémoire distribuée. Le parallélisme dans PaRSEC doit être décrit dans un langage spécifique, le JDF. Dans ce langage, le graphe est représenté sous une forme condensée, il n'y a pas besoin d'énumérer toute les tâches, il suffit de décrire les boucles de nos algorithmes pour lesquelles chaque itération correspondra à une tâche. Donc, l'ensemble des tâches du programme sont décrites dans ce langage et ce n'est que la distribution des données en mémoire distribuée qui déterminera le processus qui exécutera la tâche. PaRSEC s'occupe automatiquement des communications entre processus permettant de maintenir une cohérence entre les données. L'inconvénient majeur de ce runtime est son manque de flexibilité. Le format JDF ne permet pas de créer dynamiquement de nouvelle tâche.

## 2.7 Exemples d'ordonnanceurs

La politique d'ordonnancement aura un impact conséquent sur les performances d'un code. Chaque politique aura un surcoût différent en fonction de sa complexité algorithmique ainsi que des paramètres qu'il prend en compte.

Un ordonnanceur type tourniquet distribuera les tâches de manière uniforme sur les différents coeurs de calcul. La complexité algorithmique est donc minimale, mais aucune métrique n'est prise en charge. Nous aurons donc un ordonnancement rapide avec un surcoût très faible, mais de très mauvaise qualité.

Parmi les ordonnanceurs simples, nous pouvons aussi parler de la queue partagée. Tous les threads partagent une queue contenant les tâches pouvant être exécutées. L'équilibrage de charge se fera façon automatique, tant que la queue n'est pas vide il reste du travail à effectuer. Ce type d'ordonnancement impose une implémentation de la queue *thread-safe*, c'est à dire qui garantit les opérations d'ajout et de suppression de tâches dans la queue même lorsque plusieurs threads y accèdent simultanément. Cette implémentation aura aussi des conséquences sur le surcoût d'ordonnancement des tâches. Une implémentation à verrou aura un plus gros surcoût qu'une implémentation à base d'instructions atomiques. Ce type d'ordonnancement ne garantit pas un bon équilibrage de charge. Plusieurs cas peuvent conduire à un déséquilibre de charge :

— si les dernières tâches à ordonnancer ont des coûts très différents ;
— si une tâche qui débloque beaucoup de parallélisme est exécutée très tard.

L'algorithme d'ordonnancement HEFT est un algorithme performant et à faible complexité[THW02]. Il s'agit d'un algorithme glouton, il va essayer de toujours occuper tous les coeurs de calcul. Le graphe de tâches est parcouru en largeur et les tâches sont distribuées les unes à la suite des autres. Pour chaque tâche un temps de terminaison est estimé pour chaque thread, le thread qui aura le temps de terminaison le plus court sera en charge d'exécuter la tâche. Il existe des cas où cet algorithme n'effectue pas le meilleur ordonnancement possible. C'est par exemple le cas lorsqu'une tâche qui en débloque beaucoup d'autres est exécutée tardivement.

## 2.8 Discussion

La simulation numérique est une opération essentielle dans beaucoup d'entreprises. Ce type de simulation se base sur des modèles physiques représentés sous la forme d'équations. L'algèbre linéaire permet de résoudre ces équations. Mais l'utilisation de méthodes de résolution directe est coûteuse en temps de calcul. C'est pourquoi certaines techniques comme les méthodes itératives ont été inventées. Ces méthodes fournissent rapidement une solution acceptable à nos problèmes.

Pour obtenir de plus en plus de puissance de calcul, l'architecture des ordinateurs s'est complexifiée.

Ainsi de nombreux paradigmes de parallélisation sont apparus. Parmi ces paradigmes, la programmation à base de graphe de tâches offre le plus de flexibilité.

Les efforts entrepris pour la programmation hybride CPU/GPU ont permis l'émergence de nouveaux runtimes. Ces runtimes prennent en considération de nouveaux paramètres tels que le poids des tâches de calcul. Ce type de parallélisation est souvent utilisé en algèbre linéaire dense. Les méthodes de parallélisation actuelles peuvent aussi s'appliquer à des noyaux d'algèbre linaire creuse, mais pour la plupart d'entre eux, la granularité des tâches de calcul ne permet pas d'obtenir une parallélisation efficace.

Toutes ces solutions aident donc le programmeur à écrire un programme parallèle efficace. Mais il est toujours du devoir du programmeur de choisir un grain de calcul adapté au type d'ordonnancement choisi. Dans le chapitre suivant, nous allons nous consacrer à étudier ce problème et nous allons essayer de trouver une réponse qui satisfasse au moins les problèmes rencontrés en algèbre linéaire creuse.

# CHAPITRE 3

# UN PROBLÈME DE GRANULARITÉ

**Sommaire**



Nous avons vu dans le chapitre précédent que la programmation à base de graphe de tâches permet de paralléliser un code de calcul. De nombreux runtimes ont été créés ces dernières années pour permettre d'exécuter efficacement toutes les tâches d'un graphe. Des efforts ont été fournis pour résoudre le problème de la programmation hybride CPU/GPU. Mais les efforts concernant les problèmes à granularité fine restent trop peu nombreux. Dans ce chapitre, nous allons nous consacrer à essayer de résoudre ce problème.

Dans un premier temps, nous étudierons plus précisément la parallélisation des noyaux d'algèbre creuse d'un GMRES préconditionné par une factorisation ILU. Nous pourrons voir que ces noyaux ont une description naturelle du parallélisme mais que cette description n'est pas suffisante pour obtenir de bonnes performances. À la suite de cela, nous présenterons Taggre, notre solution pour rendre ce parallélisme efficace. Taggre étant été conçu pour être générique, nous étudierons son application sur des problèmes autres que l'algèbre linéaire creuse. Puis nous reviendrons sur les problèmes de la simulation de réservoir et nous regarderons les résultats de Taggre sur ces problèmes.

## 3.1 Parallélisation de l'algorithme GMRES préconditionné

### 3.1.1 Partie GMRES

L'algorithme GMRES est une méthode itérative utilisée pour résoudre de grands systèmes linéaires creux. La plupart des opérations sont des opérations de type BLAS1 (axpy, dot product...) et sont facilement parallélisable[ARvW03]. L'opération la plus coûteuse de l'algorithme du GMRES non préconditionné est

le produit matrice vecteur, ou *SpMV*[1]. Chaque ligne de la matrice peut être traitée indépendamment les unes des autres. Parmi les choix de parallélisation que nous avons à notre disposition, nous allons choisir du parallélisme de boucle[LZSQ09]. En effet, il permet d'utiliser une granularité adaptative contrairement au parallélisme à base de tâches qui impose le choix d'une granularité. Il permet aussi d'avoir un bon équilibrage de charge avec possibilité de vol de travail là où un paradigme par passage de messages impose un découpage fixe de la charge de travail.

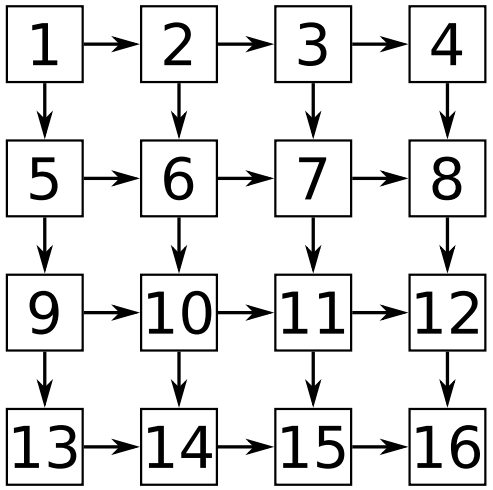
Donc de manière générale, l'algorithme du GMRES se parallélise très bien. Mais nous ne pouvons pas l'utiliser tel quel, nos matrices ne sont pas assez bien conditionnées. Il est nécessaire d'utiliser un préconditionneur pour nos matrices afin obtenir de bonnes performances. Par contre, la partie préconditionneur n'est pas toujours facilement parallélisable. Par exemple, le préconditionneur ILU(0), que nous allons utiliser, est un algorithme dont le parallélisme dépend de la structure de la matrice. Cette structure dépend elle-même du problème traité ainsi que de la numérotation des cellules (Fig. 3.1). Dans le cas d'une numérotation naturelle, le parallélisme est difficile à exploiter. Nous sommes obligés d'utiliser du parallélisme à base de tâches, potentiellement moins performant que du parallélisme de boucle. En changeant la numérotation des cellules, nous pouvons changer le graphe de tâches. Par exemple, lorsque nous factorisons une matrice nous devons procéder ligne par ligne. Les dépendances de factorisation entre les lignes de la matrice ne vont donc que dans un sens, de l'indice de ligne le plus faible vers un indice supérieur. Une numérotation rouge-noir consiste à colorier le graphe représentant notre réservoir de façon à n'avoir aucune connexion entre deux cellules d'une même couleur. Dans le cas d'un maillage 2D structuré, nous pouvons colorier facilement le graphe en parcourant les cellules une à une. La numérotation se fait ensuite en numérotant successivement toutes les cellules d'une même couleur puis en numérotant toutes les cellules de l'autre couleur avec des nombres strictement supérieurs à ceux de la première couleur. Par exemple sur la figure 3.1(c), les cellules 1 à 8 sont rouges et les cellules 9 à 16 sont noires. Les dépendances iront toujours des cellules rouges vers les cellules noires. Nous pouvons donc dans un premier temps factoriser toutes les lignes de la matrice qui correspondent aux cellules rouges. Puis dans un second temps, nous pouvons factoriser les autres lignes, celles qui correspondent aux cellules noires. Nous avons donc un parallélisme de boucle, souvent plus efficace qu'un parallélisme à base de graphe de tâches. Mais qui dans ce cas-là fournit un moins bon préconditionnement[DM89]. Dans le but de garder un bon préconditionnement, ce chapitre sera consacré à exploiter un maximum de parallélisme de l'algorithme ILU(k) tout en gardant une numérotation naturelle.

### 3.1.2 Partie préconditionneur ILU

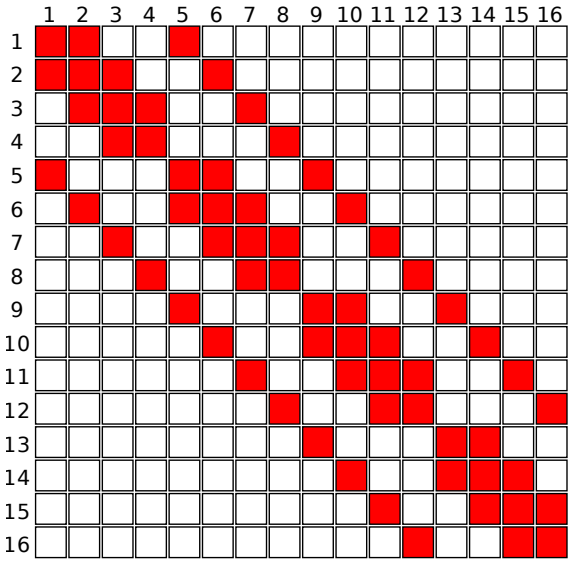
La factorisation LU en algèbre linéaire dense est une méthode pour factoriser une matrice $A$ en deux matrices $L$ et $U$. $L$ est une matrice triangulaire inférieure, toutes les valeurs au-dessus de la diagonale sont nulles. Symétriquement, $U$ est une matrice triangulaire supérieure, toutes les valeurs de $U$ en-dessous de la diagonale sont nulles. Le principal intérêt de cette factorisation est de trouver facilement $x$ dans les équations du type $Ax = y$. Dans le cas où $A = L.U$, le système d'équations $Ax = y$ est transformé en deux systèmes d'équations $L.x_{tmp} = y$ et $U.x = x_{tmp}$. La méthode utilisée pour résoudre les systèmes composés de matrices triangulaires est triviale. Il suffit de résoudre chaque équation ligne par ligne en commençant par la ligne qui n'a qu'une seule valeur non nulle. Ensuite, il faut résoudre la ligne avec deux valeurs non nulles dont une des inconnues provient de la solution précédente, et ainsi de suite jusqu'à la dernière ligne. Il existe du parallélisme à exploiter dans cet algorithme, à chaque fois qu'une inconnue est trouvée, on peut la retirer de chacune des lignes restantes à traiter [DFLL11] (voir Algo. 2).

En algèbre linéaire creuse, la transformation des éléments nuls de la matrice creuse en éléments non nuls est appelée remplissage. La factorisation LU d'une matrice $A$ creuse donnera deux matrices triangulaires denses $L$ et $U$. Or, si l'on souhaite effectuer la factorisation complète dans le cas d'une matrice d'ordre élevé ($> 10\ 000\ 000$), ces deux matrices peuvent ne pas tenir en mémoire. C'est pourquoi en algèbre linéaire creuse, on utilise une version altérée de cette factorisation que l'on appelle factorisation incomplète, ou *ILU*, dont le but est de limiter le remplissage de la matrice. En contrepartie, le résultat de la factorisation obtenu est approximatif et fournit donc un moins bon préconditionneur qu'une factorisation LU pour GMRES. L'algorithme ILU est similaire à l'algorithme LU, mais le remplissage est limité par des conditions définies par l'algorithme. Ces conditions sont principalement de deux formes : soit en limitant le remplissage avec

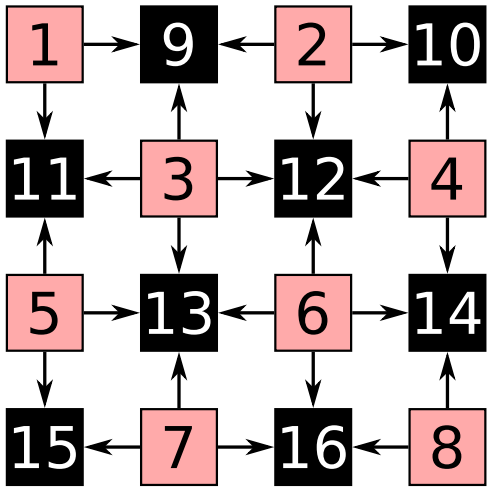
1. Sparse Matrix Vector multiply



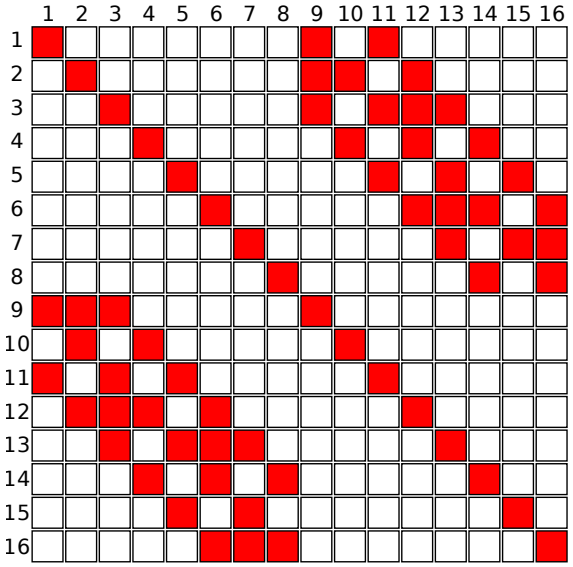

(a) Numérotation naturelle des cellules.




(b) Matrice associée à une numérotation naturelle.




(c) Numérotation rouge-noir des cellules.



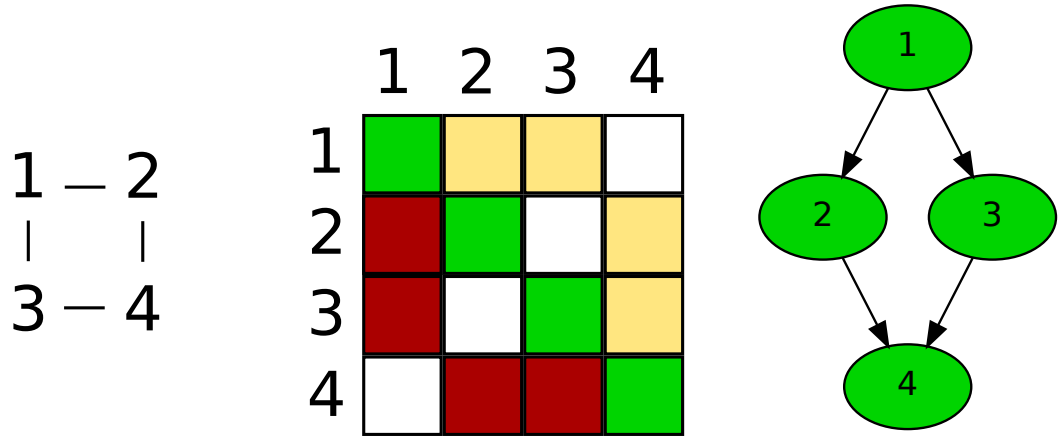

(d) Matrice associée à une numérotation rouge-noir.

FIGURE 3.1 – Impact de la numérotation sur la structure des matrices creuses.

une valeur seuil (ILUt[Saa94]), soit en limitant le niveau d'interaction entre les lignes de la matrice (ILU(k)). Dans le cas ILU(k), le paramètre k sert à limiter le niveau d'interaction entre les lignes. Avec $k = 0$, le motif des matrices $L$ et $U$ reste similaire au motif de la matrice $A$ (Algo 6). En augmentant la valeur de $k$, on augmente aussi le nombre d'interactions entre les cellules.

L'algorithme ILU offre la possibilité de factoriser certaines lignes en parallèle et ce parallélisme se représente naturellement sous la forme d'un graphe de tâches (Fig. 3.2(c)). Dans le cas ILU(0), chaque tâche représente la factorisation d'une ligne de la matrice et les dépendances entre les tâches sont données par le motif de la matrice. En effet, pour factoriser la ligne $i$, nous devons factoriser toutes les lignes $j$ inférieures à $i$ telles que l'entrée $(i, j)$ de la matrice soit non-nulle. Cette dépendance de donnée provient de la ligne 3 de l'algorithme 6. C'est cette dépendance qui nous empêche d'utiliser du parallélisme de boucle pour paralléliser la méthode ILU(0). Donc, à partir du motif des valeurs non-nulles de la matrice, nous pouvons facilement construire le graphe de tâches : la tâche $i$ corresponds à la ligne $i$ de la matrice (ligne 2 à 7 de l'algorithme 6), la liste des tâches prédécesseurs de la tâche $i$ est donnée par l'index de colonne des valeurs non-nulles avant la diagonale dans la ligne $i$ (ligne 3) et la liste des tâches successeurs de la tâche $i$ est donnée par l'index de

ligne des valeurs non-nulles au-dessous de la diagonale de la colonne $i$ (Fig. 3.2(b)).




(a) Un réservoir à 4 cellules (b) Une matrice avec 4 cellules (c) Un DAG à 4 cellules

FIGURE 3.2 – Trois représentations d'un réservoir. Les éléments en rouge dans la matrice déterminent les dépendances dans le graphe de tâches.

**Algorithme 6 :** Factorisation ILU(0) sur place.

**Données** : $M$ : matrice de dimension $n$

**pour** $i = 2$ **à** $n$ **faire**
  **pour** $k = 1$ **à** $i-1$ **et** $M_{ik} != 0$ **faire**
    $M_{ik} = M_{ik}/M_{kk}$
    **pour** $j = k+1$ **à** $n$ **et** $M_{ij} != 0$ **faire**
      $M_{ij} = M_{ij} - M_{ik}M_{kj}$
    **fin**
  **fin**
**fin**

En résumé, le parallélisme de l'algorithme ILU peut se représenter sous la forme d'un graphe de tâches. Chaque tâche représentant la factorisation d'une ligne... ce qui représente peu de calculs. En fait, la plupart des runtimes mettront plus de temps à ordonnancer la tâche que la tâche mettra à s'exécuter.

Les problèmes rencontrés pour paralléliser la factorisation incomplète d'une matrice creuse, ainsi que les résolutions triangulaires associées, sont des problèmes qui représentent bien la difficulté que l'on peut rencontrer avec une parallélisation à grain fin. La description à grain fin de ces algorithmes est naturelle, mais en pratique, une simple parallélisation utilisant des ordonnanceurs répandus, tels que Intel TBB ou OpenMP, ne donnera pas de bonnes performances à cause du faible coût de calcul d'une tâche par rapport au surcoût pris par l'ordonnanceur. En effet, chaque ordonnanceur aura son propre algorithme de distribution des tâches qui prendra différents paramètres en compte. Mais l'évaluation de cet algorithme coûte du temps et si ce temps est du même ordre de grandeur (ou plus grand) que le temps de calcul d'une tâche, l'ordonnanceur sera perçu comme une charge non désirée. On appelle cela le problème de granularité. Un ordonnanceur mettra de l'ordre de la centaine de nanosecondes à ordonnancer une tâche. Pour que ce temps devienne négligeable, il faudrait qu'il ne représente pas plus de 0,1 % du temps de calcul d'une tâche. Les tâches doivent donc durer de l'ordre de la centaine de microsecondes. Mais pour atteindre cette durée, les tâches doivent devenir plus grosses, nous devons factoriser plusieurs lignes à l'intérieur d'une tâche. Mais le choix de ces lignes n'est pas trivial, il faut limiter l'impact sur le parallélisme et ne pas changer le résultat final. Si trop de lignes à factoriser se retrouvent dans une seule tâche, de nombreuses tâches devront attendre que cette tâche finisse avant de pouvoir être exécutées même si les lignes dont elles dépendent sont déjà factorisées, nous avons donc détruit du parallélisme en augmentant le nombre de contraintes entre les tâches. Un intergiciel a été

développé dans cette thèse pour résoudre ce problème en trouvant un bon compromis entre la granularité et le parallélisme de manière transparente pour le programmeur.

De plus, l'ordre de factorisation des lignes de la matrice aura une influence sur les performances. En fonction du nombre de variables primaires, nous avons entre 24 % et 450 % de temps de factorisation en plus par rapport au meilleur temps de factorisation que nous obtenons (Tab. 3.1). La factorisation des lignes de la matrice avec un parcours linéaire des indices donne les meilleures performances. Les lignes consécutives de la matrice correspondent la plupart du temps à des cellules proches qui auront donc des interactions entre elles. La factorisation d'une ligne utilisera donc souvent une partie du résultat de la factorisation de la ligne précédente, ce résultat ayant de grandes chances d'être encore en mémoire cache. Il est donc primordial d'agréger des tâches qui factorisent des lignes consécutives de la matrice.

| Nombre de variables primaires | Temps avec un parcours linéaire des indices (s) | Temps avec un parcours non linéaire des indices (s) | Pourcentage de temps en plus |
|---:|:---:|:---:|:---:|
| 1 | 0.16 | 0.88 | 450 % |
| 3 | 0.43 | 1.43 | 230 % |
| 8 | 3.95 | 5.18 | 24 % |

TABLE 3.1 – Différence de temps d'une factorisation d'une matrice de 1 million de lignes en fonction de la méthode de parcours des indices de lignes. La matrice est stockée au format BCSR. Les lignes de la matrice sont stockées par ordre ascendant.

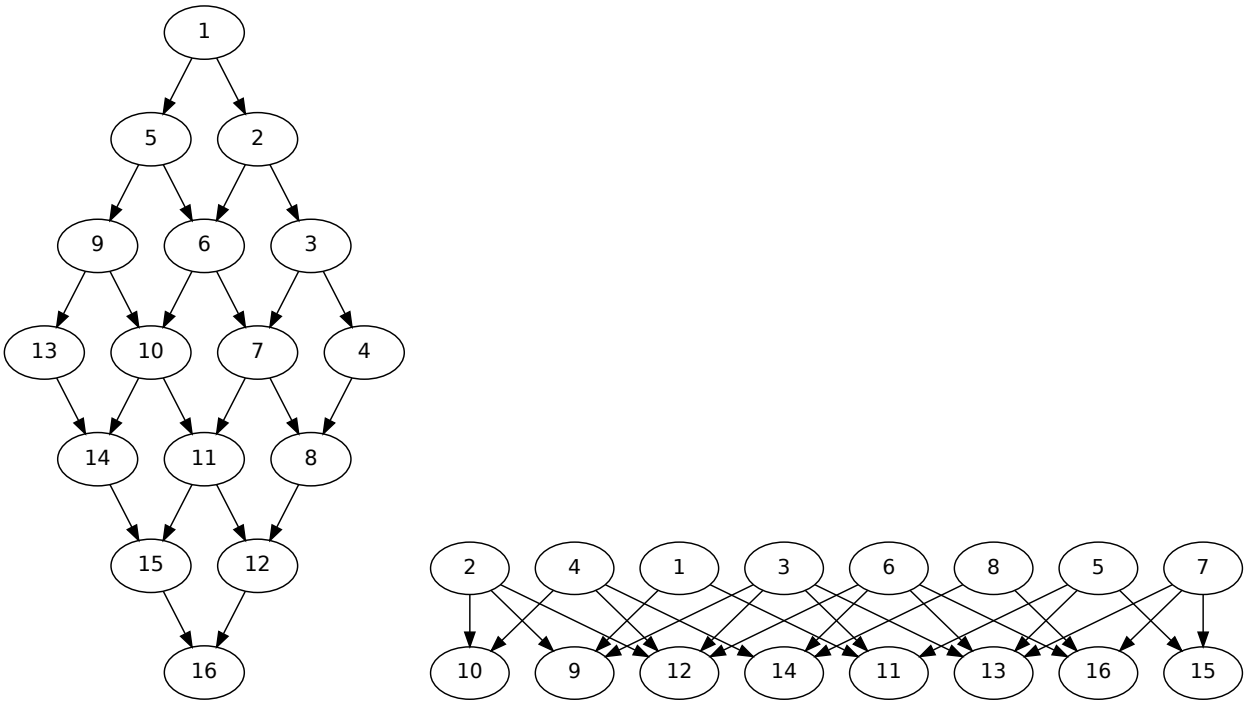
Il est aussi possible d'exploiter le parallélisme de l'algorithme ILU(0) : en agissant sur l'ordre d'élimination des inconnues (boucle extérieure de l'algorithme 2, on peut modifier la structure de la matrice ce qui aura pour effet de changer la factorisation. Une renumérotation rouge-noir permet de factoriser parallèlement la moitié des lignes de la matrice dans un premier temps, puis la seconde moitié des lignes dans un second temps. Cette technique offre énormément de parallélisme, mais aura un impact négatif sur la convergence[Doi91]. Plus récemment, une autre méthode a été développée dans [CP15] permettant de factoriser tous les éléments de la matrice en parallèle tout en gardant une numérotation naturelle. Au lieu de chercher tous les éléments non nuls des matrices $L$ et $U$, l'auteur propose de n'obtenir qu'une approximation de ces éléments. Pour cela, il considère que chaque élément peut s'écrire sous la forme d'une équation avec pour inconnues les autres éléments de la matrice. Ces équations proviennent de la reformulation de l'algorithme ILU. Il obtient donc un système d'équations non linéaires, qu'il résout avec une itération de point fixe, lui permettant ainsi de traiter tous les éléments de $L$ et de $U$ en parallèle. Chaque itération est appelée *sweep* et devra donc être effectuée plusieurs fois. En moyenne, 3 sweeps suffisent à obtenir un résultat proche de la factorisation incomplète. La convergence n'est donc que faiblement dégradée, mais il faut prendre en compte que 3 sweeps ont été nécessaires et il a donc fallu faire 3 fois plus d'opérations qu'une factorisation ILU classique.

Dans notre cas, nous nous intéressons à des renumérotations qui favorisent le regroupement des tâches, mais ne changent pas le résultat numérique.

## 3.2 Pourquoi la granularité est si importante ?

### 3.2.1 Balance parallélisme/surcoût

Les runtimes à base de tâches doivent distribuer les tâches à tous les coeurs de calcul disponibles. Mais cette action n'est pas complètement gratuite, certaines opérations sont obligatoires et leurs temps de traitement dépendent de leurs implémentations. Par exemple, un runtime peut intégrer un modèle de performance avec une gestion des priorités de tâches, ces deux fonctionnalités auront un coût sur le temps d'ordonnancement d'une tache. Généralement, plus le runtime est complexe, plus il doit faire d'opérations et donc plus il prend de temps à distribuer les tâches. L'opération la plus basique, qui est aussi l'opération essentielle au runtime, est la vérification qu'une tâche puisse être exécutée et que celle-ci s'exécute une seule et unique fois. Une implémentation basique de cette opération peut être composée d'une file partagée par tous les coeurs de calcul dans laquelle les tâches sont enfilées dès qu'un coeur de calcul peut les exécuter. Il est nécessaire que l'implémentation de cette queue soit *thread-safe* pour garantir la validité des données quand plusieurs threads l'utilisent en même temps. Malheureusement, même un runtime qui n'effectuerait que cette opération




(a) Exemple de graphe de tâches obtenu avec une numérotation naturelle.

(b) Exemple de graphe de tâches obtenu avec une numérotation rouge-noir.

FIGURE 3.3 – Différence de parallélisme en fonction de la numérotation choisie. La figure 3.1 représente les matrices associées.

aurait un surcoût de calcul lors de l'insertion/extraction de tâches dans la file. Ce surcoût peut être négligé si le temps passé à exécuter la tâche est bien plus grand que le temps passé à ordonnancer la tâche. Mais avec un parallélisme à grain très fin, ce surcoût est loin de pouvoir être considéré comme négligeable et le programmeur doit faire avec. Même les ordonnanceurs statiques ont un surcoût, les tâches sont distribuées à l'avance sur les coeurs, mais l'ordonnanceur doit toujours vérifier si toutes les dépendances de la tâche sont satisfaites. Cela peut être fait plus ou moins efficacement, mais dans tous les cas l'aspect équilibrage de charge dynamique d'un ordonnanceur dynamique est perdu[KA99].

Nous pouvons définir par grain de calcul la durée que met une tâche à être exécutée par rapport au temps que l'on a mis à l'ordonnancer. Un grain fin correspond à une durée d'exécution proche du surcoût de l'ordonnanceur alors qu'un grain grossier aura une durée d'exécution nettement supérieure. Augmenter la taille du grain d'un programme est une solution au problème de tâches trop fines. Mais cette solution réduit aussi les possibilités de parallélisme et d'équilibrage de charge fournis par l'ordonnanceur. Le parallélisme et l'équilibrage de charge sont liés, l'ordonnanceur a besoin d'avoir assez de parallélisme pour fournir un bon équilibrage de charge. Donc, trouver la granularité parfaite est souvent très difficile, voire impossible. Si la granularité est trop grossière, l'ordonnanceur ne peut pas distribuer équitablement les calculs sur tous les coeurs. Mais au contraire, si la granularité est trop fine, le surcoût de l'ordonnanceur peut nuire aux performances du programme.

Il existe des travaux qui dans le passé se sont intéressés au problème d'adaptation de la granularité au nombre d'unités de calcul disponibles. Plusieurs de ces travaux donnent des solutions au problème de granularité en utilisant des techniques de réutilisation des caches pour certaines classes de problèmes tels que la récursivité, diviser pour régner ou des divisions récursives de boucles [VTN11, Rei07, BJK+95, GLFR12, PBF10]. Des travaux comme le cadriciel SCOOP [SP99] fournissent des outils aux applications pour contrôler la granularité. Cependant, le problème de définition de plusieurs granularités doit toujours être géré par le programmeur.

Du côté théorique, l'ordonnancement général des tâches a été longuement étudié, et ce depuis de nombreuses années [KMJ94, THW99]. Les travaux sur l'adaptation de la granularité sont peu nombreux,

mais ils existent comme nous allons le voir dans la section 3.2.2.

### 3.2.2 Solutions actuelles

Lorsque l'on parle de problème de granularité, on peut penser dans un premier temps aux problèmes rencontrés avec du parallélisme de boucle. Chaque itération de la boucle étant indépendante des autres, on pourrait vouloir les ordonnancer indépendamment. Mais avec cette technique, nous ajoutons un surcoût à chaque itération. Si le coût d'une itération est trop petit par rapport à ce surcoût, cette solution n'est pas performante. Donc pour réduire ce surcoût, on pourrait regrouper des itérations ensemble et distribuer ces paquets d'itérations aux coeurs de calcul. En créant un paquet par coeur de calcul, nous avons le surcoût minimal qui permet d'utiliser tous les coeurs de calcul. Malheureusement, si la taille des paquets n'est pas exactement identique, ou si les itérations n'ont pas le même coût, voire même qu'un thread n'ait pas pu être exécuté en même temps que les autres, le temps de calcul n'est pas minimal. Il s'agit du problème d'équilibre de charge, pour obtenir un temps de calcul minimal, il faut que tous les coeurs de calcul aient travaillé pendant la même période. Pour compenser, nous créons des paquets d'itérations plus petits pour permettre de mieux équilibrer la charge entre les différents coeurs de calcul.

OpenMP propose plusieurs stratégies pour construire et distribuer ces paquets d'itérations. La stratégie *static* va couper l'espace d'itération en paquets de taille fixe et les distribuera statiquement sur tous les coeurs de calcul. La stratégie *dynamic* découpera aussi les paquets en avance, mais aura une distribution dynamique, les threads demanderont un nouveau paquet après en avoir terminé un. La stratégie *guided* découpera des paquets de différentes tailles et les distribuera dynamiquement en commençant par les plus gros. Ces stratégies permettent d'adapter l'ordonnanceur au type de problème à paralléliser. Nous pouvons voir que pour chaque stratégie nous avons utilisé la notion de paquet d'itérations pour diminuer le surcoût de l'ordonnanceur. Dans le cas d'un parallélisme de boucle, la création de paquets est simple. Mais dans le cas de parallélisme à base de graphe de tâche, les paquets sont plus durs à construire.

Certains programmes peuvent être écrits de façon à pouvoir choisir facilement une granularité de tâche. Dans ces cas-là, il suffit de faire varier tous les paramètres de granularité jusqu'à obtenir les paramètres optimaux. Il s'agit de techniques dites *autotuning* et qui ne peuvent s'appliquer qu'à un petit ensemble de problèmes.

Certains ordonnanceurs à base de tâches essaient de résoudre le problème de granularité en utilisant des approches différentes. Par exemple, X-Kaapi, présenté dans la section 2.6, a introduit le concept de tâches divisibles aussi appelé *adaptive task model* et fonctionne de la manière suivante : quand un travailleur passe dans l'état d'attente, il émet une requête de travail à un autre travailleur. Les autres travailleurs, qui sont dans l'état travail, doivent vérifier régulièrement s'ils ont reçu une requête de vol. Puis pour traiter cette requête, le travail restant est divisé en deux, la fonction divisant le travail en deux doit être écrite par le programmeur, ce n'est pas automatique. Cette fonction peut être triviale dans le cas de parallélisme de boucle ou dans le cas de parallélisme sous la forme d'arbre. Mais dans le cas d'un graphe quelconque, il n'est pas toujours possible de diviser un graphe en deux graphes totalement indépendants.

Une autre approche possible, comme donnée par Capsules[MRK08], requiert que le programmeur définisse plusieurs granularités. L'ordonnanceur pourra ensuite choisir la granularité qui s'adapte le mieux à la situation. Le programmeur doit donc architecturer son application de façon à pouvoir avoir plusieurs granularités, ce qui peut dans certains cas être difficile à exprimer de manière abstraite.

Pour grossir le grain d'un graphe, on peut aussi regrouper les tâches ensemble et les ordonnancer en une fois. Le regroupement de tâches dans un graphe a aussi été étudié. Les auteurs de l'article [GY92] proposent de créer des groupes de tâches qui seront ensuite ordonnancées sur le même processeur. Les groupes sont formés de la manière suivante : au début chaque tâche appartient à un groupe. Deux groupes sont rassemblés ensemble si leur union diminue le temps parallèle estimé. Mais ce type de regroupement est fait pour optimiser l'ordonnancement d'un graphe de tâches sur des machines à mémoires distribuées. Les tâches fines continuent d'avoir des dépendances entre elles et le surcoût lié à la gestion des dépendances des tâches fines est toujours présent.

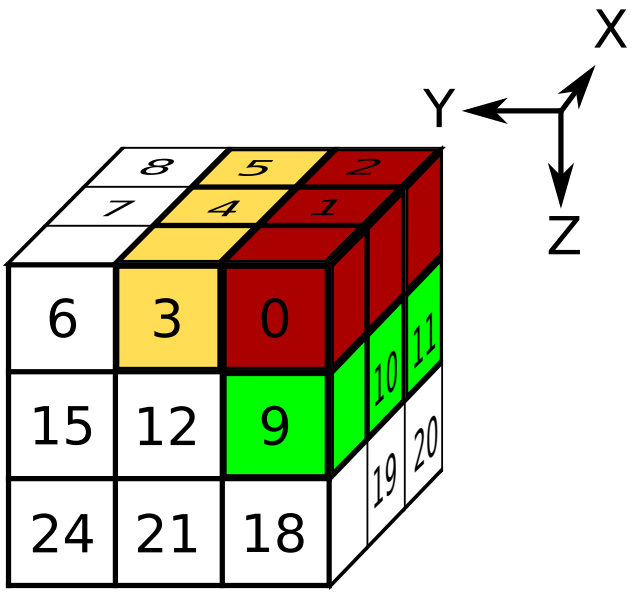



FIGURE 3.4 – Exemple d'agrégation pouvant être faite sur un cube. Ici, nous avons un cube 3x3x3 avec une numérotation naturelle des cellules. Les indices des cellules de même couleurs correspondent aux indices des lignes à factoriser ensemble. Dans cet exemple, seuls trois agrégats ont été dessinés.

## 3.3 Proposition de solution à notre problème de granularité

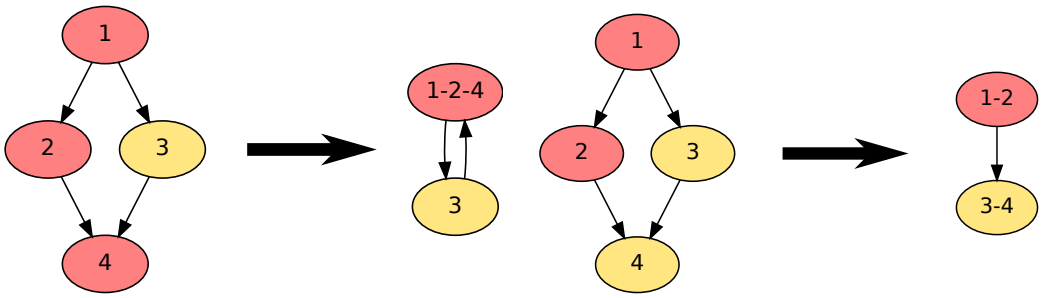
### 3.3.1 Taggre : un cadriciel pour agréger des tâches

Nous avons pour but de garder la façon naturelle de décrire le parallélisme dans les noyaux d'algèbre linéaire creuse. Malheureusement, cette granularité est parfois trop fine, l'ordonnanceur de tâches met plus de temps à choisir quel sera le processeur qui traitera la tâche que le processeur ne met à traiter la tâche. Pour obtenir des performances raisonnables, nous devons augmenter la granularité de la description du problème. Pour cela, nous proposons de créer des groupes de tâches, de considérer chaque groupe comme une seule tâche et d'ordonnancer tous ces groupes en tant que graphe de tâches pour ainsi réduire le surcoût lié à l'ordonnanceur. Au final, nous obtenons un graphe composé de moins de tâches, mais il faut faire attention à ne pas trop réduire le parallélisme fourni par le graphe.

Pour un graphe issu de la simulation de réservoir, nous connaissons une solution efficace capable de répondre à une partie du problème. Dans le cas d'un cube 3D avec une numérotation naturelle, nous pouvons changer la granularité en factorisant des groupes de lignes qui ont des coordonnées $(y, z)$ communes comme le montre l'exemple sur la figure 3.4. Malheureusement, cette méthode ne fonctionne qu'avec la numérotation naturelle et nous impose de connaître la taille du cube pour retrouver à partir d'un indice de ligne les coordonnées $(x, y, z)$ correspondantes. Nous avons donc cherché une méthode pouvant s'appliquer à n'importe quel graphe de tâches.

En partant de la représentation la plus fine sous forme de graphe de tâches du parallélisme, nous avons besoin de calculer un nouveau graphe plus grossier avec moins de tâches. La principale difficulté est de garder la propriété *acyclique* du graphe, car la présence d'un cycle introduirait un inter-blocage dans l'ordonnancement du graphe (Fig. 3.5(a)). L'autre difficulté est de maintenir assez de parallélisme pour pouvoir être capable d'utiliser au mieux les capacités de la machine (Fig. 3.5(b)).

En premier lieu, nous avons développé une nouvelle interface de programmation en C++, cette interface reprend de Intel TBB le concept d'un objet *Tâche* contenant la fonction à exécuter. À cela, nous avons ajouté la description des dépendances dans cet objet. Cette interface nous permet de décrire un graphe de tâches complet et de choisir parmi plusieurs ordonnanceurs celui qui ordonnancera le graphe. Avec cette interface, nous pouvons faire des modifications sur le graphe et le rendre plus grossier. Nous avons appelé cette interface Taggre, ce nom provient de la contraction de *Task Aggregation*. Grâce à l'utilisation d'heuristiques décrites plus loin dans le manuscrit, un programme parallèle peut continuer de décrire son parallélisme de façon naturelle, sans se soucier de la granularité. Taggre s'occupera ensuite de faire le travail nécessaire pour rendre




(a) Agrégation invalide (b) Agrégation valide

FIGURE 3.5 – Exemple de deux agrégations, le résultat de 3.5(a) ne peut pas être ordonnancé à cause du cycle. Le résultat de 3.5(b) peut être ordonnancé, mais il n'y a aucun parallélisme à exploiter.

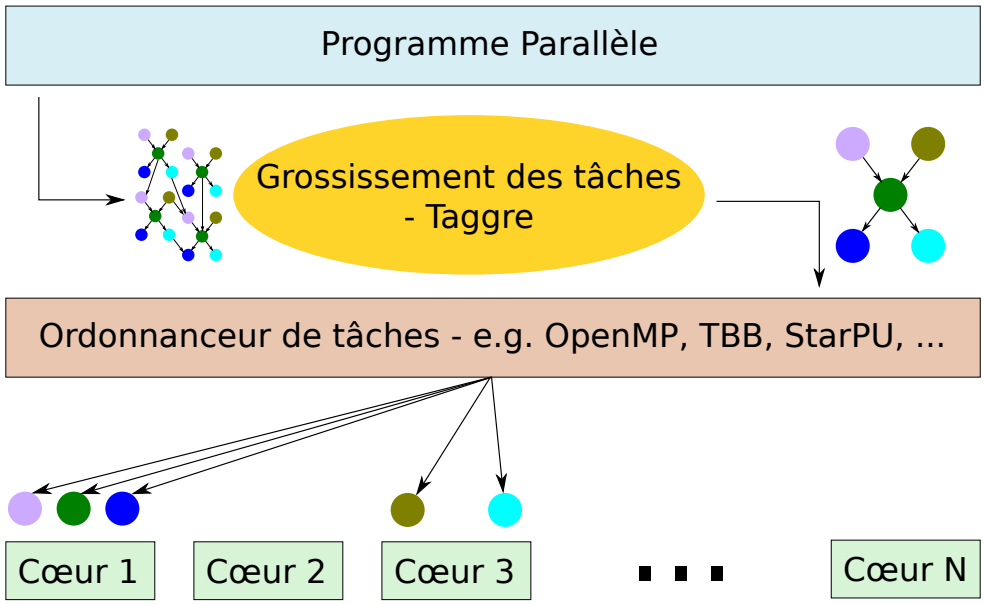



FIGURE 3.6 – Le programme parallèle fournit un graphe de tâches à Taggre. Taggre modifie le graphe. Taggre fournit le graphe à l'ordonnanceur. Le processus d'agrégation est totalement transparent pour l'ordonnanceur.

ce graphe assez grossier pour qu'un ordonnanceur puisse l'ordonnancer efficacement (Fig. 3.6).

Dans un premier temps, le programmeur crée les noeuds du graphe, ces noeuds correspondent aux tâches fines du problème. Pour chaque noeud créé, le programmeur reçoit un identifiant de noeud. Puis le programmeur déclare les arêtes du graphe en utilisant les identifiants des noeuds. Maintenant que le graphe de tâches fines est connu, Taggre peut travailler à grossir le grain. La méthode *coarse* effectuera le travail avec les paramètres que nous expliquerons plus loin dans le manuscrit. Nous avons donc un nouveau graphe composé de tâches grossières, mais ces tâches n'ont toujours pas de code à exécuter. Pour définir le code à exécuter, le programmeur doit appeler la méthode *setup* avec comme paramètre une fonction qui créera les tâches grossières. Cette fonction connaîtra les tâches fines associées à la tâche grossière et pourra ainsi optimiser le code en fonction du nombre et des types de tâches agrégées. Finalement, le programmeur exécutera toutes les tâches du graphe avec la méthode *run*. Il a aussi la possibilité d'utiliser une fonction générique avec la méthode *run_function* (voir Listing 3.1).

### 3.3.2 Les opérateurs d'agrégations

Nous appelons *opérateurs d'agrégations* les différentes heuristiques utilisées par Taggre pour grossir un graphe de tâches. Ces heuristiques ont pour règle de garder la propriété acyclique du graphe de tâches. Quatre heuristiques ont été créées, chaque opérateur s'occupe de résoudre un problème spécifique et aucun

```
1  /* Créer un objet graphe */
2  Schema schema;
3
4  /* Créer les noeuds du graphe */
5  int *mes_taches = new int[nombres_de_lignes];
6  for (int i = 0; i < nombres_de_lignes; i++)
7      mes_taches[i] = schema.new_task(1, i);
8
9  /* Définir les arêtes du graphe */
10 for (int i = 0; i < nombres_de_lignes; i++)
11     for (int j = 0; j < diagonale[i]; j++)
12         schema.declare_dependency(mes_taches[j], mes_taches[i]);
13
14 /* Grossissement du grain du graphe */
15 schema.coarse("CD(4)");
16
17 /* Créer les tâches utilisateur */
18 /* Chaque tâche reçoit trois tableaux contenant les informations
      spécifiques à la tâche */
19 schema.setup([](int nb_taches, int *ids, int *poids, int *affinites) ->
       Task*
20             {
21                 return new my_task_class(n, ids, affinities);
22             });
23
24 /* Puis nous pouvons exécuter toutes les tâches du graphe */
25 schema.run();
26
27 /* Ou utiliser une fonction pour traiter chaque tâche */
28 /* Cette méthode permet de réutiliser le graphe dans différentes
      parties du code */
29 schema.run_function([&](Task *t)
30                     {
31                         compute((my_task_class*)t);
32                     });
```

Listing 3.1 – Exemple d'utilisation de Taggre

d'entre eux ne peut créer de cycle. La création d'un cycle lors d'une agrégation intervient lorsque l'on crée un groupe de tâche dans lequel deux des tâches ont une dépendance indirecte, symbolisé par un chemin dans le graphe, et qu'au moins une des tâches de ce chemin n'appartient pas au groupe.

On pourrait se poser la question de l'utilisation d'un partitionneur de graphe comme opérateur d'agrégation. En effet, les partitionneurs de graphe essaient de créer des groupes de noeuds proches spatialement. Si nous prenons en compte ce seul paramètre, ils feraient des opérateurs de très bonne qualité. Malheureusement, les partitionneurs ne travaillent que sur des graphes non orientés. Le résultat de ces opérateurs serait donc inutilisable parce que les graphes quotients obtenus feront apparaître des cycles.

#### 3.3.2.1 Opérateur séquentiel ("S")

L'opérateur séquentiel, aussi abrégé *S* dans Taggre, est un opérateur très simple. Son but est de fusionner les tâches qui n'apportent pas de parallélisme, elles ne font qu'ajouter du surcoût d'ordonnancement au temps total de la simulation. En agrégeant ces tâches ensemble, on ne perd pas de parallélisme et on économise le temps d'ordonnancement des tâches. On peut reconnaître un groupe de deux tâches séquentielles par le fait

qu'une des tâches n'a qu'un seul successeur et l'autre tâche n'a qu'un seul prédécesseur (Fig. 3.7, Algo. 7).

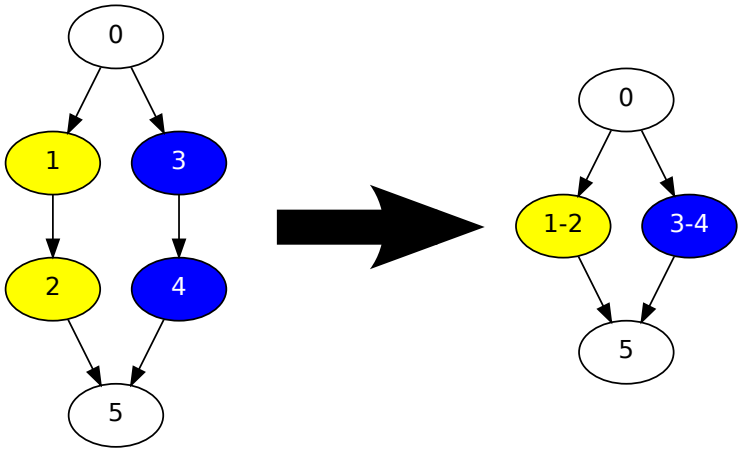



FIGURE 3.7 – Exemple d'agrégation par l'opérateur séquentiel (S).

**Algorithme 7 :** Algorithme de l'opérateur séquentiel (S).

```
   Données : DAG
1  TÂCHES = liste vide
2  mettre les tâches de DAG dans TÂCHES
3  tant que TÂCHES n'est pas vide faire
4  |  T1 = retirer le premier de TÂCHES
5  |  si Le nombre de successeurs de T1 == 1 alors
6  |  |  T2 = premier successeur de T1
7  |  |  si Le nombre de prédécesseurs de T2 == 1 alors
8  |  |  |  T2 devient T1 union T2
9  |  |  fin
10 |  fin
11 fin
```

L'opérateur S ne détruit pas de parallélisme et peut potentiellement réduire le nombre de tâches. En partant de ce postulat, on pourrait penser appliquer cet opérateur systématiquement après chaque agrégation. Mais il faut garder en tête que créer des tâches de granularité trop différentes peut impacter les politiques d'ordonnancement de tâches. Certains ordonnanceurs pourraient utiliser ces tâches fines en tant que tâches *tampons* pour les parties du graphe où il manque du parallélisme.

#### 3.3.2.2 Opérateur front ("F")

Un peu plus intéressant que l'opérateur S, l'opérateur front, abrégé *F* dans Taggre, va limiter le nombre de tâches disponibles au même moment dans l'ordonnanceur. Pour cela, il va travailler à diminuer la largeur du graphe. Certaines propriétés du graphe peuvent être corrélées aux résultats de l'ordonnanceur. Par exemple, la hauteur d'un graphe correspondra au temps minimum qu'il faudra pour traiter tout le graphe si nous avons un nombre infini d'unités de calcul. La largeur du graphe donnera un indice sur le parallélisme exploitable. Plus un graphe est large, plus il offrira de parallélisme. En effet, avec un nombre illimité de coeurs de calcul, on peut exploiter au mieux le même nombre de coeurs que la largeur du graphe. Le cas idéal serait donc un graphe de hauteur proche de 1 avec un nombre conséquent de tâches à la même hauteur. Ce cas ressemble fortement au parallélisme de boucle. Le nombre de tâches du graphe est aussi une propriété à prendre en compte, l'ordonnanceur doit avoir des structures de données efficaces capables de stocker toutes les tâches.

Un graphe fournissant énormément de parallélisme par rapport au nombre de coeurs disponibles n'aura pas forcément un meilleur équilibrage de charge par rapport à un graphe offrant un peu moins de parallélisme. D'autre part, trop de parallélisme peut conduire à la congestion des structures de données servant à maintenir

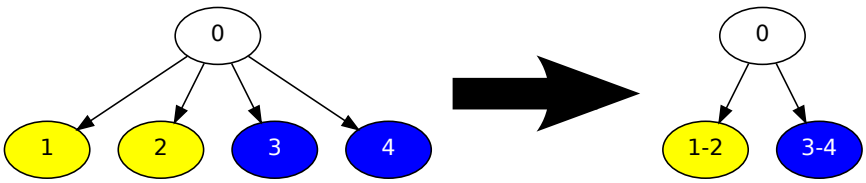



FIGURE 3.8 – Exemple d'agrégation avec l'opérateur F et le paramètre 2.

à jour les tâches prêtes à être ordonnancées. En réduisant la largeur du graphe, on peut ainsi réduire le parallélisme. Mais il faut faire attention à ne pas trop le réduire. C'est pourquoi l'opérateur F prend en paramètre la largeur du graphe souhaitée. L'algorithme de l'opérateur F consiste à faire un parcours du graphe en hauteur et à limiter le nombre de tâches par hauteur, cette limite est donnée en paramètre par le programmeur. Les tâches agrégées ensemble ont la même hauteur, il ne peut donc pas exister de chemin entre elles, il n'y a donc aucun risque de création de cycles (Fig. 3.8).

La première boucle de l'algorithme 8 va parcourir toutes les hauteurs $h$ du graphe. Nous rangeons l'ensemble de toutes les tâches de hauteur $h$ par ordre croissant du nombre de successeurs et de prédécesseurs dans une liste TÂCHES. Si plusieurs tâches ont le même nombre de successeurs et de prédécesseurs, l'ordre de création des tâches est utilisé en complément. Nous calculons le nombre moyen de tâches à agréger ensemble (MOYENNE dans l'algorithme) pour obtenir *MAX* tâches par hauteur, où *MAX* est le paramètre donné à l'algorithme. Dans le cas où MOYENNE ne serait pas un nombre entier, nous utilisons un accumulateur flottant en n'utilisant que la partie entière. La tâche dites MAÎTRE sera la première de la liste TÂCHES. Puis nous allons sélectionner une autre tâche de la liste TÂCHES pour l'agréger à MAÎTRE. Cette sélection est faite de manière à minimiser le nombre de successeurs et de prédécesseurs résultant de l'union des deux tâches. De même que pour le tri de la liste TÂCHES, si plusieurs tâches correspondent, nous sélectionnons la première créée. L'union des deux tâches devient la nouvelle tâche MAÎTRE. Nous répétons cette étape autant de fois que nécessaire afin d'agréger MOYENNE tâches entre elles. Finalement, ces opérations sont répétées autant de fois que nous souhaitons de tâches par hauteur.

**Algorithme 8 :** Algorithme de l'opérateur front (F).

```
   Données : MAX, DAG
 1 pour h allant de 0 à DAG.HAUTEUR faire
 2     TÂCHES = liste des tâches de hauteur h
 3     trier TÂCHES par nombre de successeurs et de prédécesseurs
 4     MOYENNE = taille de TÂCHES / MAX
 5     pour i allant de 0 à MAX faire
 6         MAÎTRE = retirer le premier de TÂCHES
 7         pour j allant de 0 à MOYENNE faire
 8             sélectionner MINIMAL dans TÂCHES tel que le nombre de
 9             successeurs et de prédécesseurs de MINIMAL union MAÎTRE soit le plus petit possible
10             retirer MINIMAL de TÂCHES
11             MAÎTRE devient MAÎTRE union MINIMAL
12         fin
13     fin
14 fin
```

#### 3.3.2.3 Opérateur cube (“C”)

Cet opérateur, abrégé C dans Taggre, a été créé pour être une réponse efficace à nos problèmes. Dans notre cas, le graphe à ordonnancer a exactement la même structure que le réservoir que nous souhaitons modéliser. La plupart du temps, ce modèle sera un cube 3D. En numérotation naturelle et avec un modèle 3D, une bonne agrégation consiste à agréger toutes les tâches d'un axe qui ont les mêmes coordonnées sur les

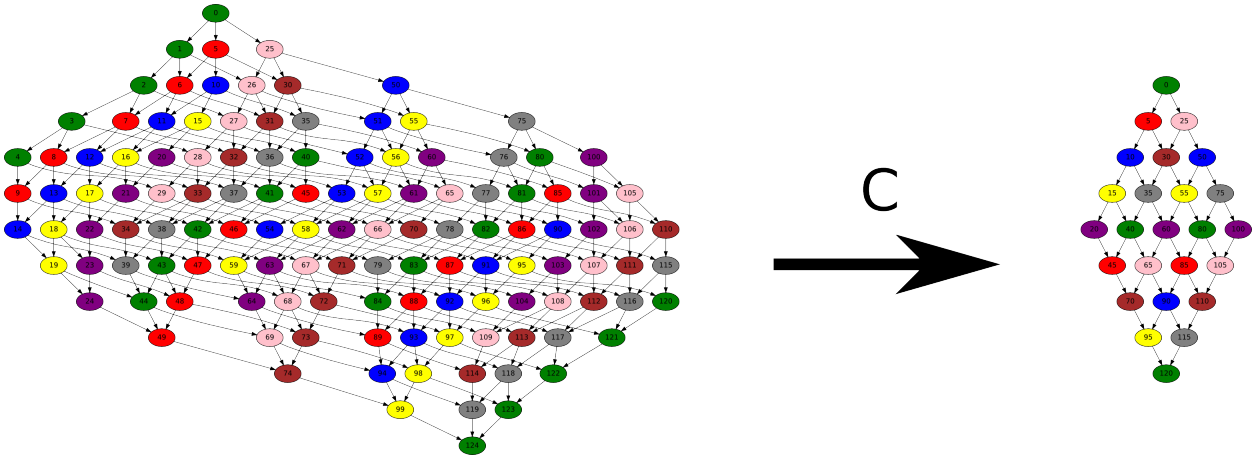



FIGURE 3.9 – Exemple d'utilisation de l'opérateur C sur un cube 5x5x5.

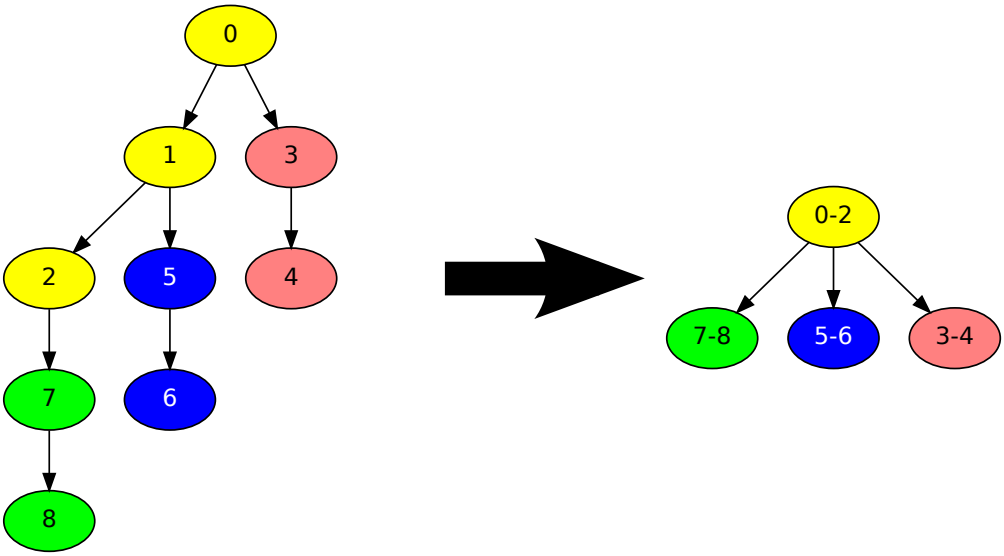



FIGURE 3.10 – Exemple d'utilisation de l'opérateur C.

deux autres axes. Cela correspond à *aplatir* notre modèle 3D en un modèle 2D (Fig. 3.9). Par exemple, un cube de 5 éléments de côté, soit 125 tâches, sera transformé en un carré de 5 éléments de côté, soit 25 tâches.

L'opérateur C pourra être utilisé pour privilégier les effets de cache liés à l'exécution de deux tâches utilisant des données mémoires proches. Le programmeur doit fournir les informations permettant de connaître les tâches pouvant bénéficier de ce type d'optimisation. Pour répondre à notre problème, nous avons supposé que deux tâches ayant un indice consécutif seront plus efficaces si elles sont exécutées l'une après l'autre. Pour fonctionner, cet opérateur a besoin que le programmeur attribue un indice unique à chaque tâche et ainsi avoir un ordre strict sur les tâches. Dans notre cas, on va utiliser la numérotation naturelle des cellules. Puis l'opérateur agrégera ensemble les tâches ayant des nombres qui se suivent ainsi qu'une dépendance entre les tâches. Par exemple sur la figure 3.10 les tâches 2 et 3 ont des nombres consécutifs, mais n'ont pas de dépendance entre elles, elles ne seront donc pas agrégées ensemble.

Comme pour les autres opérateurs, nous devons vérifier qu'aucun cycle ne sera créé. Ajoutons un prédicat à l'algorithme : pour pouvoir utiliser cet algorithme, il faut absolument que pour chaque tâche $i$, l'indice associé à la tâche $i$ soit strictement inférieur aux indices associés aux successeurs de la tâche $i$. Dans le cas d'une factorisation ILU, ce prédicat est toujours vérifié. Dans le cas général, il nous permet de nous assurer qu'aucun cycle ne sera créé. En effet, pour créer un cycle avec cet algorithme, il faudrait qu'il existe un chemin entre deux tâches agrégées qui passe par une autre tâche non agrégée. Or, pour agréger une tâche avec une autre, il faut que la différence de leurs indices soit exactement la différence minimale possible dans le graphe. Donc, en prenant en compte le prédicat, si nous agrégeons une tâche T1 avec son successeur T2, il ne peut pas exister de chemin en T1 et T2 passant par une autre tâche.

L'algorithme 9 décrit l'opérateur C et est composé de trois boucles successives. La première boucle permet de vérifier l'éligibilité du graphe à l'opérateur C. De plus cette boucle permet de connaître le PAS minimal à ajouter pour trouver la tâche suivante. La deuxième boucle va enregistrer les agrégations possibles en parcourant toutes les tâches dans un ordre quelconque et en cherchant des successeurs distants de la valeur PAS. La troisième et dernière boucle s'occupera de faire les agrégations.

**Algorithme 9 :** Algorithme de l'opérateur cube (C).

```
   Données : DAG
 1 PAS = Infini
 2 pour chaque tâche T1 de DAG faire
 3     pour chaque successeur T2 de T1 faire
 4         si indice de T2 <= indice de T1 alors
 5             retourner agrégation impossible
 6         fin
 7         si indice de T2 - indice de T1 < PAS alors
 8             PAS = indice de T2 - indice de T1
 9         fin
10     fin
11 fin
12 pour chaque tâche T1 de DAG faire
13     pour chaque successeur T2 de T1 faire
14         si indice de T2 == indice de T1 + PAS alors
15             enregistrer dans AGRÉGATS : T1 agréger avec T2
16         fin
17     fin
18 fin
19 pour chaque COUPLE dans AGRÉGATS faire
20     COUPLE.T1 devient COUPLE.T1 union COUPLE.T2
21     COUPLE.T2 référence COUPLE.T1
22 fin
```

#### 3.3.2.4 Généralisé

Les opérateurs F et C s'occupent chacun d'un des problèmes que l'on souhaite résoudre avec Taggre. Mais nous pouvons essayer de généraliser leurs propriétés dans un seul opérateur. Ainsi, nous avons construit l'opérateur G. Son but est de privilégier une direction d'agrégation qui puisse être une combinaison de la hauteur et de la largeur. Si seule la largeur est privilégiée, nous nous rapprochons de l'opérateur F. Mais ces deux algorithmes ne sont pas équivalents, la taille des tâches de l'opérateur F n'est pas fixe, elle dépend du nombre de tâches à une hauteur donnée alors que l'opérateur G crée des tâches de taille fixe. En privilégiant les tâches qui permettent de réutiliser le cache, nous retrouvons l'opérateur C. Cette sélection se fait à l'aide de la fonction de tri dans l'algorithme 10, à la ligne 9. Malheureusement cette fonction de tri n'est pas facilement définissable, elle dépendra du problème à résoudre. Il est difficile de trouver les tâches qui favorisent les effets de cache entre elles et il est tout aussi difficile d'exprimer cette relation. De plus, il aurait fallu donner toutes les informations du graphe de tâches au programmeur pour qu'il puisse construire cette fonction efficacement. Taggre a été écrit pour simplifier la programmation par graphe de tâches, nous souhaitons donc que son utilisation reste aussi simple que possible. C'est pourquoi nous n'avons pas implémenté directement cet algorithme mais à la place, nous avons préféré créer un autre opérateur qui définit une fonction de tri.

L'algorithme 10 va simuler l'exécution des tâches (lignes 6 et 13) lorsque celles-ci appartiennent à un agrégat. Seules les tâches ayant déjà eu toutes leurs dépendances satisfaites peuvent aspirer à faire parti d'un agrégat. Cette condition permet d'éviter la création de cycle. Les tâches MAÎTRE sont sélectionnées comme les plus hautes tâches disponibles grâce à la fonction de tri de la ligne 4. Si plusieurs tâches ont la même hauteur, l'ordre de création des tâches est utilisé en complément.

**Algorithme 10 :** Algorithme de l'opérateur généralisé (G).

```
Données : DAG, M : le nombre de tâches dans un agrégat
DISPONIBLES = liste vide
mettre les tâches racines de DAG dans DISPONIBLES
tant que DISPONIBLES n'est pas vide faire
    trier DISPONIBLES par ordre croissant de hauteur
    MAÎTRE = retirer le premier de DISPONIBLES
    mettre les tâches libérées par MAÎTRE dans DISPONIBLES
    COMPTEUR = 0
    tant que COMPTEUR < M et DISPONIBLES n'est pas vide faire
        appliquer une fonction de tri sur DISPONIBLES
        SUIVANT = retirer le premier de DISPONIBLES
        COMPTEUR++
        MAÎTRE devient MAÎTRE union SUIVANT
        mettre les tâches libérées par SUIVANT dans DISPONIBLES
    fin
fin
```

#### 3.3.2.5 Dézoomé

L'opérateur dézoomé est une spécification de l'algorithme G, il permet d'effectuer des agrégations assez génériques et souvent efficaces. En effet, la fonction de tri de cet opérateur va travailler à diminuer à la fois la largeur et la hauteur du graphe. Abrégé *D* dans Taggre, cet opérateur essaye de créer des agrégats de tâches proches spatialement, un peu à la manière d'un partitionneur de graphe. Le nom *dézoomé* provient du fait que la structure globale du graphe ne change que très peu pendant l'agrégation même si le nombre de tâches a lui considérablement diminué. Dans les meilleurs cas, le nombre de tâches peut être divisé par le paramètre donné par le programmeur (Fig. 3.11).

L'algorithme 10 sera utilisé pour implémenter l'opérateur dézoomé. La fonction de tri nécessaire à cet algorithme va utiliser trois critères. Le premier critère est le nombre de prédécesseurs par rapport à la tâche MAÎTRE. Les tâches qui ont beaucoup prédécesseurs se retrouveront en première position (ordre décroissant). Puis, si ce critère n'est pas discriminant, la hauteur de la tâche sera utilisée, les tâches seront triées par ordre croissant de hauteur. Finalement, l'ordre de création des tâches sera utilisé pour permettre un ordre total.

## 3.4 Application de Taggre dans un cadre général

Taggre a été conçu pour répondre aux problèmes rencontrés avec les graphes de tâches issus de la simulation de réservoir. Mais il peut aussi être utilisé sur des graphes de tâches provenant d'autres types de problèmes. Nous allons dans un premier temps évaluer les heuristiques utilisées par Taggre sur des problèmes utilisant des graphes de tâches. Puis nous étudierons les résultats que nous obtenons lors de la simulation de réservoir.

Pour évaluer l'amélioration apportée par chaque heuristique, nous avons intégré dans Taggre un simulateur minimal qui estimera le temps d'ordonnancement du graphe. Cette estimation est essentielle dans la mesure où elle permet de mesurer le parallélisme restant pouvant être extrait du graphe grossier. Pour modéliser le plus fidèlement possible les améliorations de performances liées à l'agrégation, nous prenons en compte trois paramètres. Le premier paramètre est le surcoût lié à l'ordonnancement de la tâche, il s'agit du temps à ajouter au début de chaque exécution d'une tâche. Le deuxième paramètre correspond à l'amélioration des effets de cache. Dans le cas de la factorisation ILU, il s'agit du temps gagné lorsque deux lignes consécutives de la matrice sont factorisées dans la même tâche. Ce paramètre prend la forme d'un nombre réel entre 0 et 1 qui correspond au pourcentage de temps d'exécution d'une tâche quand celle-ci est exécutée juste après une tâche dont elle peut bénéficier des effets de cache. Le troisième paramètre est le poids des tâches, il varie en fonction de la tâche qui est exécutée. Ces trois paramètres sont dépendants du problème traité. Dans les différents cas tests que nous essayerons, nous choisirons arbitrairement ces valeurs pour essayer différentes

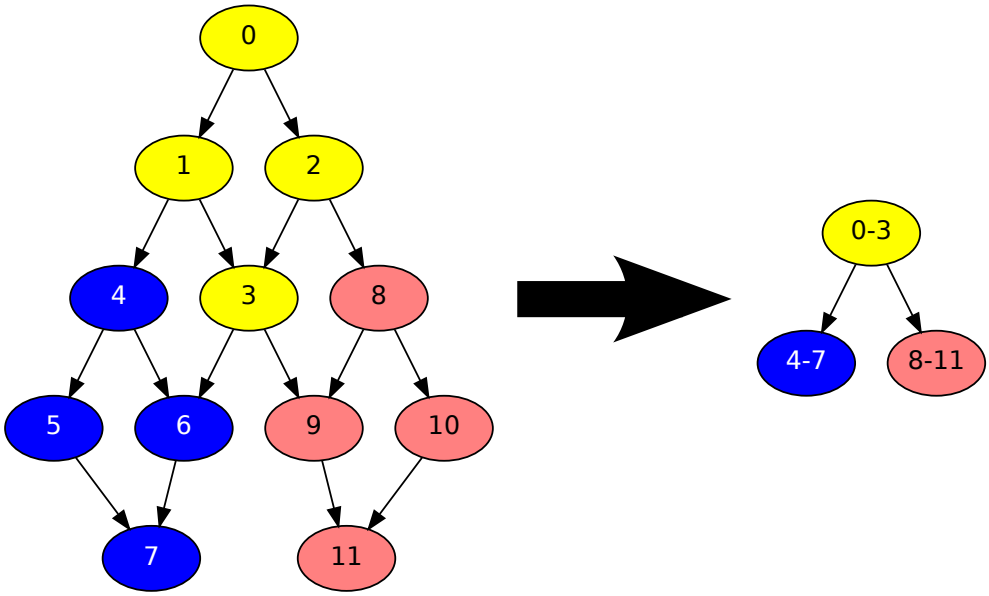



FIGURE 3.11 – Exemple d'utilisation de l'opérateur D avec le paramètre 4. Le nombre total de tâches a bien été divisé par 4.

agrégations.

### 3.4.1 Évaluation du simulateur de tâches

Le simulateur que nous utilisons va simuler l'exécution des tâches en prenant comme temps d'exécution le poids des tâches. Chaque coeur est simulé avec une structure contenant la tâche en cours d'exécution et l'heure a laquelle la tâche sera finie d'être exécutée. Nous utilisons un algorithme glouton d'ordonnancement, le coeur ayant l'heure de terminaison la plus petite libère les dépendances de sa tâche en cours et prend la première tâche disponible dans la file des tâches disponibles. Si la file est vide, on passe à l'heure de terminaison suivante. Le poids d'une tâche grossière dépend des tâches fines qui la composent. Si l'exécution de deux tâches fines est soumise à une amélioration des effets de cache, alors le poids de la deuxième tâche sera multiplié par un coefficient représentant les améliorations des effets de cache. L'heure suivante de terminaison de la tâche sera calculée de la façon suivante : $H = H + T_{ordonnancement} + P_{tache\_grossiere}$

Dans un but de valider les prochains résultats, il est important de tester la validité des résultats fournis par notre simulateur de tâches sur des problèmes dont on connaît les paramètres ainsi que le temps d'exécution obtenu avec diverses agrégations. Nous avons décidé de normaliser le coût d'une tâche à la valeur 1 et d'effectuer les calculs relativement à cette valeur. Dans le cas d'une factorisation ILU(0) d'un cube avec 3 variables primaires, les paramètres du simulateur valent 1,5 pour le surcoût d'ordonnancement d'une tâche et 0,7 pour le gain lié aux effets de cache. Ça signifie que l'ordonnanceur met 1,5 fois plus de temps à ordonnancer une tâche de poids 1 que cette tâche ne met à factoriser une ligne. Ça signifie aussi que lors de la factorisation de deux lignes consécutives, la factorisation de la deuxième ligne sera 30% plus rapide.

### 3.4.2 Les 13 nains de Berkeley

La collection des *13 nains de Berkeley*[ABC+06] est une méthode de classification des problèmes en fonction de leurs motifs de calculs et de communications. Il est intéressant de voir si Taggre peut répondre aux 13 problèmes et si c'est le cas quelle serait la stratégie d'agrégation à utiliser. En effet, le choix des opérateurs dans Taggre se fait à l'appréciation du programmeur. Malheureusement, tous les problèmes ne se prêtent pas à cet exercice. Certaines classes de problèmes ne peuvent pas se paralléliser avec du parallélisme à base de graphes de tâches, ou ne présentent aucun intérêt à être parallélisées de cette façon (MapReduce, Graph Traversal, Dynamic Programming, Backtrack and Branch-and-Bound, Graphical Models et Finite State Machines). Les problèmes rencontrés en algèbre linéaire dense sont souvent réguliers et le programmeur peut facilement choisir une granularité. La classe de problème *structured grids* peut être parallélisée avec du

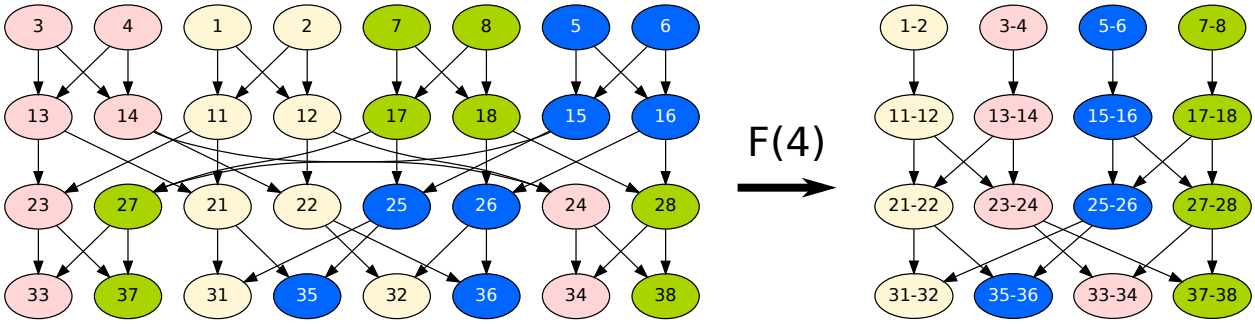



FIGURE 3.12 – Exemple d'agrégation sur un graphe de méthode spectrale.

parallélisme à base de graphe de tâches, mais il ne s'agit pas du meilleur paradigme de parallélisation pour cette classe. La même réflexion peut être faite pour la classe de problème *unstructured grids*.

La classe de problème *combinational logic* s'apparente à du parallélisme d'instructions. Cette granularité est trop faible pour être traitée par un ordonnanceur de tâches logiciel, seuls les concepteurs de microprocesseurs peuvent traiter ces problèmes.

Il ne reste donc plus que trois classes de problème pouvant utiliser Taggre, l'algèbre linéaire creuse, les méthodes N-Body et les méthodes spectrales. L'algèbre linéaire creuse comporte un ensemble très vaste d'algorithmes qui ne se parallélisent pas tous de la même manière. Dans ce manuscrit, nous étudierons plus particulièrement la parallélisation du produit matrice vecteur creux et de la factorisation ILU.

### 3.4.2.1 Méthodes N-Body

Les méthodes N-Body consistent à simuler les interactions entre particules dans un espace au cours du temps. Chaque particule aura un état particulier (masse, position, vitesse ...) et chaque interaction aura pour effet de changer cet état. Pour chaque pas de temps, il faut mettre à jour chaque particule en fonction de l'état des autres particules. Le calcul exact étant trop coûteux (complexité en $O(n^2)$), il existe des heuristiques. L'une d'entre elles consiste à diviser l'espace en plusieurs sous-espaces et de seulement simuler exactement les interactions des particules du même sous-espace et faire une approximation des particules pour les interactions avec celles d'un autre sous-espace. Des travaux de recherche se sont concentrés à traiter ce problème avec des graphes de tâches[ABC+14], et comme le graphe est un arbre, ils ont eux-mêmes fait de l'agrégation de noeuds pour obtenir une bonne granularité. Taggre n'est donc pas utile dans ce cas, le problème de granularité étant déjà traité par le programmeur.

### 3.4.2.2 Méthodes spectrales

Dans le cas des méthodes spectrales, le graphe représentant le parallélisme est plus large que haut. Le motif du graphe est en papillon, nous avons donc beaucoup de connexions entre chaque niveau du graphe, l'opérateur F pourra s'occuper de regrouper les tâches d'un même niveau ensemble. Celui-ci nous permettra de diminuer le nombre de tâches en diminuant le parallélisme (Fig.3.12). Pour évaluer le gain lié à l'agrégation, nous allons utiliser le simulateur de Taggre avec les paramètres 0,9 pour les effets de cache et 5 pour le coût d'ordonnancement. La hauteur du graphe sera le logarithme en base 2 du nombre de tâches par niveau. La table 3.2 nous montre les résultats obtenus en utilisant différents opérateurs d'agrégation. Ces nombres sont exprimés en équivalent du nombre de tâches exécutées de manière séquentielle. Donc plus le nombre est petit, meilleur est le résultat. Les opérateurs D et F offrent presque les mêmes performances. Si les paramètres du simulateur sont corrects (très petites tâches) alors l'agrégation permettra de diviser le temps de calcul par 6.

| Nombre de tâches par niveau | Types d'agrégations | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | Ø | F(6) | F(12) | F(24) | D(8) | D(16) | D(32) | D(64) |
| 32768 | 262146 | 108246 | 51714 | **39487** | 70425 | 55788 | 40980 | 46409 |
| 1024 | 1690 | 2162 | 937 | 903 | 1035 | 965 | **899** | 1033 |

TABLE 3.2 – Résultats en équivalent tâches séquentielles du simulateur d'exécution de tâches sur un graphe typique des méthodes spectrales.

#### 3.4.2.3 Collection de matrices creuses de l'université de Floride

La généralisation de l'approche proposée dans Taggre aux problèmes représentés par les 13 nains de Barkeley n'étant pas facile à montrer d'un point de vue théorique, nous nous sommes intéressés à l'utilisation de Taggre sur des exemples de problèmes correspondant à des graphes de tâches très variés. La collection de matrices creuses de l'université de Floride est un ensemble de matrices provenant de diverses simulations. Parmi toutes ces matrices, nous en avons choisi quatre ayant des motifs différents de ceux que nous pouvons retrouver en simulation de réservoir. Nous ne nous intéressons pas aux propriétés physiques de ces simulations, mais seulement aux connexions des noeuds dans le graphe. Pour obtenir un graphe de tâches à partir de ces matrices, nous n'utiliserons que la partie triangulaire basse des matrices. Une entrée dans cette partie correspond à une dépendance dans le graphe de tâches. Nous nous assurons donc d'obtenir un graphe orienté acyclique. N'ayant pas d'algorithmes spécifiques à utiliser avec ces graphes de tâches, nous en avons créé un de toutes pièces. C'est pourquoi nous allons choisir des paramètres de simulation arbitraires pour le poids des tâches et pour les effets de cache. Parmi les 4 tests choisis, il y en a 2 petits ($<$ 10 000 lignes) et 2 grands ($>$ 1 000 000 lignes). Pour chaque taille, nous avons choisi un cas qui privilégiera le paramètre des effets de cache et un autre cas qui

privilégiera le paramètre de granularité. Ces paramètres auront une incidence sur le choix des opérateurs d'agrégations. Nous testerons ces différents opérateurs grâce à notre simulateur.

| Nom de la matrice | Nombre de lignes/colonnes | Nombre de non-zéros | Paramètre des effets de cache | Paramètre de granularité |
|---:|:---:|:---:|:---:|:---:|
| Pajek/EPA | 4 772 | 8 965 | 0,95 | 0,02 |
| SNAP/roadNet-PA | 1 090 920 | 3 083 796 | 0,70 | 1 |
| Gleich/wb-cs-stanford | 9 914 | 36 854 | 0,50 | 0,001 |
| Williams/webbase-1M | 1 000 005 | 3 105 536 | 0,98 | 0,5 |

TABLE 3.3 – Descriptions des matrices utilisées.

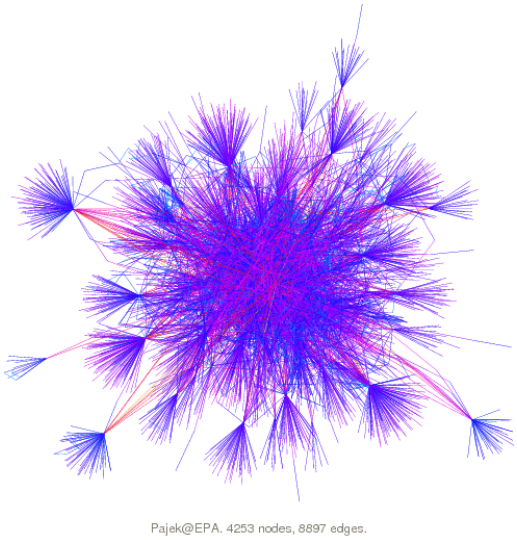
Les premiers résultats concernent la matrice Pajek/EPA, ce graphe représente les liens de pages vers *www.epa.gov*. Le graphe est plutôt petit (environ 5000 noeuds), nous n'aurons donc pas beaucoup de tâches. Nous avons choisi pour le simulateur de tâches des valeurs représentant des tâches plutôt grosses (50 fois le coût d'ordonnancement) et qui ne bénéficie presque pas des effets de cache. L'agrégation ne permet qu'un gain de 30% dans les meilleurs cas (Table 3.4). Aucun opérateur d'agrégation ne se démarque des autres.

| Types d'agrégations | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ø | D(2) | D(4) | D(8) | D(16) | D(32) | F(24) | F(36) | F(42) | F(64) | C |
| 598 | 403 | 400 | 428 | 435 | 523 | 407 | **392** | **392** | **392** | 406 |

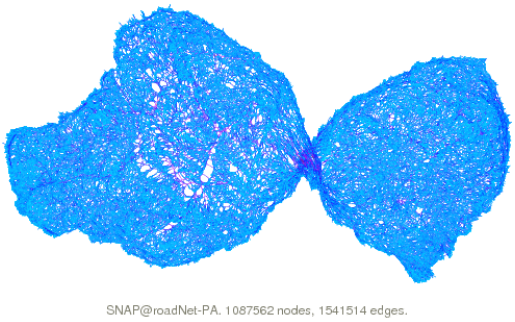
TABLE 3.4 – Résultats en équivalent tâches séquentielles du simulateur d'exécution de tâches sur Pajek/EPA sur 12 coeurs de calculs.

Les résultats suivants (Table 3.5) concernent un graphe représentant les intersections de routes de Pennsylvanie (SNAP/roadNet-PA). Le graphe est assez grand, nous pourrons agréger des noeuds ensemble. La granularité des tâches que nous avons choisie est petite, l'ordonnanceur met autant de temps à ordonnancer la tâche que la tâche met à s'exécuter. De plus, les effets de cache peuvent améliorer le temps de calcul. L'opérateur D se démarque des autres opérateurs en divisant par deux le temps de calcul (Table 3.5). L'opérateur F n'a pas fourni une bonne agrégation à cause des effets de cache. L'opérateur C a détruit du parallélisme et donne pour ce cas de très mauvais résultats.

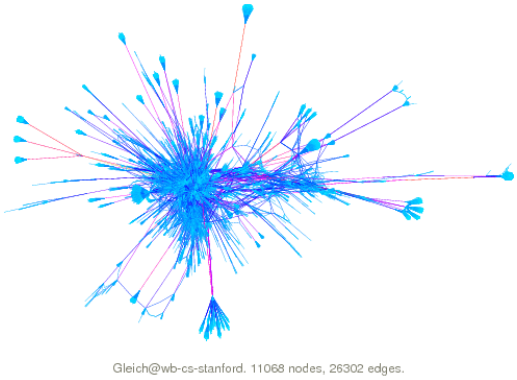
Le troisième cas test représente les liens entre les pages du site web du département informatique (*Computer Science*) de Stanford en 2001 (Gleich/wb-cs-stanford). Il s'agit d'un petit graphe d'environ 10000



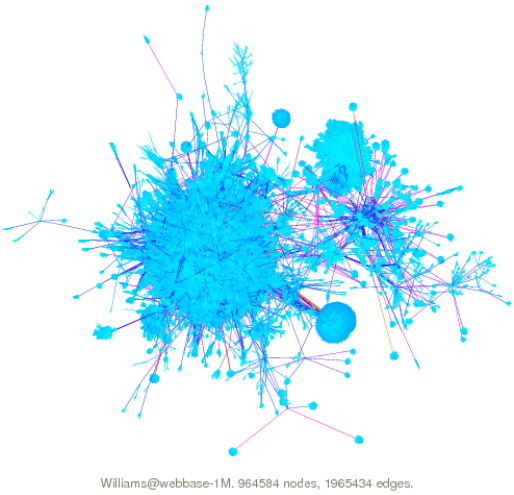

(a) Pajek/EPA




(b) SNAP/roadNet-PA




(c) Gleich/wb-cs-stanford




(d) Williams/webbase-1M

FIGURE 3.13 – Représentation du graphe de connexions des quatre matrices choisies.

| Types d'agrégations | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ø | D(16) | D(64) | D(256) | D(1500) | D(2048) | F(24) | F(36) | F(42) | F(64) | C |
| 136370 | 84244 | 75919 | 72145 | **69359** | 69607 | 92211 | 80529 | 80179 | 80190 | 181826 |

TABLE 3.5 – Résultats en équivalent tâches séquentielles du simulateur d'exécution de tâches sur SNAP/roadNet-PA sur 12 coeurs de calculs.

noeuds. Les tâches ont une granularité grossière et peuvent bénéficier d'améliorations des effets de cache. Tous les opérateurs d'agrégations donnent environ les mêmes résultats (Table. 3.6).

| Types d'agrégations | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ø | D(2) | D(4) | D(8) | D(16) | D(32) | F(24) | F(36) | F(42) | F(64) | C |
| 1267 | 765 | 729 | 733 | 753 | 923 | 739 | 736 | 736 | 737 | **722** |

TABLE 3.6 – Résultats en équivalent tâches séquentielles du simulateur d'exécution de tâches sur Gleich/wb-cs-stanford sur 12 coeurs de calculs.

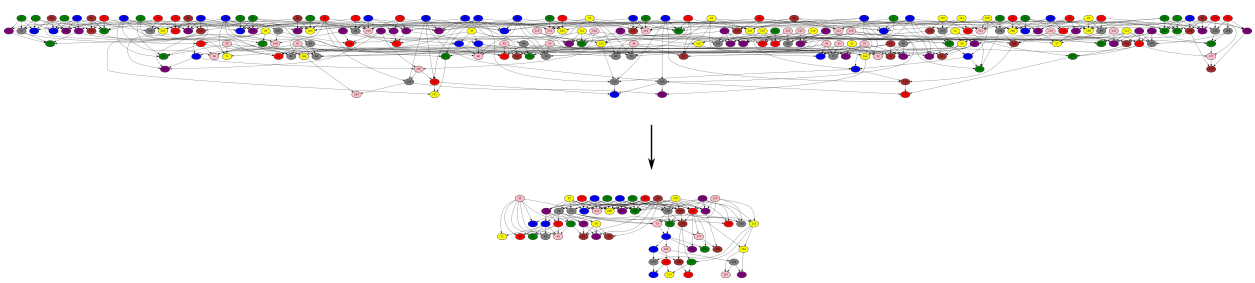


FIGURE 3.14 – Exemple de graphe d'une factorisation ILU(0) avec un ordering nested dissection d'une matrice représentant un cube 6x6x6. Nous avons appliqué l'opérateur F avec le paramètre 12.

Le dernier cas test est une matrice de connexions de site web (Williams/webbase-1M). La matrice est grande, nous avons choisi comme paramètre de tâche, une granularité fine avec un très petit gain sur les effets de cache. Les gains liés à l'agrégation sont faibles (35%) et les opérateurs D et F donnent les mêmes performances. L'opérateur C n'a pas été très utile.

| Types d'agrégations | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ø | D(16) | D(64) | D(256) | D(1500) | D(2048) | F(24) | F(36) | F(42) | F(64) | C |
| 125512 | 84898 | 82837 | 82913 | 86024 | 88551 | **82453** | 82490 | 89461 | 87427 | 124889 |

TABLE 3.7 – Résultats en équivalent tâches séquentielles du simulateur d'exécution de tâches sur Williams/webbase-1M sur 12 coeurs de calculs.

### 3.4.3 Factorisation ILU(k) avec renumérotation des cellules

Lorsque nous renumérotons les cellules, nous obtenons un graphe de tâches différent. Dans le cas d'une factorisation ILU(0), une numérotation rouge-noir (cf. section 3.1.2) donne un graphe très large avec seulement 2 niveaux de hauteur. Nous pouvons aussi utiliser une numérotation de type *nested dissection* pour obtenir un autre type de graphe. Ce graphe aura la forme d'un arbre. De plus, augmenter le niveau de remplissage de l'algorithme ILU(k) créera de nouvelles connexions dans le graphe. Nous allons utiliser notre simulateur avec une numérotation nested dissection des matrices dans le but d'évaluer les différentes stratégies d'agrégations. Nous définissons les paramètres 0,6 pour les effets de cache et 2 pour le coût d'ordonnancement. Ces paramètres correspondent aux valeurs trouvées à l'aide de mesures expérimentales lors d'une factorisation ILU(0) sur un cas à 3 variables primaires. Donc le temps d'exécution d'une tâche est 2 fois moins long que le temps d'ordonnancement. En regardant le graphe de tâches (Fig. 3.14), on s'aperçoit que le graphe est plus large que haut. L'opérateur F devrait donner de très bons résultats sur ce graphe. Mais comme les effets de cache sont importants, il est possible que l'opérateur D puisse faire aussi bien. Lorsque nous regardons les résultats d'une factorisation ILU(0) sur un cube 100x100x100 (Table. 3.8), l'opérateur F donne les meilleurs résultats. En créant plus de tâches que de coeurs de calcul par niveau, on autorise un meilleur équilibrage de charge pour les niveaux qui n'ont pas assez de tâches. L'opérateur C n'est pas du tout efficace car le graphe n'est pas assez haut.

| Types d'agrégations | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ø | D(16) | D(64) | D(256) | D(1500) | D(2048) | F(24) | F(36) | F(42) | F(64) | C |
| 250005 | 83549 | 72145 | 67706 | 65790 | 66162 | 65070 | 61493 | **61195** | 61219 | 246386 |

TABLE 3.8 – Résultats du simulateur d'exécution de tâches avec ILU(0) et une granularité très fine (3 variables primaires) sur 12 coeurs de calculs.

Supposons dans un deuxième temps que les tâches soient plus grosses et que le paramètre cache soit moins important. Il s'agit d'un cas à 8 variables primaires, pour lequel nous avons recalculé les paramètres du simulateur à partir des mesures expérimentales. Nous travaillerons donc avec les valeurs 0,8 pour les effets de cache et 0,5 pour le coût d'ordonnancement. Ordonnancer une tâche ne coûte plus que la moitié du

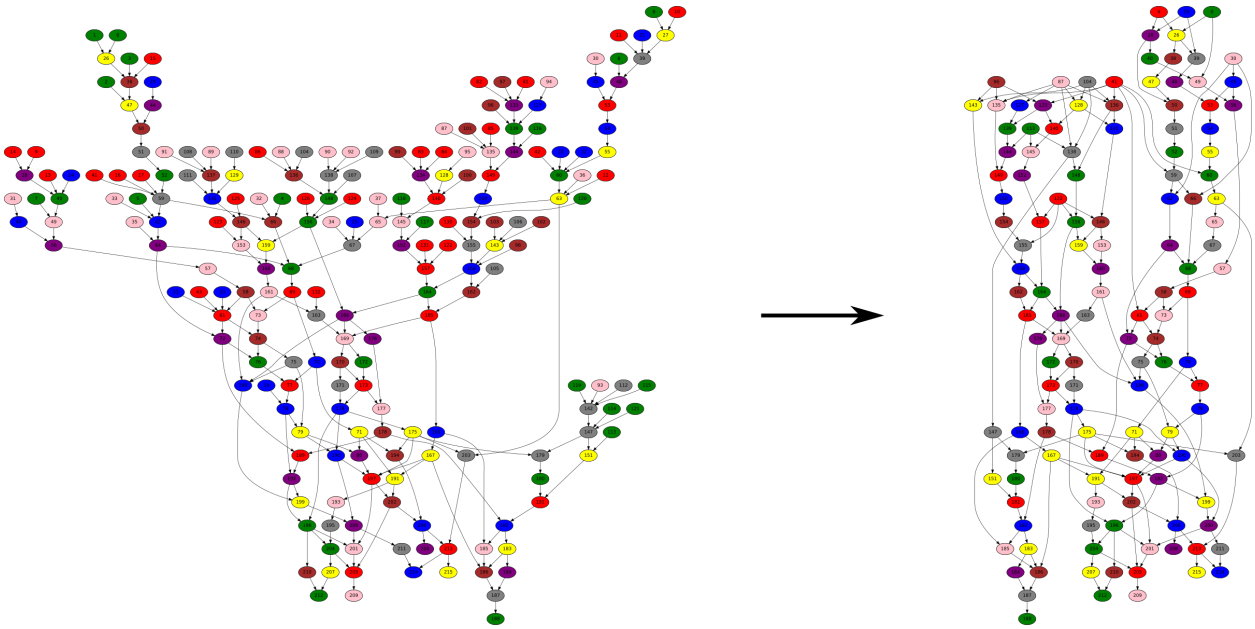


FIGURE 3.15 – Exemple de graphe d'une factorisation ILU(2) avec un ordering nested dissection d'une matrice représentant un cube 6x6x6. Nous avons appliqué l'opérateur F avec le paramètre 8.

temps de l'exécution de la tâche. Les résultats sans agrégation sont donc plus faibles avec une granularité grossière parce que le coût de l'ordonnanceur de tâches relativement au coût d'une tâche est moins important (table 3.9). Tout comme l'exemple précédent, l'opérateur F donne les meilleurs résultats. Le graphe n'est pas assez haut pour pouvoir obtenir de bons résultats avec les autres opérateurs.

| Types d'agrégations | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ø | D(16) | D(64) | D(256) | D(1500) | D(2048) | F(24) | F(36) | F(42) | F(64) | C |
| 125002 | 80849 | 77157 | 75604 | 75602 | 76250 | 73319 | 72616 | 72433 | **72251** | 123946 |

TABLE 3.9 – Résultats du simulateur d'exécution de tâches avec ILU(0) et une granularité grossière (8 variables primaires) sur 12 coeurs de calculs et un cube 100x100x100.

Nous allons maintenant augmenter le niveau de remplissage de l'algorithme ILU. Un remplissage de niveau 2 permet d'augmenter le nombre d'arêtes dans le graphe et de le rendre plus haut (Fig. 3.15). Le nombre de tâches par niveau dans le graphe n'est pas uniforme. Nous allons donc réessayer les différents opérateurs d'agrégation sur ce nouveau graphe. Nous réutilisons les paramètres du simulateur cité plus haut (0,6 pour les effets de cache et 2 pour l'ordonnancement). Les résultats de la table 3.10 nous montrent que l'opérateur F est toujours le plus efficace. Par contre, la différence avec l'opérateur D est réduite.

| Types d'agrégations | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ø | D(16) | D(64) | D(256) | D(1500) | D(2048) | F(24) | F(36) | F(42) | F(64) | C |
| 250008 | 85977 | 74584 | 69348 | 69398 | 69118 | 68156 | **67863** | 67977 | 68387 | 224098 |

TABLE 3.10 – Résultats du simulateur d'exécution de tâches avec ILU(2) et une granularité très fine (3 variables primaires) sur 12 coeurs de calculs et un cube 100x100x100.

Changeons une dernière fois les paramètres du simulateur pour augmenter le grain de calcul (0,8 pour les effets de cache et 0,5 pour le coût d'ordonnancement). Les résultats (Table. 3.11) montrent un très léger avantage pour l'opérateur F.

| Types d'agrégations | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| Ø | D(16) | D(64) | D(256) | D(1500) | D(2048) | F(24) | F(36) | F(42) | F(64) | C |
| 125004 | 82073 | 78370 | 76533 | 78233 | 78180 | 77884 | **75423** | 75452 | 75555 | 117447 |

TABLE 3.11 – Résultats du simulateur d'exécution de tâches avec ILU(2) et une granularité grossière (8 variables primaires) sur 12 coeurs de calculs.

Comme nous avons pu le voir, le choix des opérateurs dépend fortement de la structure du graphe de tâches. L'opérateur F sera à favoriser sur des graphes très larges et peu hauts. Au contraire, pour des graphes très hauts et peu larges il faudra privilégier l'opérateur C. L'opérateur D pourra être utilisé dans la plupart des cas même s'il ne fournit pas toujours les performances optimales.

## 3.5 Résultats sur des matrices de réservoir

Dans un premier temps (section 3.5.1), nous allons nous consacrer à l'amélioration de la parallélisation de la factorisation ILU(0). Puis dans la section 3.5.2, nous évaluerons les gains observés sur des factorisations de type ILU(1) et ILU(2).

### 3.5.1 Amélioration de la factorisation ILU(0) et de la résolution triangulaire

Dans le cas de la factorisation ILU, chaque tâche du graphe de tâches représente la factorisation d'une ligne de la matrice. De même, dans le cas de la résolution triangulaire, chaque tâche correspond à la résolution d'une ligne de la matrice. Cette granularité est trop fine, mais c'est voulu, elle représente la granularité que nous obtenons en décrivant naturellement l'algorithme de la factorisation. Nous avons choisi de simuler deux réservoirs, un cube généré de taille 80 cellules de côté et le réservoir SPE10 (cf. section 2.3.4). Dans le cas du cube, il est possible de choisir le nombre de variables primaires utilisées dans le calcul. Pour rappel, le nombre de variables primaires correspond au nombre de composants simulés. Chaque entrée non-nulle de la matrice correspond à l'interaction entre 2 cellules et sera composée d'une petite matrice dense $Npri * Npri$ avec $Npri$ le nombre de variables primaires. Nous avons choisi de simuler le cube avec 1 variable primaire, 3 variables primaires (modèle *black-oil*) et 8 variables primaires (modèle *compositionnel*). Pour le réservoir SPE10, c'est un cas *black-oil* à 3 variables primaires. Au final nous avons donc 4 matrices différentes. Nous ne faisons pas varier la taille du cube, car les résultats obtenus avec des cubes générés de tailles raisonnablement différentes sont équivalents. Le temps séquentiel permettant de calculer les différentes accélérations est le même pour une matrice donnée quelque soit la stratégie d'agrégation utilisée. Il s'agit du temps de traitement de chaque ligne de la matrice avec un parcours linéaire des indices, ce qui correspond au meilleur temps séquentiel que nous avons observé.

#### 3.5.1.1 Sans agrégation

Nous avons utilisé OpenMP pour tester la parallélisation à grain fin sans agrégation. OpenMP 3.0 n'ayant pas de gestion de dépendances entre les tâches, nous utiliserons son ordonnanceur de tâches auquel nous avons ajouté un système de décrémentation atomique de compteurs de dépendances pour chaque tâche. Comme le montrent les résultats de la figure 3.16, le nombre de variables primaires a une importance considérable sur les performances que nous obtenons. L'utilisation de 2 threads n'est viable que dans le cas où nous utilisons 8 variables primaires. Dans les autres cas, en utilisant de 2 à 4 threads, nous perdons du temps. Ici, le nombre de variables primaires va définir le nombre d'opérations faites par ligne de la matrice, donc plus ce nombre est grand, plus il y aura de travail à faire. Ces résultats confirment notre problème de granularité. Au final, avec l'utilisation des 12 coeurs de calcul de la machine, nous obtenons des accélérations plutôt décevantes, par exemple, pour les cas à 3 variables primaires, le code tourne environ 2,5 fois plus vite que la version séquentielle, mais utilise 12 fois plus de puissance de calcul.

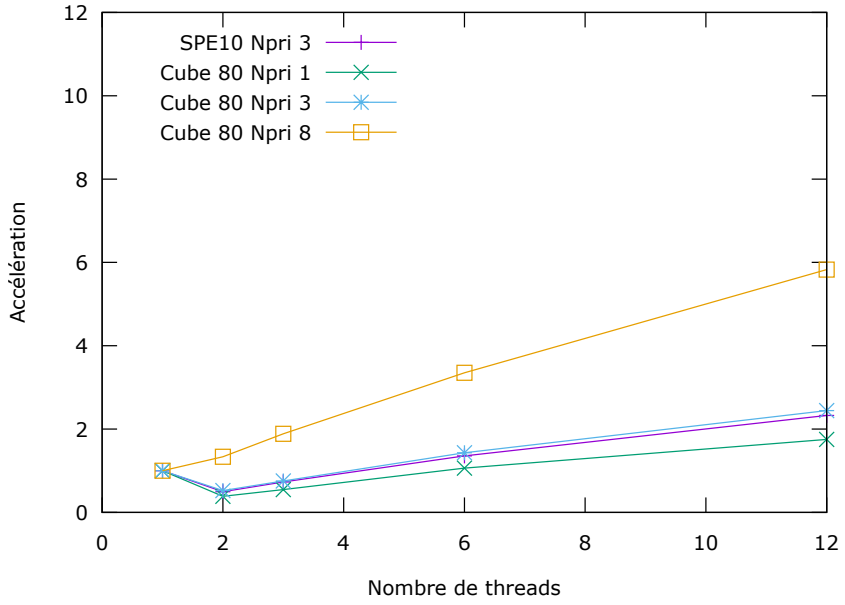



FIGURE 3.16 – Performance de la factorisation ILU(0) sur 12 coeurs sans utiliser Taggre.

#### 3.5.1.2 Avec l'opérateur F

Dans un premier temps, nous allons appliquer l'opérateur F avec le paramètre 36. Cet opérateur va donc limiter la largeur du graphe pour qu'il y ait au plus 36 tâches par hauteur du graphe. Nous aurons donc 3 fois plus de tâches que de coeurs de calcul, cela permet de diminuer fortement le nombre de tâches tout en gardant assez de parallélisme pour avoir un bon équilibrage de charge. Nous réduisons donc le parallélisme, mais nous réduisons aussi le nombre de tâches, on divise par 62 le nombre de tâches pour un cube de 80 de côté, nous passons de 512000 tâches à 8232 tâches. Pour le cas SPE10, le nombre de tâches passe de 1094421 à 12896 soit 84 fois moins de tâches. La figure 3.17 nous montre une amélioration du temps de factorisation quand le nombre de variables primaires est faible. Cela vient du fait que le surcoût d'ordonnancement de la tâche est du même ordre de grandeur que le temps de calcul de la tâche. Avec 3 variables primaires, la factorisation est 30 % plus rapide. Avec 1 variable primaire, le temps de factorisation est divisé par 2. Dans le cas où le nombre de variables primaires est élevé, il n'y a pas beaucoup d'amélioration (6 %). Ici, cet opérateur ne permet pas d'optimiser les accès mémoire caches, seul le surcoût d'ordonnancement est réduit.

#### 3.5.1.3 Avec l'opérateur D

Essayons maintenant l'opérateur D avec le paramètre 8. Cet opérateur va essayer de créer des groupes de 8 tâches assez proches dans le graphe. Nous allons donc diviser par 8 le nombre de tâches, passant ainsi de 512000 tâches à 64000 tâches. Ce qui peut paraître peu, mais cette valeur a été choisie empiriquement parmi un ensemble de tests avec différentes valeurs. Les autres valeurs donnent des résultats légèrement moins bons, par exemple le paramètre 4 offre une accélération de 3,35, tandis que le paramètre 12 offre 3,55 et finalement le paramètre 8 offre une accélération de 3,72.

Comparé à l'opérateur F, il y a très peu d'amélioration quand le nombre de variables primaires est faible (Fig. 3.18). Par contre, avec 8 variables primaires, on obtient un gain de performance qui est dû à une meilleure utilisation des caches, surtout du cache L2 avec une diminution de 10% des défauts de cache (Tab. 3.13) d'après les compteurs matériels du processeur. L'accès à ces compteurs est explicité dans la partie 4.1.1.

#### 3.5.1.4 Avec l'opérateur C

Les deux précédents opérateurs donnent déjà de meilleurs résultats que la version sans agrégation, mais nous pensons que le meilleur opérateur pour nos matrices est l'opérateur C. Il a l'avantage de réduire

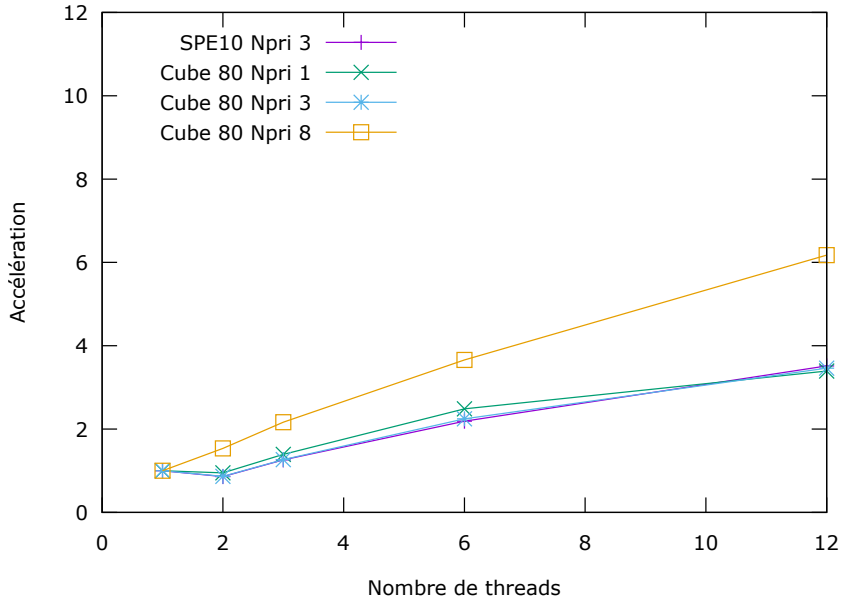



FIGURE 3.17 – Performance de la factorisation ILU(0) sur 12 coeurs avec Taggre F(36).

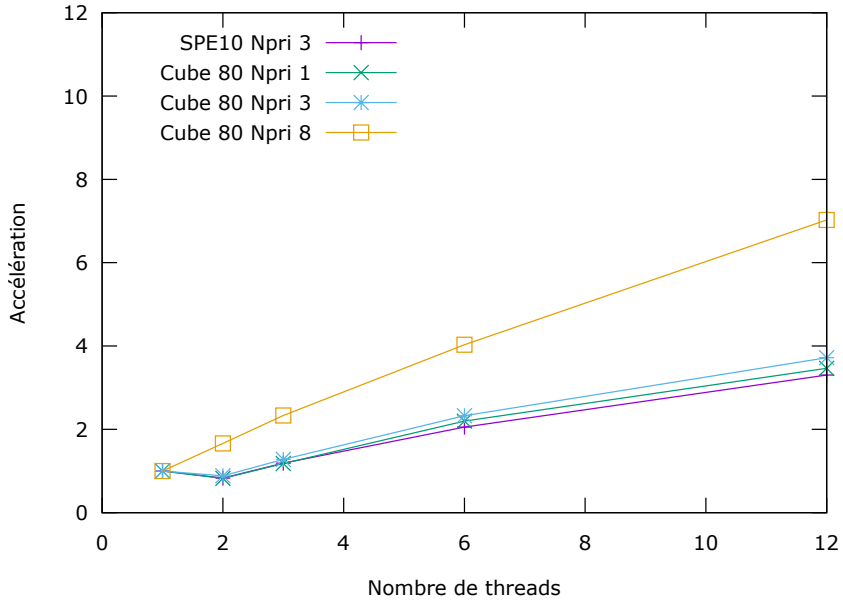



FIGURE 3.18 – Performance de la factorisation ILU(0) sur 12 coeurs avec Taggre D(8).

grandement le nombre de tâches comme l'opérateur F et il forme des groupes de tâches permettant une meilleure réutilisation des données en cache. Par contre, il se pourrait que le parallélisme en pâtisse.

Comme le montrent les résultats de la figure 3.19, tous les cas tests sont améliorés. Il y a bien deux grandes améliorations : la réduction du nombre de tâches et une meilleure réutilisation des caches. Comme pour tous les opérateurs, le nombre de tâches a été réduit, mais dans ce cas la réduction est bien plus importante. Par contre, l'amélioration des effets de cache est bien meilleure que dans les autres opérateurs, nous obtenons une diminution de 25% des défauts de cache des niveaux 1 et 2 par rapport à l'opérateur F. Cette amélioration est inexistante dans le cas de l'opérateur F, les tâches agrégées n'avaient pas une bonne réutilisabilité des données en cache. Au contraire, l'opérateur D ne réduisait que de très peu le nombre de tâches, mais améliorait la réutilisation des caches. L'opérateur C offre donc le meilleur des deux opérateurs, il produit peu de tâches et en plus ces tâches réutilisent correctement les données en cache.

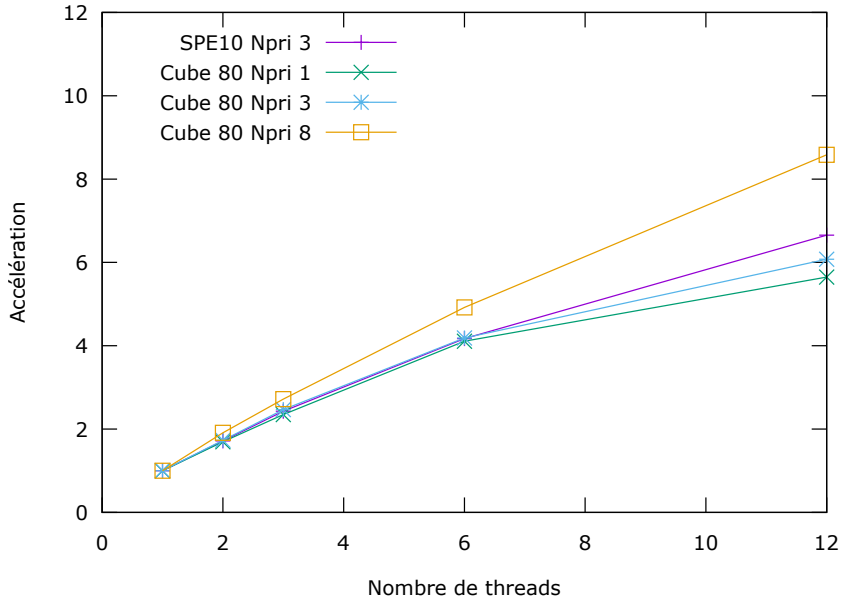



FIGURE 3.19 – Performance de la factorisation ILU(0) sur 12 coeurs avec Taggre C.

#### 3.5.1.5 Avec plusieurs opérateurs

Il est aussi possible de combiner plusieurs opérateurs, sur la figure 3.20 nous avons combiné l'opérateur C avec l'opérateur D(2). On observe une très légère perte de performance quand on a 8 variables primaires, mais dans les autres cas on observe le contraire. Le premier opérateur améliore déjà les effets de cache et augmente suffisamment la taille des tâches pour obtenir de bonnes performances, il n'y a donc presque plus d'améliorations possibles. Malgré ces optimisations, nous n'atteignons pas une accélération parfaite, avec au mieux une accélération de 8,7 pour 12 coeurs. Ce souci de performance est lié à l'architecture mémoire de la machine, nous donnerons plus d'explication dans le chapitre 4 de ce manuscrit.

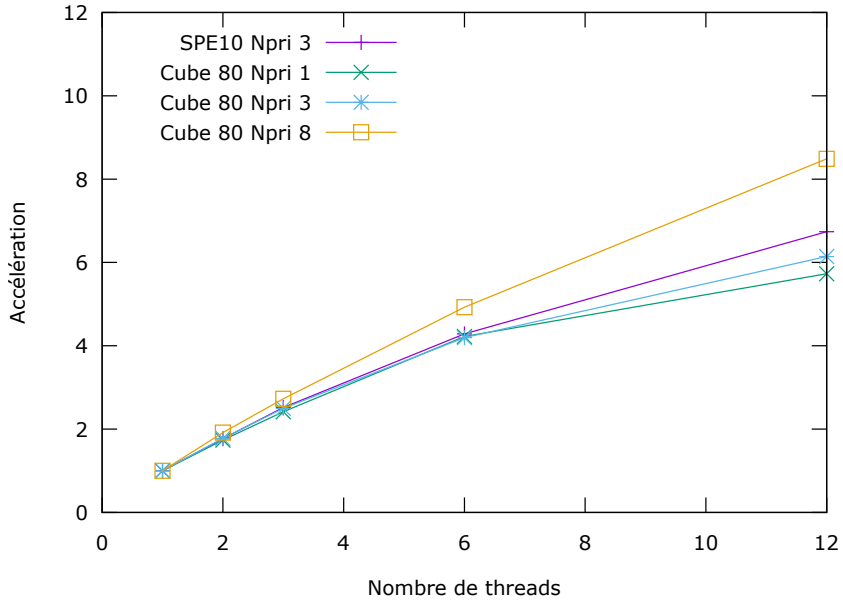



FIGURE 3.20 – Performance de la factorisation ILU(0) sur 12 coeurs avec Taggre CD(2).

#### 3.5.1.6 En résumé

| Type de matrice | Accélération | | | | | Nombre de tâches | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| | Ø | F(36) | D(8) | C | CD(2) | Ø | F(36) | D(8) | C | CD(2) |
| Cube 80 Npri 1 | 1,75 | 3,39 | 3.46 | 5.65 | 5.73 | 512000 | 8232 | 64000 | 6400 | 3200 |
| Cube 80 Npri 3 | 2,44 | 3,46 | 3.72 | 6.07 | 6.14 | 512000 | 8232 | 64000 | 6400 | 3200 |
| Cube 80 Npri 8 | 5,83 | 6,17 | 7.02 | 8.58 | 8.48 | 512000 | 8232 | 64000 | 6400 | 3200 |
| SPE10 Npri 3 | 2,32 | 3,52 | 3.30 | 6.65 | 6.73 | 1094421 | 12896 | 136887 | 36281 | 18181 |

TABLE 3.12 – Récapitulatif des résultats de la factorisation ILU(0) sur 12 coeurs de calcul suivant les opérateurs appliqués.

| | Cache L1 | Cache L2 | Cache L3 |
|---|---|---|---|
| Opérateur F | 0.009 | 1.102 | 0.096 |
| Opérateur D | 0.009 | 0.966 | 0.092 |
| Opérateur C | 0.007 | 0.873 | 0.091 |

TABLE 3.13 – Ratio du nombre de requêtes caches qui ont échouées par rapport au nombre de requêtes qui ont été faites. Résultats des compteurs matériels obtenus avec Likwid.

#### 3.5.1.7 Résultats de la résolution triangulaire

Essayons maintenant cette technique sur la partie résolution triangulaire du code. Les performances sans agrégation sont bien en dessous de la partie factorisation (Fig. 3.21). Même en utilisant 12 threads, seule la version avec 8 variables primaires donne de meilleurs résultats que la version séquentielle. Encore une fois, il s'agit d'un problème de granularité. Dans le cas de la résolution de $Ly = b$, aussi appelé descente (cf. section 2.3.1), le graphe à ordonnancer est le même que pour la factorisation, mais les tâches sont plus petites. Pour nos problèmes, la structure de la matrice étant symétrique, la remontée (résolution de $Ux = y$) correspond au même graphe avec les arêtes inversées.

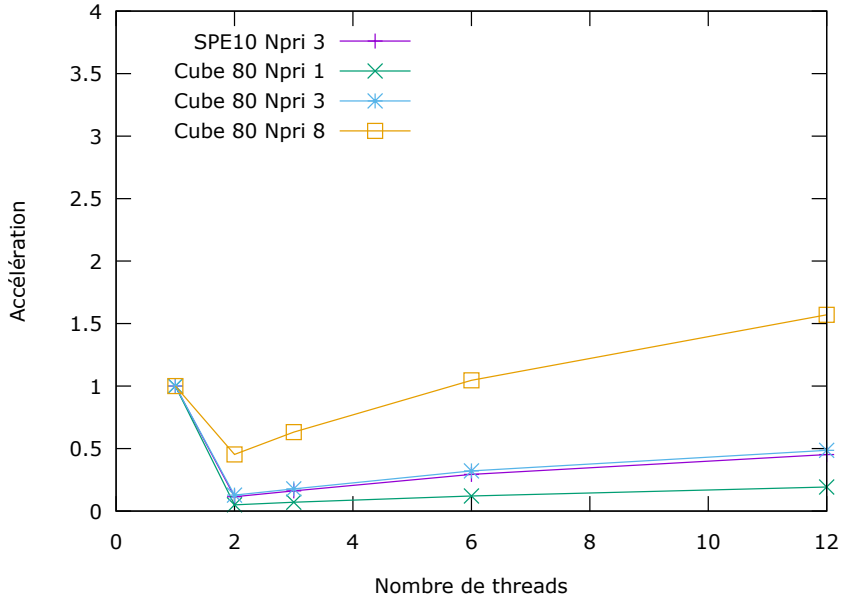



FIGURE 3.21 – Performance de la résolution triangulaire ($k = 0$) sur 12 coeurs sans utiliser Taggre.

Nous avons effectué une comparaison des différentes stratégies d'agrégation pour la résolution triangulaire, et les résultats étaient similaires à ceux de la factorisation ILU(0). Si nous regardons les résultats avec la meilleure agrégation que nous ayons, l'agrégation CD(2), on obtient des résultats légèrement meilleurs, mais nous sommes encore loin de l'accélération parfaite (Fig. 3.22). On arrive ici aux limites de notre méthode, la

granularité n'est pas encore suffisante pour permettre d'obtenir de très bonnes performances. Mais si nous augmentons encore la granularité, il n'y aura plus assez de parallélisme pour avoir un bon équilibrage de charge et le temps final sera supérieur.

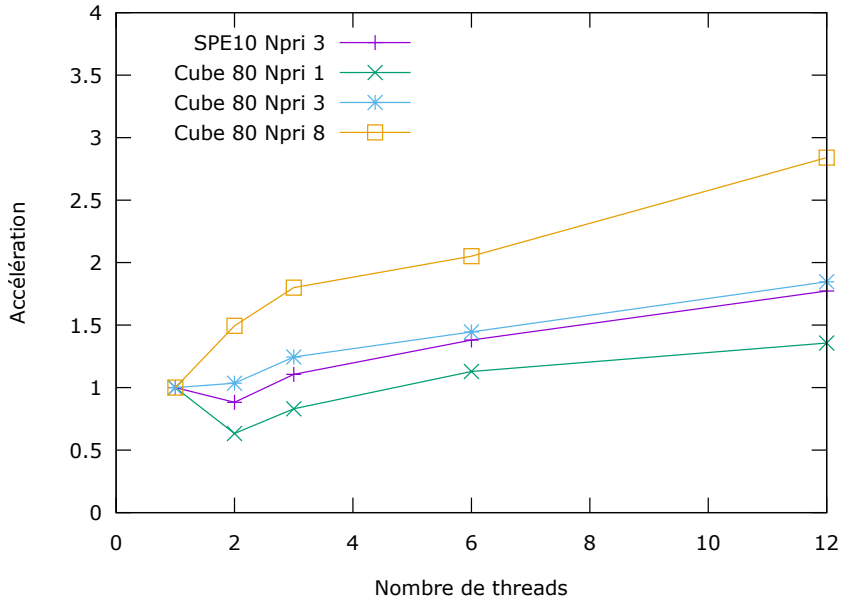



FIGURE 3.22 – Performance de la résolution triangulaire ($k = 0$) sur 12 coeurs avec Taggre CD(2).

### 3.5.2 Application à la factorisation ILU(k)

Nous souhaitons maintenant appliquer notre méthode d'agrégation à des factorisations ILU(k) avec $k$ supérieur à 0. Plus le paramètre $k$ est grand, plus il y aura de remplissage dans la matrice. Ce remplissage aura deux incidences majeures sur le graphe de tâches :

— des dépendances supplémentaires entre les tâches, donc moins de parallélisme à exploiter ;
— des éléments supplémentaires à traiter dans les tâches, donc une meilleure granularité.

Nous pouvons remarquer que sans utiliser d'agrégation, les factorisations ILU(1) et ILU(2) offrent de meilleures accélérations que la factorisation ILU(0) (Table. 3.14). En effet, les tâches étant plus grosses, le surcoût de l'ordonnanceur est moins important. En utilisant un opérateur C, nous obtenons quasiment toujours les meilleures performances. Seul le cas d'une factorisation ILU(2) avec 8 variables primaires donne une performance légèrement meilleure avec l'opérateur D. Dans ce cas, l'opérateur C n'a pas laissé suffisamment de parallélisme pour que l'ordonnanceur puisse faire efficacement son travail. L'opérateur F donne aussi de très bons résultats montrant que dans le cas des factorisations ILU(1) et ILU(2) les effets de cache entre les tâches sont moins importants que dans le cas d'une factorisation ILU(0)

| | | Types d'agrégations | | | |
|---|---|---|---|---|---|
| | | Ø | C | D(12) | F(36) |
| ILU(1) | Cube 80 Npri 1 | 2,36 | **5,95** | 3,98 | 3,64 |
| | Cube 80 Npri 3 | 3,75 | **7,50** | 5,24 | 4,36 |
| | Cube 80 Npri 8 | 7,60 | **9,63** | 8,85 | 7,51 |
| ILU(2) | Cube 80 Npri 1 | 3,73 | **7,11** | 5,91 | 5,23 |
| | Cube 80 Npri 3 | 5,32 | **8,49** | 7,33 | 5,84 |
| | Cube 80 Npri 8 | 8,26 | 9,59 | **9,72** | 8,28 |

TABLE 3.14 – Accélération de la factorisation ILU(k) sur 12 coeurs. Le temps de référence séquentiel est le même quelque soit la stratégie d'agrégation pour un cube et un niveau de k fixé.

Par contre, pour les résolutions triangulaires associées, augmenter la valeur de $k$ ne modifie pas que très peu les accélérations par rapport au meilleur temps séquentiel (Table. 3.15). Il y a plus d'éléments à traiter par ligne mais ces éléments ne sont que très peu réutilisés, il n'y a pas d'amélioration des effets cache. Nous continuons donc à avoir des problèmes de bande passante mémoire.

| | | Types d'agrégations | | | |
|---|---|---|---|---|---|
| | | Ø | C | D(12) | F(36) |
| k=1 | Cube 80 Npri 1 | 0,34 | **1,47** | 0,88 | 0,91 |
| | Cube 80 Npri 3 | 0,95 | **2,32** | 1,71 | 1,68 |
| | Cube 80 Npri 8 | 2,13 | **2,75** | 2,50 | 2,42 |
| k=2 | Cube 80 Npri 1 | 0,36 | **1,39** | 0,87 | 0,83 |
| | Cube 80 Npri 3 | 1,26 | **2,59** | 2,05 | 1,89 |
| | Cube 80 Npri 8 | 2,33 | **2,83** | 2,67 | 2,51 |

TABLE 3.15 – Accélération de la résolution triangulaire ($k > 0$) sur 12 coeurs. Le temps de référence séquentiel est le même quelque soit la stratégie d'agrégation pour un cube et un niveau de k fixé.

### 3.5.3 Surcoût d'agrégation

L'application des opérateurs d'agrégation a un surcoût. Il est nécessaire de savoir à partir de quel moment le surcoût d'agrégation devient plus petit que le temps gagné par l'agrégation. Dans notre cas l'agrégation est faite au début du programme et reste valide tant que la structure du réservoir ne change pas. Le temps passé à appliquer les opérateurs d'agrégation va dépendre du nombre de tâches, des opérateurs et de l'ordre des opérateurs. L'opérateur C mettra 1,5 $\mu s$ à traiter une tâche, l'opérateur F mettra 1,9 $\mu s$ et l'opérateur D mettra 2,5 $\mu s$. Par exemple pour la matrice SPE10 il faut 1,67 s pour appliquer l'opérateur C, 2,94 s pour appliquer l'opérateur D(8) et 2.19s pour appliquer CD(2). Toujours pour le cas SPE10, il devient rentable d'utiliser Taggre à partir d'environ 8 factorisations ou 10 résolutions triangulaires (Tab. 3.16). Les 10 résolutions triangulaires peuvent être faites dans une résolution du GMRES.

| Matrice | Agrégation | Temps agrégation (s) | Gain factorisation (s) | Gain résolution (s) |
|---|---|---|---|---|
| SPE10 | C | 1,67 | 0,188 | 0,171 |
| SPE10 | D(8) | 2,94 | 0,087 | 0,124 |
| SPE10 | CD(2) | 2,19 | 0,189 | 0.173 |
| Cube 100 | C | 1,49 | 0,167 | 0,163 |
| Cube 100 | D(8) | 2,52 | 0,095 | 0,132 |
| Cube 100 | CD(2) | 2,52 | 0,167 | 0,165 |

TABLE 3.16 – Temps d'application des opérateurs d'agrégation et les gains de temps obtenus sur les opérations d'algèbre linéaire.

L'évolution actuelle des processeurs conduit à avoir de plus en plus de coeurs de calcul. Ces coeurs supplémentaires pourraient faire diminuer le temps de calcul. Or, l'application des opérateurs d'agrégation étant séquentielle, la différence de temps entre l'agrégation et le calcul augmentera. Il faudra donc faire plus de calcul pour amortir le coût de l'agrégation. Pour pallier ce problème, la solution serait de paralléliser les algorithmes d'agrégation.

### 3.5.4 Méthode de Jacobi par blocs

Afin d'obtenir une évaluation de la borne maximale de l'accélération qu'il est possible d'obtenir sur une machine donnée, nous proposons de connaître l'accélération obtenue par une méthode de Jacobi par blocs et de la comparer à notre solution. Dans cette méthode, nous considérons le cas où chaque coeur effectue la factorisation ILU(0) d'un sous bloc diagonal de la matrice. Cette méthode de préconditionnement induit une factorisation et une résolution triangulaire totalement indépendante sur chaque coeur. Elle n'est évidemment

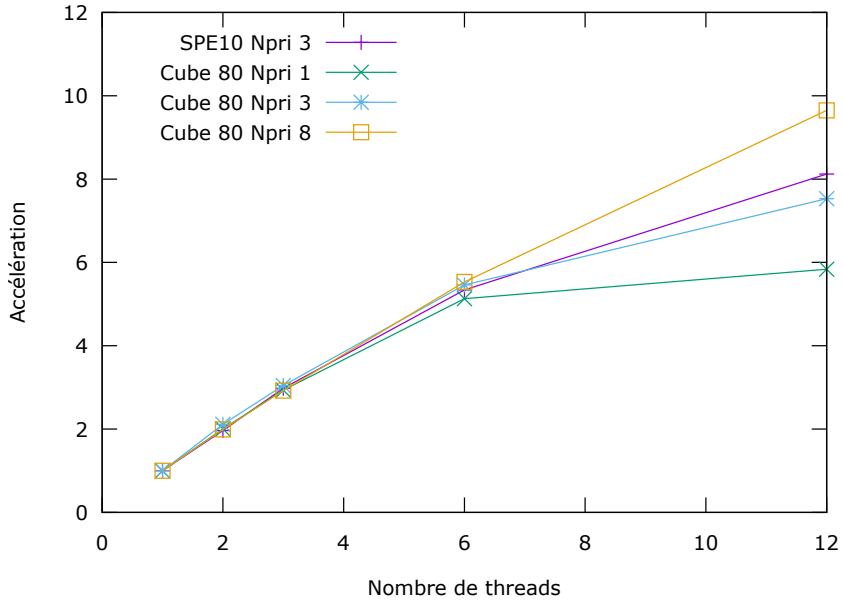



FIGURE 3.23 – Performance de la factorisation ILU(0) sur 12 coeurs en utilisant une méthode de Jacobi par blocs.

pas équivalente numériquement à effectuer une factorisation ILU de toute la matrice. Plus on utilise de coeurs, plus ce préconditionneur est inefficace numériquement[BG04]. Mais l'intérêt cette évaluation est que l'accélération obtenue donne une borne supérieure de l'accélération de la factorisation ILU sur la matrice complète. Cette borne existe puisqu'individuellement chaque coeur utilise une factorisation dont la bande passante mémoire est approximativement égale à celle de l'algorithme ILU globale.

Malgré un parallélisme idéal, nous n'obtenons pas une accélération parfaite pour la factorisation ILU(0) (Fig. 3.23). De plus, les performances de la résolution triangulaire sont basses malgré un parallélisme total (Fig. 3.24). C'est un problème que l'on rencontre souvent dans les codes d'algèbre linéaire creuse. Si l'on compare les accélérations obtenues avec celles obtenues avec l'agrégation de tâches, on peut voir des résultats légèrement meilleurs pour la méthode Jacobi par blocs. Notre solution d'agrégation n'est pas optimale, mais cette différence d'accélération n'est pas seulement due à la granularité du calcul.

Il est nécessaire de regarder du côté de la bande passante mémoire pour trouver des explications. Nous allons maintenant mesurer cette bande passante à l'aide des compteurs matériels. La bande passante utilisée lors de la factorisation parallélisée par une méthode de Jacobi par blocs est au mieux de 16 Go/s, avec les threads nous utilisons 12 Go/s de bande passante locale dont 5 Go/s de bande passante distante. Cette bande passante distante est le résultat des effets NUMA. Dans le cas des processus MPI utilisés par la méthode Jacobi par blocs, chaque processus a son propre espace mémoire et la localité des données est assurée sous réserve d'un placement statique et optimal des processus. Mais dans le cas des threads, l'espace mémoire est partagé entre deux bancs mémoires et si les accès mémoire ne sont pas optimisés, il y aura des transferts entre les bancs NUMA. Ces transferts auront pour effet d'augmenter la latence des accès mémoire et de réduire le nombre de cycles d'horloge par instruction (CPI [1]), ici on a un CPI de 0,75 pour la version MPI contre 1,08 pour la version threadée (résultats obtenus avec les compteurs matériels). L'explication de la différence de performance entre ces deux versions se situe au niveau de la mémoire, on est limité par la bande passante.

### 3.5.5 Discussion

Même si un algorithme peut exposer du parallélisme naturellement, il n'est pas toujours possible de l'exploiter correctement. Soit parce qu'il n'y a pas assez de parallélisme pour obtenir un bon équilibrage de charge, soit parce que la granularité du problème n'est pas suffisamment grosse pour négliger les surcoûts liés aux runtimes. Notre proposition permet de manière presque transparente d'apporter une solution au problème

1. Clock Per Instruction

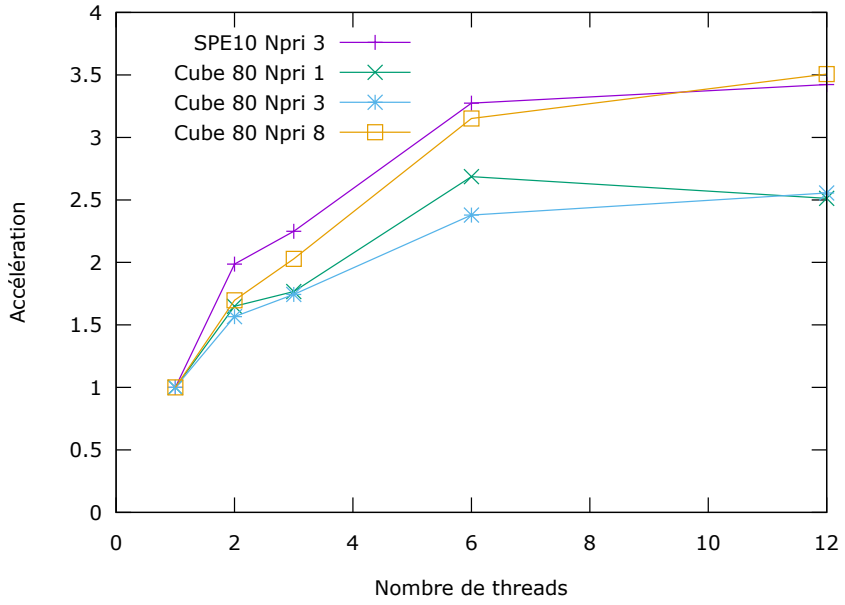



FIGURE 3.24 – Performance de la résolution triangulaire sur 12 coeurs en utilisant une méthode de Jacobi par blocs.

de la granularité. Mais cette solution n'est pas parfaite, le choix des opérateurs d'agrégation est laissé au programmeur. Ce choix se fait suivant trois différents critères :

— la forme du graphe de tâches ;
— le rapport entre surcoût d'ordonnancement et coût de la tâche ;
— l'amélioration des effets de cache suivant l'ordre d'ordonnancement des tâches.

Un graphe plus large que haut permettra de sacrifier un peu de parallélisme pour grossir le grain de calcul. Au contraire, un graphe pas assez large n'offrira pas la possibilité d'augmenter le grain de calcul. Si le surcoût d'ordonnancement est supérieur au coût d'exécution d'une tâche alors réduire le nombre de tâches augmentera presque systématiquement les performances du programme. Si l'ordre d'ordonnancement des tâches n'a que très peu d'effet, l'opérateur F reste le meilleur candidat. Dans le cas contraire, les opérateurs C et D sont à privilégier.

L'utilisation du simulateur de tâches pourrait être étendue pour permettre de choisir automatiquement les bons opérateurs avec les bons paramètres. Mais l'espace des possibilités à tester est considérable, le temps nécessaire à tester toutes les possibilités serait trop grand. Par contre, nous pouvons imaginer une heuristique qui en fonction des propriétés du graphe, de la granularité du calcul et des effets de cache liés à l'ordre d'ordonnancement des tâches propose un opérateur à appliquer. Le simulateur pourra ensuite tester cette possibilité et donner une estimation du gain théorique.

Le temps d'application des opérateurs sur le graphe de tâches est aussi un problème, il faut réutiliser le nouveau graphe grossier un certain nombre de fois avant de pouvoir considérer cette étape comme négligeable. Dans le cas où le graphe changerait souvent au cours de l'exécution, cette approche ne fonctionne plus, il faut changer d'algorithme. Une fois les bons opérateurs trouvés, on peut commencer à avoir des résultats proches de l'optimal. C'est le cas pour notre problème, mais il reste encore une différence de performance à rattraper. Cette différence est due aux effets NUMA, certains accès mémoire auront une latence plus grande que les autres. Il faut donc optimiser le placement des données, le chapitre suivant se concentrera sur l'optimisation des placements mémoires sur machine NUMA.

# CHAPITRE 4

# LIMITATION DE BANDE PASSANTE MÉMOIRE

**Sommaire**



Les algorithmes que nous essayons de paralléliser ont des problèmes de limitations par la bande passant mémoire. C'est pourquoi ce chapitre est consacré à l'atténuation de ces limitations. Nous passerons en revue les performances de nos algorithmes sur la machine Rostand à l'aide des compteurs matériels. Puis nous verrons les différentes solutions proposées par le système d'exploitation pour améliorer la localité mémoire sur des machines NUMA. Nous proposons ensuite une solution permettant d'obtenir une gestion des affinités mémoires au niveau du graphe de tâches. Nous testons cette solution sur un noeud de la machine Rostand ainsi que sur la machine Manumanu. Et ce chapitre se clôturera sur un retour d'expérience de l'utilisation du Xeon Phi d'Intel.

## 4.1 Étude du produit matrice vecteur creux

### 4.1.1 Passage à l'échelle

Comme dit dans le chapitre précédent, l'algorithme du GMRES non préconditionné se parallélise bien parce qu'il est essentiellement composé d'opérations sur des vecteurs. L'opération la plus coûteuse est le produit d'une matrice par un vecteur (SpMV). Notre implémentation du SpMV est optimisée pour prendre en compte la structure bloc des entrées de la matrice quand le nombre de variables primaires est supérieur à 1. Nos matrices sont stockées au format BCSR, les éléments contenus dans les petits blocs denses sont donc contigus en mémoire. Cela nous permet d'optimiser les effets de cache et nous pouvons aussi utiliser le jeu d'instructions vectorielles de nos processeurs.

Malgré les optimisations des effets de cache, le SpMV est toujours limité par la bande passante mémoire. Les courbes d'accélération du SpMV nous montrent que le gain de performance n'est pas linéaire avec le nombre de coeurs (Fig. 4.1). Le constat est même pire que ça, on arrive difficilement à une accélération de 2 sur 12 coeurs lorsqu'on utilise une seule variable primaire.

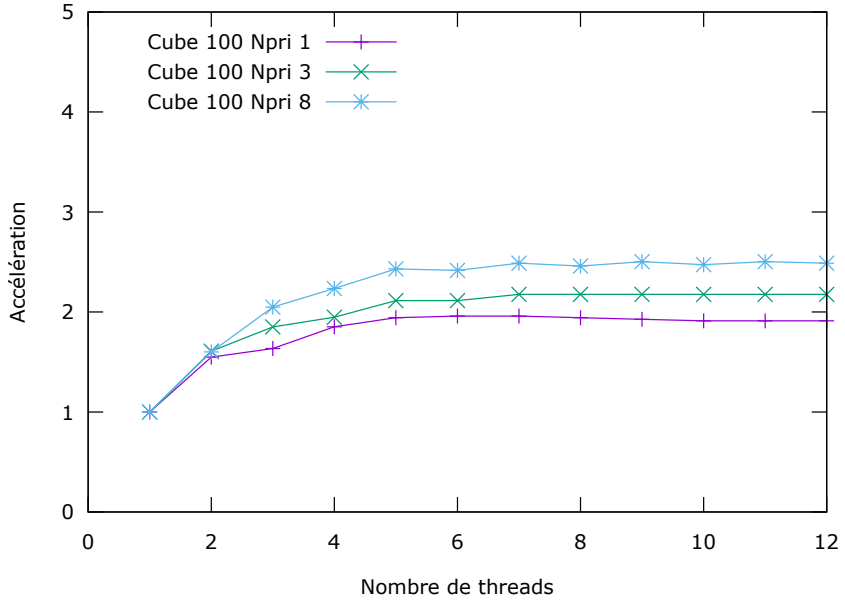



FIGURE 4.1 – Accélération du produit matrice vecteur creux sur Rostand en mémoire partagée.

Pour comprendre ce qui se passe, nous allons utiliser les compteurs matériels de la machine. Ces compteurs sont intégrés au processeur et on y accède via des registres spéciaux seulement disponibles en espace noyau. De nombreux logiciels proposent une interface d'accès à ces compteurs, mais certains nécessitent le chargement d'un module noyau.

Linux propose un outil d'analyse de performance natif nommé *perf*. Cet outil se présente sous la forme d'un exécutable qui se lance juste avant notre programme. Nous devons choisir nous même les compteurs qui nous intéressent. Malheureusement, la version du noyau Linux installé sur nos machines ne nous permet pas d'accéder à tous les compteurs. Il manque essentiellement les compteurs de type *uncore* qui permettent de mesurer le nombre de requêtes mémoire faites à un banc NUMA distant.

PAPI [1] fournit une interface de programmation qui fournit une abstraction pour l'accès aux compteurs matériels. Cette bibliothèque utilise l'interface perfmon sous Linux, elle utilise donc le même module noyau que perf. En plus de faciliter l'accès aux compteurs, elle propose comme fonctionnalité de simuler des accumulateurs de compteurs. Mais son mode de fonctionnement dans un environnement multi-threadé n'est pas suffisamment documenté, nous avons donc cherché un autre outil.

Intel propose aussi un outil d'analyse de performances, il s'agit de Intel VTune. Cet outil est hautement configurable avec des profils prédéfinis. L'exécution du code échantillonnée avec les valeurs des compteurs matériels associés à chaque échantillon. Cet outil est très complet, mais requiert l'achat d'une licence d'utilisation ainsi que l'ajout d'un module noyau. Nous n'avons donc pas pu utiliser cet outil car les administrateurs de notre grappe de serveur ne souhaitent pas installer le module noyau.

Nous avons préféré utiliser Likwid, un outil d'analyse de performance léger qui simplifie l'accès aux compteurs matériels. Le principal avantage de Likwid est qu'il contient des profils déjà configurés calculant des métriques utiles. Par exemple, le profil *MEM* nous donnera directement les mesures de bande passante mémoire locale et distante en divisant le nombre d'accès mémoire par le temps de calcul. Le nombre d'accès mémoire provient des compteurs *UNC_QMC_NORMAL_READS_ANY*, *UNC_QMC_WRITES_FULL_ANY*, *UNC_QHL_REQUESTS_REMOTE_READS* et *UNC_QHL_REQUESTS_REMOTE_WRITES*. Il y a aussi la possibilité d'interagir avec Likwid depuis le code d'une application pour ne mesurer qu'une portion du code. Nous utiliserons cette fonctionnalité pour ne mesurer que les routines qui nous intéressent.

1. Performance Application Programming Interface

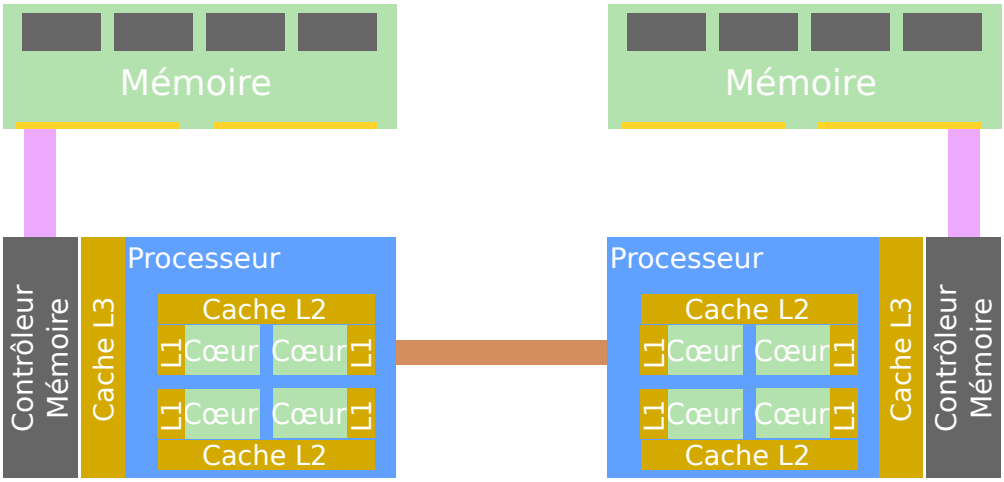



FIGURE 4.2 – Exemple d'une architecture NUMA à deux processeurs.

Pour mesurer les effets NUMA sur le SpMV, nous effectuons 100 SpMV et mesurerons le nombre d'accès mémoire. Si l'on regarde la valeur des compteurs matériels dans la table 4.1, on s'aperçoit que la moitié de la bande passante en lecture de chaque banc NUMA est utilisée par des accès distants. Donc environ la moitié des lectures mémoires sont faites avec une plus grande latence. Nous pouvons aussi remarquer un déséquilibre de charge. Le thread responsable de l'initialisation du code s'est exécuté sur le banc NUMA 1. Les pages mémoires servant à stocker la matrice étaient donc quasiment toutes sur ce banc NUMA.

| | Lecture locale | Lecture distante | Écriture locale | Écriture distante |
|---|---|---|---|---|
| Banc NUMA 0 | 8,68E+07 | 9,18E+07 | 6,24E+07 | 2,29E+07 |
| Banc NUMA 1 | 4,08E+09 | 2,60E+09 | 2,68E+08 | 5,64E+07 |

TABLE 4.1 – Nombre d'accès mémoire effectués lors de 100 produits matrice vecteur creux.

### 4.1.2 Limitations mémoire

Pour comprendre les résultats précédents, nous devons étudier brièvement l'architecture des ordinateurs. Les performances d'un ordinateur dépendent essentiellement de deux choses : le processeur et la mémoire. Quand un programme souhaite lire une donnée en mémoire, il subit une pénalité mémoire. Cette pénalité est la conséquence de deux contraintes :

— la latence : la différence de temps entre la demande d'accès à la mémoire et la réception du premier octet ;
— la bande passante : le nombre maximum d'octets par seconde que le bus peut envoyer/recevoir

De nos jours, les ordinateurs ont différents types de mémoire. Il y a la mémoire cache, très proche des unités de calcul, elle a une faible latence (quelques cycles), mais sa taille est limitée à quelques Mo. Ensuite, il y a la mémoire RAM avec une latence de quelques centaines de cycles, mais avec une taille de plusieurs Go. Si nous regardons la structure mémoire de deux bancs NUMA de la figure 4.2, on s'aperçoit que la distance physique des coeurs de calcul et de la mémoire est liée à la latence des accès mémoire. Pour limiter le plus possible les pénalités mémoire, il est nécessaire que tous les coeurs de calcul utilisent la mémoire qui est la plus proche. Par exemple, sur une machine à deux bancs NUMA, les accès à la mémoire du banc NUMA 1 depuis les coeurs du banc NUMA 0 coûteront plus cher que les accès à la mémoire du banc NUMA 0. Un autre problème se situe au niveau de la bande passante. La bande passante totale de la machine est distribuée entre chaque banc NUMA. Pour exploiter efficacement la totalité de la bande passante, il faut répartir les données que l'on va utiliser sur tous les bancs mémoires. Des interfaces de programmation existent pour connaître et améliorer la localité des données. Il ne suffit pas de faire une répartition équitable, il faut aussi au moment de l'accès aux données que tous les coeurs de calcul accèdent à des données différentes qui doivent se situer dans la mémoire qui leur est la plus proche.

## 4.2 Gestion actuelle des machines NUMA

De manière générale, quand un programme alloue de la mémoire, il reçoit un pointeur sur de la mémoire virtuelle. Cette mémoire virtuelle est unique à chaque processus et est partagée entre les threads d'un même processus. La relation entre la mémoire virtuelle et la mémoire physique est faite par le système d'exploitation. Il est responsable de la bonne gestion de la mémoire physique. Lorsqu'un processus accède à de la mémoire virtuelle qui n'a pas encore été mappée à de la mémoire physique, une faute de page est générée, on appelle ça toucher une page. Cette faute est traitée par le système d'exploitation, il s'occupera de trouver une page mémoire libre dans la mémoire physique et il la fera correspondre à une adresse virtuelle. Lors des accès mémoire du processus, la traduction de l'adresse virtuelle vers l'adresse physique est faite par un des composants du processeur, l'unité de gestion mémoire ou MMU [1]. De cette façon, le processus ne voit que l'espace d'adressage virtuel et ne connaît rien de l'espace d'adressage physique. Le système d'exploitation peut donc changer l'emplacement physique d'une page sans affecter le fonctionnement du processus de manière totalement transparente. Toutefois, l'emplacement physique d'une page virtuelle peut impacter les performances du processus sur les architectures NUMA. Cet impact dépendra des connexions entre le banc mémoire physique où se situe la page et le coeur de calcul qui fait tourner le processus.

Sur une machine NUMA, lorsqu'une faute de page arrive, le système d'exploitation doit choisir sur quel banc NUMA il placera la page. Avec Linux, il y a au moins trois politiques d'allocations de disponibles :

— *First Touch* : La mémoire est allouée sur le banc NUMA du coeur de calcul qui y accède en premier. Il s'agit de la politique par défaut.
— *Bind* : La mémoire est allouée sur banc NUMA spécifié en paramètre.
— *Interleaved* : Les allocations mémoires sont entrelacées parmi tous les bancs NUMA disponibles.

Sur Linux, ces politiques peuvent être choisies avec l'appel système *mbind*, ou avec l'outil en ligne de commande *numactl*. La version 3.13 de Linux apporte un nouveau mécanisme de gestion de la mémoire sur les machines NUMA, il s'agit d'AutoNUMA. Ce mécanisme a pour but d'optimiser le placement des pages NUMA tout au long de l'exécution d'un processus (voir section 4.2.4).

D'autres systèmes d'exploitation peuvent avoir leurs propres ensembles de politiques d'allocations NUMA. Solaris, par exemple, fournit aussi la politique *next-touch* [LH05]. Avec cette politique, les pages mémoires physiques seront déplacées près du prochain coeur de calcul qui y accédera (voir section 4.2.3). De nombreuses bibliothèques proposent des interfaces de programmation permettant d'abstraire le placement des pages lors d'une allocation mémoire. Par exemple, la libNUMA[Kle05] va abstraire les appels systèmes Linux. Il existe aussi des bibliothèques qui ajouteront de nouvelles politiques d'allocations ainsi que des allocations 2D optimisées[PRCMC11].

### 4.2.1 First touch

La politique d'allocation mémoire par défaut sous Linux s'appelle *First Touch*. La traduction littérale serait le *premier toucher*, ce nom se réfère au fait qu'avec cette allocation le noyau associe une nouvelle page physique à une page virtuelle qu'à partir de la première utilisation de cette page. La page physique sera choisie avec comme priorité de prendre une page dans la mémoire la plus proche du thread qui souhaite utiliser cette page. L'idée derrière cette allocation n'est pas mauvaise, dans le cas d'un processus mono-thread dont l'affinité processeur est fixée à un banc mémoire, la localité mémoire sera toujours optimale. Dans le cas d'un processus multi-threads dont les threads ont une affinité fixe, s'ils allouent eux-mêmes leurs mémoires et ne font presque aucun partage entre eux, cette allocation fonctionne toujours. Mais tous les programmes ne sont pas écrits pour fonctionner de cette façon. Imaginons un programme qui soit écrit pour avoir une phase d'initialisation séquentielle, avec toutes les allocations dont il aura besoin dans cette phase, alors toute la mémoire physique sera allouée sur un seul banc NUMA. Comme toutes les données se retrouvent exclusivement sur un seul banc NUMA, tous les threads devront se partager la bande passante mémoire de ce banc alors que les bandes passantes des autres bancs NUMA seront utilisées.

Il existe d'autres cas où ce type d'allocation ne permet pas d'obtenir le maximum de bande passante mémoire de la machine. Par exemple dans le cas d'une application multi-processus dont tous les processus ont une affinité processeur identique et fixée à un seul banc NUMA. Seulement une partie de la bande passante

1. Memory Management Unit

mémoire sera utilisée. Il y a aussi le cas où les processus ont une affinité processeur leurs permettant d'utiliser n'importe quel coeur de la machine. L'allocation des pages mémoires peut se faire sur un banc NUMA, puis le noyau décide de changer le processus de banc NUMA, et tous les calculs sont faits avec une mauvaise localité mémoire. De plus, si le thread de calcul change souvent de coeur de calcul, l'utilisation des caches de faibles niveaux (L1 et L2) ne sera pas optimale. Il est donc important de toujours fixer l'affinité processeur d'un thread à un coeur de calcul. Dans notre programme, l'initialisation des données est séquentielle. Donc avec une politique d'allocation first touch, toutes les données se retrouvent sur un seul banc NUMA. Nous avons donc plusieurs choix pour distribuer les données :

— soit nous réécrivons la partie initialisation pour qu'elle soit faite en parallèle ;
— soit nous essayons une autre politique d'allocation qui correspond mieux à notre problème.

La première solution est compliquée à mettre en oeuvre et pourrait introduire de nouveaux bogues dans le code. La deuxième solution nécessite moins de changement, nous avons donc essayé cette solution.

### 4.2.2 Interleaved memory

First touch n'étant pas parfait, il est nécessaire d'avoir d'autres politiques d'allocation. La politique *Interleaved memory* distribue uniformément les pages mémoire sur tous les bancs mémoire en mode tourniquet. Cette distribution est faite par le noyau du système d'exploitation au moment où une page mémoire est utilisée pour la première fois par le programme. Sur un système d'exploitation utilisant Linux comme noyau, il suffit d'utiliser la commande *numactl –interleave=all ./programme* pour utiliser cette politique d'allocation dans tout le programme. En plus d'avoir très peu d'impact sur le code source d'une application, la politique d'entrelacement mémoire montre des atténuations des effets NUMA dans le cas général. En moyenne, il n'y a pas d'amélioration de la latence, mais la bande passante est améliorée grâce à l'utilisation simultanée de tous les liens mémoire par rapport à une initialisation séquentielle avec une politique first touch. Ainsi, il est généralement intéressant d'expérimenter cette politique, avant d'étudier la question des optimisations NUMA. Dans notre cas, cette politique nous donnait de meilleures performances que la politique first touch, mais les résultats n'étaient pas suffisants.

### 4.2.3 Next touch

L'idée du First touch n'est pas mauvaise, mais elle impose une phase d'initialisation parallèle. Au lieu de récrire toute l'initialisation d'un programme, il pourrait être intéressant d'utiliser les phases de calculs pour distribuer la mémoire sur tous les bancs NUMA. C'est pour cela que la politique d'allocation *next touch* a été créée. Le programmeur choisit un ensemble de pages mémoires qu'il pense mal placées et définit une politique d'allocation next touch sur ces pages. Lors du prochain accès mémoire à l'une de ces pages, le noyau s'occupera, si besoin, de déplacer la page mémoire vers le banc NUMA le plus proche du processeur faisant cet accès. Ainsi nous pouvons obtenir une amélioration de la localité mémoire sans avoir à récrire certaines parties du code. Cette politique aurait pu apporter des performances supplémentaires à notre code, mais n'étant pas disponible dans le noyau Linux, malgré des propositions d'extensions [LBB10, GF09], nous n'avons pas pu l'utiliser telle quelle. À la place, nous avons implémenté une solution similaire qui consiste à choisir manuellement l'emplacement des pages mémoires.

### 4.2.4 AutoNUMA

Linux n'a pas adopté la politique d'allocation next touch. À la place, les développeurs de Linux ont choisi d'implémenter un autre mécanisme pour améliorer la localité NUMA : *AutoNUMA*. Ce mécanisme va analyser périodiquement une portion de la mémoire d'un processus, et de la même façon que la politique next touch, le prochain accès à une page de cette portion entrainera un déplacement de la page. Pour détecter l'accès à une page, le noyau utilise la MMU. Il supprime la relation adresse virtuelle vers adresse physique de la MMU et peut ainsi recevoir un signal lors du prochain accès. Ce mécanisme à un surcoût, c'est pourquoi il n'est pas appliqué sur toute la mémoire d'un coup. Le noyau donne la possibilité de modifier plusieurs paramètres de ce mécanisme, mais ces paramètres sont globaux. Parmi ces paramètres, il y a *scan_delay* et *scan_size*. Tous les "scan_delay", les "scan_size" pages suivantes sont traitées. Une fois arrivé au bout de l'espace d'adressage, le scanner recommence au début de l'espace d'adressage. La variable scan_delay

change de valeur en fonction du nombre de pages déplacées. Elle diminue quand il y a beaucoup de fautes NUMA et augmente quand les pages sont bien placées, c'est pourquoi le surcoût de cette méthode est difficilement calculable. Ainsi, une application dont les threads accèdent toujours à la même mémoire aura automatiquement un bon placement mémoire.

Ce mécanisme comporte plusieurs défauts. Sa configuration est globale au système, on ne peut pas l'activer seulement pour un processus particulier. Les paramètres ne peuvent être définis que par l'administrateur de la machine. Le mécanisme s'applique sur toute la mémoire, même les zones mémoire peu utilisées. Dans le cas de notre application, cette politique permet d'améliorer la localité mémoire sans changer une ligne de code. Mais le surcoût lié à l'analyse du placement des pages mémoires peut devenir problématique.

### 4.2.5 Un processus MPI par banc NUMA

Les problèmes rencontrés sur les machines NUMA proviennent essentiellement de la mémoire partagée. Lorsque deux threads partagent un espace mémoire et que ces deux threads s'exécutent sur des bancs NUMA différents, il y aura potentiellement des accès mémoire non performants. Nous pouvons donc résoudre le problème des effets NUMA en utilisant plusieurs processus. En utilisant un processus MPI par banc NUMA et une politique d'allocation first touch ou bind, on se retrouve toujours avec des accès mémoire sur le banc mémoire le plus proche. Mais la problématique reste similaire à la problématique du choix entre la parallélisation en mémoire distribuée et la parallélisation en mémoire partagée, ce n'est pas toujours possible ou performant. Dans le cas où les algorithmes en mémoire distribuée ne passeraient pas à l'échelle, il est nécessaire de noter que l'utilisation de cette solution multipliera le nombre de processus MPI par le nombre de bancs NUMA. Notre application utilise déjà du parallélisme en mémoire distribuée, il n'y aura donc pas de modification à effectuer pour cette solution. Par contre, les algorithmes que nous utilisons donnent de meilleurs résultats avec peu de processus MPI, nous cherchons donc à limiter le nombre de processus MPI.

## 4.3 Gérer le NUMA directement dans l'ordonnanceur

### 4.3.1 Statuts des ordonnanceurs actuels

La gestion du placement des pages mémoires n'est pas utile si le code qui utilisera ces données ne s'exécute pas sur un coeur associé au bon banc NUMA. Dans le cas de la programmation par tâche, chaque tâche doit connaître le banc NUMA qui lui est le plus favorable. Cette information pourra être ensuite donnée à l'ordonnanceur de tâches qui s'occupera de placer correctement la tâche.

ForestGOMP[Bro10] est le résultat d'un travail de recherche autour du support des machines à mémoires hiérarchiques dans OpenMP. Il utilise le parallélisme de boucle imbriqué pour adresser les différents niveaux hiérarchiques de la machine. La bibliothèque de threads Marcel[Thi05] est utilisée pour la création de thread en espace utilisateur. Dans une première boucle for sur le nombre de noeuds NUMA, le programmeur déclare les données auxquelles il va accéder. Dans une deuxième boucle for imbriquée, le programmeur effectue les calculs. L'utilisation de threads en espace utilisateur permet de créer un grand nombre de threads sans trop impacter la performance. En créant un nombre conséquent de threads, nous pouvons obtenir un bon équilibrage de charge. Cet équilibrage de charge sera possible grâce à l'utilisation de BubbleSched[TNW07] qui permet de créer des bulles de threads pour pouvoir déplacer un ensemble de threads sur un nouveau coeur de calcul d'un coup. Lorsque le travail vient à manquer, c'est-à-dire qu'il n'y a plus assez de bulles pour occuper tous les coeurs de calcul, on peut exploser une bulle qui se transformera en plusieurs bulles jusqu'à atteindre la granularité d'un thread. Les informations mémoires étant associées aux bulles, il est possible de choisir la meilleure bulle à éclater, celle qui nous fournira les meilleurs accès mémoire. La gestion des allocations mémoires de ForestGOMP repose sur MaMI [1], une bibliothèque écrite spécialement pour exploiter les machines NUMA dans Marcel. MaMI implémente la politique d'allocation next touch et permet aussi la migration explicite des données. ForestGOMP se base donc sur le principe que le programmeur est le mieux placé pour connaître l'utilisation mémoire de son programme. Ces travaux montrent qu'une meilleure gestion des allocations mémoires, même manuelle, permet de gagner en performance.

1. Marcel Memory Allocator

PaRSEC est un cadriciel de parallélisation par tâche qui fonctionne aussi en mémoire distribuée. Il est l'un des seuls ordonnanceurs de tâches à offrir un réel support des architectures NUMA. Par contre son support est une analogie avec la programmation en mémoire distribuée. En effet, le support du NUMA est fait avec les structures Virtual Process (VP) de PaRSEC, ce qui peut correspondre à avoir un processus MPI par banc NUMA. Mais ce n'est pas si grave, le vol de tâche entre VP existe. Il conserve donc l'aspect équilibrage de charge des solutions multithreadées. Par contre, cette solution ne convient toujours pas à résoudre notre problème, nous essayons d'avoir le moins possible de parallélisme en mémoire distribuée.

MAi [1][RMC+09] fournit une interface de placement des données. Par rapport à MaMI, MAi implémente des stratégies génériques d'allocations des pages mémoires. Mais MAi ne fournit pas d'ordonnanceur de tâches, nous ne pourrons donc pas exécuter les tâches sur les noeuds de calcul où la mémoire est allouée.

D'autres tentatives d'extension de la spécification OpenMP existent. L'article [BCC+00] présente l'ajout de trois nouvelles directives à OpenMP. La première directive se concentre sur la migration des données lors du prochain accès à ces données, il s'agît d'une implémentation de la politique d'allocation next touch. La deuxième directive concerne la distribution des pages d'une zone mémoire. Cette distribution peut être faite par bloc, entrelacé sur différents bancs NUMA dans plusieurs dimensions. La troisième directive permet de prévenir l'ordonnanceur que la distribution des itérations d'une boucle doit tenir compte des allocations NUMA. Les performances obtenues sur une factorisation LU dense sont encourageantes, une accélération quasi parfaite jusqu'à 16 processeurs là où une version OpenMP classique est deux fois moins performante. Ces travaux nous prouvent qu'une gestion fine des allocations NUMA couplée à un ordonnanceur prenant en compte les affinités mémoires permet d'exploiter efficacement une machine NUMA. Malheureusement, seul le parallélisme de boucle est traité, il n'y a pas gestion du parallélisme à base de graphe de tâches.

### 4.3.2 NATaS : ordonnancer des tâches sur une machine NUMA

Aucune des solutions proposées dans la section précédente ne correspondait à notre besoin, nous avons créé notre propre ordonnanceur de tâches. Nous l'avons appelé NATaS, il s'agit de l'acronyme *Numa Aware Task Scheduler*. Celui-ci est très basique, il ne prend en compte que l'affinité mémoire des tâches, la gestion des dépendances est laissée à Taggre, tout comme avec les ordonnanceurs OpenMP et TBB. Pour ordonnancer les tâches avec la meilleure affinité mémoire possible, nous utilisons un conteneur de tâches thread-safe par banc NUMA. Ce container permet à plusieurs threads d'insérer/retirer des tâches en limitant la contention. Le vol de tâche entre conteneurs a aussi été implémenté, il existe une option par tâche pour autoriser ou non le vol de tâche. Dans le cas du parallélisme de boucle, une option permettant de donner une tâche spécifiquement à un thread a été implémentée.

Contrairement à la plupart des autres ordonnanceurs de tâches, nous n'utilisons pas un conteneur de tâches par thread, mais un conteneur par banc NUMA. Nous avons donc plus de contention sur cette structure et une queue basique ne serait pas assez efficace. À la place, nous utilisons un conteneur sans verrou, entièrement construit autour de l'instruction compare-and-swap. Nous limitons donc les processeurs capables de faire fonctionner NATaS à ceux ayant l'instruction compare-and-swap. Cette structure ne nous permet pas d'obtenir les mêmes performances que l'utilisation d'une queue par thread, mais elle a l'avantage de mieux fonctionner pour l'équilibrage de charge à l'intérieur d'un banc NUMA. Le choix des structures de données a été fait pour obtenir de bonnes performances sur la machine cible composée de deux bancs NUMA.

NATaS fournit aussi une API permettant de gérer les allocations mémoires et leurs placements. Il permet de faire différents types d'allocations tels que :

— distribuer régulièrement la mémoire en mode bloc ;
— entrelacer les pages mémoires ;
— spécifier l'emplacement mémoire.

Ces allocations font miroir aux différentes politiques de gestion mémoire (first touch, interleave et bind). Dans le cas d'un parallélisme de boucle avec une distribution statique, on distribuera la mémoire régulièrement. Cet espace mémoire sera découpé en autant de blocs qu'il y a de bancs NUMA sur la machine. Puis chaque bloc sera placé sur un banc NUMA.

Taggre utilisera l'interface de NATaS pour améliorer les performances sur des machines NUMA. Il fournira au programmeur une interface simplifiée permettant de déplacer la mémoire juste en déclarant pour

1. Memory Affinity Interface

```
1  /* Créer un objet graphe */
2  Schema schema;
3
4  /* Créer les noeuds et les arêtes du graphe, puis grossir le grain du
       graphe et création des tâches utilisateurs */
5  ...
6
7
8  /* Distribution équitable des tâches sur les bancs NUMA */
9  schema.distribute_on_numa_nodes();
10
11 /* Simulation d'une exécution pour déclarer les données utilisées */
12 schema.run_function([&](Task *t)
13                      {
14 /* Enregistre une information d'utilisation d'une donnée par une tâche
       */
15 /* La banc NUMA est déduit à partir du thread appelant */
16                          schema.register_numa_data(&my_array[t->id],
17                                                    t->data_size);
18                      });
19
20 /* Calcule les données devant être déplacées et les déplace sur le
       meilleur banc NUMA possible */
21 schema.move_numa_data();
22
23 /* Exécution efficace */
24 schema.run();
```

Listing 4.1 – Exemple de code permettant de déplacer la mémoire sur une machine NUMA avec Taggre

chaque tâche la mémoire utilisée dans celle-ci. La connaissance complète du graphe de tâches permet des améliorations notables sur la distribution mémoire. Dans un premier temps, on va équilibrer la distribution des tâches sur les bancs NUMA en attribuant une affinité NUMA aux tâches (Fig. 4.3). Comme le graphe sera déroulé de haut en bas lors de son exécution, il parait naturel de distribuer les tâches par hauteur. En supposant que les tâches produisent des données et que ces données sont passées en paramètre aux tâches successeurs dans le graphe, on peut essayer d'optimiser le placement NUMA. Cette affinité sera choisie en fonction de la hauteur de la tâche dans le graphe et des affinités NUMA de ses prédécesseurs. Le but étant d'avoir à hauteur fixée, un nombre égale de tâche par banc NUMA tout en minimisant les accès en lecture distante. Une fois l'affinité NUMA de toutes les tâches définie, il ne nous reste plus qu'à connaître les données utilisées. Donc dans un deuxième temps, le programmeur indiquera les données utilisées dans chaque tâche. Pour cela, il lui suffira de simuler l'exécution du code, de désactiver le vol de tâches et d'appeler la fonction d'enregistrement mémoire. Nous obtiendrons ainsi une mémoire équitablement distribuée et des accès mémoire optimisés.

Le listing 4.1 montre comment à partir d'un graphe déclaré dans Taggre, on obtient un graphe dont les tâches sont distribuées sur les différents bancs NUMA. La première méthode à appeler est *distibute_on_numa_nodes*. Cette méthode va définir une affinité NUMA pour chaque tâches du graphe. Puis nous appelons, pour chaque tâche grâce à la méthode *run*, une fonction qui enregistre les données utilisées dans la tâche. Finalement, la méthode *move_numa_data* déplace les données pour faire en sorte que les données se retrouvent sur le banc NUMA qui les utilisera le plus.

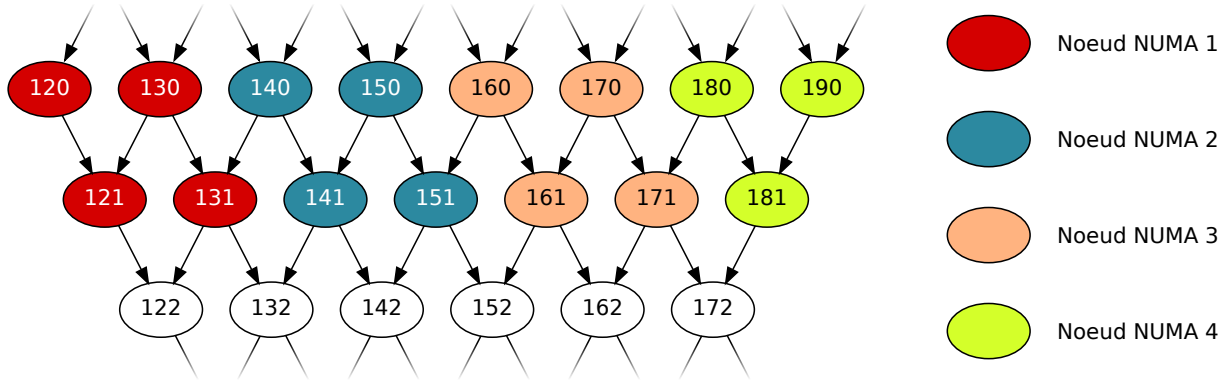



FIGURE 4.3 – Exemple d'application de l'algorithme de distribution des tâches en action, la couleur des tâches détermine leur affinité NUMA. Les tâches en blanc ne sont pas encore traitées.

## 4.4 Résultats

Nous allons maintenant présenter les résultats obtenus sur trois des noyaux de calcul du GMRES. Dans un premier temps nous présenterons les résultats du produit matrice vecteur creux. Puis nous présenterons les performances de la factorisation ILU(0) ainsi que les résolutions triangulaires associées. Pour évaluer les gains de notre solution, nous allons à chaque fois présenter les résultats obtenus avec les différentes stratégies d'allocations disponibles. La stratégie consistant à utiliser la mémoire distribuée nous indiquera une approximation de la performance maximale que nous pouvons atteindre en mémoire partagée. Puis nous évaluerons les politiques d'allocation first touch et interleave. Nous pourrons ensuite comparer ces résultats à la solution que nous avons développée : NATaS. Et en dernier, nous testerons la politique d'équilibrage NUMA automatique mis en place dans les versions récentes du noyau Linux. Les différents tests seront effectués à la fois sur Rostand et sur Manumanu, seul le test de la politique d'équilibrage NUMA automatique sera effectué sur une machine différente.

Nous utiliserons trois cas tests qui correspondent tous à une matrice représentant un cube de 100 éléments de côté. Nous faisons seulement varier le nombre de variables primaires pour faire varier la taille en mémoire du cas test ainsi que la possibilité de réutiliser des données en cache. La variation de la taille du cube n'aurait permis que de faire varier suffisamment la taille des cas tests. Nous utiliserons des cas à 1, 3 et 8 variables primaires qui nous donneront respectivement trois matrices dont les tailles mémoires seront de 52 Mo, 470 Mo et 3,33 Go.

### 4.4.1 Multiplication matrice vecteur creuse

La multiplication d'une matrice creuse par un vecteur est une opération dont le rapport nombre d'opérations par le nombre d'octets lus est petit. Dans le cas d'une matrice scalaire, ce ratio vaut environ $1/10$ en double précision. Pour chaque valeur non-nulle de la matrice, il faut lire cette valeur, l'indice de la colonne et la valeur contenue dans le vecteur à l'indice de la colonne. Il faut ensuite multiplier les deux valeurs ensemble et l'ajouter à un accumulateur, ce qui fait en double précision 2 opérations pour 20 octets lus, l'accumulateur pouvant tenir dans un registre, il n'est pas compté dans le nombre d'octets lus. Si nous utilisons trois variables primaires, chaque entrée de la matrice est un bloc 3 par 3. Nous devons donc lire ce bloc (9*8 Octets), lire l'indice de colonne (4 Octets) et finalement lire 3 valeurs dans le vecteur (3*8 Octets). Pour chaque valeur du bloc nous avons 2 opérations à faire (2*9), nous avons donc un ratio de $18/100$ soit environ $1/5,5$. Avec huit variables primaires, le ratio monte à environ $1/4,5$.

Le *roofline model*[WWP09] est un modèle de performance permettant de connaître la puissance de calcul maximale pouvant être atteinte par un algorithme sur une machine. Ce modèle est composé d'une courbe représentant les performances de la machine. En abscisse, nous avons le ratio du nombre d'opérations par le nombre d'octet lus depuis la mémoire, qui permettra de décrire les algorithmes, l'unité est en opérations par octets. En ordonnée, nous avons la capacité de calcul de la machine, l'unité est en opérations par seconde. Pour construire ce modèle, nous avons besoin de connaître deux paramètres de la machine :

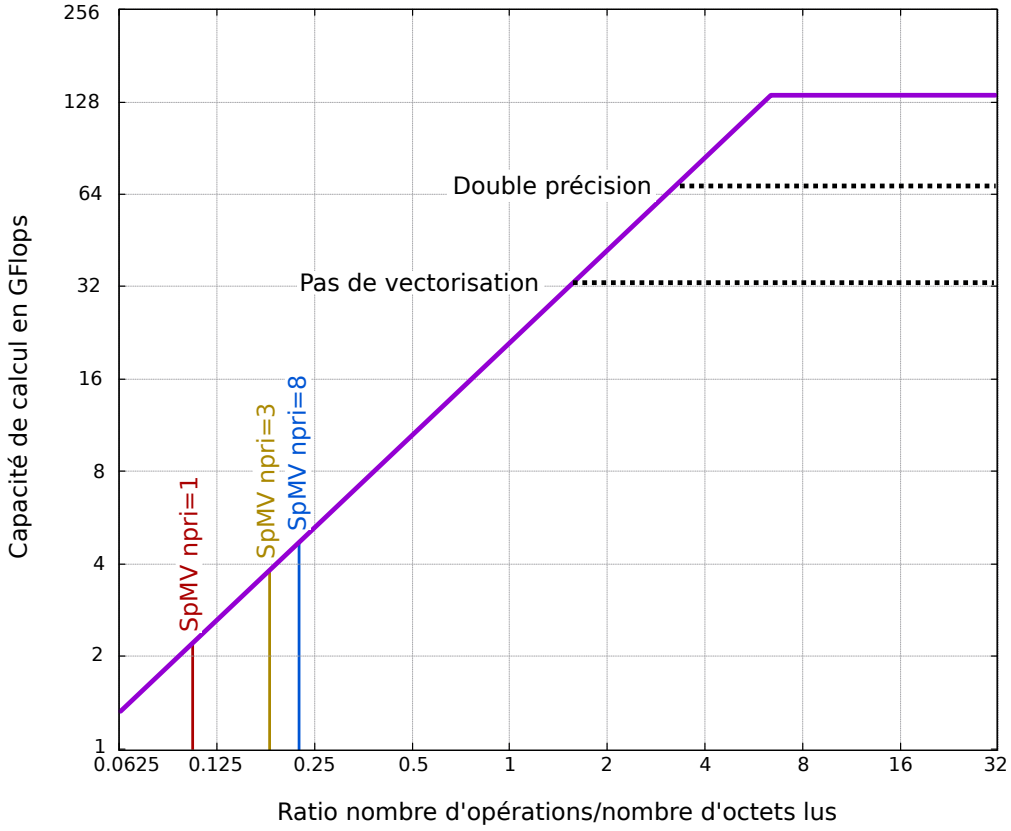



FIGURE 4.4 – Roofline model de Rostand avec les différents produits matrice vecteur creux.

— la puissance crête, en GFlops, qui correspond au nombre maximal d'opérations flottantes que la machine est capable de faire en une seconde ;
— la bande passante mémoire, en Go/s, qui correspond au débit maximal.

La première partie de la courbe correspond aux algorithmes limités par la bande passante mémoire, il s'agit d'une fonction linéaire dont la formule est :

$$f(x) = bande\_passante * x, \quad 0 \leqslant x \leqslant \frac{puissance\_crete}{bande\_passante} \tag{4.1}$$

La deuxième partie de la courbe correspond aux algorithmes limités par la puissance de calcul de la machine, les valeurs correspondent à la puissance crête de la machine.

Pour construire le roofline model d'un noeud de Rostand, nous allons procéder de la façon suivante : dans un premier temps nous allons mesurer la bande passante maximale de la machine. Pour cela nous avons utilisé le benchmark STREAM[McC07]. Sur Rostand, nous obtenons une bande passante de 21 Go/s en prenant en compte les 2 bancs NUMA. Puis, dans un second temps, nous allons calculer la capacité de calcul maximale de la machine. Pour calculer cette capacité, il faut multiplier le nombre de coeurs de calcul par le nombre maximal d'opérations faites dans une instruction et multiplier le tout par la fréquence d'horloge. Chaque noeud de Rostand étant composé de 12 coeurs cadencés à 2,80 GHz et du jeu d'instructions SSE 4.2 permettant d'effectuer 4 opérations flottantes en simple précision à la fois, cela donne 134,4 GFlops. Le nombre d'opérations simultanées en double précision est divisé par 2, donc on peut avoir au maximum 67,2 GFlops et si nous n'utilisons pas le jeu d'instructions vectorielles, nous pouvons obtenir au maximum 33,6 GFlops (Fig. 4.4).

Une fois le roofline model construit, nous pouvons donc placer le produit matrice vecteur creux. Les performances du SpMV dépendent du nombre de variables primaires, nous avons donc placé sur le roofline model trois SpMV en fonction du nombre de variables primaires utilisées. Ces trois points nous indiquent que les performances du SpMV seront limitées par la bande passante mémoire. L'utilisation du jeu d'instructions vectorielles aura donc très peu d'impact sur nos performances. Nous devons nous concentrer sur les accès mémoire et surtout dans notre cas, sur les effets NUMA.

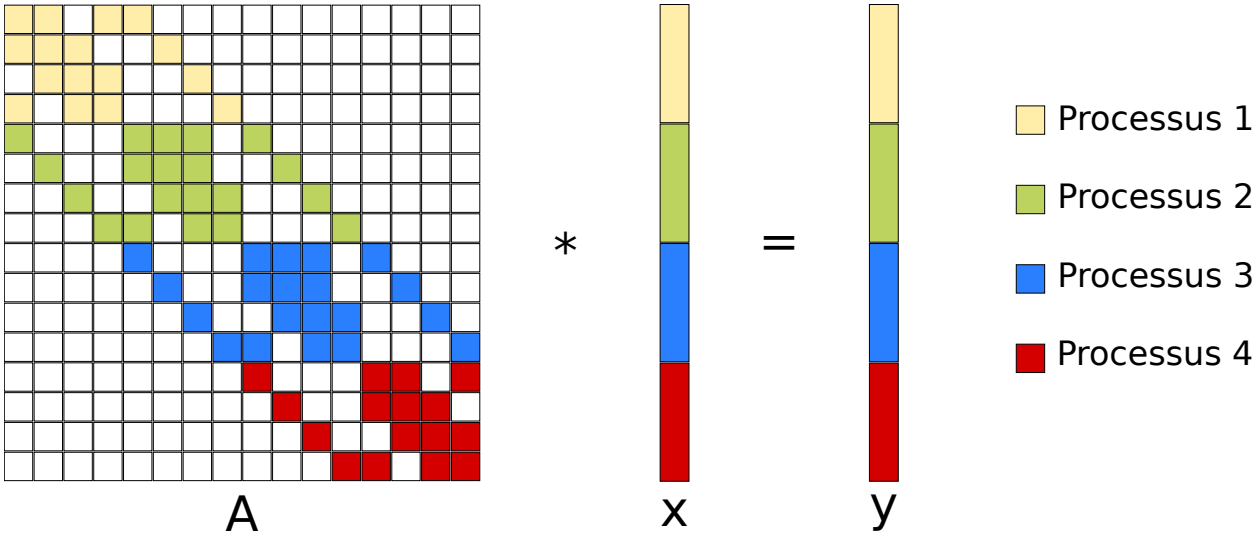



FIGURE 4.5 – Distribution des données utilisées par le produit matrice vecteur creux.

#### 4.4.1.1 Mémoire distribuée

Le SpMV présente du parallélisme de boucle sur les lignes de la matrice. Nous pouvons multiplier indépendamment chaque ligne de la matrice par un vecteur et stocker directement le résultat dans un autre vecteur. Pour paralléliser un produit matrice vecteur creux en mémoire distribuée, il faut tout d'abord distribuer la matrice sur tous les processus. Dans notre cas, cette distribution provient du partitionnement effectué pour la méthode Jacobi par bloc (section 2.3.3). Chaque processus devra s'occuper d'un ensemble de lignes de la matrice. Cette répartition sera statique et les indices des lignes seront les mêmes pour la distribution des vecteurs (Fig. 4.5). Nous pouvons voir que certains éléments du vecteur $x$ doivent être partagés. Pour cela, l'algorithme démarrera par une phase de communication de ces éléments. Puis chaque processus pourra effectuer la multiplication de ces lignes de matrice et stocker le résultat dans un vecteur $y$ local. Hormis la phase de communication au début de l'algorithme, le SpMV se parallélise très bien en mémoire distribuée. Cette phase de communication peut être, dans certains cas, recouverte par du calcul. Il suffit de multiplier en premier les lignes de la matrice qui ne dépendent pas des éléments distants. Dans le cas d'une matrice issue d'un maillage, cela correspond aux cellules intérieures. Puis une fois les éléments reçus, il est possible de multiplier les lignes restantes.

La performance atteinte en mémoire distribuée correspond à la performance crête du SpMV. En effet, de par sa nature distribuée, les pénalités mémoires NUMA sont minimales. Chaque processus allouera la mémoire dont il a besoin sur son propre banc NUMA et si celui-ci n'est pas déplacé, tous ces accès mémoire seront optimaux. Nous pouvons donc estimer cette performance sera la borne maximale à atteindre lorsque l'on travaillera en mémoire partagée.

Le roofline model prédit un algorithme limité en performance par la bande passante mémoire. Or, cette bande passante mémoire est partagée entre les coeurs d'un même banc NUMA. L'accélération obtenue sera donc limitée par la bande passante mémoire. Sur la machine Rostand, la bande passante mémoire limite grandement cette accélération (Fig. 4.6(d)). Avec un cas à 8 variables primaires, nous obtenons une accélération maximale de 3,8. La capacité de calcul mesuré avec 12 coeurs est de 4,96 GFlops, cela correspond à la prédiction du roofline model. L'utilisation d'un seul processeur (6 coeurs de calcul) donne aussi les mêmes résultats que ceux produits par le roofline model correspondant.

Sur Manumanu, nous avons beaucoup plus de bancs NUMA, ce qui signifie que nous aurons plus de bande passante mémoire à notre disposition. Nous pouvons donc espérer avoir de meilleurs résultats que sur Rostand. Il faut aussi prendre en compte une bande passante mémoire plus élevée sur les bancs NUMA de Manumanu que sur ceux de Rostand. Nous avons choisi d'allouer les processus MPI en mode compact, c'est-à-dire qu'ils sont distribués de façon à utiliser un minimum de noeuds NUMA. Nous devrions ainsi voir les différents paliers tous les 8 coeurs qui correspondent à l'ajout de bande passante mémoire d'un nouveau processeur. Ces paliers sont difficilement visibles sur la courbe des résultats (Fig. 4.7(d)). Mais les valeurs numériques sont plus expressives, par exemple avec 1 variable primaire, l'accélération sur 7 et 8 coeurs est la

même (4,72), l'utilisation d'un neuvième coeur la fait augmenter (5,34). Sur 1 banc NUMA, nous avons une accélération de 6 avec 8 variables primaires. Cette accélération monte à 110 avec l'utilisation des 20 bancs NUMA et des 160 coeurs. Les performances par banc NUMA sont meilleures que sur Rostand. Le passage à l'échelle entre 1 banc NUMA et 20 bancs NUMA est quasiment parfait.

#### 4.4.1.2 First touch

Nous allons maintenant nous concentrer sur la parallélisation du SpMV en mémoire partagée. La mémoire est allouée sur un seul banc NUMA et le travail est partagée par une directive *#pragma omp parallel for*. Sur la machine Rostand, nous obtenons difficilement une accélération de 2,5 sur 12 coeurs en ayant 8 variables primaires (Fig. 4.6(a)). Cette accélération descend à 1,9 en ayant 1 variable primaire, toujours sur 12 coeurs de calcul. Ces résultats sont à comparer avec ceux obtenus en mémoire distribuée. Nous n'obtenons que 65 % de la puissance maximale que nous devrions avoir. Le SpMV étant limité par la bande passante mémoire, l'utilisation d'un seul banc NUMA pour les accès mémoire ne nous permet pas d'exploiter toute la puissance de la machine.

Sur la machine Manumanu, ces effets sont amplifiés (Fig. 4.7(a)). Nous obtenons les meilleures performances en utilisant 8 coeurs avec une accélération de 5-6. Utiliser plus de 8 coeurs pour effectuer le SpMV fait perdre du temps, les données étant toutes sur le premier banc NUMA, nous utilisons uniquement la bande passante de ce banc avec des latences d'accès plus ou moins longues. Les résultats en mémoire distribuée sont meilleurs. Pour obtenir les mêmes performances qu'en mémoire distribuée, nous devons optimiser les accès mémoire.

#### 4.4.1.3 Interleave

Pour diminuer les effets NUMA, nous pouvons utiliser la politique d'allocation interleave. Cette politique va distribuer uniformément les pages mémoires sur les différents bancs NUMA. Nous allons donc augmenter la bande passante mémoire en ne modifiant que la latence mémoire moyenne. Sur Rostand, nous obtenons un gain de performance d'environ 20 %, mais les performances sont toujours en dessous des performances obtenues en mémoire distribuée (Fig. 4.6(b)).

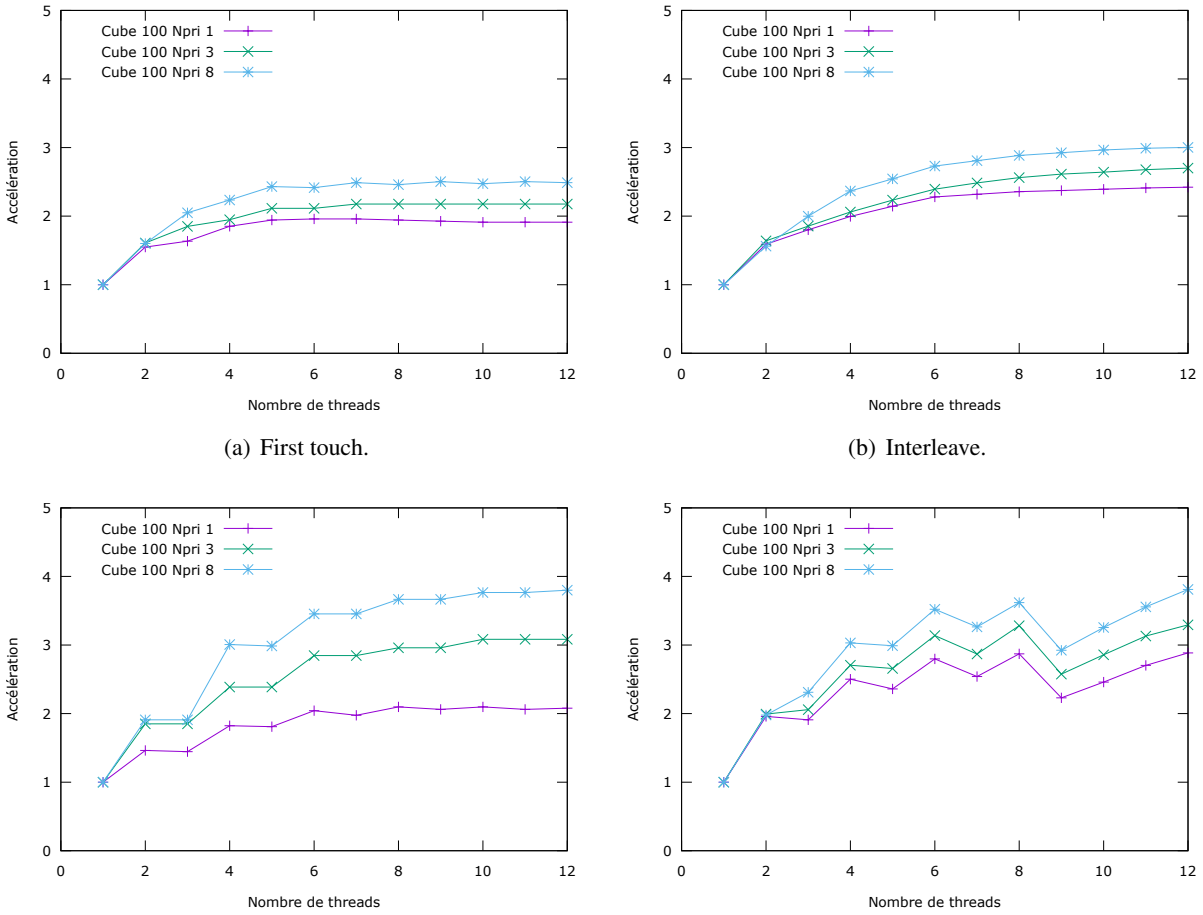
Sur Manumanu, on obtient de bons résultats jusqu'à 16 coeurs (Fig. 4.7(b)). Au-delà, nous commençons à utiliser le SGI® NUMAlink™[Gro12] et les temps de latence des accès mémoire augmentent. En effet, la majorité des accès mémoire se font sur des bancs NUMA distants. Au final, les résultats de l'allocation interleave sur Manumanu sont proches des résultats de l'allocation first touch. Ce n'est donc pas la bonne solution pour exploiter les performances de cette machine.

#### 4.4.1.4 NATaS

NATaS distribue les pages mémoires utilisées par la matrice en mode bloc de lignes. Nous divisons le nombre de lignes par le nombre de bancs NUMA pour obtenir la taille d'un bloc. Puis chaque bloc est alloué sur un banc NUMA différent. En plus d'optimiser le placement des pages, NATaS répartira la charge de travail en prenant en compte la localité mémoire. Sur le cas à 8 variables primaires, nous obtenons les mêmes performances qu'en mémoire distribuée (Fig. 4.6(c)). Pour 3 variables primaires, nous obtenons 95 % des performances de la version en mémoire distribuée. Par contre, nous n'obtenons que 70 % avec 1 variable primaire.

Malheureusement, sur la machine Manumanu, malgré un placement optimal des pages mémoires, NATaS ne fait pas aussi bien que MPI (Fig. 4.7(c)). Les compteurs matériels nous indiquent qu'un nombre non négligeable d'accès à une mémoire distante sont faits. Or, nous avons vérifié l'emplacement physique des pages mémoires de la matrice ainsi que celles des vecteurs et celles-ci sont bien placées. Les accès à la mémoire distante proviennent donc de l'ordonnanceur de tâches. C'est en grande partie l'architecture logicielle de NATaS qui en est la cause. Nous effectuons trop d'accès à un ensemble de variables partagées par tous les threads. Par exemple, nous avons un compteur de tâches en cours qui nous permet de savoir s'il existe encore des tâches pouvant être volées ou débloquées. Le vol de travail est aussi consommateur d'accès mémoire distant. Actuellement, lorsqu'un thread tombe à court de tâche, il parcourt les structures servant à maintenir la liste des tâches disponibles sur les autres bancs NUMA. Pour améliorer considérablement

les performances, il faudrait construire un ordonnanceur utilisant moins de variables partagées. Ce type d'ordonnanceur étant difficile à écrire et nos machines de calculs étant essentiellement composées de 2 bancs NUMA, nous ne nous sommes pas intéressé à ce genre d'ordonnanceur. L'utilisation de 2 bancs NUMA sur Manumanu nous donne une accélération de 10, tout comme la version en mémoire distribuée. Au-delà de 2 bancs NUMA, les performances sont légèrement meilleures que la version OpenMP, atteignant une accélération de 32.




(a) First touch.

(b) Interleave.

(c) NATaS.

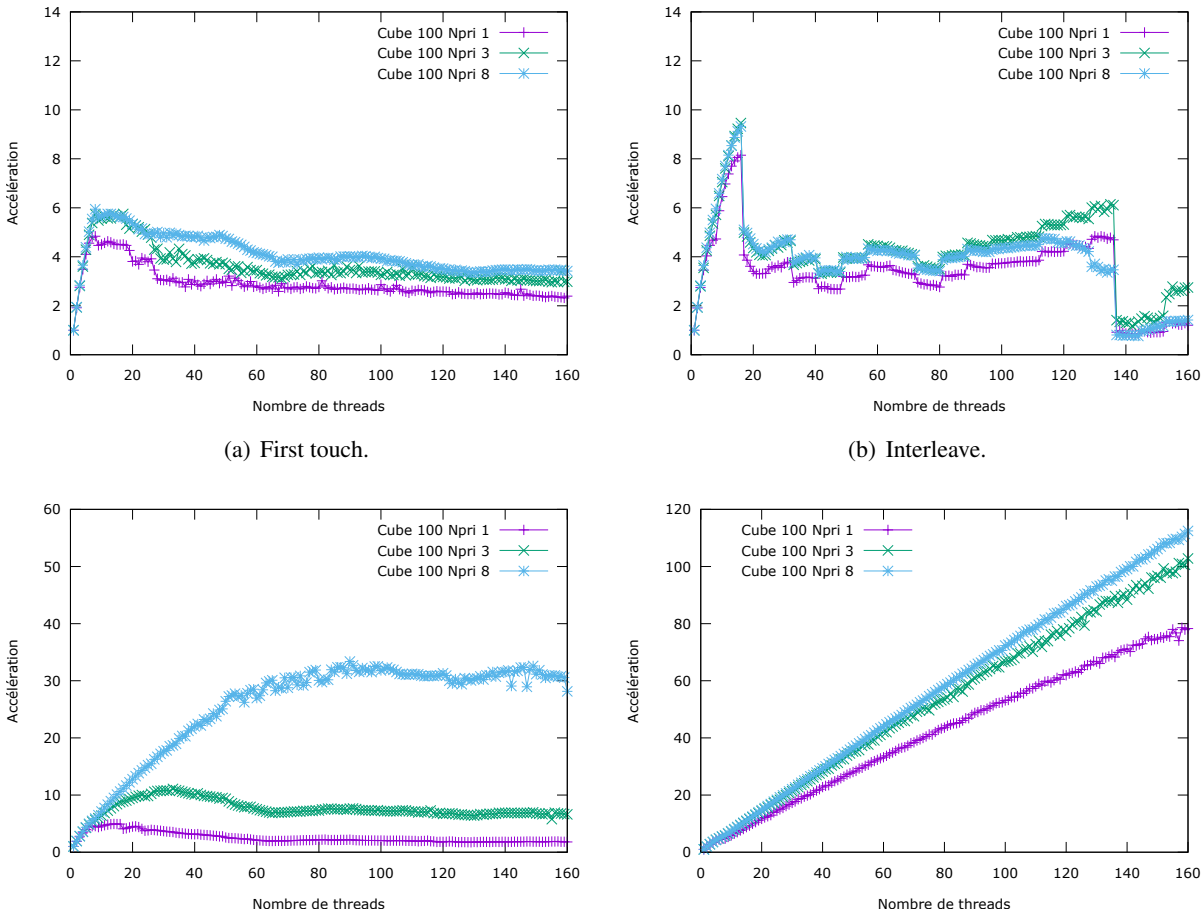
(d) MPI.

FIGURE 4.6 – Performances du produit matrice vecteur creux sur Rostand. Les threads sont distribués de façon équilibrée sur le plus de bancs NUMA possible. Par exemple, pour 6 threads, il y a 3 threads par banc NUMA.

#### 4.4.1.5 Équilibrage automatique NUMA

Les noyaux Linux récents proposent un équilibrage de charge automatique des pages mémoires. Malheureusement, nous ne pouvons pas utiliser les grappes de serveurs à notre disposition pour tester cette fonctionnalité. La version de Linux disponible sur ces machines n'est pas assez récente, la fonctionnalité AutoNUMA n'est apparue que dans la version 3.13 du noyau. À la place, nous allons utiliser une machine de bureau contenant deux processeurs Intel Xeon X5660, chaque banc NUMA dispose de 6 coeurs de calculs et de 24 Go de mémoire vive. La version de Linux utilisée est la 3.18.

Cette méthode ne fonctionne que lorsque le programme est exécuté suffisamment longtemps pour avoir le temps d'analyser toute la mémoire utilisée. Nous allons chercher l'accélération maximale que nous pouvons atteindre avec cette solution. Il ne nous est donc pas utile de faire varier le nombre de coeurs de calcul, nous utiliserons les 12 coeurs de calcul de la machine. Nous allons plutôt faire varier le nombre de SpMV pour faire varier le temps d'exécution du programme et laisser au noyau assez de temps pour déplacer les pages.




(a) First touch.

(b) Interleave.

(c) NATaS.

(d) MPI.

FIGURE 4.7 – Performances du produit matrice vecteur creux sur Manumanu. Les threads sont distribués de façon à utiliser le moins de bancs NUMA possible. Par exemple, pour 6 threads, il n'y a qu'un banc NUMA utilisé. À partir de 9 threads le banc NUMA le plus proche du premier banc NUMA commence à être utilisé.

Avec un nombre de répétitions faibles, AutoNUMA donne les mêmes performances que l'allocation first touch (Fig. 4.8). Au bout de 8 répétitions, soit environ 1,36 seconde, nous commençons à voir une amélioration des performances. Vers 384 répétitions, soit environ 1 minute, nous obtenons la performance crête d'AutoNUMA qui correspond aussi à la performance obtenue avec l'allocation interleave. Même si les données se retrouvent suffisamment bien placées, AutoNUMA conserve un surcoût d'exécution à cause de l'analyse mémoire en continue. De plus, l'ordonnanceur de tâches ne prend pas en compte les informations sur le placement des données, on conserve donc une exécution dégradée. Il est nécessaire de rappeler que l'allocation interleave donnait de bonnes performances avec l'utilisation de 2 bancs NUMA. Les meilleurs résultats sont obtenus avec NATaS. Il serait aussi intéressant de tester la méthode AutoNUMA sur Manumanu en utilisant plus de 2 bancs NUMA, nous pourrions savoir si les résultats sont meilleurs qu'avec NATaS. Malheureusement, nous ne pouvons pas changer le noyau utilisé sur cette machine.

### 4.4.2 Factorisation et résolution triangulaire

Le parallélisme de la factorisation ILU et de la résolution triangulaire est le même. La résolution triangulaire peut donc réutiliser le graphe grossier calculé pour la factorisation. Ce graphe grossier n'utilise pas forcément la stratégie d'agrégation qui lui est la plus favorable, mais nous avons pu observer que les stratégies d'agrégation utilisées pour ILU donnent aussi de très bons résultats pour la résolution triangulaire. Par contre, la fonction exécutée à l'intérieur des tâches sera différente, dans un cas il s'agira de factoriser une ou plusieurs lignes de la matrice, et dans l'autre cas il s'agira de résoudre une ou plusieurs équations linéaires

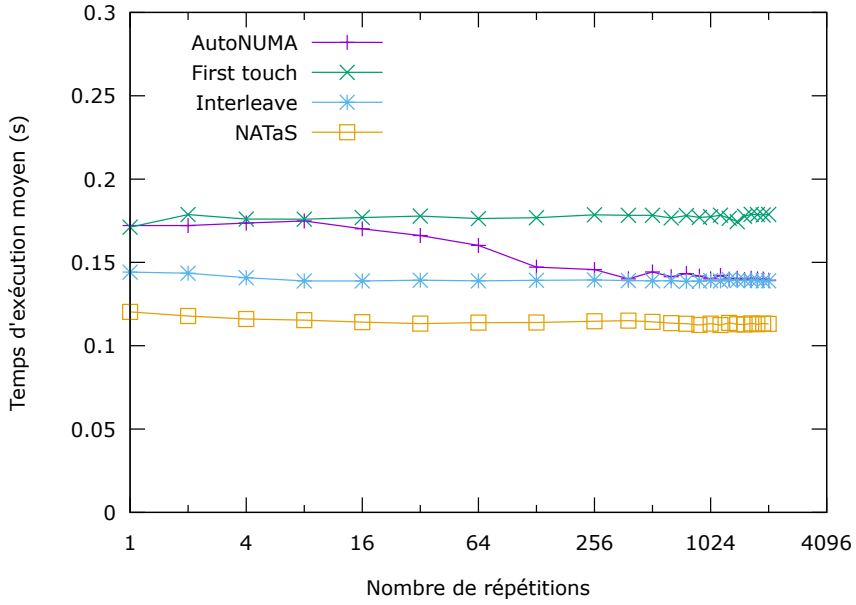



FIGURE 4.8 – Temps d'un produit matrice vecteur creux sur Linux 3.18 en mémoire partagée avec 12 coeurs. Nous utilisons une matrice représentant un cube 100 avec 8 variables primaires.

à une inconnue.

Les pages mémoires composant la matrice sont distribuées en respectant l'affinité mémoire des tâches associées. Taggre a défini cette affinité en distribuant les tâches sur les bancs NUMA en équilibrant le nombre de pages par banc NUMA pour chaque hauteur du graphe. Nous avons ainsi une distribution à peu près équitable des pages mémoires au fil du déroulement du graphe. Le même mécanisme qui sert à utiliser un seul graphe de tâches pour la factorisation et la résolution triangulaire est utilisé ici pour choisir les pages mémoires à déplacer. Pour cela, le programmeur doit créer une fonction qui enregistre auprès de Taggre les données utilisées, cette fonction sera celle exécutée par toutes les tâches du graphe. Puis Taggre s'occupera d'optimiser le placement des pages mémoires en effectuant les appels systèmes correspondants. Malheureusement, nous ne pouvons pas distribuer toutes les pages de manière optimale, car il arrive que certaines pages soient utilisées par plusieurs tâches ayant des affinités mémoires différentes. Dans un tel cas, nous choisissons de placer la page mémoire sur le banc NUMA ayant le plus de tâches.

#### 4.4.2.1 Mémoire distribuée

En mémoire distribuée, nous utilisons une méthode de Jacobi par blocs pour obtenir du parallélisme. Chaque processus factorise un bloc de la matrice en parallèle. Nous n'effectuons donc pas le même calcul que lors d'une factorisation en mémoire partagée. Il y a un peu plus de calcul à faire en mémoire partagée, ces calculs correspondent aux éléments en dehors des blocs qui sont ignorés avec la méthode de Jacobi par blocs. Par contre, cette méthode nous permettra d'évaluer une borne maximale de performances que nous pourrons obtenir en mémoire partagée. En effet, les types d'opérations sont les mêmes, seul l'ordre de traitement des lignes de la matrice change avec l'ajout de dépendances entre chaque ligne.

Sur la machine Rostand, la factorisation ILU(0) atteint une accélération de 9,9 (Fig. 4.9(d)) et la résolution triangulaire une accélération de 3,7 sur 12 coeurs de calcul (Fig. 4.11(d)). La factorisation est moins limitée par la bande passante mémoire que la résolution triangulaire. Sur la machine Manumanu, cette accélération monte à 93 pour la factorisation (Fig. 4.10(d)) et à 73 pour la résolution triangulaire (Fig. 4.12(d)) pour 160 coeurs de calcul. Si nous n'utilisons que deux processeurs de 8 coeurs de calcul chacun sur Manumanu, nous obtenons des accélérations de 15,18 pour la factorisation et 7,2 pour la résolution triangulaire. Nous pouvons donc voir que la résolution triangulaire passe mieux à l'échelle que la factorisation.

#### 4.4.2.2 First touch

Nous allons maintenant tester la politique d'allocation first touch sur la factorisation et sur la résolution triangulaire. Les matrices et les vecteurs seront donc alloués sur un seul banc NUMA, celui du thread qui a exécuté le code d'initialisation. Tous les accès mémoire passeront ensuite par ce banc NUMA, nous n'utiliserons donc qu'une partie de la bande passante mémoire de la machine. Nous avons aussi utilisé Taggre pour grossir le grain de calcul.

Sur la machine Rostand, nous obtenons au mieux une accélération de 8,7 (Fig. 4.9(a)) sur 12 coeurs pour la factorisation et une accélération de 2,8 pour la résolution triangulaire (Fig. 4.11(a)). Nous obtenons donc des performances en dessous des performances obtenues en mémoire distribuée. Il faut aussi prendre en compte que l'utilisation de plusieurs threads implique une gestion des dépendances entre les tâches de calcul.

Sur la machine Manumanu, nous obtenons le même type d'accélération que pour le produit matrice vecteur creux. Tant que nous utilisons moins de 2 bancs NUMA, nous obtenons une accélération de 10 pour la factorisation (Fig. 4.10(a)) et une accélération de 6,2 pour la résolution triangulaire (Fig. 4.12(a)). Au-delà de 16 threads, les performances chutent, les temps de latence des accès mémoires deviennent trop grands.

#### 4.4.2.3 Interleave

Pour essayer de diminuer les effets NUMA, nous activons la politique d'allocation mémoire interleave. Les pages mémoires sont donc distribuées uniformément entre chaque banc NUMA. Nous pouvons donc utiliser plus de bande passante qu'avec l'allocation first touch précédente.

Sur Rostand, la factorisation donne des résultats légèrement moins bons qu'avec une politique d'allocation first touch (Fig. 4.9(b)) Par contre, nous obtenons une amélioration entre 3 % et 30 % de la résolution triangulaire pour un nombre faible de variables primaires (Fig. 4.11(b)). Même si nous avons la possibilité d'utiliser plus de bande passante mémoire, ce n'est pas pour cela qu'elle sera mieux utilisée. En moyenne, la moitié des accès mémoire auront toujours une latence plus élevée.

Sur Manumanu, la factorisation se comporte de la même façon que sur Rostand (Fig. 4.10(b)). De même, l'accélération maximale de la résolution triangulaire est meilleure avec une politique d'allocation interleave (Fig. 4.12(b)). Comme pour la politique first touch, après 16 threads, les performances chutent.

#### 4.4.2.4 NATaS

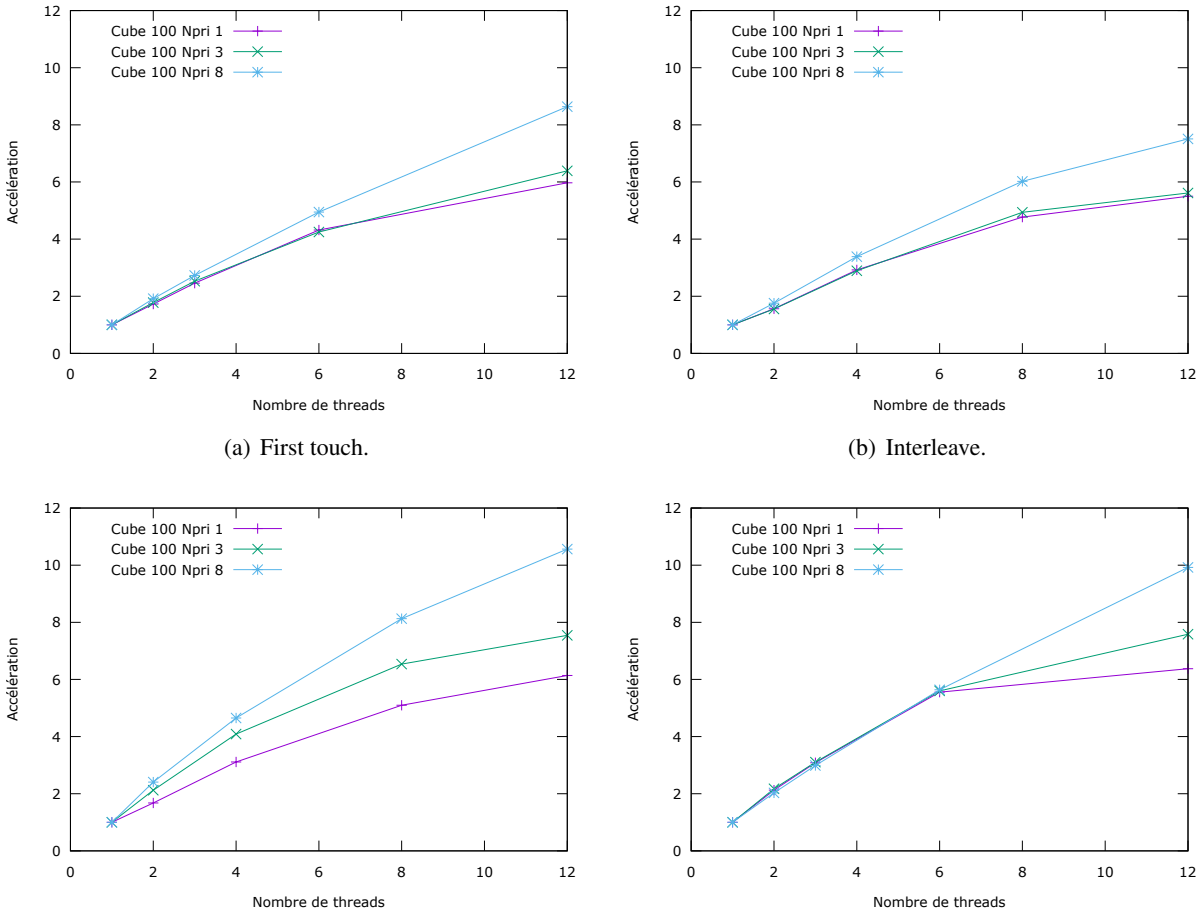
À la différence des autres ordonnanceurs, NATaS va tenir compte de l'affinité NUMA des tâches. Cette affinité a été définie par Taggre de telle sorte à équilibrer la charge sur les différents bancs NUMA. Donc, nous limitons le nombre d'accès aux bancs NUMA distants. La granularité de placement mémoire étant d'une page, il n'est pas possible d'allouer toutes les données correctement. Certaines données seront donc à cheval sur deux bancs NUMA différents. De plus, dans le cas de la résolution triangulaire, les vecteurs seront eux aussi à cheval sur plusieurs bancs NUMA.

NATaS offre de meilleures performances sur Rostand par rapport à la politique d'allocation interleave. La factorisation est 40% plus rapide avec 8 variables primaires (Fig. 4.9(c)) et la résolution triangulaire est 23% plus rapide (Fig. 4.11(c)). Avec 1 variable primaire, nous n'obtenons pas de gain sur la résolution triangulaire.

Sur Manamanu, les résultats sont meilleurs qu'avec les précédentes allocations (Fig. 4.10(c) et 4.12(c)). Par contre, au-delà de deux processeurs, les performances s'effondrent aussi. Cette fois l'ordonnanceur en est la cause, les données sont bien réparties et chaque thread accède à des données locales. Seul l'ordonnanceur accède à des données distantes pour des besoins de synchronisation. Ces accès ne sont pas gênants dans un cas où la variation de latence est faible tels que deux processeurs dans un même groupe. Mais cette variation devient gênante dès que l'on accède à un groupe NUMA distant.

#### 4.4.2.5 Équilibrage automatique NUMA

L'équilibrage automatique des pages mémoires ne donne pas de bonnes performances sur la factorisation (Fig. 4.13). Cette méthode d'allocation est la moins efficace de toutes. Avec un nombre suffisant d'itérations, l'allocation interleave donne les mêmes performances que l'allocation first touch. L'utilisation de NATaS reste la solution qui donne les meilleures performances.



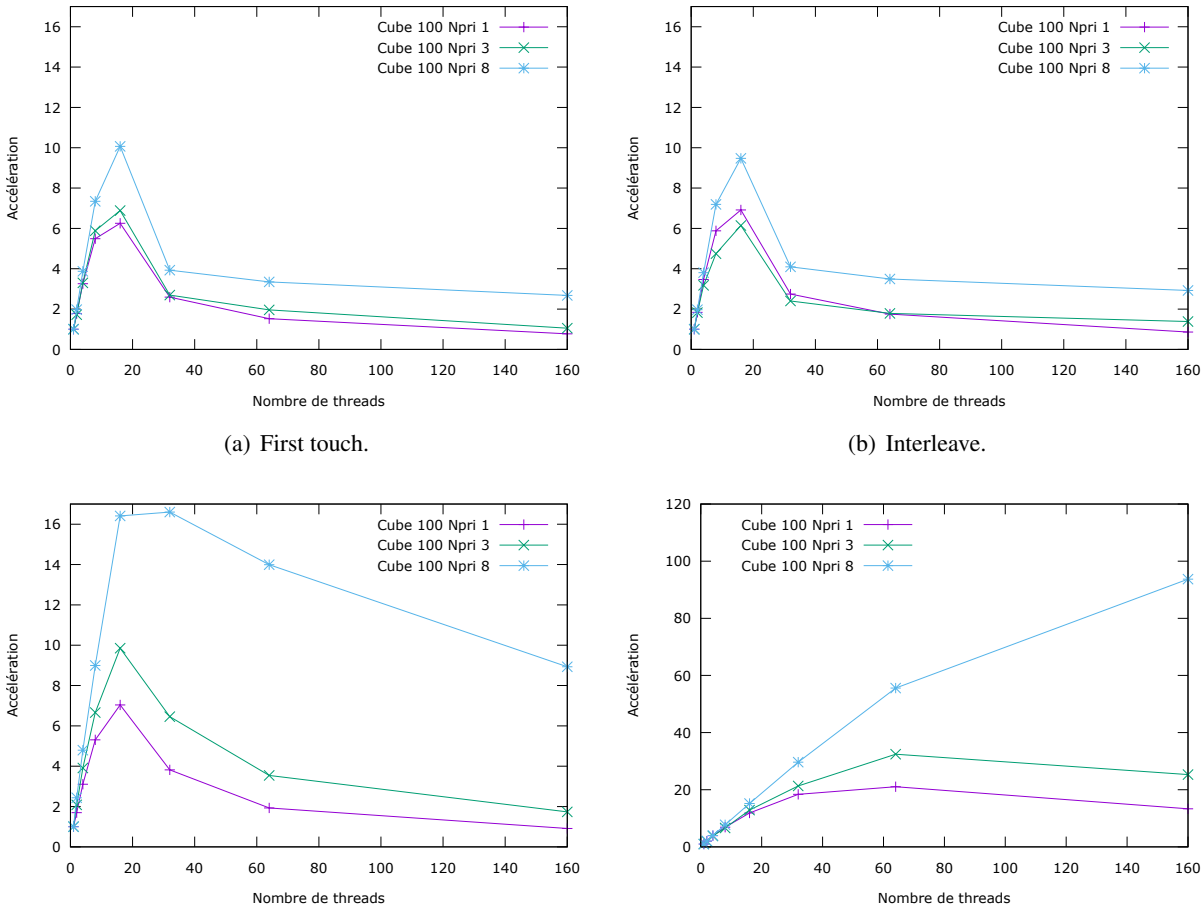

(a) First touch.

(b) Interleave.

(c) NATaS.

(d) MPI.

FIGURE 4.9 – Performances de la factorisation ILU(0) sur Rostand. Les threads sont distribués de façon équilibrée sur le plus de bancs NUMA possible.



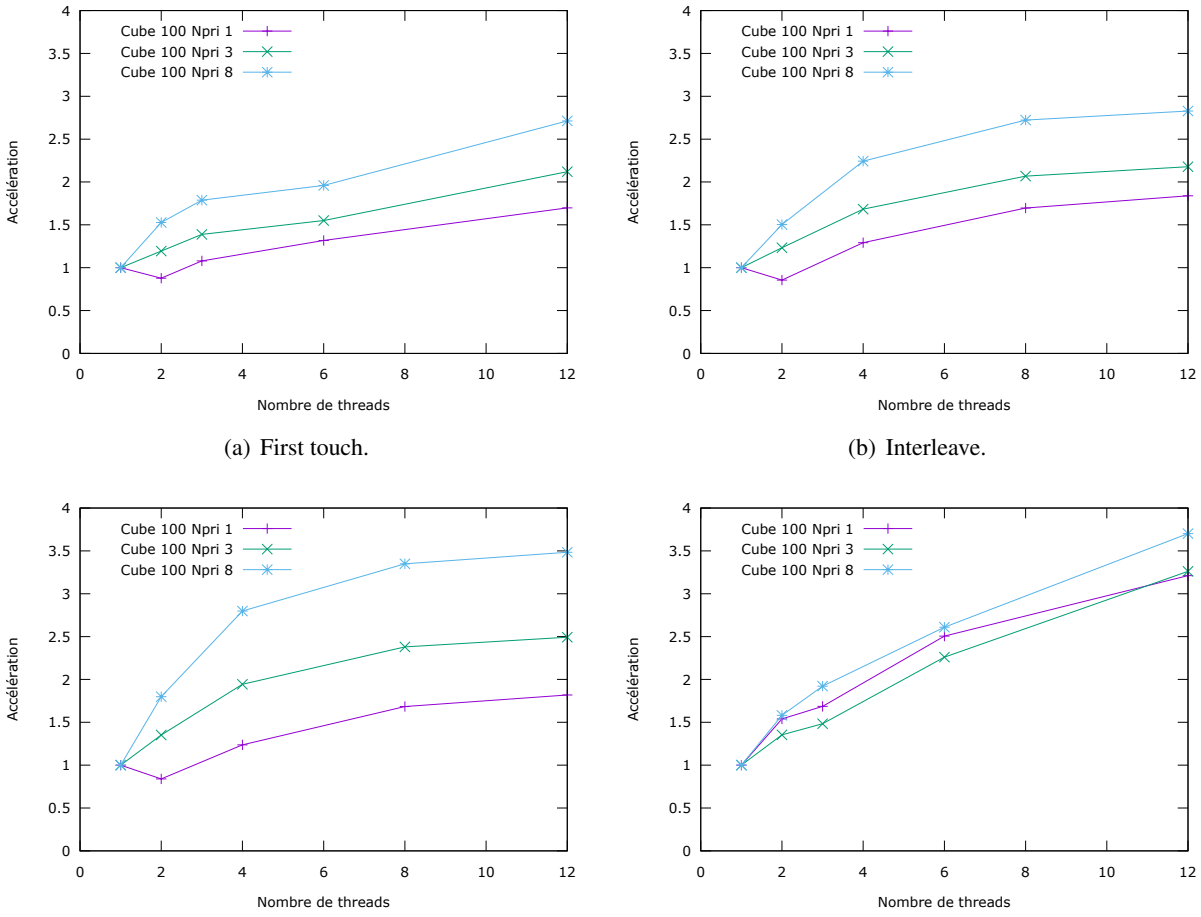

(a) First touch.

(b) Interleave.

(c) NATaS.

(d) MPI.

FIGURE 4.10 – Performances de la factorisation ILU(0) sur Manumanu. Les threads sont distribués de façon à utiliser le moins de bancs NUMA possible.



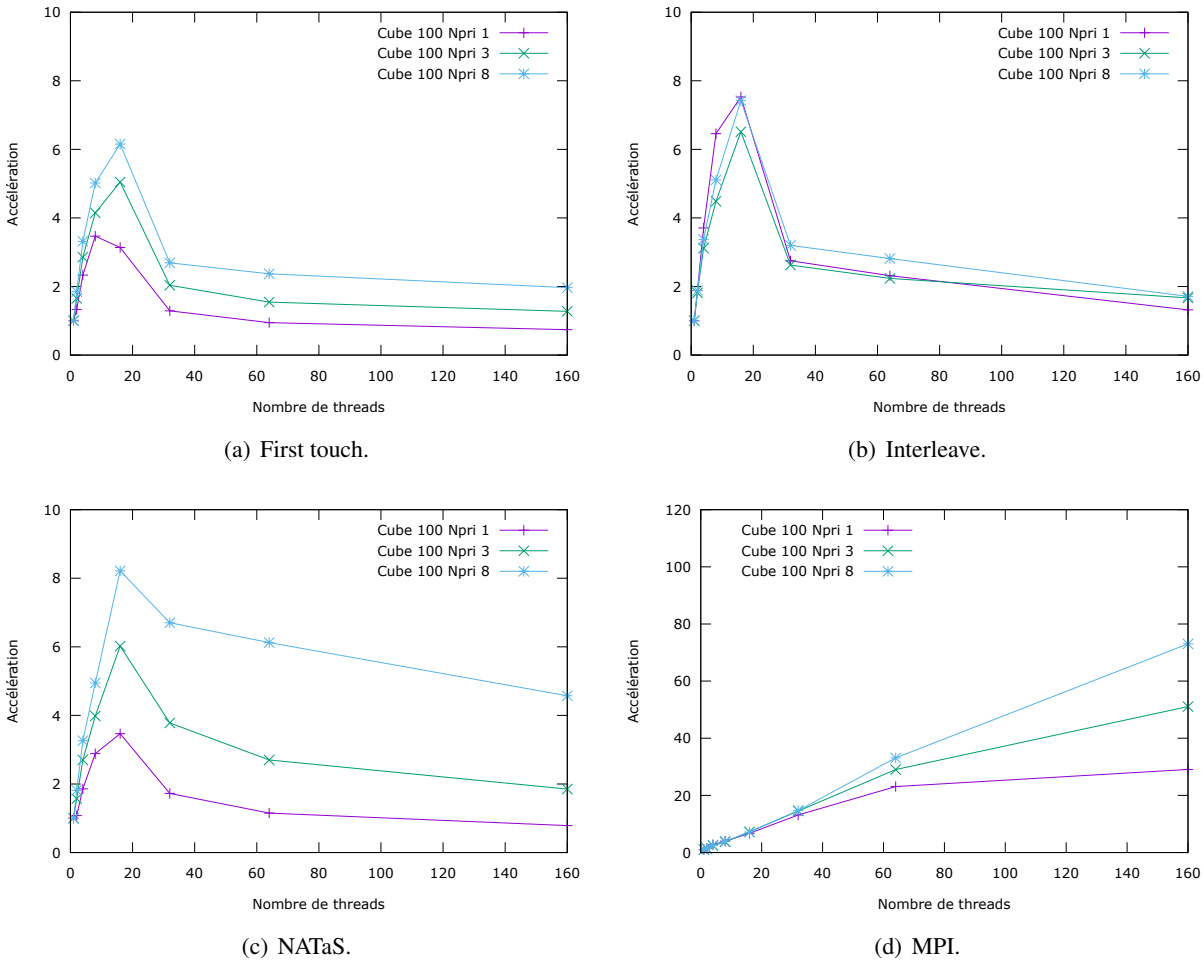

(a) First touch.

(b) Interleave.

(c) NATaS.

(d) MPI.

FIGURE 4.11 – Performances de la résolution triangulaire sur Rostand. Les threads sont distribués de façon équilibrée sur le plus de bancs NUMA possible.




(a) First touch.

(b) Interleave.

(c) NATaS.

(d) MPI.

FIGURE 4.12 – Performances de la résolution triangulaire sur Manumanu. Les threads sont distribués de façon à utiliser le moins de bancs NUMA possible.

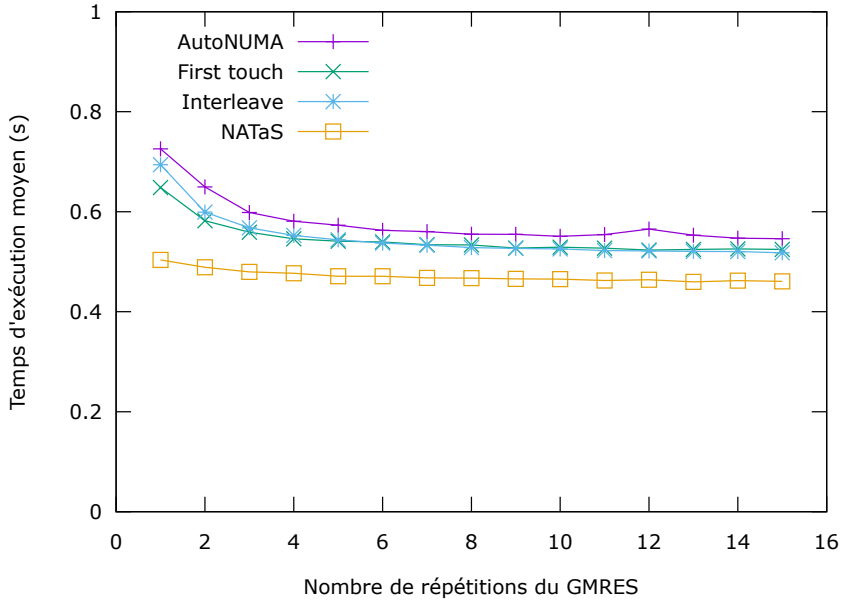



FIGURE 4.13 – Temps moyen d'une factorisation sur Linux 3.18 en mémoire partagée avec 12 coeurs. Nous utilisons une matrice représentant un cube 100 avec 8 variables primaires.

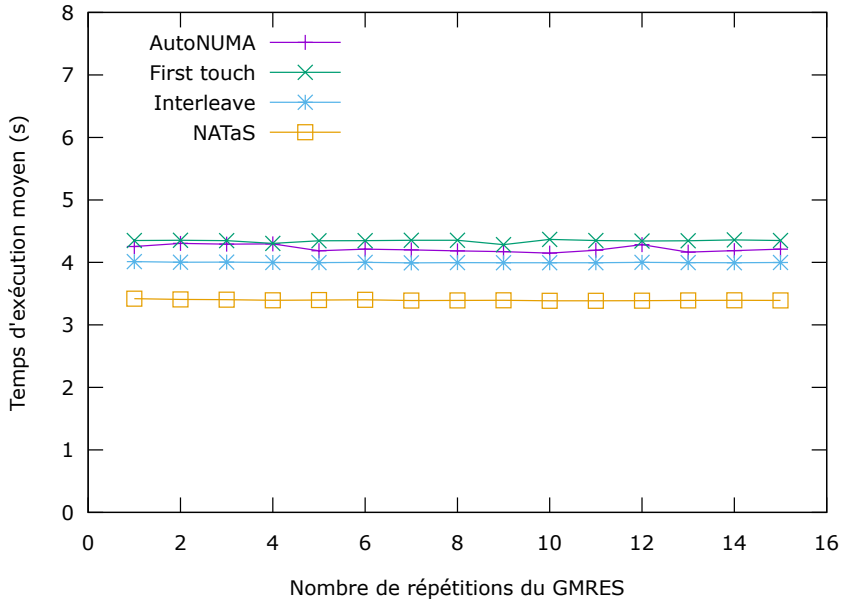



FIGURE 4.14 – Temps moyen d'une résolution triangulaire sur Linux 3.18 en mémoire partagée avec 12 coeurs. Nous utilisons une matrice représentant un cube 100 avec 8 variables primaires.

Par contre, la résolution triangulaire se comporte comme le SpMV (Fig. 4.14). La politique d'allocation AutoNUMA offre des performances intermédiaires aux politiques d'allocations first touch et interleave. Encore une fois, l'utilisation de NATaS est la plus efficace.

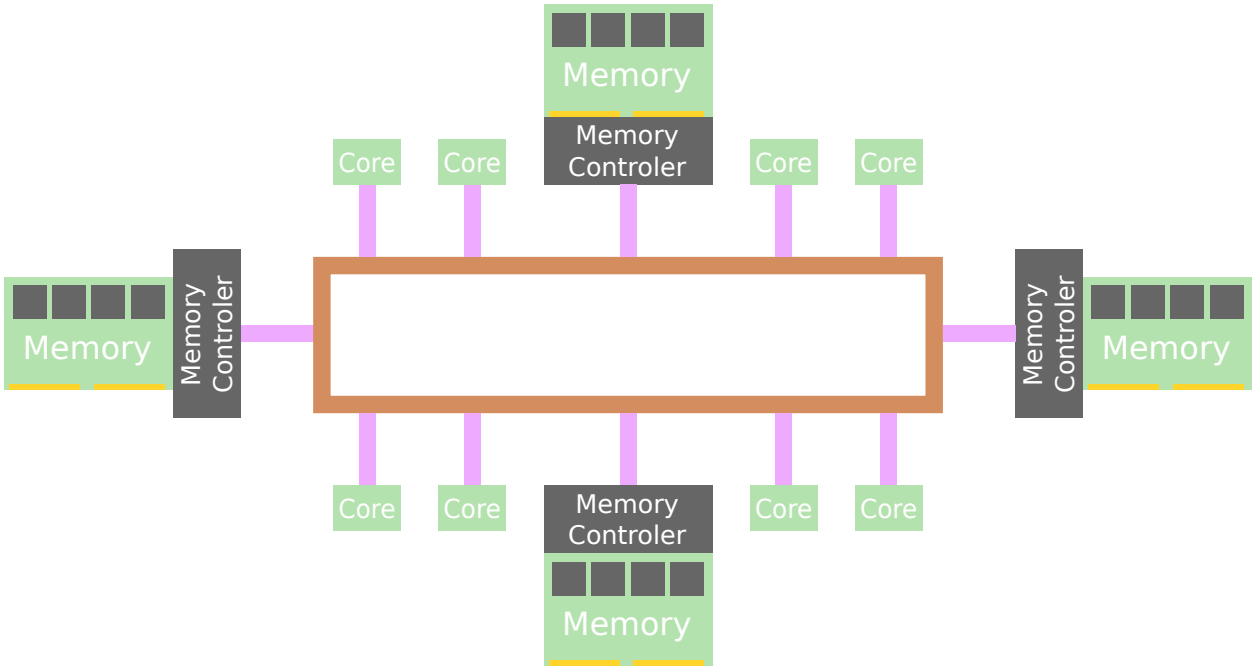



FIGURE 4.15 – Architecture en anneau du Xeon Phi.

### 4.4.3 Retour d'expérience sur le Xeon Phi

Le Xeon Phi est un co-processeur conçu par Intel. Il s'agit d'une carte branchée en PCI express et qui permet de faire du calcul déporté. Ce co-processeur est composé de 60 coeurs de calcul généralistes. Ces coeurs de calcul supportent le jeu d'instructions x86 et ont la particularité de pouvoir maintenir 4 contextes d'exécution simultanément. Il s'agit de la technologie HyperThreading ou SMT [1]. Un changement de contexte est effectué à chaque cycle. L'utilisation de plusieurs threads par coeur à l'avantage de pouvoir masquer les temps d'attentes mémoire. La mémoire du Xeon Phi est distribuée le long d'un anneau bi-directionnel (Fig. 4.15) sur lequel les coeurs de calcul sont aussi directement connectés[Rah13].

Cette agencement de la mémoire nous fournit de très bons débits mémoire comparé à l'agencement de la carte mère hôte. Pour cette raison, il peut être intéressant de tester nos noyaux d'algèbre linéaire creuse sur cet accélérateur.

L'accélération maximale est de 120 par rapport à un coeur du Xeon Phi (Fig. 4.16). Mais il faut rappeler que ces coeurs ne sont pas faits pour exécuter du code séquentiel. Les instructions sont exécutées dans l'ordre (in-order) et il n'y a donc pas de parallélisme d'instructions. De plus, la fréquence d'horloge est basse (1,2 GHz) et le thread est exécuté un cycle sur 2 (600 MHz). Il est donc très simple d'obtenir une bonne accélération par rapport à un coeur du Xeon Phi.

Par contre, si nous comparons le temps qu'il a fallu au Xeon Phi pour faire la multiplication par rapport au temps que mettent les deux processeurs hôtes à faire la même opération, le Xeon Phi ne met que 2 fois moins de temps (0,110 s sur 2 processeurs Xeon contre 0.058 s pour le Xeon Phi avec 8 variables primaires). Ce qui est déjà bien mais ces résultats ne sont pas suffisants pour utiliser des Xeon Phi. Pour comprendre pourquoi, il faut regarder le fonctionnement du Xeon Phi. En effet, il y a deux modes de programmation du Xeon Phi :

— le mode natif, qui consiste à faire tourner un noyau Linux sur le Xeon Phi et à l'utiliser comme un noeud de calcul ;
— le mode déporté, qui consiste à l'utiliser comme un accélérateur, à la manière d'un GPU.

Dans le mode natif, le code du simulateur de réservoir doit aussi tourner sur le Xeon Phi. Or les parties séquentielles du code s'exécutent environ 10 fois moins vite que sur un processeur classique. Donc les accélérations que nous obtenons sur les noyaux de calcul parallèle seront perdues à cause des parties séquentielles du code.

Quant au mode déporté, pour pouvoir exécuter les noyaux de calcul sur le Xeon Phi, nous devons

1. Simultaneous Multi Threading

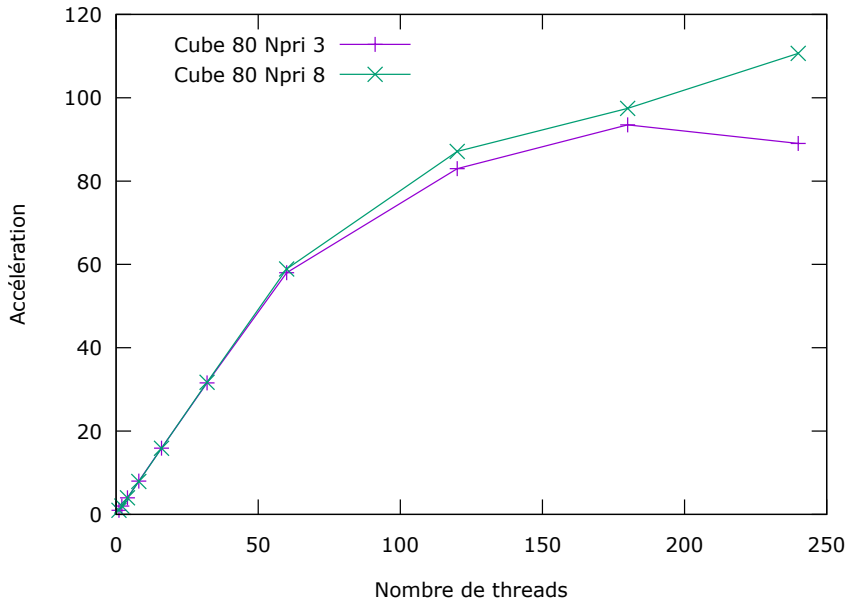



FIGURE 4.16 – Résultat sur produit matrice vecteur creux sur Xeon Phi.

transférer les données qui seront utilisées et/ou produites. Dans le cas du produit matrice vecteur creux, nous devons d'abord transférer la matrice et le vecteur, puis à la fin du calcul nous devons récupérer un vecteur. Or, ces transferts coûtent du temps et ce temps cumulé à l'exécution du noyau de calcul est plus long que l'exécution du code sur les processeurs hôtes. De plus, ce mode ajoute de la complexité au code qui doit maintenant avoir un support des transferts mémoires. Cette complexité peut bien entendu être masquée par un runtime à base de tâches qui s'occupera des transferts mémoires à notre place. Mais le travail nécessaire reste trop important par rapport aux gains obtenus.

### 4.4.4 Discussion

Les effets NUMA sont vraiment importants dans le cas d'une application limitée par la bande passante mémoire. La table 4.2 nous montre l'impact de la localité des données sur les performances d'un programme. Une mauvaise distribution des pages mémoires peut conduire à une sous-exploitation de la bande passante. La politique d'allocation interleaved limite ce problème, on est sûr que tous les liens mémoires sont utilisés, mais on n'a aucun contrôle sur l'amélioration de la localité des accès mémoire. Malgré cela, on obtient un gain important de performance dans certains cas tels que le produit matrice vecteur creux et la résolution triangulaire. Les politiques d'allocations du type next-touch et AutoNUMA résolvent une bonne partie du problème en améliorant la localité mémoire. Mais ces politiques ne nous permettent pas d'avoir un contrôle fin de l'accès aux données d'un thread.

La gestion des affinités NUMA directement dans l'ordonnanceur de tâches, nous permet de mieux répartir la charge mémoire. La localité mémoire en devient meilleure et une bonne distribution des tâches donne de très bonnes performances. L'utilisation d'un seul banc NUMA nous montre que l'ordonnanceur NATaS est moins bon que l'ordonnanceur Intel OpenMP. Sur un nombre important de bancs NUMA, NATaS ne passe pas à l'échelle. Cet ordonnanceur a été écrit spécifiquement pour des machines à 2 bancs NUMA. Les gains que nous observons avec l'utilisation de plusieurs bancs NUMA sont bien dus à une amélioration de la localité mémoire.

Malgré les bonnes performances que nous offre NATaS, on pourrait se demander s'il s'agit de la meilleure solution. En effet, le placement des tâches n'est pas entièrement optimal, de même que l'équilibrage de charge. Avec les algorithmes actuels, il est impossible de supprimer complètement les accès distants, nous ne pouvons que les limiter. Seule la solution utilisant un processus MPI par noeud NUMA permettrait de supprimer les accès distants. Mais cette suppression peut engendrer un algorithme moins efficace lié à la gestion des communications intra-noeud. Donc une meilleure solution pourrait être d'améliorer les ordonnanceurs existants en leur ajoutant une meilleure prise en charge des architectures NUMA.

| | | Cube 100 Npri 1 | Cube 100 Npri 3 | Cube 100 Npri 8 |
|---|---|---|---|---|
| SpMV | First Touch | 1.91 | 2.18 | 2.49 |
| | Interleave | 2.42 | 2.70 | 3.00 |
| | NATaS | 2.08 | 3.08 | 3.80 |
| | MPI | 2.89 | 3.29 | 3.81 |
| Facto | First Touch | 5.97 | 6.39 | 8.64 |
| | Interleave | 5.50 | 5.62 | 7.51 |
| | NATaS | 6.14 | 7.54 | 10.56 |
| | MPI | 6.37 | 7.58 | 9.92 |
| TRSV | First Touch | 1.69 | 2.12 | 2.71 |
| | Interleave | 1.84 | 2.18 | 2.83 |
| | NATaS | 1.82 | 2.49 | 3.48 |
| | MPI | 3.21 | 3.26 | 3.70 |

TABLE 4.2 – Accélérations obtenues sur Rostand avec 12 coeurs en fonction de la politique d'allocation mémoire. La version MPI ne fait pas exactement le même calcul, elle permet juste d'obtenir une indication sur l'accélération maximale que nous pouvons atteindre.

Sur une machine avec beaucoup de bancs NUMA, l'ordonnanceur NATaS ne passe pas à l'échelle. Sa structure interne n'est pas assez distribuée, l'utilisation de compteurs globaux de tâches est loin d'être idéal. Malgré une implémentation sous-optimale, NATaS offre de meilleures performances que les ordonnanceurs habituels. Il est donc essentiel d'optimiser les accès mémoire des applications limitées par la bande passante mémoire sur la machine NUMA. Le modèle de placement guidé des pages mémoires proposé par notre approche Taggre et NATaS offre de bonnes améliorations sur une application limitée par la bande passante mémoire. Par contre, l'extension à des architectures massivement multi-coeurs et NUMA demanderait des algorithmes de placement parallèles qui passent à l'échelle pour NATaS.

# CHAPITRE 5

# CONCLUSIONS ET PERSPECTIVES

**Sommaire**



## 5.1 Conclusion

La parallélisation des méthodes de résolution de grands systèmes linéaires creux est cruciale pour réduire les temps de calcul de nombreuses applications scientifiques. En particulier en simulation de réservoir, cette étape consomme plus de 80% du temps de calcul. Dans cette thèse, nous avons proposé des bibliothèques de parallélisation à grain fin pour les algorithmes élémentaires d'algèbre linéaire creuse qui sont utilisés dans les méthodes de résolution itératives. Notre but était de concevoir un cadre de programmation qui étend le modèle de programmation proposé par Intel TBB ou OpenMP à des algorithmes où le coût de calcul d'une tâche est faible et où la bande passante limite la performance. Nous avons conçu une approche transparente pour le développeur qui permet de prendre en compte des phénomènes complexes tels que la granularité des tâches où le placement des pages mémoires. Pour évaluer notre approche, nous nous sommes concentrés sur les algorithmes qui sont représentatifs des méthodes itératives. Ainsi, nous nous sommes donc focalisés à améliorer cette partie en commençant par la factorisation ILU, un des préconditionneurs du GMRES. La parallélisation de la factorisation ILU(k) s'exprime facilement sous la forme d'un graphe de tâches, mais la granularité des tâches ne permet pas d'obtenir de bonnes performances. La plupart des ordonnanceurs actuels mettent plus de temps à ordonnancer une tâche que la tâche ne met à exécuter son code. Pour pouvoir modifier facilement cette granularité, nous avons élaboré un nouveau cadriciel que nous avons appelé Taggre. Ce cadriciel prend en entrée un graphe de tâches à grain très fin et utilise des algorithmes pour grossir le grain de tâche en créant des groupes de tâches. Pour cela, il utilise des opérateurs d'agrégations que nous avons définis dans le chapitre 3. Chaque opérateur aura un objectif particulier d'optimisation du graphe. En combinant ces opérateurs, nous pouvons obtenir un nouveau graphe à grain grossier. Ce graphe pourra être ensuite utilisé par un ordonnanceur de tâches. Taggre a permis d'obtenir une parallélisation multi-threads efficace de la factorisation ILU(0) du GMRES préconditionné. Cela permettra, à puissance de calcul égale, de réduire le nombre de processus MPI utilisés en les remplaçant par des threads. Cette diminution a un impact à la fois sur la convergence de la méthode itérative ainsi que sur d'autres algorithmes pouvant eux aussi utiliser une parallélisation hybride pour obtenir de meilleures performances.

Cette méthode d'agrégation de tâches est générique. Dans le cadre de cette thèse, nous l'avons appliquée à la factorisation ILU(k) ainsi qu'aux résolutions triangulaires associées. Nous l'avons aussi expérimentée sur des graphes un peu plus généraux grâce à un simulateur d'exécution de tâches. Mais cette méthode pourrait aussi s'appliquer à d'autres noyaux d'algèbre linéaire creuse ainsi qu'à d'autres genres de problèmes représentant le parallélisme sous la forme d'un graphe de tâches.

Nous nous sommes ensuite concentrés sur l'optimisation des accès mémoire. L'architecture de type NUMA des machines que nous utilisons nous a conduit à créer un nouvel ordonnanceur de tâches prenant en compte la localité mémoire des tâches de calcul. Nos algorithmes étant limités par la bande passante mémoire, l'amélioration de la localité mémoire a conduit à une amélioration directe des performances. Malheureusement, cet ordonnanceur a été développé avec pour objectif de fonctionner sur des machines avec 2 bancs NUMA. L'utilisation d'une machine ayant plus de bancs NUMA a montré les limites de

cet ordonnanceur. Malgré ces limites, les résultats obtenus restent meilleurs que ceux obtenus avec un ordonnanceur classique. Les résultats de ces travaux ont été présentés au workshop PDSEC de la conférence IPDPS2013[RHAT13].

Actuellement, la performance du simulateur de réservoir dépend d'un solveur AMG qui est utilisé comme préconditionneur dans le CPR. La parallélisation du préconditionneur ILU n'a donc que très peu d'impact mais les méthodes utilisées pour sa parallélisation pourront être appliquées à d'autres noyaux d'algèbre linéaire creuse.

Durant cette thèse, nous avons aussi eu l'occasion d'essayer notre code de calcul sur un coprocesseur Intel Xeon Phi. La bande passante mémoire de ces coprocesseurs étant plus élevée que celle des processeurs que nous utilisons, nous avons obtenu de très bonnes performances sur les principaux noyaux de calculs du code. Par contre, l'utilisation du mode natif pour la partie séquentielle du code contrebalance les gains obtenus.

## 5.2 Perspectives

Taggre sera utilisé pour paralléliser efficacement de nouvelles routines d'algèbre linaire creuse dans le simulateur de réservoir. L'ordonnanceur de tâches que nous avons écrit peut aussi être amélioré. Il n'a qu'une seule fonctionnalité, la gestion de l'affinité NUMA. Des stratégies d'ordonnancement plus intelligentes pourraient être implémentées pour améliorer ses performances. De plus, les structures de données actuelles ne permettent pas de l'utiliser efficacement sur plus de 2 bancs NUMA.

Les architectures actuelles tendent à avoir de plus en plus de coeurs de calcul par processeur. Le Xeon Phi nous offre une vision de ces futurs processeurs et des moyens de programmation à notre disposition. Malheureusement la version actuelle ne propose pas de bonnes performances séquentielles. La prochaine version aura une exécution dans le désordre (out-of-order) et une meilleure fréquence d'horloge qui devraient résoudre le problème des parties de code séquentielles.

La solution Capsules[MRK08], présentée dans la section 3.2.2, permet d'utiliser une parallélisation par graphe de tâches en définissant plusieurs grains de calcul. Il pourrait être intéressant de coupler Taggre à Capsules. Taggre s'occupera de définir plusieurs granularités avec une nouvelle granularité après chaque application d'un opérateur. Capsules pourra ensuite utiliser ces différentes granularités pour ordonnancer efficacement le graphe de tâches.

Les opérateurs d'agrégations ont été créés pour résoudre les problèmes rencontrés en simulation de réservoir. Le choix des opérateurs se fait à l'appréciation du programmeur. Ce choix dépendra essentiellement de la forme du graphe de tâches ainsi que des propriétés des tâches. Parmi ces propriétés, la granularité des tâches est importante. Si la granularité des tâches est vraiment trop fine, il faudra privilégier des opérateurs qui créent de gros groupes de tâches. Par contre pour une granularité moins fine, nous pouvons utiliser des opérateurs qui favorisent les interactions entre les tâches. Nous proposons deux solutions pour rendre ce choix automatique avec chacun des avantages et des défauts.

La première solution est *l'auto-tuning*, elle consiste à parcourir l'espace des possibilités d'agrégations et à choisir la meilleure. Le principal problème vient de la taille de l'espace à parcourir. Pour réduire cette taille, nous pouvons utiliser des techniques d'optimisation pour éviter de tester des cas trop absurdes. Nous pouvons aussi utiliser le simulateur d'exécution des tâches pour avoir une approximation du temps de calcul. Mais cette approximation n'est pas aussi fiable qu'une vraie exécution car nous ne prenons pas en compte tous les paramètres.

La deuxième solution consiste à faire une analyse statique des tâches ainsi que du graphe de tâches. Cette analyse est très complexe à mettre en oeuvre et les critères d'évaluation des graphes sont à définir. Par contre, une fois tout le travail d'analyse effectué, le choix des opérateurs est rapide.

Le problème de définition de la fonction de tri de l'opérateur généralisé se pose toujours. Deux solutions peuvent être développées :

— la création de greffons pour Taggre permettant d'écrire du code C++ et de l'intégrer directement ;
— la création d'un langage permettant de décrire les tâches qui sera interprété à l'exécution.

La première méthode a l'avantage d'être simple à mettre en oeuvre, mais elle peut être difficile à utiliser. La deuxième méthode est plus complète mais elle a l'inconvénient de demander beaucoup de travail au niveau de l'implémentation.

# BIBLIOGRAPHIE


[ABC+06] Krste Asanovic, Ras Bodik, Bryan Christopher Catanzaro, Joseph James Gebis, Parry Husbands, Kurt Keutzer, David A Patterson, William Lester Plishker, John Shalf, Samuel Webb Williams, et al. The landscape of parallel computing research : A view from berkeley. Technical report, UCB/EECS-2006-183, EECS Department, University of California, Berkeley, 2006.

[ABC+14] Emmanuel Agullo, Bérenger Bramas, Olivier Coulaud, Eric Darve, Matthias Messner, and Toru Takahashi. Task-based FMM for heterogeneous architectures. Research Report RR-8513, April 2014.

[ABD+90] E. Angerson, Z. Bai, J. Dongarra, A. Greenbaum, A. McKenney, J. Du Croz, S. Hammarling, J. Demmel, C. Bischof, and D. Sorensen. Lapack : A portable linear algebra library for high-performance computers. In *Supercomputing '90., Proceedings of*, pages 2–11, Nov 1990.

[ABGL13] Emmanuel Agullo, Alfredo Buttari, Abdou Guermouche, and Florent Lopez. Multifrontal QR factorization for multicore architectures over runtime systems. In Felix Wolf, Bernd Mohr, and Dieter Mey, editors, *Euro-Par 2013 Parallel Processing*, volume 8097 of *Lecture Notes in Computer Science*, pages 521–532. Springer Berlin Heidelberg, 2013.

[ACD+09] Eduard Ayguade, Nawal Copty, Alejandro Duran, Jay Hoeflinger, Yuan Lin, Federico Massaioli, Xavier Teruel, Priya Unnikrishnan, and Guansong Zhang. The design of OpenMP tasks. *IEEE Transactions on Parallel and Distributed Systems*, 20(3) :404–418, 2009.

[ARvW03] Cliff Addison, Yuhe Ren, and M. van Waveren. OpenMP issues arising in the development of parallel BLAS and LAPACK libraries. *Scientific Programming*, 11(2) :95–104, 2003.

[ATNW11] Cédric Augonnet, Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. StarPU : A unified platform for task scheduling on heterogeneous multicore architectures. *Concurrency and Computation : Practice and Experience, Special Issue : Euro-Par 2009*, 23(2) :187–198, Feb. 2011.

[BBAC14] L. Boillot, G. Bosilca, E. Agullo, and H. Calandra. Task-based programming for seismic imaging : Preliminary results. In *Proceedings of the 16th IEEE International Conference on High Performance Computing and Communications (HPCC)*, pages 1259–1266, 2014.

[BBD+11] G. Bosilca, A. Bouteiller, A. Danalis, M. Faverge, H. Haidar, T. Herault, J. Kurzak, J. Langou, P. Lemarinier, H. Ltaief, P. Luszczek, A. YarKhan, and J. Dongarra. Flexible development of dense linear algebra algorithms on massively parallel architectures with DPLASMA. In *Proceedings of the 25th IEEE International Symposium on Parallel & Distributed Processing Workshops and Phd Forum (IPDPSW'11), PDSEC 2011*, pages 1432–1441, Anchorage, United States, mai 2011.

[BBD+13] George Bosilca, Aurélien Bouteiller, Anthony Danalis, Mathieu Faverge, Thomas Hérault, and Jack Dongarra. PaRSEC : A programming paradigm exploiting heterogeneity for enhancing scalability. *Computing in Science and Engineering*, 15(6) :36–45, November 2013.

[BCC+00] John Bircsak, Peter Craig, RaeLyn Crowell, Zarka Cvetanovic, Jonathan Harris, C. Alexander Nelson, and Carl D. Offner. Extending OpenMP for NUMA machines. In *Supercomputing, ACM/IEEE 2000 Conference*, pages 48–48. IEEE, 2000.

[BG04] Petter Bjorstad and William Gropp. *Domain decomposition : parallel multilevel methods for elliptic partial differential equations*. Cambridge university press, 2004.

[BJK+95] Robert D. Blumofe, Christopher F. Joerg, Bradley C. Kuszmaul, Charles E. Leiserson, Keith H. Randall, and Yuli Zhou. Cilk : an efficient multithreaded runtime system. In *Proceedings of the fifth ACM SIGPLAN symposium on Principles and practice of parallel programming*, PPOPP '95, pages 207–216, New York, NY, USA, 1995. ACM.

[BMD+11] Javier Bueno, Luis Martinell, Alejandro Duran, Montse Farreras, Xavier Martorell, Rosa M. Badia, Eduard Ayguade, and Jesús Labarta. Productive cluster programming with OmpSs. In *Proceedings of the 17th international conference on Parallel processing - Volume Part I*, Euro-Par '11, pages 555–566, Berlin, Heidelberg, 2011. Springer-Verlag.

[BPR+97] Ronald F. Boisvert, Roldan Pozo, Karin Remington, Richard F. Barrett, and Jack J. Dongarra. Matrix Market : A web resource for test matrix collections. In *Proceedings of the IFIP TC2/WG2.5 Working Conference on Quality of Numerical Software : Assessment and Enhancement*, pages 125–137, London, UK, UK, 1997. Chapman & Hall, Ltd.

[Bro10] François Broquedis. *De l'exécution d'applications scientifiques OpenMP sur architectures hiérarchiques*. PhD thesis, Université Bordeaux 1, December 2010.

[CB01] M.A. Christie and M.J. Blunt. Tenth SPE comparative solution project : A comparison of upscaling techniques. *SPE Reservoir Evaluation & Engineering*, 4(04) :308–317, 2001.

[CP08] Cédric Chevalier and François Pellegrini. PT-Scotch : A tool for efficient parallel graph ordering. *Parallel Computing*, 34(6) :318–331, 2008.

[CP15] Edmond Chow and Aftab Patel. Fine-grained parallel Incomplete LU factorization. *SIAM Journal on Scientific Computing*, 37(2) :C169–C193, 2015.

[DBB07] Romain Dolbeau, Stéphane Bihan, and François Bodin. HMPP : A hybrid multi-core parallel programming environment. In *Workshop on General Purpose Processing on Graphics Processing Units (GPGPU 2007)*, 2007.

[DFLL11] Jack J. Dongarra, Mathieu Faverge, Hatem Ltaief, and Piotr Luszczek. Achieving numerical accuracy and high performance using recursive tile LU factorization. Research report, 2011.

[DM89] Iain S. Duff and Gerard A. Meurant. The effect of ordering on preconditioned conjugate gradients. *BIT Numerical Mathematics*, 29(4) :635–657, 1989.

[Doi91] S. Doi. On parallelism and convergence of Incomplete LU factorizations. *Appl. Numer. Math.*, 7(5) :417–436, June 1991.

[GF09] B. Goglin and N. Furmento. Enabling high-performance memory migration for multithreaded applications on linux. In *IEEE International Symposium on Parallel Distributed Processing, 2009*, IPDPS '09, pages 1–9. IEEE Computer Society, May 2009.

[GLFR12] Thierry Gautier, Fabien Lementec, Vincent Faucher, and Bruno Raffin. X-Kaapi : a multi paradigm runtime for multicore architectures. Technical Report RR-8058, Feb. 2012.

[Gro12] SGI Group. *SGI NUMAlink, Industry Leading Interconnect Technology*, Sept. 2012.

[GY92] Apostolos Gerasoulis and Tao Yang. A comparison of clustering heuristics for scheduling directed acyclic graphs on multiprocessors. *Journal of Parallel and Distributed Computing*, 16(4) :276–291, 1992.

[int09] *Intel Math Kernel Library. Reference Manual*. Intel Corporation, Santa Clara, USA, 2009. ISBN 630813-054US.

[KA99] Yu-Kwong Kwok and Ishfaq Ahmad. Static scheduling algorithms for allocating directed task graphs to multiprocessors. *ACM Computing Surveys (CSUR)*, 31(4) :406–471, 1999.

[Kar03] George Karypis. Multi-constraint mesh partitioning for contact/impact computations. In *Proceedings of the 2003 ACM/IEEE conference on Supercomputing*, page 56. ACM, 2003.

[KK98] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on scientific Computing*, 20(1) :359–392, 1998.

[Kle05] Andi Kleen. A NUMA API for linux. *Novel Inc*, 2005.

[KMJ94] A. A. Khan, C. L. McCreary, and M. S. Jones. A comparison of multiprocessor scheduling heuristics. In *Proceedings of the 1994 International Conference on Parallel Processing, volume II*, pages 243–250, 1994.

[LBB10] Stefan Lankes, Boris Bierbaum, and Thomas Bemmerl. Affinity-on-next-touch : An extension to the linux kernel for NUMA architectures. In Roman Wyrzykowski, Jack Dongarra, Konrad Karczewski, and Jerzy Wasniewski, editors, *Parallel Processing and Applied Mathematics*, volume 6067 of *Lecture Notes in Computer Science*, pages 576–585. Springer Berlin Heidelberg, 2010.

[Lei09] Charles E. Leiserson. The Cilk++ concurrency platform. *The Journal of Supercomputing*, 51 :522–527, 2009.

[LH05] Henrik Löf and Sverker Holmgren. affinity-on-next-touch : increasing the performance of an industrial PDE solver on a cc-NUMA system. In *Proceedings of the 19th annual international conference on Supercomputing*, ICS '05, pages 387–392, New York, NY, USA, 2005. ACM.

[LHKK79] C. L. Lawson, R. J. Hanson, D. R. Kincaid, and F. T. Krogh. Basic linear algebra subprograms for fortran usage. *ACM Trans. Math. Softw.*, 5(3) :308–323, September 1979.

[LRT79] Richard J Lipton, Donald J Rose, and Robert Endre Tarjan. Generalized nested dissection. *SIAM journal on numerical analysis*, 16(2) :346–358, 1979.

[LSAT13] B. Lize, G. Sylvand, E. Agullo, and S. Thibault. A task-based H-matrix solver for acoustic and electromagnetic problems on multicore architectures. In *SciCADE, the International Conference on Scientific Computation and Differential Equations*, 2013.

[LY14] Hatem Ltaief and Rio Yokota. Data-driven execution of fast multipole methods. *Concurrency and Computation : Practice and Experience*, 26(11) :1935–1946, 2014.

[LZSQ09] Shengfei Liu, Yunquan Zhang, Xiangzheng Sun, and RongRong Qiu. Performance evaluation of multithreaded sparse matrix-vector multiplication using OpenMP. In *High Performance Computing and Communications, 2009. HPCC'09. 11th IEEE International Conference on*, pages 659–665. IEEE, 2009.

[McC07] John D. McCalpin. STREAM : Sustainable memory bandwidth in high performance computers. Technical report, University of Virginia, Charlottesville, Virginia, 1991-2007. A continually updated technical report. http ://www.cs.virginia.edu/stream/.

[MRK08] HasnainA. Mandviwala, Umakishore Ramachandran, and Kathleen Knobe. Capsules : Expressing composable computations in a parallel programming model. In Vikram Adve, MaríaJesús Garzarán, and Paul Petersen, editors, *Languages and Compilers for Parallel Computing*, volume 5234, chapter Lecture Notes in Computer Science, pages 276–291. Springer Berlin Heidelberg, 2008.

[MVM09] Frederic P. Miller, Agnes F. Vandome, and John McBrewster. *Advanced Configuration and Power Interface : Open Standard, Operating System, Power Management, Cross- Platform, Intel Corporation, Microsoft, Toshiba, ... Sleep Mode, Hibernate (OS Feature), Synonym*. Alpha Press, 2009.

[nVi12] nVidia. *CUBLAS Library User Guide*. nVidia, v5.0 edition, October 2012.

[Ope08] OpenMP Architecture Review Board. OpenMP application program interface version 3.0, May 2008.

[Ope13] OpenACC-Standard. The OpenACC application programming interface, May 2013.

[PBF10] Artur Podobas, Mats Brorsson, and Karl-Filip Faxén. A comparison of some recent task-based parallel programming models. In *3rd Workshop on Programmability Issues for Multi-Core Computers*, Pisa, Italy, 2010.

[PR97] François Pellegrini and Jean Roman. Sparse matrix ordering with scotch. In Bob Hertzberger and Peter Sloot, editors, *High-Performance Computing and Networking*, volume 1225 of *Lecture Notes in Computer Science*, pages 370–378. Springer Berlin Heidelberg, 1997.

[PRCMC11] Christiane Pousa Ribeiro, Márcio Castro, Jean-François Méhaut, and Alexandre Carissimi. Improving memory affinity of geophysics applications on NUMA platforms using minas. In JoséM.LaginhaM. Palma, Michel Daydé, Osni Marques, and JoãoCorreia Lopes, editors, *High Performance Computing for Computational Science – VECPAR 2010*, volume 6449 of *Lecture Notes in Computer Science*, pages 279–292. Springer Berlin Heidelberg, 2011.

[Rah13] Rezaur Rahman. *Intel® Xeon Phi™ Coprocessor Architecture and Tools : The Guide for Application Developers*. Apress, 2013.

[Rei07] James Reinders. *Intel threading building blocks*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, first edition, 2007.

[RHAT13] Corentin Rossignon, Pascal Henon, Olivier Aumage, and Samuel Thibault. A NUMA-aware fine grain parallelization framework for multi-core architecture. In *Parallel and Distributed Processing Symposium Workshops & PhD Forum (IPDPSW), 2013 IEEE 27th International*, pages 1381–1390. IEEE, 2013.

[RHJ09] Rolf Rabenseifner, Georg Hager, and Gabriele Jost. Hybrid MPI/OpenMP parallel programming on clusters of multi-core smp nodes. In *Parallel, Distributed and Network-based Processing, 2009 17th Euromicro International Conference on*, pages 427–436. IEEE, 2009.

[RMC+09] Christiane Pousa Ribeiro, J Mehaut, Alexandre Carissimi, Marcio Castro, and Luiz Gustavo Fernandes. Memory affinity for hierarchical shared memory multiprocessors. In *Computer Architecture and High Performance Computing, 2009. SBAC-PAD'09. 21st International Symposium on*, pages 59–66. IEEE, 2009.

[Ros13] Corentin Rossignon. Optimisation du produit matrice-vecteur creux sur architecture GPU pour un simulateur de réservoir. In *21èmes Rencontres Francophones du Parallélisme (RenPar'21)*, Grenoble, France, January 2013.

[Saa94] Yousef Saad. ILUT : A dual threshold Incomplete LU factorization. *Numerical linear algebra with applications*, 1(4) :387–402, 1994.

[Saa96] Yousef Saad. *Iterative Methods for Sparse Linear Systems*. PWS, Boston, MA, USA, Jul. 1996.

[SP99] João Luís Sobral and Alberto José Proença. Dynamic grain-size adaptation on object oriented parallel programming the SCOOPP approach. In *Proceedings of the 13th International Symposium on Parallel Processing and the 10th Symposium on Parallel and Distributed Processing*, IPPS '99/SPDP '99, pages 728–732, Washington, DC, USA, 1999. IEEE Computer Society.

[Thi05] Samuel Thibault. A flexible thread scheduler for hierarchical multiprocessor machines. In *Second International Workshop on Operating Systems, Programming Environments and Management Tools for High-Performance Computing on Clusters (COSET-2)*, Cambridge, USA, June 2005.

[THW99] Haluk Topcuoglu, Salim Hariri, and Min-You Wu. Task scheduling algorithms for heterogeneous processors. In *Proceedings of the Eighth Heterogeneous Computing Workshop*, HCW '99, page 3, Washington, DC, USA, 1999. IEEE Computer Society.

[THW02] Haluk Topcuoglu, Salim Hariri, and Min-you Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Parallel and Distributed Systems, IEEE Transactions on*, 13(3) :260–274, 2002.

[TNW07] Samuel Thibault, Raymond Namyst, and Pierre-André Wacrenier. Building portable thread schedulers for hierarchical multiprocessors : The BubbleSched framework. In *Euro-Par 2007 Parallel Processing*, volume 4641, pages 42–51. Springer, 2007.

[VTN11] H. Vandierendonck, G. Tzenakis, and D. S. Nikolopoulos. A unified scheduler for recursive and task dataflow parallelism. In *Parallel Architectures and Compilation Techniques*, PACT '11, pages 1–11, Oct. 2011.

[WD98] R. Clint Whaley and Jack J. Dongarra. Automatically tuned linear algebra software. In *Proceedings of the 1998 ACM/IEEE Conference on Supercomputing*, SC '98, pages 1–27, Washington, DC, USA, 1998. IEEE Computer Society.

[WKL+85] JR Wallis, RP Kendall, TE Little, et al. Constrained residual acceleration of conjugate residual methods. In *SPE Reservoir Simulation Symposium*. Society of Petroleum Engineers, 1985.

[WWP09] Samuel Williams, Andrew Waterman, and David Patterson. Roofline : An insightful visual performance model for multicore architectures. *Commun. ACM*, 52(4) :65–76, April 2009.