Algèbre linéaire creuse pour le logarithme discret

Ce chapitre présente le problème d'algèbre linéaire qui résulte des calculs de logarithme discret. L'algèbre linéaire considérée ici correspond à la résolution de systèmes linéaires creux sur des corps finis, c'est-à-dire des calculs exacts. De nombreux aspects des méthodes issues des calculs numériques ne sont pas applicables dans ce contexte. Le point principal de divergence des ces deux types d'algèbre linéaire est que dans le contexte numérique les méthodes utilisées tendent à converger vers la solution recherchée, alors que dans le cas exact, sur un corps fini, il n'est pas possible d'approcher ou de converger vers la solution.
Une partie importante de ce chapitre a fait l'objet de la publication [Jel14b].

Sommaire

3.1 Présentation du problème

Les systèmes d'équations linéaires que nous allons étudier ici sont issus des algorithmes de calcul d'index. Pour une description plus détaillée de ces algorithmes, le lecteur peut se référer au chapitre 1. Dans ces algorithmes, l'étape de *crible* fournit des relations entre les logarithmes de certains éléments. Par la suite, l'étape de *filtrage* en quelque sorte « nettoie » et « prépare » le système à résoudre. Nous avons besoin de résoudre ce système pour pouvoir calculer dans l'étape de *logarithme individuel* le logarithme discret de n'importe quel élément du corps, à partir des logarithmes déjà connus.

Nous calculons le logarithme discret dans un sous-groupe multiplicatif d'un corps fini $\mathbb{F}_q$. L'ordre de ce sous-groupe est premier et est noté ℓ. On rappelle que ℓ est un diviseur de $q-1$ (voir sous-section 1.4.5). Les entrées de la phase d'algèbre linéaire sont :

— le nombre premier ℓ ;
— une matrice carrée et singulière A, à N lignes et colonnes et définie dans le corps fini $(\mathbb{Z}/\ell\mathbb{Z})$.

Nous cherchons un élément non trivial du noyau de la matrice, autrement dit une solution de l'équation

$$Aw \bmod \ell = 0 \tag{3.1}$$

Nous allons considérer des matrices issues des calculs avec les algorithmes FFS et NFS. Les propriétés que nous allons décrire peuvent ne pas être valides pour des matrices issues d'autres algorithmes de calcul d'index.

Fabrication de la matrice par l'étape de filtrage

L'étape de filtrage est une étape de pré-calcul qui permet de réduire la taille de la matrice. La description que nous fournissons du filtrage est basée sur les travaux [Cav02, chap. 3] et [Bou13].

À l'entrée du filtrage, la matrice est très grande et très creuse (environ 20 coefficients non nuls par ligne). La matrice n'est généralement pas carrée ; elle contient plus de lignes que de colonnes (c'est-à-dire plus d'équations que d'inconnues). L'objectif du filtrage est de diminuer la taille de la matrice sans trop la densifier ; la densité finale correspond à quelques centaines de coefficients non nuls par ligne. On désire généralement pour la suite de l'algèbre linéaire une matrice carrée.

Cette étape de pré-calcul tend à augmenter le nombre moyen de coefficients non nuls par ligne, mais permet de diminuer la taille de la matrice. La diminution de la taille est importante pour la résolution effective du système linéaire. D'une part, elle réduit la quantité de mémoire nécessaire à la représentation et au traitement de la matrice. D'autre part, elle diminue le coût de la résolution effective, sachant que les algorithmes de résolution effective, comme nous allons le voir dans la prochaine section, ont une complexité au moins quadratique, si ce n'est cubique, en la taille de la matrice. Au fur et à mesure que le filtrage agit sur la matrice, le coût estimé de la résolution effective diminue. Le filtrage est arrêté lorsque le coût recommence à augmenter.

Il existe une autre méthode pour diminuer la taille de la matrice, qui est celle de l'*élimination gaussienne structurée* (SGE pour Structured Gaussian Elimination) [LO90, PS92]. Les approches SGE et celle du filtrage sont similaires ; elles se distinguent de par la stratégie et les critères qu'elles emploient pour le choix des lignes à combiner ou à supprimer.

Dans la table 3.1, nous illustrons l'évolution de la taille de la matrice pendant le filtrage avec deux exemples de matrices issues de calculs concrets de logarithme discret ; une matrice FFS

qui correspond à la résolution du logarithme discret dans $\mathbb{F}_{2^{809}}$ et une matrice NFS issue de la résolution du logarithme discret dans un corps premier $\mathbb{F}_{p_{155}}$, où p_{155} est un nombre premier de 155 chiffres décimaux.

Calcul	FFS pour $\mathbb{F}_{2^{809}}$	NFS pour $\mathbb{F}_{p_{155}}$
#relations uniques / #inconnues	80 963 931 / 39 357 159	26 835 094 / 14 891 504
Poids moyen des lignes avant le filtrage	21.8	24.2
Taille de la matrice après le filtrage (N)	3 602 667	2 561 574
Poids moyen des lignes après le filtrage	100	105

TABLE 3.1 – Évolution de la matrice pendant le filtrage avec les exemples dans $\mathbb{F}_{2^{809}}$ et $\mathbb{F}_{p_{155}}$.

3.2 Caractéristiques des entrées

Le nombre ℓ est un « grand » nombre premier, de l'ordre de quelques centaines de bits. D'un point de vue cryptographique, le nombre ℓ doit être suffisamment grand de sorte à ce que le sous-groupe correspondant résiste à des attaques de type Pollard rho (voir sous-section 1.4.1 pour de plus amples explications).

La matrice A est grande. Sa dimension N peut aller de centaines de milliers à des dizaines de millions de lignes, si on se réfère aux calculs récents de logarithme discret.

La matrice est creuse. Une analyse asymptotique donne une densité indicative de $O\left(\log^2 N\right)$ coefficients par ligne. En pratique, pour les problèmes étudiés, cette densité est de quelques centaines de coefficients par ligne.

La table 3.2 donne les caractéristiques des entrées de l'algèbre linéaire correspondant aux exemples précédents.

Calcul	FFS pour $\mathbb{F}_{2^{809}}$	NFS pour $\mathbb{F}_{p_{155}}$
Taille de ℓ (bits)	202	511
Taille de la matrice (N)	3 602 667	2 561 574
Nombre de coefficients non nuls	360 266 822	268 965 377
Poids moyen des lignes	100	105

TABLE 3.2 – Caractéristiques des matrices utilisées pour les exemples dans $\mathbb{F}_{2^{809}}$ et $\mathbb{F}_{p_{155}}$.

3.2.1 Nature des coefficients

La matrice A est définie sur le corps $(\mathbb{Z}/\ell\mathbb{Z})$. Toutefois, la majorité des coefficients sont « petits », dans le sens où ils peuvent être représentés par un entier de petite taille, tenant dans un mot machine (éventuellement signé). Ces coefficients correspondent à des exposants dans les relations trouvées dans la phase de *crible*.

Pour les matrices issues d'un calcul FFS, tous les coefficients sont « petits ». Les courbes de la figure 3.1 montrent la répartition des valeurs des coefficients dans la matrice-exemple de FFS pour $\mathbb{F}_{2^{809}}$. En abscisse, on représente les valeurs des coefficients et en ordonnée le nombre d'apparition dans la matrice.

Tous les coefficients sont compris entre -35 et 36. En échelle linéaire, il y a deux grands pics qui correspondent aux valeurs -1 et 1 et deux autres pics moins importants pour les valeurs -2 et 2. En effet, 92.7% des coefficients sont des ± 1 et 4.5% des coefficients sont des ± 2. Quand

nous passons en échelle logarithmique, nous observons une répartition quasi triangulaire que nous n'expliquons pas ici ; l'explication pourrait être trouvée avec une étude statistique de l'étape de filtrage.

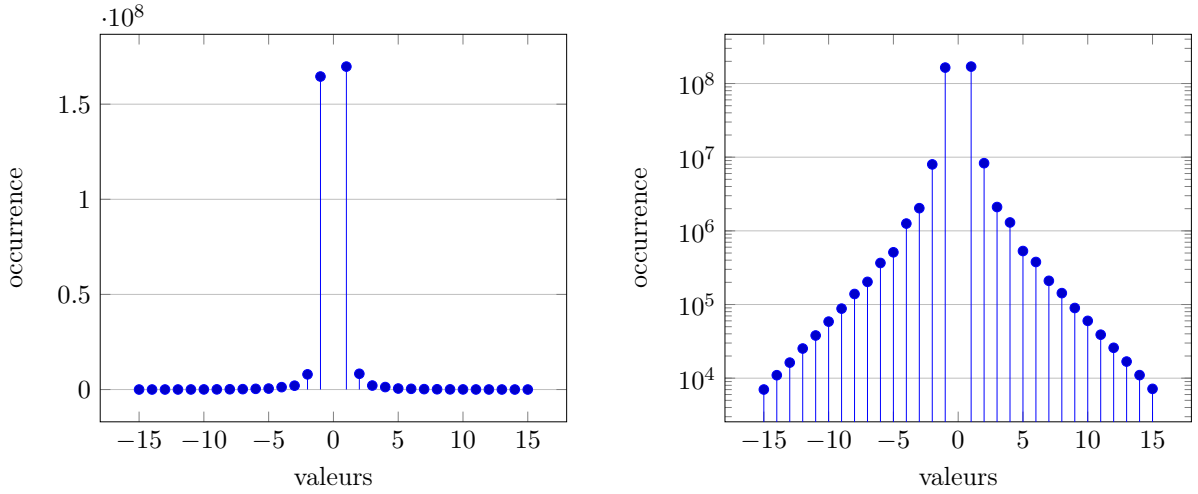


FIGURE 3.1 – Répartition des valeurs des coefficients dans la matrice-exemple de FFS pour $\mathbb{F}_{2^{809}}$ en échelle linéaire (à gauche) et en échelle logarithmique (à droite).

Pour les matrices issues d'un calcul NFS, la majorité des coefficients sont « petits ». Toutefois, en plus des coefficients petits, nous avons des colonnes particulières (dites *colonnes de caractères*). Ces colonnes sont denses et contiennent des éléments qui sont « grands », c'est-à-dire leur représentant dans $[0, \ell[$ occupe autant de mots machine que ℓ (voir sous-sous-section 1.4.4). Le nombre de ces colonnes ne dépasse généralement pas la dizaine ; en l'occurrence pour la matrice-exemple de NFS pour $\mathbb{F}_{p_{155}}$, il y a 5 *colonnes de caractères*. Dans cette matrice, 95% des coefficients sont « petits ».

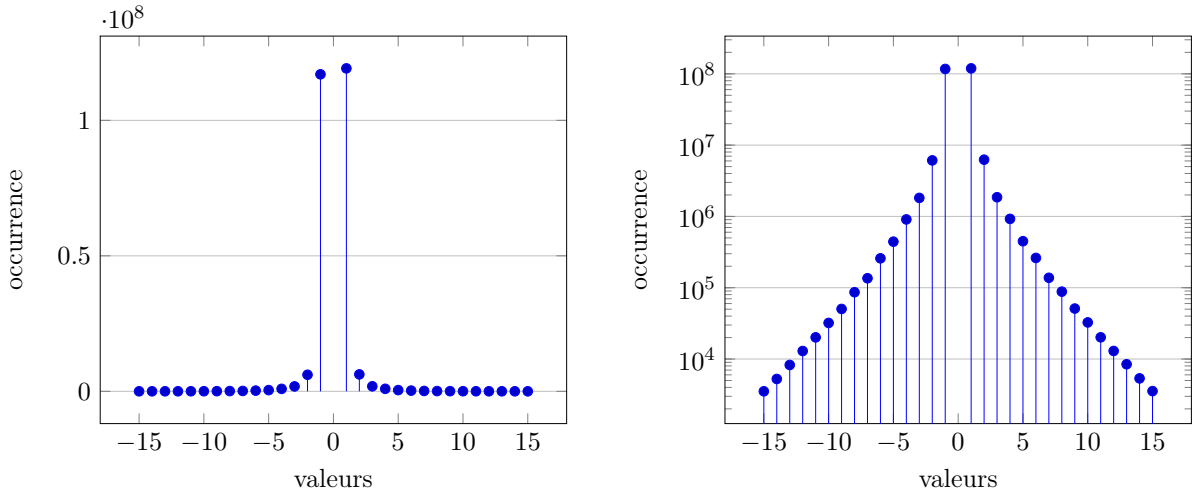


FIGURE 3.2 – Répartition des valeurs des coefficients « petits » dans la matrice-exemple de NFS pour $\mathbb{F}_{p_{155}}$ en échelle linéaire (à gauche) et en échelle logarithmique (à droite).

Les courbes de la figure 3.2 montrent la répartition des valeurs des coefficients « petits ».

Tous les coefficients sont compris entre -35 et 35. De même que pour la matrice-exemple pour FFS, nous observons les deux pics correspondant aux valeurs -1 et 1 ainsi que la répartition quasi triangulaire des valeurs.

3.2.2 Distribution des coefficients non nuls

La distribution des coefficients non nuls de la matrice creuse n'est pas régulière. Toutefois, les coefficients non nuls sont plus ou moins localisés dans certaines parties de la matrice. Les figures 3.3 et 3.4 montrent les distributions typiques des matrices issues des algorithmes FFS et NFS. Ces figures ont été obtenues en calculant les densités de blocs contenant quelques centaines de lignes et de colonnes (800 pour la figure 3.3 et 600 pour la figure 3.4). Un bloc est ensuite visualisé par un point.

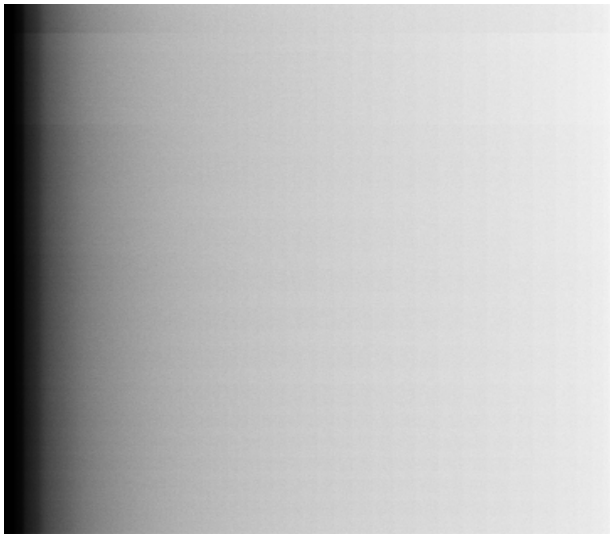


FIGURE 3.3 – Distribution des coefficients non nuls dans la matrice-exemple de FFS pour $\mathbb{F}_{2^{809}}$.

Dans la matrice-exemple issue de FFS, les premières colonnes de la matrice sont relativement denses, ensuite la densité des colonnes diminue progressivement. On note l'évolution de la densité des colonnes comme suit :

Indices des colonnes	Densité des colonnes	Contribution des colonnes au nombre total de coefficients non nuls de la matrice
0-76	>10%	22.5%
77-475	1-10%	10.6%
476-4948	0.1-1%	13.4%
4949-68580	0.01-0.1%	17.6%
68581-3602666	<0.01%	35.9%

La densité des lignes ne change pas considérablement.

FIGURE 3.4 – Distribution des coefficients non nuls dans la matrice-exemple NFS pour $\mathbb{F}_{p_{155}}$.

Comme avec la matrice FFS, la densité des colonnes de la matrice NFS diminue progressivement, sauf pour les *colonnes de caractères*, qui sont denses (ces colonnes sont représentées dans la figure 3.4 par un trait continu fin à l'extrême droite de la matrice). La densité des colonnes évolue comme suit :

Indices des colonnes	Densité des colonnes	Contribution des colonnes au nombre total de coefficients non nuls de la matrice
0-76	>10%	22.2%
77-602	1-10%	12.7%
603-6122	0.1-1%	13%
6123-75183	0.01-0.1%	16.2%
75183-2561568	<0.01%	31.1%
2561568-2561573	100%	4.8%

3.3 Du parallélisme pour résoudre le problème

Résoudre ce problème d'algèbre linéaire nécessite, comme nous allons le détailler dans ce chapitre, des calculs lourds. En effet, les algorithmes de résolution de tels problèmes ont une complexité au moins quadratique en la dimension de la matrice N. Rappelons que, pour résoudre des logarithmes discrets dans des groupes de plus en plus grands, les matrices sont de plus en plus grandes. Par ailleurs, ces grandes matrices posent des problèmes de stockage, avec des tailles de données qui peuvent atteindre quelques dizaines de gigaoctets. Par conséquent, il est important de pouvoir *paralléliser* les calculs d'algèbre linéaire, pour que ce calcul puisse être distribué en plusieurs sous-calculs, où chaque sous-calcul est exécuté sur un nœud de calcul. Une distribution du calcul initial sur plusieurs unités calculatoires diminue le temps d'exécution et les exigences en terme de puissance de calcul et de mémoire par rapport à un calcul qui aurait été fait uniquement sur une seule unité calculatoire.

Il existe quatre niveaux de parallélisme pour la résolution de systèmes linéaires creux sur les corps finis :

1. Le niveau algorithmique : les algorithmes de résolution dits « par blocs » permettent de distribuer le calcul de l'algèbre linéaire en plusieurs calculs indépendants. Ce parallélisme est détaillé dans la section 3.5, en particulier celui de l'algorithme Wiedemann par blocs (voir les sous-sections 3.5.3 et 3.5.4).
2. Le niveau du produit matrice-vecteur sur plusieurs nœuds : l'opération centrale dans les algorithmes de résolution est un produit de la matrice creuse par un vecteur. Le chapitre 4 explore comment distribuer cette opération sur plusieurs nœuds de calcul, où chaque nœud effectue un produit partiel.
3. Le niveau du produit partiel : si le produit partiel est effectué sur une architecture hautement parallèle, par exemple un processeur graphique, plusieurs *threads* vont collaborer pour calculer le produit partiel. Le chapitre 5 discute la stratégie de parallélisation de ce produit.
4. Le niveau de l'arithmétique sur les corps finis : une opération arithmétique sur un grand corps fini, dont un élément occupe plusieurs mots machine, peut être parallélisée. En l'occurrence, le système de représentation RNS (Residue Number System) permet de distribuer une opération arithmétique en plusieurs « petites » opérations indépendantes. Le chapitre 6 détaille cet aspect.

3.4 Solveurs directs et itératifs

Pour résoudre des systèmes d'équations linéaires, deux familles d'algorithmes peuvent être utilisées :

— les méthodes directes ;
— les méthodes itératives.

Nous allons étudier l'efficacité de ces méthodes dans notre contexte. Parmi ces méthodes, certaines ne s'adaptent pas forcément très bien aux systèmes définis sur des grands corps finis.

3.4.1 Les méthodes directes

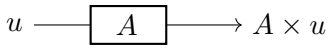
Ces méthodes sont des algorithmes classiques pour l'algèbre linéaire numérique, sur des nombres réels. Parmi ces méthodes, nous trouvons l'élimination gaussienne et les décompositions Cholesky, LU et QR [FM67, Wes68, PTV+92].

Les méthodes directes requièrent $O\left(N^{\omega}\right)$ opérations dans le corps, où l'exposant ω est égal à 3 dans le cas du produit matriciel naïf, et à 2.807 si l'algorithme de Strassen [Str69] est utilisé. Dans notre contexte, la matrice correspondant au système linéaire est creuse, mais appliquer les méthodes directes tendra à la densifier. Ceci augmente la quantité de mémoire nécessaire pour représenter la matrice. En effet, la complexité en espace mémoire des méthodes directes atteint $O\left(N^2\right)$, car la matrice devient dense. Ainsi, en employant une méthode directe, la mémoire nécessaire pour traiter la matrice de FFS pour $\mathbb{F}_{2^{809}}$ serait passée de 3 Go à 13 To.

3.4.2 Les méthodes itératives

Parmi les méthodes itératives, il y a la méthode du gradient conjugué [HS52] et les algorithmes de Lanczos [Lan52] et de Wiedemann [Wie86].

Les méthodes itératives sont des méthodes dîtes *black-box*, dans le sens où elles n'agissent pas sur la matrice ; elles utilisent la matrice uniquement pour effectuer une opération de produit matrice-vecteur.



Les méthodes nécessitent $O(N)$ produits matrice-creuse–vecteur. Leur complexité en espace mémoire est en $O(N\gamma)$, où γ désigne le nombre moyen de coefficients non nuls par ligne.

Si un produit matrice-creuse–vecteur est effectué en moins de $O\left(N^{\omega-1}\right)$ opérations dans le corps, les méthodes itératives ont une meilleure complexité asymptotique. Cette condition est réalisable, vu que les matrices issues du calcul de logarithme discret contiennent un nombre petit de coefficients non nuls. Le nombre moyen γ de coefficients non nuls par ligne est négligeable devant la dimension N. Ainsi, la complexité du produit matrice-creuse–vecteur, qui est égale à $O(N\gamma)$, est meilleure que $O\left(N^{\omega-1}\right)$, lorsque $\gamma < N^{\omega-2}$. Par conséquent, les méthodes itératives sont asymptotiquement plus rapides que les méthodes directes pour notre contexte.

Nous pouvons conclure que pour résoudre les systèmes linéaires issus de calcul de logarithme discret, les arguments de la complexité en mémoire et de la complexité en temps sont en faveur des méthodes itératives par rapport aux méthodes directes.

3.5 Algorithmes itératifs de résolution d'algèbre linéaire

3.5.1 Algorithme de Lanczos

L'algorithme de Lanczos [Lan52] a été inventé pour résoudre des systèmes linéaires avec des coefficients réels. Il se base sur le procédé d'orthogonalisation de Gram-Schmidt, utilisé sur $\mathbb{R}$ pour construire une base de vecteurs orthogonaux à partir d'un ensemble de vecteurs non liés. L'algorithme de Lanczos adapte ce procédé d'orthogonalisation pour des espaces de Krylov. On définit un espace de Krylov K à partir d'un vecteur arbitraire v et de la matrice symétrique $B = {}^tAA$ comme le sous-espace engendré par les vecteurs $v, Bv, B^2v, \ldots$.

On définit la forme bilinéaire symétrique suivante de $K \times K$ vers $\mathbb{Z}/\ell\mathbb{Z}$:

$$\langle x, y\rangle = {}^txBy \text{ pour } x, y \in K \tag{3.2}$$

L'algorithme de Lanczos construit une suite de vecteurs $w_0, w_1, \ldots, w_k$ dont les éléments sont orthogonaux :

$$w_0 = v \tag{3.3}$$

$$w_1 = Bw_0 - \frac{\langle Bw_0, Bw_0\rangle}{\langle w_0, Bw_0\rangle} w_0 \tag{3.4}$$

$$\vdots$$

$$w_k = Bw_{k-1} - \frac{\langle Bw_{k-1}, Bw_{k-1}\rangle}{\langle w_{k-1}, Bw_{k-1}\rangle} w_{k-1} - \frac{\langle Bw_{k-1}, Bw_{k-2}\rangle}{\langle w_{k-2}, Bw_{k-2}\rangle} w_{k-2} \tag{3.5}$$

L'algorithme se termine lorsqu'il trouve un vecteur isotrope pour l'application bilinéaire (le vecteur $x \in K$ est un vecteur isotrope si et seulement si $\langle x, x\rangle = 0$). Alors, nous pouvons déduire un vecteur du noyau de la matrice A avec une bonne probabilité. Le cas problématique arrive lorsque nous trouvons un vecteur qui satisfait ${}^t(Ax)Ax = 0$ et que $Ax \neq 0$, ce qui arrive avec une faible probabilité lorsque nous travaillons dans un corps fini.

La dimension du sous-espace de Krylov est proche de N. La complexité de l'algorithme est alors celle de N itérations. Si on ne tient pas compte des coûts des produits scalaires, chaque itération correspond à deux produits matrice-vecteur, une multiplication par la matrice A et une multiplication par sa transposée ${}^tA$. Pour une étude plus approfondie de l'algorithme de Lanczos, on oriente le lecteur vers l'article [EK97].

3.5.2 Algorithme de Wiedemann

L'algorithme de Wiedemann prend deux vecteurs aléatoires $x, y \in (\mathbb{Z}/\ell\mathbb{Z})^N$. Il est composé de 3 étapes essentielles :

1. *Produits scalaires* : on calcule les $2N$ premiers termes de la suite $(a_i)_{i\in\mathbb{N}} \in (\mathbb{Z}/\ell\mathbb{Z})^{\mathbb{N}}$, où $a_i = {}^t x A^i y$.
2. *Générateur linéaire* : on calcule le polynôme minimal de la séquence, qui est le polynôme $F(X) = \sum_{i=0}^{d} f_i X^i$ de plus petit degré d tel que $\sum_{i=0}^{d} f_i a_{k+i} = 0$ pour tout $k \geq 0$. Le degré d est proche de N.
3. *Évaluation* : on calcule $w = F(A)z$, où z est un vecteur arbitraire de $(\mathbb{Z}/\ell\mathbb{Z})^N$.

La suite calculée dans l'étape *Produits scalaires* est une suite récurrente linéaire. Son polynôme minimal F est un diviseur du polynôme minimal μ_A de la matrice A. Supposons que les deux polynômes sont égaux. Puisque la matrice a un noyau non trivial, son polynôme minimal s'écrit sous la forme $\mu_A(X) = X^k P(X)$. L'exposant k est petit et est généralement égal à 1. Nous avons calculé le polynôme w comme $P(A)z$. Il existe alors un entier i qui est au plus égal à k tel que $A^i w = 0$. Par conséquent $A^{i-1}w$ est un vecteur non trivial de la matrice.

Si la matrice n'est pas singulière, le sortie w de l'algorithme est le vecteur nul. Sinon (c'est-à-dire si la dimension du noyau de la matrice est non nulle), alors le vecteur w est avec une grande probabilité un élément non nul du noyau de la matrice. La probabilité d'échec d'une exécution de l'algorithme de Wiedemann est en $O\left(\frac{1}{\ell}\right)$ [Wie86].

Les noms des étapes (en italique) correspondent aux appellations originales qui apparaissent dans l'article de Wiedemann [Wie86]. Dans le logiciel `CADO-NFS` [CADO] et dans notre implémentation d'algèbre linéaire, on utilise les noms `Krylov`, `Lingen` et `Mksol` pour désigner les programmes correspondant aux étapes *Produits scalaires*, *Générateur linéaire* et *Évaluation* respectivement.

En théorie, x est tiré aléatoirement dans $(\mathbb{Z}/\ell\mathbb{Z})^N$. Toutefois, en pratique, on le prend dans la base canonique, ou éventuellement de faible poids de Hamming, de sorte à pouvoir simplifier le produit scalaire entre ${}^t x$ and $A^i y$. Au lieu d'effectuer un produit scalaire complet, on prend juste l'élément (ou les éléments) de $A^i y$ qui correspond (ou correspondent) à la coordonnée (ou aux coordonnées) non nulle(s) de x. Cette approche ne pose empiriquement pas de problème avec les matrices qu'on étudie.

L'algorithme de Wiedemann requiert $3N$ produits matrice-vecteur pour les étapes *Produits scalaires* et *Évaluation*. Pour l'étape *Générateur linéaire*, on peut utiliser l'algorithme de Berlekamp-Massey [Ber68, Mas69] ou l'algorithme d'Euclide, dont la complexité est $O\left(N^2\right)$ opérations dans le corps, ou encore un algorithme asymptotique rapide tel que l'algorithme Half-gcd [Knu70, BGY80] dont la complexité est $O\left(N(\log N)^2\right)$ opérations dans le corps. Un produit matrice-creuse–vecteur est effectué en $O\left(N\gamma\right)$ opérations dans le corps, avec γ le nombre moyen de coefficients non nuls par ligne. Ainsi, les complexités relatives aux étapes *Produits scalaires* et *Évaluation* sont quadratiques en la dimension N. Dans ce mémoire, nous reviendrons plus en détails sur l'accélération des étapes *Produits scalaires* et *Évaluation* qui sont les étapes dominantes du calcul. Plus spécifiquement, nous allons étudier l'optimisation de l'opération produit matrice-creuse–vecteur, qui est centrale dans ces deux étapes. Plus de détails sur le calcul du générateur linéaire sont donnés dans l'article [Tho02].

À présent, nous allons nous intéresser aux versions « par blocs » des algorithmes de résolution. Coppersmith [Cop93] et Montgomery [Mon95] ont élaboré simultanément la version par blocs

de l'algorithme de Lanczos. La version par blocs de l'algorithme de Wiedemann a été introduite par Coppersmith [Cop94]. Dans ce recueil, nous nous contenterons de détailler l'algorithme de Wiedemann par blocs, parce qu'il présente certaines propriétés de parallélisme dont nous tirerons avantage, alors que paralléliser l'algorithme de Lanczos nécessite des contraintes plus strictes. En l'occurrence, l'algorithme de Wiedemann par blocs peut être exécuté sur différents clusters indépendants, alors que l'algorithme de Lanczos par blocs ne peut être exécuté que sur un seul cluster.

3.5.3 Algorithme de Wiedemann par blocs

L'algorithme de Wiedemann par blocs remplace le vecteur $y \in (\mathbb{Z}/\ell\mathbb{Z})^N$ par un bloc de n vecteurs $y^{(0)}, \ldots, y^{(n-1)}$, chacun dans $(\mathbb{Z}/\ell\mathbb{Z})^N$, et d'une manière similaire, remplace le vecteur x par un bloc de m vecteurs. La séquence de scalaires a_i est alors remplacée par une séquence de matrices $m \times n$. Les étapes de l'algorithme deviennent :

1. *Produits scalaires* : cette étape calcule les $\lceil \frac{N}{n} \rceil + \lceil \frac{N}{m} \rceil$ premiers termes de la séquence $(a_i)_{i \in \mathbb{N}}$.
2. *Générateur linéaire* : on cherche un générateur linéaire pour la séquence précédente. La sortie de *Générateur linéaire* est composé de n générateurs $F^{(0)}, \ldots, F^{(n-1)}$, chacun un polynôme dans $\mathbb{Z}/\ell\mathbb{Z}$ de degré plus petit que $\lceil \frac{N}{n} \rceil$.
3. *Évaluation* : on calcule l'élément suivant $w = \sum_{j=1}^{n} F^{(j)}(A) y^{(j)}$.

Les paramètres (n, m) sont désignés dans la littérature par les terminologies *blocking factors* ou *blocking parameters*.

Le nombre total des itérations des étapes *Produits scalaires* et *Évaluation* devient $\lceil \frac{2N}{n} \rceil + \lceil \frac{N}{m} \rceil$. Chaque itération est dominée par le temps d'un produit matrice–bloc-de-n-vecteurs. Si m et n sont du même ordre de grandeur et négligeables devant $\log N$, la complexité de l'étape *Générateur linéaire* devient $O\left(nN^2\right)$ en utilisant l'algorithme Berlekamp-Massey matriciel et $O\left(nN \log N (\log N + \log p)(n^{\omega-2} + \log N) \log\log N\right)$ avec l'algorithme rapide [DMT74, BL94, Tho02].

3.5.4 Intérêt des blocs

Dans cette section, supposons que les paramètres m et n sont égaux. Nous reviendrons dans la sous-section suivante sur le choix des valeurs pour m et n.

Les algorithmes par blocs présentent deux avantages principaux. Le premier avantage est l'amélioration la probabilité de succès. Ce bénéfice est d'autant plus important pour les systèmes définis sur $\mathbb{F}_2$, où la probabilité d'échec de l'algorithme de Wiedemann simple est trop élevée. Cette probabilité d'échec devient négligeable avec l'algorithme de Wiedemann par blocs. Une argumentation détaillée peut être trouvée dans [Tho03, Vil97].

Le second avantage consiste à remplacer le produit matrice-vecteur par le produit matrice–bloc-de-vecteurs. En effet, dans l'algorithme de Wiedemann simple, les étapes *Produits scalaires* et *Évaluation* sont composées de 3N itérations ; chaque itération correspond à un produit de la matrice creuse par un vecteur. L'algorithme de Wiedemann par blocs divise le nombre des itérations par n ; et chaque itération correspond à un produit de la matrice par un bloc de n vecteurs. Or, effectuer un produit d'une matrice creuse par un bloc de n vecteurs coûte souvent moins que n produits de la même matrice par un vecteur. En effet, lorsque qu'on traite simultanément

un bloc de vecteurs, la lecture des coefficients de la matrice est « factorisée ». Ceci est particulièrement intéressant pour les systèmes définis sur $\mathbb{F}_2$ où l'on prend les *blocking parameters* des multiples de 64 et où l'on encode un bloc de 64 vecteurs dans un mot machine. L'arithmétique est alors plus efficace que si nous multiplions par des vecteurs de bits.

On peut aussi tirer avantage autrement de l'opération du produit matrice–bloc-de-vecteurs en distribuant cette opération en plusieurs produits matrice-vecteur ou plusieurs produits matrice–bloc-de-vecteurs. En effet, remarquons que lorsqu'on effectue le produit d'une matrice par un bloc de n vecteurs de la forme $v = A \times u$, la j^{e} colonne du bloc de sortie, qu'on note $v^{(j)}$, dépend uniquement de la j^{e} colonne du bloc d'entrée, notée $u^{(j)}$. Ainsi, le calcul $v = A \times u$ peut être distribué en n tâches parallèles, chaque tâche calcule $v^{(j)} = Au^{(j)}$. La figure 3.5 schématise cette distribution sur un exemple avec $n = 3$. En distribuant l'opération centrale des étapes *Produits scalaires* et *Évaluation*, on arrive à distribuer le calcul de chaque étape en n calculs parallèles et indépendants qui ne nécessitent ni synchronisation, ni communication, excepté à la fin lorsqu'on combine tous les résultats.

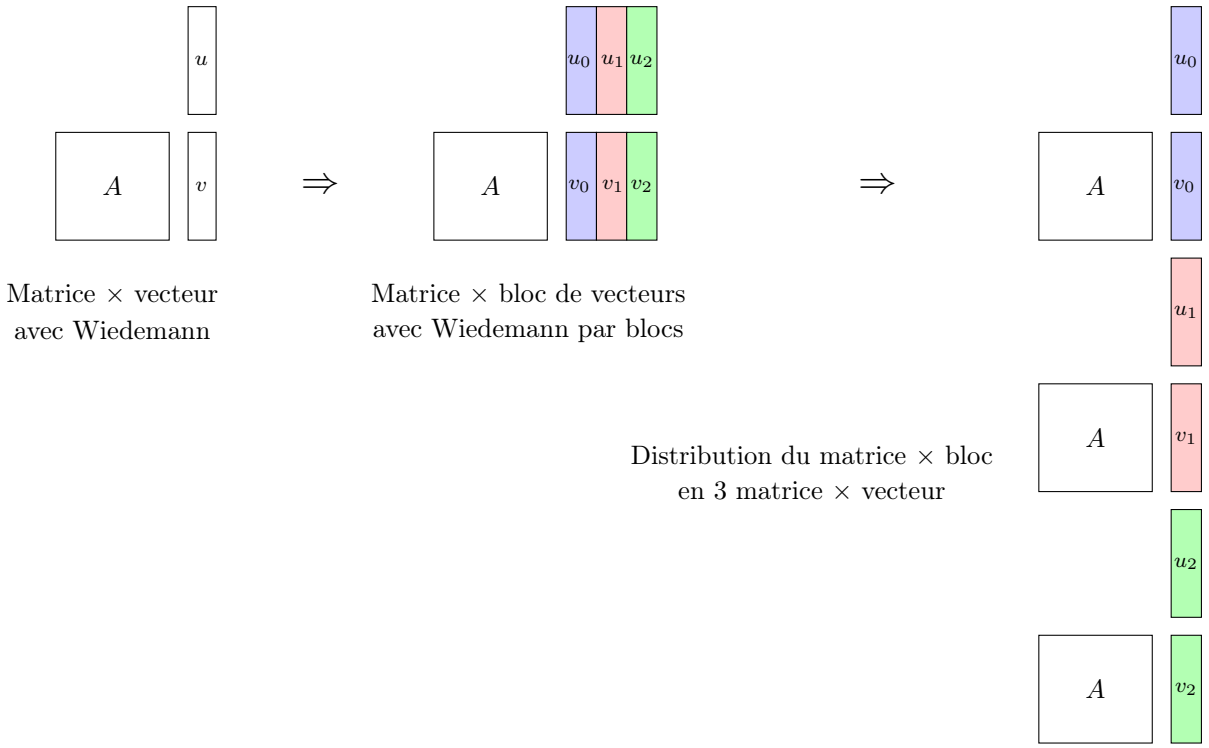


FIGURE 3.5 – Évolution de l'opération centrale entre Wiedemann simple et Wiedemannn par blocs et distribution du produit matrice × bloc en plusieurs produits matrice × vecteur.

Pour récapituler, nous venons de détailler deux façons possibles pour tirer avantage de l'utilisation des blocs. La première façon consiste à réduire la complexité totale des deux étapes les plus importantes, en continuant d'avoir une exécution séquentielle. La seconde est de créer du parallélisme en distribuant le calcul de chaque étapes en n calculs parallèles, chaque calcul étant exécuté sur un nœud de calcul. Ces deux façons ne sont pas mutuellement exclusives. Typiquement, lorsque on travaille dans $\mathbb{F}_2$, on prend généralement $n = 64n'$. Ainsi, on distribue le calcul sur n' calculs parallèles ; chaque calcul effectue simultanément le produit de la matrice par 64 vecteurs, représentés dans un seul mot machine.

3.5.5 Choix des paramètres (n, m)

En théorie, il y a une liberté totale dont le choix des valeurs pour les paramètres. Toutefois, prendre des (n, m) très grands va rendre l'étape *Générateur linéaire* difficile. En effet, à N constant, la complexité de cette étape est linéaire ou davantage en $(n + m)$.

Pour le choix de n, lorsque nous disposons de n' unités (nœuds) de calcul, on prend souvent n égal à un multiple de n'. Pour les systèmes définis sur $\mathbb{F}_2$, comme nous l'avons mentionné, on prend $n = 64n'$. Pour les systèmes définis sur des grands corps finis, on prend $n = n'$ ou bien $n = 2n'$ si l'architecture du nœud permet de traiter simultanément et d'une manière efficace deux vecteurs, par exemple avec des opérations SIMD.

Pour le choix de m, il n'y a pas de contrainte particulière. On tend généralement à prendre m plus grand que n, car si nous prenons les vecteurs du bloc x dans la base canonique (voir la remarque dans la sous-section 3.5.2), multiplier par x devient très peu cher. Augmenter m devient alors intéressant, vu que ceci permet de réduire le nombre d'itérations de l'étape *Produits scalaires* sans surcoût (rappel : ce nombre est égal à $\lceil \frac{N}{n} \rceil + \lceil \frac{N}{m} \rceil$).

Le choix des paramètres (m, n) est aussi défini par des contraintes d'implémentation. Par exemple, dans le logiciel `CADO-NFS` [CADO] m est choisi égal à $2n$ pour une question d'efficacité de l'exécution parallèle du programme `Lingen`, qui se parallélise d'autant mieux que $\gcd(n, m)$ est grand.

3.6 Conclusion

Nous avons présenté le problème d'algèbre linéaire et les spécificités liées aux matrices qui interviennent dans les calculs de logarithme discret. Nous avons détaillé les différents niveaux de parallélisme que nous pourrons utiliser pour résoudre efficacement ce problème sur des architectures pensées pour le calcul parallèle. Nous avons montré l'intérêt d'utiliser dans notre contexte les algorithmes spécifiques au caractère creux et nous avons détaillé comment l'emploi des blocs dans ces algorithmes permet de distribuer les calculs d'algèbre linéaire.

Chapitre 4

Paralléliser le produit matrice-vecteur sur plusieurs nœuds

Dans ce chapitre, nous allons discuter la parallélisation du produit matrice-vecteur sur plusieurs nœuds de calcul. Nous allons présenter le modèle de calcul et décrire les différentes étapes d'un produit matrice-vecteur parallèle. Une parallélisation efficace nécessite de distribuer des charges de travail équilibrées sur les nœuds de calcul et de minimiser les coûts de communications. Dans notre application, la question de l'équilibre des charges est d'autant plus importante, à cause du caractère creux des matrices considérées.
Les travaux détaillés dans ce chapitre ont été publiés dans [Jel14b].

Sommaire

4.1 Modèle

Nous supposons disposer d'un ensemble de *nœuds* de calcul, identiques, organisés dans une *grille* bidimensionnelle carrée (i.e., chaque case de la grille correspond à un nœud) de taille $t \times t$. Les nœuds sont interconnectés par un *réseau* de communication. Chaque nœud est identifié par ses coordonnées (i, j) dans la grille.

Dans ce modèle, la notion de *nœud* de calcul est une notion qui fait abstraction de la nature du *nœud*. Un *nœud* peut correspondre à un cœur dans une machine, à une machine indépendante ou à une carte graphique. On insiste sur le fait de ne pas confondre cette notion avec la notion du nœud de cluster, telle que nous l'avons utilisée dans le chapitre 2 et qui correspond à une machine connectée au réseau.

L'objectif est de pouvoir utiliser cette grille de calcul pour paralléliser le produit matrice-creuse–vecteur (SpMV pour Sparse-Matrix–Vector product). Plus précisément, les entrées sont la matrice creuse A et un vecteur d'entrée, dense, qu'on note u. Les *nœuds* de calcul collaborent ensemble pour pouvoir calculer le vecteur de sortie $v \leftarrow Au$. Le produit est itératif, dans le sens où le vecteur de sortie de l'itération k devient le vecteur d'entrée de l'itération $(k + 1)$.

4.2 Distribution de la charge de travail

La matrice A est divisée en *sous-matrices* carrées, de même taille $\frac{N}{t} \times \frac{N}{t}$, de sorte que chaque sous-matrice est assignée à un *nœud* de la grille.

La distribution particulière des coefficients non nuls dans la matrice creuse fait que les nœuds ont des charges de travail déséquilibrées. Les nœuds qui ont les sous-matrices les plus denses mettent plus de temps pour traiter leurs sous-matrices que les nœuds qui ont les sous-matrices les plus creuses.

Pour le type particulier des matrices dont nous disposons, le problème de déséquilibre peut être résolu d'une manière efficace. Pour corriger le problème, nous appliquons des permutations des lignes et des colonnes, de telle sorte à ce que la distribution des coefficients non nuls pour toutes les sous-matrices soient proches (ces distributions sont aussi proches de celle de la matrice A).

Une possibilité pour obtenir cette permutation est de trier les colonnes par leurs poids (on rappelle que le poids est le nombre de coefficients non nuls) et de les distribuer d'une manière uniforme sur les nœuds, puis de procéder de la même façon avec les lignes. Ceci est possible par le fait que l'écart-type du poids des lignes est beaucoup plus petit que celui du poids des colonnes.

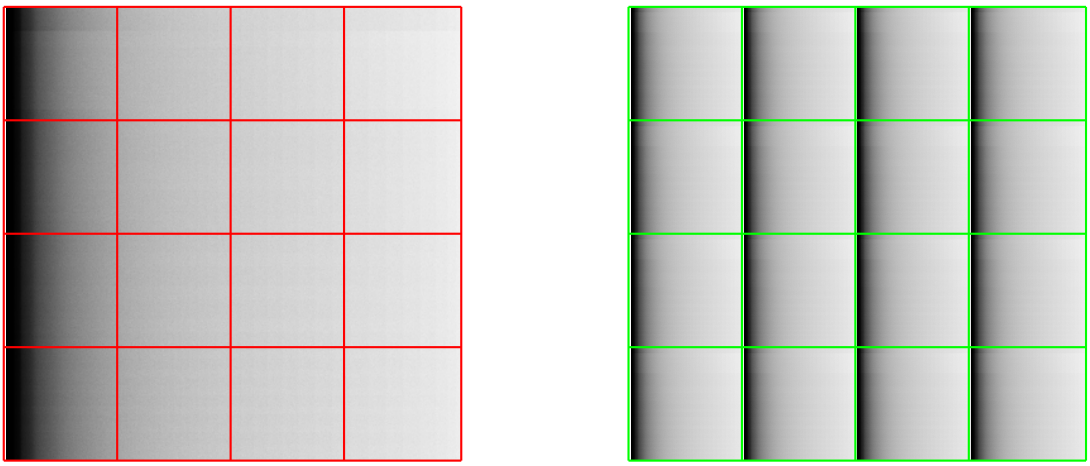
Nous illustrons cette technique d'équilibrage sur la matrice-exemple du calcul de FFS dans $\mathbb{F}_{2^{809}}$ (voir table 3.2). Nous supposons que t est égal à 4, c'est-dire que nous voulons diviser la matrice en 16 sous-matrices. La figure 4.1a montre la distribution des coefficients non nuls dans la matrice initiale et la variation de la densité des sous-matrices. La matrice totale contient 360.2 M coefficients non nuls ; une distribution équitable des charges de travail nécessiterait que chaque sous-matrice contienne 6.25% des coefficients non nuls, soit environ 22.5 M coefficients. Toutefois, les densités de la sous-matrice la plus dense et de la sous-matrice la moins dense sont comme suit :

— la sous-matrice la moins dense (en haut à droite) contient 2.1 M coefficients non nuls, soit 0.6% des coefficients non nuls de la totalité de la matrice ;
— la sous-matrice la plus dense (en bas à gauche) contient 81.8 M coefficients non nuls, soit 22.7% des coefficients ;

Nous appliquons le procédé de permutation de lignes et de colonnes. La figure 4.1b visualise la distribution des coefficients non nuls dans la matrice obtenue et dans ses sous-matrices. Maintenant, les densités de la sous-matrice la plus dense et de la sous-matrice la moins dense deviennent :

— la sous-matrice la plus dense contient 22.7 M coefficients non nuls, soit 6.3% des coefficients non nuls de la totalité de la matrice ;
— la sous-matrice la moins dense contient 22.2 M coefficients non nuls, soit 6.16% des coefficients ;

D'une part, les sous-matrices ont des densités relativement proches. D'autre part, les distributions spatiales des coefficients non nuls sont proches. Ainsi, des nœuds identiques mettent quasiment le même temps pour traiter les sous-matrices.



(a) Matrice initiale. (b) Matrice permutée.

FIGURE 4.1 – Distribution des coefficients non nuls pour une matrice initiale et une matrice permutée (matrice de FFS pour $\mathbb{F}_{2^{809}}$).

Les permutations des lignes et des colonnes ne changent pas le rang de la matrice et n'ont par conséquent pas d'impact sur la solution du système initial. Nous cherchons un élément du noyau de la matrice permutée, duquel un élément du noyau de la matrice A pourra par la suite être facilement déduit.

4.3 Schéma de calcul/communication

Nous raisonnons sur les communications entre nœuds selon le modèle de passage de messages (voir sous-section 2.4.2), même si dans la pratique, d'autres opérations (accès direct à la mémoire DMA) sont utilisées. Chaque processus qui est exécuté sur le nœud de calcul possède une mémoire privée et les processus effectuent des transferts de type envoi/réception pour s'échanger leurs données.

Au début d'une itération, un *nœud* (i, j) stocke dans sa mémoire la sous-matrice A_{ij} et le j^{e} fragment u_j du vecteur d'entrée u. À la fin de l'itération, le *nœud* (i, j) a besoin d'avoir dans sa mémoire le j^{e} fragment v_j du vecteur de sortie pour pouvoir calculer l'itération suivante. La longueur d'un fragment des vecteurs d'entrée ou de sortie est $\frac{N}{t}$ éléments.

Le produit matrice-vecteur parallèle est effectué en trois étapes, selon le schéma suivant :

1. *SpMV* : Chaque *nœud* (i, j) calcule le SpMV partiel $A_{ij}u_j$.
2. *Réduction* : Chaque *nœud* diagonal (i, i) collecte et somme les résultats partiels des nœuds de la ligne i. En effet, chaque nœud (i, j) calcule une contribution à v_i (i^{e} fragment de v) ; et la somme des contributions des nœuds de la ligne correspond à v_i.
3. *Broadcast* : Chaque *nœud* diagonal (i, i) diffuse (broadcast) son fragment v_i à tous les nœuds de la colonne i.

Dans la figure 4.2, nous présentons un exemple d'une exécution du schéma précédent pour 4 nœuds. La matrice A est divisée en 4 sous-matrices. Dans la figure, les 4 nœuds, colorés en gris, sont numérotés de 0 à 3.

Dans la partie gauche de la figure, sont indiqués les 4 sous-matrices, les fragments du vecteur d'entrée u_0 et u_1 et les fragments du vecteur de sortie v_0 et v_1 qui vérifient :

$$\begin{cases} v_0 &= A_{00}u_0 + A_{01}u_1 \\ v_1 &= A_{10}u_0 + A_{11}u_1 \end{cases}$$

Dans la partie centrale, nous montrons la répartition des calculs sur la grille.

Dans la partie droite, nous détaillons les données intermédiaires présentes dans chaque *nœud* après chaque étape.

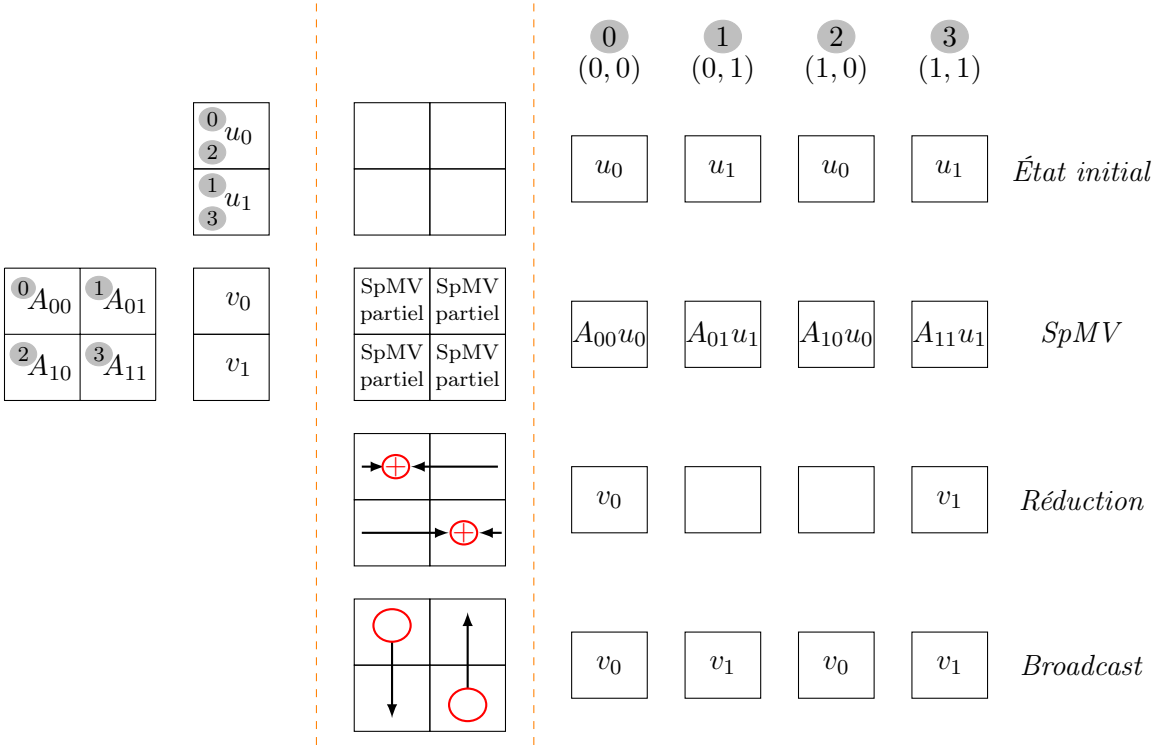


FIGURE 4.2 – Schéma de calcul/communication pour une division 2×2 de la matrice, en utilisant les opérations *Reduction/Broadcast*.

Dans le schéma précédent, pour une ligne donnée, un *nœud* collecte les $(t-1)$ autres résultats partiels et pour une colonne donnée, un *nœud* diffuse le fragment vers les $(t-1)$ autres nœuds. Ce schéma souffre ainsi du fait que les charges de communication ne sont pas distribuées d'une manière équilibrée.

Il est possible de paralléliser les opérations *Réduction/Broadcast*, typiquement en utilisant les opérations *ReduceScatter/AllGather*. On remplacera l'opération de *Réduction*, qui met le résultat combiné dans un nœud, par l'opération *ReduceScatter*, qui divisera le résultat combiné en plusieurs parties, chacune combinée dans un nœud [Tho12b, section 13.2].

Cette parallélisation apporte deux avantages : le premier est que tous les nœuds participeront aux opérations de communication, le second est que la taille des fragments à communiquer dans les étapes analogues à la réduction et au broadcast sera divisée par t. Toutefois, en utilisant un schéma parallèle, la sortie de l'itération sera permutée, i.e., les fragments du vecteur de sortie v ne seront pas distribués de la même manière que l'étaient les fragments du vecteur d'entrée u, au début de l'itération [Tho12a].

Reprenons dans la figure 4.3 l'exemple de la matrice distribuée sur 4 nœuds et détaillons l'exécution du schéma de Calcul/Communication avec les opérations *ReduceScatter/AllGather*. Maintenant, dans la partie gauche de la figure, nous considérons t^2 fragments pour chaque vecteur tels que :

$$u_0 = \begin{bmatrix} u_{00} \\ u_{01} \end{bmatrix}, \; u_1 = \begin{bmatrix} u_{10} \\ u_{11} \end{bmatrix}, \; v_0 = \begin{bmatrix} v_{00} \\ v_{01} \end{bmatrix} \text{ et } v_1 = \begin{bmatrix} v_{01} \\ v_{11} \end{bmatrix}$$

Nous observons qu'à la sortie de l'itération, les fragments du vecteur v ont été permutés. Ainsi, au lieu d'avoir effectué $v = Au$, nous avons calculé $v = \Sigma_t Au$, où Σ_t est une matrice de permutation qui dépend du paramètre t, ce qui n'affecte pas la résolution du système, comme nous l'avons déjà vu avec l'équilibrage des charges de travail dans la section 4.2.

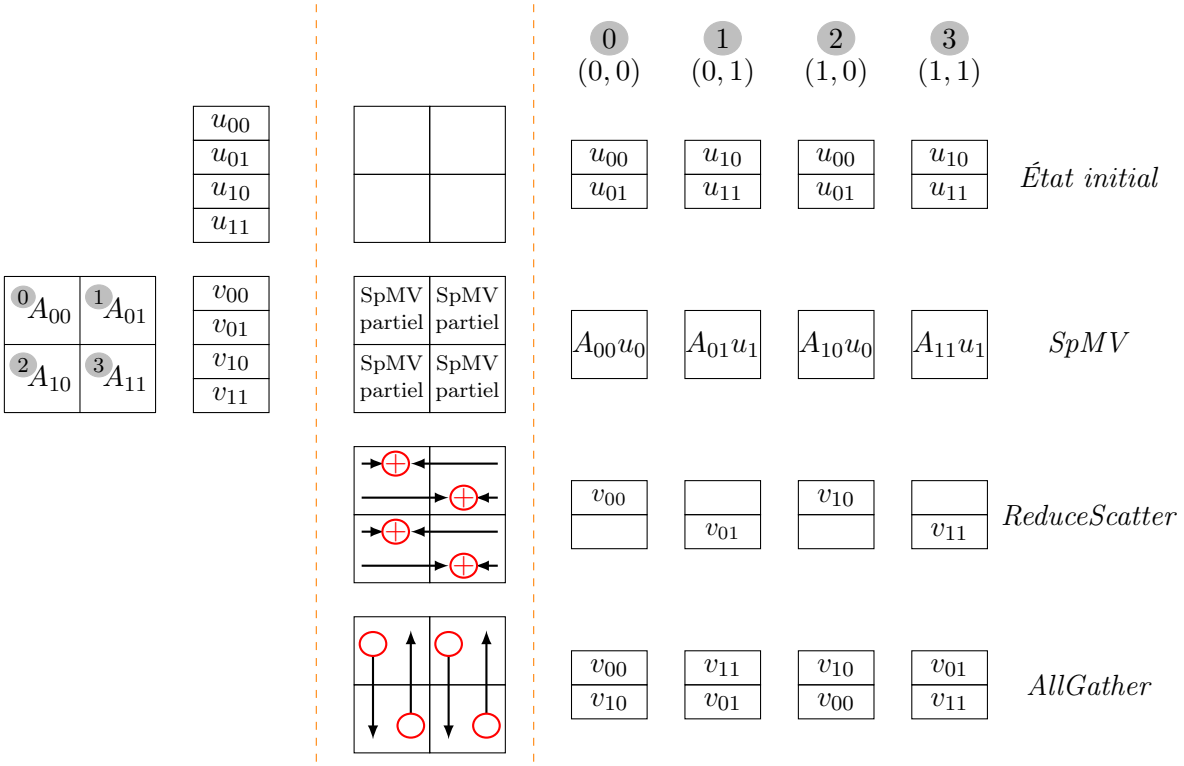


FIGURE 4.3 – Schéma de calcul/communication pour une division 2×2 de la matrice, en utilisant les opérations *ReduceScatter/AllGather*.

Dans ce chapitre, nous considérons le produit partiel sous-matrice–vecteur comme une *boîte noire*, c'est-à-dire une sous-routine, qui prend une sous-matrice et un fragment et qui retourne leur produit. Le chapitre 5 détaille comment cette sous-routine est implémentée. Maintenant, nous nous intéressons au partage des données entre les nœuds de calcul.

4.4 Communication entre les nœuds de calcul

Nous utilisons le modèle de passage de messages (avec l'interface MPI) pour implémenter la parallélisation du produit matrice-vecteur. Le choix de MPI est justifié par le fait qu'il est adapté pour des communications aussi bien dans le cas où les nœuds de calcul sont des GPU que dans le cas où les nœuds correspondent à des CPU. Toutefois, on souligne que MPI n'est pas le modèle le plus efficace pour les communications CPU intra-nœud (nous avons déjà abordé ce point dans la sous-section 2.4.2).

Chaque processus MPI est exécuté sur un *nœud* et effectue un produit partiel. Nous utilisons les *Communications Collectives* pour l'échange de données entre les processus. Nous rappelons qu'une *Communication Collective* est une méthode de communication qui fait intervenir tous les processus MPI qui appartiennent à un ensemble donné, appelé *Communicateur*. Ici, nous utilisons en particulier les collectives :

- `MPI_Reduce` : un processus collecte et combine des données réparties sur plusieurs processus.
- `MPI_Bcast` : un processus diffuse des données vers plusieurs processus.
- `MPI_Reduce_Scatter` : il s'agit d'une réduction, où le résultat final est dispersé sur plusieurs processus.
- `MPI_AllGather` : il s'agit de collecter, par plusieurs processeurs, des données réparties.

4.5 Répartition des processus MPI

Nous avons déjà introduit dans la section 4.1 la notion abstraite de *nœud* de calcul, qui correspond à une unité qui exécute un processus MPI et qui peut correspondre à un GPU ou un cœur d'un CPU. Maintenant, nous allons utiliser la notion de nœud du cluster, qui est plutôt une définition matérielle et qui correspond à une machine dans le cluster et qui peut contenir plusieurs GPU ou plusieurs cœurs et qui peut par conséquent exécuter plusieurs processus MPI.

Si nous considérons les communications entre deux processus MPI qui sont exécutés en parallèle sur le cluster, nous distinguons deux cas, le cas *intra-nœud* où les deux processus sont exécutés sur un même nœud du cluster et le cas *inter-nœud* où les deux processus sont exécutés sur deux nœuds distincts. Les communications *intra-nœud* sont plus efficaces que les communications *inter-nœud*, vu que les communications *inter-nœud* sont effectuées à travers le réseau, alors que les communications *intra-nœud* sont effectuées à travers la mémoire partagée, s'il s'agit de cœurs d'un CPU ou à travers le PCI Express (PCIe), s'il s'agit de GPU (voir section 2.4). Lorsque des opérations de communication sont effectuées entre plusieurs processus MPI, il est possible d'avoir simultanément les deux cas. Dans cette section, nous allons discuter la stratégie de répartition des processus MPI de sorte à tirer avantage le plus possible de l'efficacité des communications *intra-nœud* par rapport aux communications *inter-nœud*.

Pour illustrer cette stratégie, nous considérons un exemple, celui d'une parallélisation d'un produit matrice-vecteur sur un ensemble de 4 nœuds ; chaque nœud est composé de 2 processeurs Intel Xeon E5-2650 (2 GHz) avec 8 cœurs dans chaque processeur, soit 16 cœurs par nœud. Les

nœuds sont connectés par des connexions InfiniBand FDR à 56 Gbit/s (voir sous-section 2.4.1). Le calcul du produit matrice-vecteur est effectué en parallèle par 64 processus MPI. Nous utilisons la matrice décrite dans la table 7.8, qui contient 7 M lignes et colonnes, et la bibliothèque OpenMpi-1.7.3 et nous allons mesurer les latences des opérations de collecte dans une ligne et de diffusion dans une colonne. Le but est de minimiser les temps de communications, par conséquent la somme des ces deux opérations.

Dans la sous-figure 4.4a, est illustrée la répartition par défaut des 64 processus. Avec cette configuration, nous avons obtenu les temps suivants :

— temps de la collecte : 0.18 s ;
— temps de la diffusion : 0.65 s.

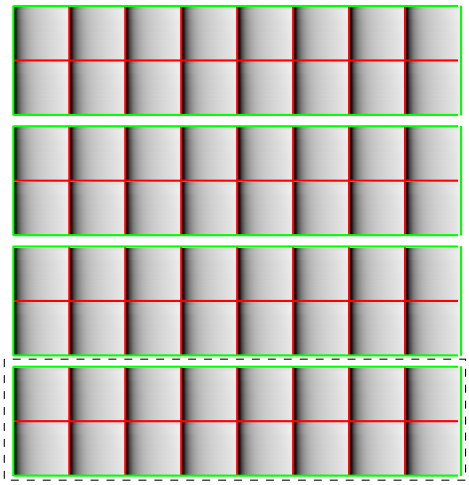
La collecte des résultats dans une ligne est « rapide » vu que tous les processus s'exécutent sur le même nœud, alors que la diffusion dans une colonne est « lente », vu qu'elle se fait à travers le réseau par des processus s'exécutant sur les 4 nœuds.

Nous proposons de répartir les processus MPI de sorte à ce que les processus qui sont exécutés sur des cœurs appartenant à un même nœud soient groupés en blocs carrés, c'est-à-dire lorsqu'on minimise les frontières entre les processus s'exécutant sur des nœuds distincts (voir la sous-figure 4.4b). Avec cette répartition, nous diminuons les communications *inter-nœuds* au profit des communications *intra-nœuds*. Ce qui donne les mesures suivantes :

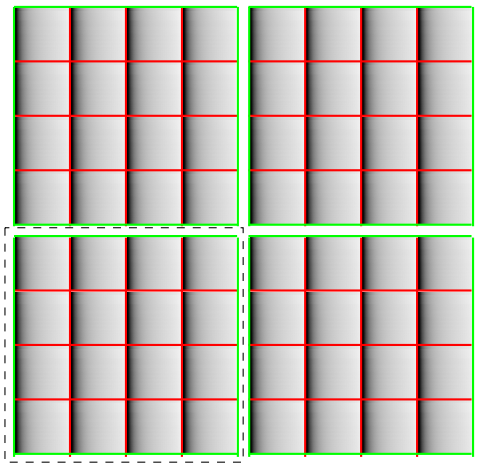
— temps de la collecte : 0.22 s ;
— temps de la diffusion : 0.18 s.

Que ce soit pour la collecte dans une ligne ou la diffusion dans une colonne, les 8 processus MPI qui communiquent appartiennent à deux groupes, où chaque groupe est exécuté sur un nœud.

La seconde répartition donne une accélération d'un facteur 2.1 sur la somme des latences de deux opérations de communication, par rapport à la répartition par défaut.



(a) Répartition par défaut des processus.



(b) Répartition choisie des processus.

FIGURE 4.4 – Répartition des processus MPI sur les cœurs pour une matrice divisée en 8×8 et distribuée sur 4 nœuds, chacun contenant 16 cœurs.

4.6 Conclusion

Dans ce chapitre, nous avons discuté de la parallélisation du produit matrice-vecteur sur plusieurs nœuds de calcul, ce qui correspond au second niveau de parallélisme pour le calcul de l'algèbre linéaire. Nous avons détaillé le produit matrice-vecteur parallèle et les stratégies pour l'accélérer, en considérant le produit partiel du *nœud* comme une boîte noire. Dans les chapitres suivants, nous allons étudier comment effectuer ce produit partiel et détailler les niveaux de parallélisme qui lui sont associés.