Chapitre de l'ouvrage *Parallélisme et applications irrégulières*, **HERMES, Paris 1995, p. 71–84**

Algorithmique PRAM II
Algorithmes de travail optimal et probabilistes

Jean-Louis ROCH et Gilles VILLARD
(LMC/IMAG) [1].

1 Introduction

Nous avons expliqué dans le chapitre précédent les techniques de base pour la construction d'algorithmes parallèles très rapides (de temps poly-logarithmique) sur le modèle PRAM [2]. Lors de la programmation sur une machine réelle, avec ses propres caractéristiques, un tel algorithme peut être réorganisé (ordonnancement des calculs qui le constituent) d'un très grand nombre de façons différentes, pour permettre de s'adapter au mieux au grain de parallélisme de la machine cible.

L'intérêt d'un algorithme PRAM très parallèle (dans $\mathcal{NC}$) est alors sa *portabilité* intrinsèque, *i.e.* son aptitude à être implémenté de façon simple sur une machine donnée pour obtenir un programme performant. La recherche de la meilleure variante pour une machine donnée, nécessaire à l'obtention de performances, concerne alors essentiellement le regroupement des tâches de grain trop fin – adaptation de grain – et l'ordonnancement. Par exemple, les algorithmes par lignes ou par colonnes pour le produit matrice-vecteur peuvent être vus comme des variantes d'implémentation de l'algorithme parallèle de grain le plus fin [23].

Mais pour que cet algorithme apporte des performances sur une machine à p processeurs, encore faut-il qu'il soit de travail optimal. Or, le passage d'un algorithme parallèle à un algorithme parallèle de travail optimal est bien sûr loin d'être trivial.

Une première méthode pour obtenir des algorithmes de travail optimal est de construire un algorithme moins parallèle (de profondeur polynomiale et non plus poly-logarithmique), mais qui conserve la notion d'extensibilité, c'est-à-dire que l'accélération augmente proportionnellement au nombre de processeurs lorsque la taille du problème est suffisamment grande.
Dans le paragraphe 2, les outils introduits dans [26] pour classifier de tels algorithmes sont présentés. L'inconvénient de cette approche est que l'algorithme impose des contraintes lors de l'ordonnancement, et donc peut perdre en portabilité par rapport à un algorithme de profondeur poly-logarithmique.

Une deuxième méthode consiste à relâcher les contraintes sur le modèle PRAM : plutôt que de chercher un algorithme qui soit toujours dans $\mathcal{NC}$ de travail optimal, on cherche ici à construire un algorithme qui donne la solution du problème P_n (et cela quelle que soit l'instance, c'est-à-dire les données en entrée) *presque toujours* en temps parallèle poly-logarithmique avec un travail optimal.

Il ne faut pas confondre les algorithmes probabilistes, qui donnent (mais pas toujours) la solution exacte d'un problème, et les algorithmes numériques qui donnent

1. Nous remercions Jean-Marc Vincent pour son expertise probabiliste – mais pas aléatoire – et Nathalie Revol et Thierry Gautier pour les indications et corrections déterministes efficaces et performantes.

2. Ce chapitre est construit comme une suite au chapitre [22], et s'appuie sur les notions qui y sont présentées.

une approximation de la solution d'un problème. Notamment, les algorithmes numériques de type Monte-Carlo, qui fournissent avec une bonne probabilité une approximation aussi précise que désirée de la solution à partir de tirages aléatoires, sont des algorithmes probabilistes. La cible des algorithmes probabilistes est en général les problèmes de décision, comme tester la primalité d'un entier, ou déterminer si une matrice est inversible. Plus généralement, si par exemple on veut trier un ensemble, ou ranger les éléments d'une liste par ordre dans un tableau, une solution approchée n'est pas satisfaisante. La construction d'un algorithme probabiliste pour un tel problème vise alors à construire la vraie solution du problème, avec un algorithme très performant quelle que soit l'instance du problème (*i.e.* le pire cas ne peut pas être caractérisé), mais en acceptant – avec une probabilité faible – de ne pas obtenir de réponse.

Les techniques qui régissent la construction d'algorithmes parallèles déterministes sont bien sûr utilisées pour la construction d'algorithmes parallèles probabilistes, mais des techniques spécifiques sont nécessaires pour introduire le tirage aléatoire dans l'algorithme, et pour mesurer la complexité de l'algorithme : ces techniques sont proches de celles utilisées en algorithmique séquentielle probabiliste.

Dans ce chapitre, construit surtout à partir de [15, 11, 6], nous introduisons les différents types d'algorithmes probabilistes tout d'abord dans le cadre séquentiel (§3) puis parallèle (§4). Les paragraphes suivants sont consacrés à la présentation de trois techniques de base pour la construction d'algorithmes probabilistes :

- diviser pour paralléliser probabiliste (§5),
- introduction d'aléatoire par homomorphisme sur les entiers (§6),
- introduction d'aléatoire à partir d'une caractérisation algébrique (§7).

Ces techniques sont illustrées par des exemples caractéristiques d'algorithmes parallèles probabilistes.

Pour des compléments sur les différents résultats de probabilité utilisés, le lecteur pourra se référer à [8].

2 Algorithmes de travail optimal

En pratique, comme le nombre de processeurs réellement disponibles est faible, un algorithme parallèle de travail optimal, même s'il n'est pas de temps polylogarithmique, s'avère intéressant pour autant qu'il reste "extensible", c'est-à-dire qu'il soit possible de tirer parti de la présence d'un nombre quelconque de processeurs pour une taille suffisante des entrées. Pour classifier de tels algorithmes, une extension de la notion d'algorithme "efficace" dans le cadre de $\mathcal{NC}$ est proposée [16] [26].

2.1 Algorithmes d'accélération polynomiale

Soit $\mathcal{P}\backslash$ un problème, et soit $T_s(n)$ le temps du meilleur algorithme séquentiel résolvant ce problème. Soit A un algorithme parallèle résolvant ce problème avec un coût $O_{//}\left(T_{//}(n), H(n)\right)$, *i.e.* en temps parallèle $T_{//}(n)$ avec $H(n)$ processeurs.

L'algorithme A est dit d'*accélération polynomiale* [15] (ou encore *extensible*) s'il existe une constante $\epsilon < 1$ telle que :

$$T_s(n^\epsilon) = O\left(T_{//}(n)\right) .$$

Soit $W(n) = T_{//}(n)H(n)$ le travail de l'algorithme A. Pour préciser la surcharge de travail induite par un algorithme parallèle d'accélération polynomiale, Snir introduit la notion d'*inefficacité*. Ce terme est préféré à efficacité, puisque, de par le principe de Brent [22], un algorithme parallèle ne peut pas être *plus* rapide théoriquement qu'un algorithme séquentiel.

L'*inefficacité* de A est dite :

- *constante* si
$$W(n) = O(T_s(n)).$$
On peut alors parler d'*efficacité constante*, le terme d'*inefficacité constante* étant mal adapté pour qualifier un algorithme performant.

- *poly-logarithmique* si
$$W(n) = O\left(T_s(n)\log^{O(1)}(T_s(n))\right).$$

- *polynomiale* si
$$W(n) = O\left(T_s(n)^{O(1)}\right).$$

2.2 Illustration : inversion de matrice

Considérons comme problème $\mathcal{P}\backslash$ l'inversion de matrices dans un corps de caractéristique nulle.

En parallèle, il existe des algorithmes d'inversion de matrice dans $\mathcal{NC}^{\in}$ depuis longtemps [7], mais ces algorithmes ne sont pas de travail optimal. Le meilleur algorithme dans NC^2 est de travail $n^{\omega+\frac{n}{2}}$ [19].

En séquentiel, l'algorithme de Gauss fournit un algorithme de complexité n^3. Une parallélisation directe de cet algorithme (par une approche "Diviser pour paralléliser") fournit un algorithme de travail n^3, mais de complexité temporelle n . Même si cet algorithme ne prouve pas l'appartenance à $\mathcal{NC}$ du problème, il est d'accélération polynomiale (donc extensible) et d'efficacité constante par rapport à l'algorithme de Gauss séquentiel : il est donc intéressant en pratique.

De même, une parallélisation directe de l'algorithme de Strassen [27] – version "Divide & Conquer" de Gauss, de complexité n^ω, $\omega \simeq 2,38$ – fournit un algorithme de complexité $O_{//}\left(n, n^{\omega-1}\right)$. On obtient ainsi un algorithme d'accélération polynomiale et d'efficacité constante par rapport au meilleur algorithme séquentiel connu.

Pour conclure ce paragraphe, il est à noter que des algorithmes parallèles probabilistes de temps poly-logarithmique $O(\log^2 n)$ et de travail optimal sont connus [13].

3 Classification des algorithmes probabilistes

Deux types d'algorithmes probabilistes sont distingués, aussi bien en séquentiel qu'en parallèle : les algorithmes de Monte-Carlo et les algorithmes de Las Vegas. Pour ces deux types d'algorithmes, nous introduisons une définition intuitive, puis une plus formelle.

De manière générale, un algorithme probabiliste utilise un oracle qui est capable de fournir des nombres aléatoires (avec des tirages indépendants), et de même distribution de probabilité.

Soit P un problème, qui à une entrée x associe une sortie $f(x)$.

3.1 Algorithme de Las Vegas

Un algorithme de Las Vegas délivre en sortie soit le bon résultat, soit l'indication qu'il lui est impossible de donner le résultat : on parle alors d'*échec*. La caractéristique fondamentale est que l'algorithme de Las Vegas ne doit donner un échec qu'avec une probabilité plus petite qu'une constante – disons $\frac{1}{2}$ –, et ce indépendamment de la taille n de l'entrée et de la valeur de l'entrée.

À la différence des algorithmes déterministes évalués avec une complexité en moyenne, cela n'a donc pas de sens de parler de pire cas pour un algorithme de Las Vegas. Si l'on exécute deux fois consécutives un algorithme de Las Vegas sur une même entrée x_0 arbitraire, la première exécution peut très bien générer une indication d'échec, et la deuxième fournir le résultat.

De par l'indépendance des tirages, si la probabilité d'échec d'une exécution de l'algorithme est plus petite que p, la probabilité d'échec de k exécutions (successives ou en parallèle) sera plus petite que p^k. Donc, même si l'échec reste toujours possible, sa probabilité peut être rendue aussi petite que désirée, en augmentant soit le temps avec plusieurs exécutions séquentielles, soit le nombre de processeurs avec plusieurs exécutions parallèles.

3.2 Algorithme de Monte-Carlo

À la différence d'un algorithme de Las Vegas, un algorithme de Monte-Carlo peut donner une réponse erronée sans indication d'échec – on parle d'erreur – mais cela avec une probabilité plus petite qu'une constante (disons $\frac{1}{2}$). Ici encore, de par l'indépendance des tirages, cette probabilité peut être rendue, avec plusieurs exécutions, aussi petite que désirée.

S'il existe un algorithme "verifier(y)" pour vérifier si la sortie y de l'algorithme de Monte-Carlo est exacte ou erronée, il est alors possible de construire un algorithme de Las Vegas en faisant suivre l'algorithme de Monte-Carlo de la vérification de la correction de la sortie qu'il délivre.

Le problème est que cette vérification peut être plus coûteuse que l'algorithme de Monte-Carlo, et conduise alors à un algorithme de Las Vegas inefficace. Le principal inconvénient d'un algorithme de Monte-Carlo est donc qu'il n'est pas possible de déterminer si la sortie qu'il délivre est correcte ou non.

3.3 Illustration sur l'exemple du quicksort

Nous considérons ici un premier exemple simple d'algorithme probabiliste, afin de préciser les différences entre algorithmes de Monte-Carlo et de Las Vegas. L'étude détaillée de cet exemple sera reprise au paragraphe 5.2.

De manière générique, les algorithmes de tri de type "quicksort" commencent par partitionner la séquence à trier en deux sous-séquences autour d'un élément pivot (d'un côté les éléments plus petits que le pivot, de l'autre les plus grands), puis trient récursivement les deux sous-séquences :

```
Tri( n, X  = (x_1, ..., x_n) ) ==
  si n == 1 alors retourner X
  sinon
    piv := ChoisirPivot(n, X) ;
    [X_inf, X_sup] := SeparerInfSup(X, X(piv) ) ;
    retourner QuickSort(X_inf) ^ ( X(piv) ) ^  QuickSort(X_sup) ;
  fin si ;
fin Tri ;
```

où `SeparerInfSup` range dans le tableau `X_inf` les éléments de `X` inférieurs ou égaux à `X(piv)` – à l'exception de `X(piv)` – et dans le tableau `X_sup` les éléments de `X` strictement supérieurs à `X(piv)`. L'opérateur ^ dénote l'opération de concaténation de séquences.

La complexité de l'algorithme est liée à la procédure de choix du pivot. L'idéal est de choisir le pivot qui sépare exactement en deux sous-tableaux de taille $\frac{n}{2}$, mais ce choix est coûteux.

Il est bien connu qu'un choix déterministe arbitraire du pivot (premier élément par exemple, ou encore pseudo-médiane) permet d'obtenir une profondeur *moyenne* des appels récursifs de $O(\log n)$, au prix d'un pire cas quadratique (par exemple, si le tableau en entrée est trié dans le cas du premier élément en pivot). On parle alors de *complexité moyenne*.

Un choix probabiliste du pivot (par un tirage aléatoire d'un indice entre 1 et n) permet d'assurer que le pire cas ne se reproduira pas – indéfiniment tout au moins – sur la même séquence en entrée. On parle alors d'algorithme de Sherwood [6].

Un *algorithme de Monte-Carlo* consisterait alors à n'effectuer des appels récursifs que sur une profondeur de $K \log n$ (K constante), et à retourner le tableau obtenu alors. Le coût de cet algorithme est $O(n \log n)$, et il délivre en sortie soit un tableau trié, soit un tableau non trié (erreur).

Si l'on ajoute à cet algorithme de Monte-Carlo la vérification que le tableau est trié (de coût $O(n)$), on obtient un *algorithme de Las Vegas* qui retourne soit le tableau trié soit l'indication d'échec lorsque la vérification indique que le tableau résultat est non trié.

Remarque. Ici, la procédure qui permet de vérifier que le résultat est correct est très simple, et par suite, l'algorithme de Monte-Carlo n'a pas d'intérêt par rapport à celui de Las Vegas. Mais pour d'autres problèmes (notamment en algèbre linéaire), nous verrons que la vérification de la correction de la solution peut être très coûteuse. On pourra donc alors aussi s'intéresser à la construction d'algorithmes de Monte-Carlo.

3.4 Une définition plus formelle

Nous reprenons ici la définition proposée dans [15] dans le cadre de problèmes caractérisés par une relation binaire $\mathcal{R}$, entre l'ensemble E des entrées et celui S des sorties. Sur une entrée $x \in E$ le problème consiste à trouver une sortie $y \in S$ telle que $x\mathcal{R}\dagger$. Un algorithme déterministe résolvant ce problème qui reçoit en entrée un élément $x \in E$ délivre en sortie l'une des deux réponses suivantes :

(1) un élément $y \in S$ tel que $x\mathcal{R}\dagger$,

(2) l'indication qu'il n'existe aucun élément $y \in S$ qui soit tel que $x\mathcal{R}\dagger$.

Un algorithme probabiliste peut renvoyer, outre ces deux réponses :

(3) l'indication d'échec, dans le cas où il lui est impossible de déterminer s'il existe ou non un élément $y \in S$ vérifiant $x\mathcal{R}\dagger$.

Si, sur une entrée x, il existe un élément y vérifiant $x\mathcal{R}\dagger$, un algorithme de Las Vegas ou un algorithme de Monte-Carlo fournissent tous deux soit la réponse (1) avec une probabilité supérieure à une constante p – disons $\frac{1}{2}$ –, soit la réponse (3) avec une probabilité inférieure à $(1-p)$.
Par contre, s'il n'existe aucun élément $y \in S$ vérifiant $x\mathcal{R}\dagger$, un algorithme de Las Vegas fournit soit la réponse (2), soit le constat d'échec (3), avec une probabilité

inférieure à $1-p$. Un algorithme de Monte-Carlo retourne quant à lui toujours le constat d'échec (3) [3].

Exemple. Si l'on reprend l'exemple du tri, le problème de décision associé peut être : étant donnés deux tableaux de n éléments X_1 et X_2 en entrée, X_1 et X_2 contiennent-ils exactement les mêmes éléments ?
Si l'on utilise l'algorithme de Las Vegas présenté précédemment pour trier les deux tableaux X_1 et X_2 en entrée, puis que l'on compare les deux tableaux – dans le cas du succès du tri de ces deux tableaux –, on obtient alors un algorithme de Las Vegas, qui est capable de décider, avec une probabilité suffisante, si les deux tableaux sont différents ("zero-error").
Si maintenant on utilise l'algorithme de Monte-Carlo présenté précédemment pour trier X_1 et X_2, alors, même si l'on peut décider avec une probabilité suffisante que les deux tableaux sont les mêmes (dans le cas où les deux tris fournissent deux tableaux identiques), il est impossible de décider, avec cet algorithme, si les deux tableaux sont différents ("one-sided error").

4 Modèles PRAM probabilistes et classes $\mathcal{RNC}$ et $\mathcal{ZNC}$

En séquentiel, la complexité d'un algorithme probabiliste est liée au temps – et à l'espace mémoire – nécessaire à l'obtention d'une solution correcte avec une probabilité sufisamment grande (disons supérieure à $\frac{1}{2}$, ou, ce qui est encore mieux, qui tend vers 1 quand la taille du problème tend vers l'infini).
En parallèle, comme pour les algorithmes déterministes [22], la complexité est mesurée en termes du temps parallèle $T_{//}(n)$ nécessaire à l'obtention d'une telle solution et du nombre de processeurs $H(n)$ nécessaires pour obtenir ce temps. Le travail – *i.e.* le produit $T_{//}(n) \times H(n)$ – est noté dans la suite $W(n)$.

Pour introduire de l'aléatoire dans les algorithmes, il est nécessaire d'ajouter aux modèles parallèles des primitives permettant d'effectuer des tirages aléatoires.

4.1 PRAM probabiliste

Une PRAM probabiliste est une PRAM où chaque processeur est doté d'une opération "alea" qui permet de générer des entiers aléatoires tirés selon une loi uniforme dans $\{1,\ldots,n^{O(1)}\}$, n étant la taille des entrées (autrement dit un nombre aléatoire ayant $O(\log n)$ bits). Cette primitive est de temps unité. Un tel nombre peut être stocké dans un emplacement mémoire, et manipulé en temps constant. Les nombres aléatoires tirés par des processeurs distincts sont *indépendants*.

Les classes de complexité définies sur le modèle PRAM (EREW^k, CREW^k, CRCW^k) ont leurs équivalents probabilistes, préfixés par "Z" [4] pour les algorithmes Las Vegas, et par "R" [5] pour les algorithmes Monte-Carlo.

Par exemple, la classe ZCREW^k est l'ensemble des problèmes qui peuvent être résolus par un algorithme de Las Vegas s'exécutant sur une CREW-PRAM probabiliste (lecture concurrente, écriture exclusive) en temps $O(log^k n)$ avec $n^{O(1)}$ processeurs. Un tel algorithme ne doit fournir un constat d'échec qu'avec une probabilité inférieure à une constante (strictement inférieure à 1), $\frac{1}{2}$ par exemple.

3. C'est pourquoi un algorithme de Las Vegas est appelé algorithme "zero-error" alors qu'un algorithme de Monte-Carlo est appelé algorithme "one-sided error" [15].
4. "Z" pour "Zero-error"
5. "R" pour "Random"

4.2 Circuit probabiliste

Un circuit booléen probabiliste à n entrées est un circuit booléen à n entrées auquel on ajoute, en nouvelles entrées, $n^{0(1)}$ bits aléatoires, indépendants, et générés selon une loi uniforme dans $\{0, 1\}$.

Les classes de complexité $\mathcal{NC}^{\|}$ (et $\mathcal{AC}^{\|}$) définies sur le modèle booléen (et de même sur les circuits arithmétiques) ont leurs équivalents probabilistes, préfixés encore par "Z" pour les algorithmes de Las Vegas, et par "R" pour les algorithmes de Monte-Carlo.

Par exemple, la classe $\mathcal{RNC}^{\|}$ est l'ensemble des problèmes qui peuvent être résolus par une famille log-uniforme (cf [22]) de circuits booléens probabilistes $(B_n)_{n\in I\!N}$, telle que B_n a $n^{O(1)}$ entrées, est de profondeur $\log^k n$ et comporte $n^{O(1)}$ portes booléennes. Ce circuit délivre sur ses sorties la solution – erronée ou exacte – du problème (exacte avec une probabilité supérieure à $\frac{1}{2}$).

5 Diviser pour paralléliser probabiliste

La technique "Diviser Pour Régner" parallèle [22] permet la construction d'algorithmes performants, pour autant qu'il soit possible de réduire – efficacement – la résolution du problème initial à celle de sous-problèmes de tailles *équivalentes*, de façon à garantir un temps parallèle poly-logarithmique.

Or la recherche d'une telle réduction peut entraîner un coût important, soit en ce qui concerne la découpe du problème en sous-problèmes, soit en ce qui concerne la recombinaison des résultats des sous-problèmes pour construire le résultat final.

L'approche probabiliste consiste alors à trouver un moyen aléatoire permettant de découper le problème initial en sous-problèmes, de telle manière que leurs tailles soient équivalentes avec une bonne probabilité. Une telle découpe est intéressante si elle permet une découpe et une reconstruction très rapides (idéalement de temps parallèle constant).

5.1 Tri probabiliste

Par exemple, pour le problème du tri, l'introduction de redondance permet la construction d'un algorithme de tri non efficace (d'inefficacité polynomiale), mais de temps parallèle $O(\log n)$. La technique "diviser pour paralléliser" permet la construction directe d'un algorithme de tri par partition-fusion [20]. Mais, s'il est facile de découper en temps constant un tableau de taille n en deux sous-tableaux de taille $\frac{n}{2}$, la construction du tableau trié par fusion des deux tableaux triés obtenus après résolution est plus difficile. Typiquement, une fusion bitonique a une complexité $O_{//}(\log n, n)$. L'algorithme obtenu a donc un temps parallèle $\log^2 n$ – et non plus $O(\log n)$ comme le tri par sélection – et un travail $O(n \log^2 n)$: bien qu'efficace (inefficacité poly-logarithmique), il n'est donc pas optimal [6].

Le choix déterministe d'une partition rend donc ici difficile l'obtention d'un algorithme optimal.

Pourtant, l'algorithme de quicksort présenté au paragraphe 3.3 contient du parallélisme, et son travail – $O(n \log n)$ – est optimal. L'inconvénient est que, du point de vue parallèle, il est possible d'obtenir un mauvais choix de pivot à chaque étape de partitionnement, ce qui peut entraîner n étapes dans la découpe parallèle.

L'étude probabiliste consiste alors à étudier sous quelles conditions le nombre d'étapes est garanti avec une bonne probabilité comme étant poly-logarithmique,

6. Il est à noter que la fusion bitonique peut être rendue optimale [2], mais le temps parallèle reste $O(\log^2 n)$. Il existe par ailleurs des algorithmes parallèles de fusion de complexité temporelle $O(\log\log n)$ [11].

sans changer le travail de l'algorithme. Nous obtiendrons ainsi (5.2) un algorithme probabiliste de complexité

$$O_{//}\left(\log^2 n, \frac{n}{\log n}\right)$$

donc de travail optimal.

Remarque. Pour obtenir directement plus de parallélisme, on peut alors envisager le choix aléatoire parallèle de plusieurs pivots. Cette stratégie permet de construire un algorithme probabiliste de tri de complexité [11]

$$O_{//}(\log n, n)$$

donc non seulement de travail optimal mais aussi de temps parallèle optimal.

5.2 Quicksort parallèle

L'algorithme 3.3 se décompose de la manière suivante :

1. Calcul du pivot X_{piv} (tirage aléatoire) avec une complexité $O_{//}(1,1)$. Soit k la position du pivot dans le tableau trié.

2. Partition en deux sous-tableaux X_{inf} et X_{sup} : tous les éléments peuvent être comparés en parallèle à X_{piv} en coût $O_{//}(1,n)$. Pour ranger les éléments de X_{inf} dans les $k-1$ premières cases du tableau X, il suffit de leur associer une marque valant 1, les éléments de X_{sup} étant associés à une marque valant 0. Le calcul de la somme préfixée des marques (ou une application des sauts de pointeurs – *list ranking* –) permet alors de les ranger dans les $k-1$ premières cases de X avec un coût $O_{//}\left(\log n, \frac{n}{\log n}\right)$. On procède de la même façon pour ranger les éléments de X_{sup} dans les $(n-k)$ dernières cases de X, avec le même coût.

3. Tri récursif en parallèle des deux sous-tableaux X_{inf} et X_{sup}.

La complexité totale, après équilibre des travaux, de la partition – étapes 1 et 2 – est donc $O_{//}\left(\log n, \frac{n}{\log n}\right)$ et son travail est donc $O(n)$ sur une EREW-PRAM.

La complexité de l'algorithme est alors liée à la profondeur des appels récursifs – étape 3 –, *i.e.* au nombre de partitions qui sont effectuées séquentiellement.

Un mauvais choix systématique du pivot (qui délivre par exemple, à chaque partition, le plus petit élément du tableau à partitionner comme pivot) est possible : dans ce pire cas, le nombre d'étapes séquentielles de partition est $O(n)$. Nous allons montrer qu'un tel choix est très peu probable.

5.3 Analyse probabiliste de la complexité

Soit $\text{Prob}_{\text{échec}}(t)$ la probabilité que le tableau ne soit pas trié après t étapes de partition de l'algorithme parallèle. Soit $\text{Prob}^{(e)}_{\text{échec}}(t)$ la probabilité qu'un élément arbitraire e du tableau à trier soit mal placé après t étapes.
D'après l'inégalité de Boole (ou propriété de σ-additivité [8]), la probabilité de l'union de n événements est plus petite que la somme des probabilités de ces événements. On en déduit que :

$$\text{Prob}_{\text{échec}}(t) \leq n\text{Prob}^{(e)}_{\text{échec}}(t) \tag{1}$$

On veut montrer que la probabilité que le tableau ne soit pas trié après un nombre proportionnel à $\log n$ partitions est faible : posons $t = K \log n$. Il s'agit donc de déterminer une constante K qui soit telle que

$$\frac{\mathrm{Prob}^{(e)}_{\text{échec}}(t)}{n}$$

soit suffisamment petite.

Soit n_j la taille du sous-tableau contenant e après j étapes de partition. Le problème revient à calculer la probabilité avec laquelle, après une étape de partition sur n_j éléments, les deux sous-tableaux obtenus sont de taille proportionnellement plus petite que n_j, soit une taille plus petite que $\frac{3n_j}{2}$ ($\frac{n_j}{2}$ étant impossible). Une telle partition est appelée partition *réussie*.
Une partition échoue (n'est pas réussie) si et seulement si l'un des sous-tableaux est de taille plus petite que $\frac{n_j}{4}$, c'est-à-dire que le pivot a été tiré parmi les $\frac{n_j}{4}$ éléments les plus petits ou les $\frac{n_j}{4}$ éléments les plus grands. La probabilité d'un tel tirage est $\frac{1}{2}$. On en déduit donc :

$$Prob\left(n_{j+1} \leq \frac{3n_j}{4}\right) \geq \frac{1}{2} \tag{2}$$

Après $\log_{\frac{4}{3}} n$ étapes de partitions réussies, l'élément e est correctement placé dans le tableau trié.
Pour que l'élément e ne soit pas correctement placé après t partitions, il est donc nécessaire d'avoir effectué un nombre de partitions réussies strictement inférieur à $\log_{\frac{4}{3}} n$.
Les tirages successifs étant indépendants, les événements correspondant aux partitions réussies suivent une distribution de Bernouilli : cette distribution modélise le jeu de pile ou face – succès avec probabilité p ou échec avec probabilité $(1-p)$ –, avec ici une probabilité de succès $p = \frac{1}{2}$ (2).
Soit X_t une variable aléatoire qui dénombre les partitions réussies des sous-tableaux contenant l'élément e après les t premières étapes de partition, on a alors :

$$\mathrm{Prob}^{(e)}_{\text{échec}}(t) = \mathrm{Prob}(X_t \leq \log_{\frac{4}{3}} n) \tag{3}$$

Par propriété des distributions de Bernouilli avec une probabilité d'échec de p, X suit une distribution binomiale d'où :

$$\mathrm{Prob}(X_t \leq x) = \sum_{i=0}^{x} C_t^i p^i (1-p)^{t-i}.$$

Cette probabilité peut être majorée en utilisant la borne de Chernoff [8], qui permet de quantifier les événements rares – théorie des grandes déviations – :

$$\mathrm{Prob}\left(X_t \leq (1-\epsilon)pt\right) \leq e^{-\frac{\epsilon^2 tp}{2}} \tag{4}$$

où ϵ est une constante comprise entre 0 et 1.

Ici $p = \frac{1}{2}$, et $t = K \log_{\frac{4}{3}} n$, la base $\frac{4}{3}$ étant choisie par commodité. Posons $\epsilon = \frac{K-2}{K}$, qui est bien compris entre 0 et 1 en choisissant $K > 2$. En remplaçant dans (3) on obtient :

$$\mathrm{Prob}\left(X_t \leq \log_{\frac{4}{3}} n\right) \leq e^{-\frac{\epsilon^2 K \log_{\frac{4}{3}} n}{4}}. \tag{5}$$

Comme $e^{\log_{\frac{4}{3}} n} = n^{\frac{1}{\log \frac{4}{3}}}$, il reste d'après (1) à déterminer K tel que :

$$\left(\frac{K-2}{K}\right)^2 \frac{K}{\log \frac{4}{3}} > 1. \tag{6}$$

$K = 4$ convient. On en déduit donc que la probabilité que le tableau ne soit pas trié après $4 \log_{\frac{4}{3}} n$ étapes de l'algorithme parallèle tend vers 0 quand n tend vers $+\infty$. Comme la complexité d'une étape est $O_{//}\left(\log n, \frac{n}{\log n}\right)$, la complexité totale de l'algorithme probabiliste de tri proposé est avec une grande probabilité

$$O_{//}\left(\log^2 n, \frac{n}{\log n}\right),$$

et cet algorithme est de travail optimal.

6 Homomorphisme sur les entiers

Nous avons vu qu'un algorithme probabiliste PRAM est toujours construit à partir d'un oracle générant des entiers aléatoires (ou des bits dans le cas des circuits). Pour introduire des entiers dans un problème $\mathcal{P}$ – de décision ici [7] – défini dans un ensemble E, une méthode consiste à transformer le problème initial pour le ramener à un problème équivalent (isomorphe) $\mathcal{P}_{\mathbb{Z}}$ dans les entiers – ou les rationnels –, en construisant un homomorphisme $\Phi : E \to \mathbb{Z}$ adapté. Si le problème $\mathcal{P}_{\mathbb{Z}}$ calcule en sortie un entier $s_{\mathbb{Z}}$, la solution s au problème $\mathcal{P}$ initial doit alors s'exprimer comme un test de nullité de cet entier, par exemple $s = (s_{\mathbb{Z}} =?0)$.

Pour obtenir un algorithme parallèle pour résoudre $\mathcal{P}$, il faut construire un algorithme parallèle $\mathcal{A}_{\mathbb{Z}}$ pour résoudre le problème $\mathcal{P}_{\mathbb{Z}}$. Nous supposerons dans ce qui suit, pour éviter les problèmes [21], que $\mathcal{A}_{\mathbb{Z}}$ ne contient pas de branchements – ou au plus un nombre fini indépendamment de la taille des entrées – : $\mathcal{A}_{\mathbb{Z}}$ est donc un circuit arithmétique et non un circuit booléen-arithmétique [28].
Généralement, les entiers manipulés par l'algorithme $\mathcal{A}_{\mathbb{Z}}$ peuvent être grands. Par suite, même si l'algorithme $\mathcal{A}_{\mathbb{Z}}$ a un travail optimal en terme d'opérations arithmétiques sur $\mathbb{Z}$, il peut s'avérer très inefficace en arithmétique booléenne, si les entiers sont plus grands que $O(1)^n$. Pour limiter le coût de l'arithmétique entière, on peut alors effectuer les calculs *modulairement*, *i.e.* calculer $\mathcal{A}_{\mathbb{Z}}$ dans $\mathbb{Z}/p\mathbb{Z}$, où p est un entier choisi aléatoirement (plus petit que $n^{O(1)}$). p peut être choisi premier ou non, selon les besoins (notamment si l'algorithme nécessite ou non des divisions). On construit ainsi un algorithme de Monte-Carlo :

1. tirer au hasard un entier p dans $\{1, \ldots, M\}$,
2. effectuer les calculs de l'algorithme $\mathcal{A}_{\mathbb{Z}}$ dans $\mathbb{Z}/p\mathbb{Z}$: on obtient en sortie $s_{\mathbb{Z}p} = s_{\mathbb{Z}} \mathrm{mod} p$,
3. si $s_{\mathbb{Z}p} \neq 0$ alors $s_{\mathbb{Z}} \neq 0$: on peut alors calculer sans échec la solution du problème $\mathcal{P}$.
   Sinon, on ne peut pas décider si $s_{\mathbb{Z}}$ est nul ou pas, puisque, si cet entier est un multiple non nul de p, l'algorithme modulaire renvoie $s_{\mathbb{Z}p} = 0$, alors que $s_{\mathbb{Z}} \neq 0$: on a donc ici une possibilité d'échec.

Nous montrerons, sur l'exemple des chaînes de caractères, que la probabilité d'échec de cet algorithme de Monte-Carlo peut être rendue petite lorsque la taille des entiers manipulés par l'algorithme déterministe $\mathcal{A}_{\mathbb{Z}}$ est bornée par $n^{O(1)}$.

Obtention d'un algorithme de Monte-Carlo efficace. Une remarque importante est que, si l'algorithme $\mathcal{A}_{\mathbb{Z}}$ a un travail optimal en terme d'opérations

7. Cette technique est généralisable, comme nous le verrons dans l'exemple du filtrage de chaînes, à des problèmes qui ne se décrivent pas initialement comme un problème de décision.

arithmétiques sur $\mathbb{Z}$, alors l'algorithme de Monte-Carlo est efficace : son inefficacité est au plus poly-logarithmique, les opérations modulo p pouvant être effectuée séquentiellement en temps inférieur à $O(\log p \log\log p \log\log\log p)$, donc borné par $\log^{1+\epsilon} n$, avec $\lim_{n\to\infty} \epsilon = 0$. De plus, il est toujours possible de précoder les calculs dans une table (à $M \times M$ entrées). L'utilisation de la technique "2 pour 3" pour réduire les additions permet, si le circuit $\mathcal{A}_{\mathbb{Z}}$ n'admet que des chemins comportant un nombre borné de multiplications, d'ajouter alors seulement un terme $O(\log n)$ à la complexité de l'algorithme, qui ne change donc pas son travail. Dans la suite, nous supposerons que la complexité du produit de deux entiers de $n^{O(1)}$ bits est $O(\log n)$, en négligeant le facteur $\log\log n \log\log\log n$ dû à cette opération.

Transformation en algorithme de Las Vegas. S'il existe un algorithme pour vérifier la justesse de la solution, il peut être transformé en algorithme de Las Vegas. Par ailleurs, il est toujours possible de le transformer en un algorithme déterministe, par utilisation du théorème chinois des restes : il suffit de lancer plusieurs exécutions de l'algorithme avec suffisamment de nombres – relativement premiers ici –, pour pouvoir remonter par interpolation la solution du problème sur les entiers : mais si les entiers manipulés sont grands, on obtient un travail exponentiel par rapport à celui de l'algorithme probabiliste.

Cette technique de calcul par homomorphisme est illustrée ci-après sur le problème de la recherche de chaîne de caractères dans un fichier.

6.1 Filtrage de chaînes de caractères

Soit $E = (\{0,1\}^\star, \wedge, \epsilon)$ l'ensemble des chaînes de caractères (codées en binaire), $\wedge$ désignant l'opération de concaténation de chaînes et ϵ la chaîne vide (*monoïde*). Étant données en entrée deux chaînes $Y = (y_0, \ldots, y_n)$ et $X = (x_0, \ldots, x_m)$ (on supposera $m \leq n$), le problème consiste à trouver toutes les occurrences de la chaîne X dans Y, autrement dit l'ensemble K :

$$K = \{k \leq n / y_k \ldots y_{k+m} = X\}$$

Une étude complète de ce problème est effectuée dans [1]. En séquentiel, le premier algorithme optimal de complexité $O(n+m)$ est dû à Boyer et Moore [5], mais il est peu parallèle. Il est facile de construire un algorithme parallèle en introduisant de la redondance : il suffit de comparer, en parallèle pour tout k, la chaîne $Y_k = (y_k \ldots y_{k+m})$ à la chaîne X – avec un coût $O_{//}\left(\log m, \frac{nm}{\log m}\right)$ – puis de compresser les résultats obtenus (par un algorithme de list-ranking ou de somme préfixée, de la même façon que pour le quicksort) pour former l'ensemble K avec un coût $O_{//}(\log n, \frac{n}{\log n})$. Le problème appartient donc à $\mathcal{NC}^\infty$, mais l'algorithme ainsi construit n'est pas efficace. L'inefficacité provient ici de la localisation des occurrences de X dans Y (de travail nm) et non de la compression (de travail n).

Nous présentons ici un algorithme probabiliste efficace pour localiser toutes les occurrences de X dans Y, dû à Karp et Rabin [14] [1]. Des extensions de cet algorithme pour des problèmes de filtrage multi-dimensionnel sont présentés dans [11].

6.1.1 Plongement du problème dans les entiers

L'espace E de départ étant muni d'une loi $\wedge$ associative, non commutative et qui possède un élément neutre ϵ, il est nécessaire de construire un espace F – construit à partir des entiers – qui possède au moins une telle opération (et éventuellement une structure plus riche).

L'espace des matrices carrées de dimension 2, muni du produit de matrices, a une telle structure. Pour pouvoir rendre l'opération de concaténation inversible, on plonge le problème dans l'espace $\mathcal{E}$

$$\mathcal{E} = \{\mathcal{M} \in \mathcal{M}_{\in,\in} / \det(\mathcal{M}) = \infty\}$$

des matrices de déterminant 1, ce qui permettra de les inverser. Dans la suite, on note I la matrice identité de $\mathcal{E}$. L'homorphisme de structure Φ doit vérifier :

$$\begin{cases} \Phi(\epsilon) = I \\ \Phi(a \wedge b) = \Phi(a)\Phi(b) \quad \forall (a,b) \in E^2 \end{cases} \tag{7}$$

Φ est alors complètement caractérisé par la donnée de $\Phi('0')$ et $\Phi('1')$. Pour que Φ soit injective – de façon à ce que deux chaînes distinctes soient associées à deux matrices distinctes – on pose :

$$\Phi('0') = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \qquad \Phi('1') = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{8}$$

Il est facile de montrer que Φ ainsi construite est un homorphisme injectif de structure, donc que E est isomorphe à $\Phi(E) \subset \mathcal{E}$.

Remarque. Concaténer un '0' à gauche d'une séquence s peut se calculer de la façon suivante :

$$\Phi('0' \wedge s) = \Phi('0')\Phi(s) = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \begin{pmatrix} a & b \\ c & d \end{pmatrix} = \begin{pmatrix} a & b \\ a+c & b+d \end{pmatrix}$$

ce qui équivaut à remplacer la deuxième ligne de $\Phi(s)$ par la somme de ses deux lignes.
De même,

- concaténer un '1' à gauche d'une séquence s équivaut à remplacer la ligne 1 par la somme des 2 lignes,
- concaténer un '0' à droite d'une séquence s équivaut à remplacer la colonne 1 par la somme des 2 colonnes,
- concaténer un '1' à droite d'une séquence s équivaut à remplacer la colonne 2 par la somme des 2 colonnes.

Soit $Y_k = (y_k \dots y_{k+m})$, $M_X = \Phi(X)$ et $M_k = \Phi(Y_k)$. Les matrices M_k peuvent être calculées optimalement en parallèle (en comptant le nombre d'opérations sur les entiers, et non sur les booléens).
En effet : posons $A_k = \Phi(y_0 \dots y_k)$, $0 \leq k \leq n$. Les matrices A_k sont les préfixes – pour l'opération produit de matrices dans $\mathcal{E}$ – des matrices $\Phi(y_k)$. Elles peuvent donc être calculées avec un coût $O_{//}\left(\log n, \frac{n}{\log n}\right)$: la remarque précédente montre en effet que l'opérations de base – le produit de deux matrices $\Phi(a)$ et $\Phi(b)$ – se ramène à 2 additions.
Posons $A_{-1} = I$. On a alors :

$$M_k = A_{k-1}^{-1} A_{k+m} \qquad \forall\, 1 \leq k \leq n$$

et les matrices M_k peuvent être obtenues à partir des matrices A_k avec un coût de $O_{//}\left(\log n, \frac{n}{\log n}\right)$ et donc un travail $O(n)$ optimal.

L'algorithme s'écrit finalement :

1. Calcul de $\Phi(x_1, \dots, x_m) = M_X$

2. Calcul de $A_k = \Phi('y_0 \dots y'_k)$, $0 \leq k \leq n$ (calcul de préfixes)

3. Calcul de A_k^{-1}, inverse de A_k (de déterminant 1)

4. Calcul de $M_k = A_{k-1}^{-1} A_{k+m}$, $0 \leq k \leq n$

5. Pour $k = 0, \dots, n-m$, comparaison de M_k et M_X, pour déterminer toutes les occurrences de X dans Y.

Par récurrence, si s est une chaîne de i caractères, les coefficients de $\Phi(s)$ sont bornés par 2^i. Par suite, les entiers manipulés par cet algorithme sont plus petits que 2^n, et donc très grands. La technique introduite dans cette section consiste alors à construire un algorithme probabiliste de Monte-Carlo en effectuant non pas les opérations arithmétiques exactes (trop coûteuses), mais modulo un nombre p petit, tiré au hasard, plus petit que $n^{O(1)}$ (*i.e.* ayant au plus $O(\log n)$ bits).

Cela revient à remplacer Φ par l'homomorphisme Φ_p :

$$\Phi_p(x) = \Phi(x) \mod p \qquad\qquad \forall x \in E$$

Φ_p n'étant alors plus injectif, il est possible d'avoir deux chaînes a et b distinctes qui vérifient $\Phi_p(a) = \Phi_p(b)$. Dans la suite, nous allons montrer que ceci ne peut se produire qu'avec une probabilité inférieure à une constante.

L'algorithme précédent, avec Φ_p, délivre donc en sortie un sur-ensemble de K : certaines sous-chaînes de Y obtenues en sortie sont différentes de X, mais toutes les occurrences de X apparaissent en sortie. On a donc ainsi construit un algorithme de Monte-Carlo : l'analyse de complexité présentée ci-dessous montre que sa probabilité d'échec peut être rendue aussi petite que désiré. Il est donc dans REREW[1] et de travail optimal.

Pour le transformer en un algorithme de Las Vegas, il suffit de vérifier que les sorties de l'algorithme de Monte-Carlo sont bien des occurrences de X, en examinant une à une les chaînes de caractères obtenues.

6.1.2 Analyse de la complexité

Soit p un nombre premier et Φ_p l'homomorphisme de structure induit.

Soit $\tilde{K} = \{k / Y_k \neq X\}$. Le cardinal de $\tilde{K}$ est plus petit que $(n-m)$. L'algorithme proposé filtre incorrectement (*échec*) toutes les chaînes Y_k $(k \in \tilde{K})$ qui vérifient :

$$\Phi_p(Y_k) - \Phi_p(X) = 0$$

Posons $N_k = \Phi(Y_k) - \Phi(X)$: il y a donc échec s'il existe $k \in \tilde{K}$ tel que :

$$N_k \mod p = 0,$$

ce qui équivaut à dire que p divise $N_{X,Y} = \prod_{k \in \tilde{K}} N_k$. La probabilité que l'algorithme de Monte-Carlo produise un échec est donc bornée par :

$$\text{Prob(échec)} = \text{Prob}(p \text{ divise } N_{X,Y})$$

Nous avons vu que, si s est une chaîne de longueur i, les coefficients de $\Phi(s)$ sont bornés par 2^i. Par suite, $N_{X,Y} \leq 2^{nm}$.

$N_{X,Y}$ admet donc au plus nm facteurs premiers.

Soit p un nombre premier *tiré uniformément* dans l'ensemble des nombres premiers plus petits que α, où α est un entier donné (plus petit que $n^{O(1)}$). Le nombre $\pi(\alpha)$ d'entiers premiers plus petits que α vérifie :

$$\frac{\alpha}{\log_e \alpha} \leq \pi(\alpha) \leq 1.3 \frac{\alpha}{\log_e \alpha}$$

La probabilité que l'algorithme de Monte-Carlo produise un échec est donc bornée par :

$$\text{Prob}(\text{échec}) \leq \frac{nm}{\pi(\alpha)}$$

Posons $\alpha = n^2m$. La probabilité d'échec, lorsque p est un nombre premier tiré uniformément dans $\{2, \ldots, n^2m\}$ est donc bornée par

$$\frac{3\log n}{n}$$

qui tend asymptotiquement vers 0.

Exemple. Pour $n = 10^6$ $m = 10^2$: si l'on tire p au hasard parmi les nombres premiers plus petits que 10^{14} (codables exactement dans un flottant double précision), la probabilité d'échec de l'algorithme de Monte-Carlo proposé est donc inférieure à 10^{-4}.

On a donc obtenu un algorithme de Monte-Carlo dans EREW[1] qui filtre toutes les occurrences d'une chaîne de longueur m dans une chaîne de longueur n avec une complexité

$$O_{//}\left(\log n, \frac{n}{\log n}\right)$$

et donc un travail optimal.

6.1.3 De Monte-Carlo à Las Vegas

On suppose ici que la chaîne X n'est pas périodique : il y a donc au plus $\frac{n}{m}$ occurrences de X dans Y.

Pour rendre l'algorithme précédent déterministe, il suffit de vérifier que toutes les chaînes trouvées par l'algorithme de Monte-Carlo précédent sont bien égales à la chaîne X. La comparaison de l'égalité de deux chaînes de longueur m peut se faire optimalement en $O_{//}\left(\log m, \frac{m}{\log m}\right)$.

Mais il se peut que l'algorithme de Monte-Carlo délivre $O(n)$ occurrences, avec une très faible probabilité comme nous l'avons montré précédemment. Dans ce cas, le coût de la vérification déterministe est $O_{//}\left(\log m, \frac{nm}{\log m}\right)$, et rend l'algorithme inefficace.

Il est donc préférable de construire, à partir de cette vérification possible, un algorithme de Las Vegas :

1. Appliquer l'algorithme de Monte-Carlo précédent.
2. Si l'algorithme délivre en sortie plus de $\frac{n}{m}$ occurrences alors délivrer en sortie un constat d'échec.
3. Sinon, vérifier toutes les occurrences trouvées, et délivrer en sortie toutes les occurrences Y_k égales à X.

La complexité de cet algorithme est

$$O_{//}\left(\log n, \frac{n}{\log n}\right)$$

et il est donc optimal. De plus, il ne délivre un constat d'échec qu'avec une probabilité inférieure à celle d'échec de l'algorithme de Monte-Carlo, donc très faible.

7 Test de nullité d'un polynôme

De nombreux problèmes peuvent être ramenés au problème de décidabilité de la nullité d'une expression algébrique [28], dont les coefficients sont définis à partir des entrées. Le test de nullité d'un polynôme peut être ramené au problème de l'évaluation de ce polynôme, donc d'une expression ou d'un programme arithmétique. Des techniques générales [21] permettent de construire des algorithmes parallèles déterministes qui évaluent efficacement un polynôme, même représenté sous forme de circuit.

Le problème est que, du fait du nombre d'indéterminées du polynôme, il peut être nécessaire de l'évaluer en un nombre exponentiel de points pour décider de manière déterministe de sa nullité.

La propriété suivante, dûe à Schwartz [25] (dont une démonstration peut être trouvée dans [11]), permet de construire un algorithme de Monte-Carlo pour décider de la nullité d'un polynôme à partir de son évaluation en un nombre réduit de points.

Propriété [25]. *Soit $P(x_1, \ldots, x_n)$ un polynôme à n variables, à coefficients dans un corps K, et de degré d et soit I un sous-ensemble fini de K de cardinal c. Soit $(\alpha_1, \ldots, \alpha_n)$ un n-uplet de K^n tiré uniformément dans I^n.*
Si P n'est pas identiquement nul, alors :

$$Prob(P(\alpha_1, \ldots, \alpha_n) = 0) \leq \frac{d}{c} \tag{9}$$

Cette propriété permet la construction d'un algorithme de Monte-Carlo pour un problème donné dès qu'il est possible de le réduire au test de nullité d'un polynôme. Il suffit de construire un algorithme parallèle qui permet d'évaluer le polynôme en un point, avec en entrée de l'algorithme ce point. En tirant aléatoirement les valeurs des entrées de l'algorithme d'évaluation, et en le calculant sur ces entrées aléatoires, on obtient un algorithme probabiliste. Pour avoir une probabilité d'erreur inférieure à $\frac{1}{2}$ (ou qui tend vers 0 quand la taille du problème augmente), il suffit d'effectuer le tirage aléatoire des valeurs dans un sous-ensemble suffisamment grand.

Cette technique algébrique s'applique à de nombreux problèmes, et notamment en théorie des graphes [11] et en calcul algébrique [4]. Deux exemples en algèbre linéaire sont présentés ci-après : la vérification d'un produit de matrices et le calcul du rang d'une matrice.

7.1 Vérification probabiliste du produit de deux matrices

Soient A, B et C trois matrices carrées de dimension n à coefficients dans un anneau R. Le problème consiste ici à vérifier si $AB = C$.

Un algorithme déterministe pour ce problème revient à calculer un produit de matrices, et a donc une complexité arithmétique[8]

$$O_{//}\left(\log n, \frac{M(n)}{\log n}\right).$$

La propriété (9) permet de construire un algorithme de Monte-Carlo de complexité arithmétique

$$O_{//}\left(\log n, \frac{n^2}{\log n}\right)$$

donc beaucoup plus performant, en le ramenant à des produits matrice-vecteur.

8. $M(n)$ étant le nombre de processeurs nécessaires au calcul d'un produit de matrices carrées de dimension n en temps $O(\log n)$: $n^2 \leq M(n) \leq n^3$.

Il est clair que $AB = C$ si et seulement si

$$(AB - C)x = 0 \quad \forall x \in R^n. \tag{10}$$

Posons $x = [x_1, \ldots, x_n]^t$ et $P(x) = ABx - Cx$. L'équation (10) se ramène à vérifier si le polynôme P, à n indéterminées $(x_1, \ldots, x_n)$ et de degré 1, est identiquement nul.

Cette propriété permet alors d'écrire l'algorithme de Monte-Carlo suivant. Soit I un sous-ensemble fini de R, de cardinal c.

1. Tirer uniformément un vecteur $\alpha = [\alpha_1, \ldots, \alpha_n]^t$ dans I^n.
2. Calculer $u_1 = B\alpha$, $u_2 = Au_1$ et $u_3 = C\alpha$. Ce calcul, qui consiste en trois produits matrice vecteur, a un coût : $O_{//}\left(\log n, \frac{n^2}{\log n}\right)$.
3. Si $u_2 - u_3 = 0$ alors retourner *vrai* sinon retourner *faux*.

D'après (9), la probabilité d'échec de cet algorithme (cas où il renvoie *vrai* alors que $AB \neq C$) est bornée par $\frac{1}{c}$. En choisissant $I = \{0, 1\}$ (éléments neutres pour l'addition et la multiplication dans R), on obtient donc une probabilité d'échec pour cet algorithme inférieure à $\frac{1}{2}$. En réitérant k fois cet algorithme, on obtient une probabilité d'échec inférieure à $\frac{1}{2^k}$.

On a ainsi construit un algorithme de Monte-Carlo dans EREW[1], et de travail optimal $O(n^2)$.

7.2 Rang et formes normales

Soit A une matrice carrée de dimension n à coefficients dans un corps K. On s'intéresse à calculer le rang de A, c'est-à-dire le nombre de lignes linéairement indépendantes dans A. Nous présentons ici l'algorithme de Las Vegas introduit dans [4] pour résoudre ce problème.

Le rang de A est la dimension du mineur de A inversible de plus grande dimension. Autrement dit, si A est de rang r, il existe une matrice de permutation de lignes L et une matrice de permutation de colonnes C qui sont telles que le mineur principal de dimension r de la matrice LAC est de déterminant non nul, alors que les déterminants de tous les autres mineurs principaux de dimension supérieure à r de LAC sont nuls.

7.2.1 Un algorithme de Monte-Carlo

Soit r le rang de A, et soient $(L_{i,j})$ et $(C_{i,j})$ $(1 \leq i, j \leq n)$ les coefficients des matrices L et C.

Soit Δ_i le mineur principal de dimension i de LAC, et $\delta_i(L, C)$ son déterminant. δ_i s'écrit comme un polynôme, dont les indéterminées sont les $(L_{i,j})$ et $(C_{i,j})$, et de degré $2n$. Cela provient du fait que les coefficients de LAC sont de degré 2, et que le déterminant d'une matrice est une forme multi-linéaire de ses coefficients.

Nous supposons ici pour simplifier que K est de cardinal infini. Par propriété du rang, il existe au moins une paire de matrices (L, C) telle que les mineurs principaux de dimension strictement inférieure à r de LAC sont de déterminant non nuls. Par suite, les polynômes δ_i, $1 \leq i \leq r$, ne peuvent pas être identiquement nul :

$$\delta_i(L, C) \neq 0 \quad \text{pour} \quad 1 \leq i \leq r.$$

Le rang de A est alors le plus grand entier i, $1 \leq i \leq n$, tel que le polynôme δ_i n'est pas identiquement nul.

Soit I un sous-ensemble fini de K, de cardinal c. De la même façon que pour l'exemple précédent, il est alors possible de construire l'algorithme de Monte-Carlo suivant pour calculer le rang :

1. Tirer aléatoirement deux matrices L et C, *i.e.* $2n^2$ coefficients dans I ($L_{i,j}$ et $C_{i,j}$, $1 \leq i, j \leq n$).
2. Calculer le produit $D = LAC$.
3. Pour $i = 1, \ldots, n$, calculer $d_i = \det(D_i)$, et poser $d_0 = 1$.
4. Retourner $s = \mathrm{Max}_{k=0,\ldots,n}\{k/d_k \neq 0\}$.

Dans tous les cas, cet algorithme renvoie un entier plus petit que le rang (par exemple si l'on tire pour L la matrice nulle). Il y a échec lorsque l'entier retourné est strictement plus petit que le rang r de la matrice A. La probabilité de cet échec est celle que l'évaluation d_i du polynôme δ_i donne une valeur nulle, alors que le polynôme δ_i n'est pas identiquement nul. D'après (9), la probabilité d'un tel échec est bornée par $\frac{2n}{c}$.

En choisissant pour I un sous-ensemble de cardinal $c = 4n$ (c est bien borné par un polynôme en n) on obtient donc un algorithme de Monte-Carlo de probabilité d'échec bornée par $\frac{1}{2}$. Les tirages étant indépendants, pour avoir une probabilité d'échec inférieure à $\frac{1}{2^k}$, il suffit alors d'appliquer (successivement ou en parallèle) k fois l'algorithme précédent, et de retourner comme résultat le maximum des entiers retournés par les k exécutions, ou de choisir parmi $2^{k+1}n$ éléments.

Finalement, comme le calcul du déterminant – dont le coût domine la complexité de l'algorithme – appartient à EREW_K^2, on en déduit que le calcul du rang dans un corps K quelconque[9] appartient à REREW_K^2.

7.2.2 De Monte-Carlo à Las Vegas

On utilise ici comme point de départ l'algorithme de Monte-Carlo (M_k) précédent de probabilité d'échec $\frac{1}{2^k}$, obtenu soit par un choix de I de cardinal $c = 2^{k+1}n$, soit par k exécutions de l'algorithme.

Soit s le rang retourné par (M_k), avec A en entrée. Pour construire un algorithme de Las Vegas, il reste à exhiber un mineur de A inversible, et de montrer que sa dimension s est maximale.

L'algorithme (M_k) permet facilement de construire un sous-ensemble I_L, de cardinal s, de lignes linéairement indépendantes. Soit $s_i = \mathrm{Max}_{k=0,\ldots,i}\{k/d_k \neq 0\}$, $i = i, \ldots, n$. Il suffit de choisir pour I_L tous les indices i tels que $s_i > s_{i-1}$: en effet, la ligne i est alors linéairement indépendante avec toutes les lignes d'indices inférieurs.
Soit alors A', la matrice carrée de dimension n dont la ligne i est la ligne i de A lorsque $i \in I_L$, et nulle sinon. Soit s' le rang calculé par l'algorithme (M_k), avec A' en entrée.

Une technique analogue à celle utilisée pour calculer I_L permet de calculer un sous-ensemble I_C de colonnes de A linéairement indépendantes.

Si I_L et I_C sont de cardinaux différents, alors il y a eu échec et l'algorithme de Las Vegas retourne un constat d'échec.

Sinon, un candidat comme mineur maximal de A est la sous-matrice carrée A_s de A de dimension s, formée par les coefficients de A d'indices de lignes dans I_L et de colonnes dans I_C. A_s est inversible. Il reste donc à montrer que ce mineur est

9. **Cas où K est de cardinal fini.** Si K est de cardinal fini, [4], le choix $I = K$ ne suffit pas toujours pour assurer une probabilité d'échec inférieure à une constante strictement inférieure à 1. De manière générale, la technique consiste alors à augmenter artificiellement le cardinal de K en le plongeant dans une extension de cardinal suffisant [10].

maximal. Soit $\tilde{A}_s(i,j)$, pour $i \notin I_L$ et $j \notin I_C$ la matrice de carrée de dimension $s+1$, formée par la matrice A_s bordée en bas (resp. à droite) par les coefficients de la ligne i (resp. de la colonne j) de A et d'indices de colonne dans I_C (resp. de ligne dans I_L). A_s est un mineur de rang maximal si et seulement si toutes les matrices $\tilde{A}_s(i,j)$ ($i \notin I_L$, $j \notin I_C$), sont de déterminant nul.

Si tel est le cas, l'algorithme de Las Vegas retourne s comme rang, et sinon retourne un constat d'échec.

La probabilité du constat d'échec est alors plus petite – d'après l'inégalité de Boole – que $2\frac{1}{2^k}$, donc plus petite que $\frac{1}{2}$ pour $k=2$.

La complexité de cet algorithme de Las Vegas pour le rang est

$$O_{//}\left(\log^2 n, nM(n)\right).$$

Il est donc d'inefficacité polynomiale, puisqu'en séquentiel, à partir d'une adaptation de l'algorithme de Strassen, la complexité de ce problème est $O(M(n))$.

7.2.3 Algorithmes déterministes et formes normales

Dans l'algorithme de Las Vegas précédent, même si le travail de l'algorithme n'est pas augmenté de plus d'un facteur constant par rapport à l'algorithme de Monte-Carlo, la construction de la vérification introduit des calculs supplémentaires et la construction d'objets qui ne sont pas nécessités par l'algorithme de Monte-Carlo.

Cet algorithme est important : la technique présentée ici est à la base des algorithmes probabilistes efficaces connus pour ce problème. Il est à noter que le calcul du rang d'une matrice dans un corps quelconque a été montré dans NC^2_F (algorithme déterministe de Mulmuley [18]), en utilisant une toute autre caractérisation du problème. Mais aucun algorithme déterministe efficace n'est connu pour ce problème.

Ces deux remarques sont générales en algèbre linéaire, et tout particulièrement pour le calcul des formes normales :

- à partir d'algorithmes de Monte-Carlo efficaces pour le calcul de la forme de Jordan par exemple [10], la construction d'un algorithme de Las Vegas pour les mêmes problèmes nécessite la construction des matrices de passage. Cette construction est coûteuse.
- des algorithmes probabilistes [12] pour ce problème ont été trouvés avant des algorithmes déterministes [24]. Des algorithmes probabilistes efficaces sont connus [10].

Notons que, de même que pour le rang, pour Jordan encore des algorithmes déterministes mais inefficaces existent qui utilisent de tout autres méthodes pour résoudre le problème [24, 9].

8 Conclusion

Nous avons introduit dans ce chapitre des techniques générales qui permettent la construction d'algorithmes probabilistes : "diviser pour régner" probabiliste, homomorphisme sur les entiers, vérification d'identités polynômiales. Ces techniques sont utiles pour construire :

- des algorithmes parallèles pour des problèmes pour lesquels de tels algorithmes ne sont pas connus en déterministe,
- des algorithmes performants, de travail optimal ou efficace.

Ces techniques peuvent être utilisées pour un très large spectre de problèmes, et tout particulièrement en algèbre linéaire [3] et en combinatoire [11].

Les différents exemples didactiques explicités dans ce chapitre montrent que de tels algorithmes sont basés sur une approche du problème différente de celles visant directement à la construction d'algorithmes déterministes : en algorithmique probabiliste, on s'intéresse à exhiber des instances du problème pour lesquelles il est possible de construire une solution déterministe performante, et auxquelles une transformation aléatoire d'une entrée quelconque permet de se ramener avec une probabilité suffisante.

On peut alors s'intéresser à rendre déterministe un algorithme probabiliste, en se ramenant de manière déterministe et efficace à ces "bonnes" instances. Sous certaines hypothèses particulières, des transformations systématiques ont été exhibées [17] pour rendre déterministe un algorithme probabiliste, sans augmenter d'un facteur plus que poly-logarithmique le travail de l'algorithme.

Références

[1] Aho (A.-V.). – Algorithms for finding patterns in strings. *In : Algorithms and Complexity*, éd. par van Leuwen (J.), pp. 253–300. – Elsevier, 1990.

[2] Bilardi (G.) et Nicolau (A.). – Bitonic Sorting with $O(n \log n)$ Comparisons. *In : Proc. 20th Conf. Information Science and Systems*, pp. 320–326. – Princeton, NJ, 1986.

[3] Bini (D.) et Pan (V.). – *Numerical and Algebraic Computations with Matrices and Polynomials*. – Birkhauser, 1992.

[4] Borodin (A.), von zur Gathen (J.) et Hopcroft (J.). – Fast parallel matrix and gcd computations. *Information and Control*, vol. 52, 1982, pp. 241–256.

[5] Boyer (R.-S.) et Moore (J.-S.). – A Fast String Searching Algorithm. *Communications ACM*, vol. 20, n° 10, 1977, pp. 62–72.

[6] Brassard (G.) et Bratley (P.). – *Algorithmique : Conception et Analyse*. – Masson, 1987.

[7] Csánky (L.). – Fast parallel matrix inversion algorithms. *SIAM J. Comput.*, vol. 5, 1976, pp. 618–623.

[8] Dacunha-Castelle (D.) et Duflo (M.). – *Probabilités et Statistiques. Tome 1.* – Masson, 1982.

[9] Gautier (T.) et Roch (J.). – PAC++ System and Parallel Algebraic Numbers Computation. *In : First International Symposium on Parallel Symbolic Computation (PASCO'94)*, éd. par Hong (H.), p. 145.

[10] Giesbrecht (M.). – *Nearly optimal algorithms for canonical matrix forms* . – Thèse de PhD, Department of Computer Science, University of Toronto, 1993.

[11] Jájá (J.). – *An Introduction to Parallel Algorithms*. – Addison-Wesley, 1992.

[12] Kaltofen (E.), Krishnamoorthy (M.-S.) et Saunders (B.). – Parallel algorithms for matrix normal forms. *Linear Algebra and its Applications*, vol. 136, 1990, pp. 189–208.

[13] Kaltofen (E.) et Pan (V.). – Processor efficient parallel solutions of linear systems over an abstract field. *In : Proc. Third Annual ACM Symposium on Parallel Algorithms and Architectures*. pp. 180–191. – ACM Press.

[14] Karp (R.-M.) et Rabin (M.-O.). – Efficient Randomized Pattern-Matching Algorithms. *IBM J. Reserach Develop.*, vol. 31, n° 2, 1987, pp. 249–260.

[15] Karp (R.-M.) et Ramachandran (V.). – Parallel algorithms for shared-memory machines. *In : Algorithms and Complexity*, éd. par van Leuwen (J.), pp. 869–932. – Elsevier, 1990.

[16] Kruskal (C.-P.), Rudolph (L.) et Snir (M.). – *A Complexity Theory of Efficient Parallel Algorithms.* – Rapport technique n° RC 13572, IBM Research Division, 1988.

[17] Luby (M.). – Removing Randomness in Parallel Computation without a Processor Penalty. *J. Computer and System Sciences*, vol. 47, 1993, pp. 250–286.

[18] Mulmuley (K.). – A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, vol. 7, n° 1, 1987, pp. 101–104.

[19] Pan (V.). – Complexity of computations with matrices and polynomials. *SIAM Review*, vol. 34, n° 2, June 1992, pp. 225–262.

[20] Plateau (B.), Rasse (A.), Roch (J.-L.) et Verjus (J.-P.). – Parallélisme. – 1994. Polycopié ENSIMAG, Grenoble, France – Première édition en 1991 –.

[21] Revol (N.). – *Complexité de l'évaluation parallèle des circuits arithmétiques.* – Thèse de PhD, INP Grenoble, Août. 1994.

[22] Roch (J.-L.). – Complexité Parallèle et Algorithmique PRAM. *In : Algorithmes Parallèles : Analyse et Conception*, éd. par C. Roucairol et al., chap. 5, pp. 105–126. – Hermès, 1994.

[23] Roch (J.-L.) et Trystram (D.). – Méthodologies pour la programmation efficace d'applications parallèles. *La lettre des calculateurs distribués - numéro spécial Actes de l'Ecole SPI Lyon juillet 94*, 1994.

[24] Roch (J.-L.) et Villard (G.). – Parallel computations with algebraic numbers, a case study: Jordan normal form of matrices. *In : Parallel Architectures and Languages Europe 94, Athens Greece.*

[25] Schwartz (J.). – Fast Probabilistic Algorithms for Verification of Polynomial Identities. *J. ACM*, vol. 27, n° 4, 80 1980, pp. 701–717.

[26] Snir (M.). – Scalable Parallel Computers and Scalable Parallel Codes : From Theory to Practice. *In : Parallel Architectures and their Efficient Use.* pp. 176–184. – Springer Verlag.

[27] Strassen (V.). – Gaussian elimination is not optimal. *Numerische Math.*, 1969, pp. 354–356.

[28] von zur Gathen (J.). – Parallel arithmetic computations : a survey. *In : Proc. 12th Int. Symp. Math. Found. Comput. Sci., Bratislava.* pp. 93–112. – LNCS 233, Springer-Verlag.