Complexité Parallèle et Algorithmique PRAM

DEA Mathématiques Appliquées et DEA Informatique
Université J. Fourier - Grenoble

Jean-Louis Roch
LMC-IMAG, 46 Av. F. Viallet F38031 Grenoble Cedex
Jean-Louis.Roch@imag.fr

Octobre 1995

1. Je remercie Gilles Villard, Nathalie Revol et Thierry Gautier pour leur participation à la construction de ces chapitres, et Jean-Marc Vincent pour son expertise probabiliste.

Avant propos

Ce polycopié comprend deux chapitres qui servent de base à ce cours qui a par ailleurs été présenté dans deux écoles C^3-CAPA (*Conception et Analyse d'Algorithmes Parallèles*) : à cette occasion le premier chapitre est paru dans l'ouvrage *"Algorithmes parallèles : Analyse et conception", chapitre 5, édition Hermès, 1994* et le deuxième chapitre dans l'ouvrage *"Parallélisme et applications irrégulières", chapitre 1, Hermès, 1995.*

Le premier chapitre correspond à un résumé étendu du cours, et est distribué à titre de support. Les principaux résultats théoriques présentés ici sont démontrés en cours. La bibliographie contient les principales références sur lesquelles le cours est basé. D'autres résultats, non présentés dans cet article, sont étudiés pendant le cours, notamment en ce qui concerne les techniques liées à la NC-réductibilité et à la contraction algébrique. Des applications, notamment à la reconnaissance de langages hors-contextes et en algèbre linéaire ne sont pas non plus, bien que traitées en cours, abordées dans ce polycopié.

Le deuxième chapitre est ciblé sur la construction d'algorithmes performants, utilisant des mécanismes fins (Polyalgorithmes et Diviser pour régner en cascade, techniques probabilistes). D'autres résultats concernant notamment la complexité en communication d'un algorithme et l'évaluation du coût sur des modèles de machines plus réalistes que la PRAM (BSP [52] et LogP [17] par exemple) sont présentés en cours.

Chapitre 1

Modèles, complexité et algorithmes de base

1.1 Introduction.

Le but de cet article est de dégager les principales techniques qui permettent de développer des algorithmes parallèles dans le cadre du modèle PRAM. Le but poursuivi est de chercher pour un problème donné un algorithme parallèle utilisant un grand (mais "raisonnable") nombre de processeurs qui résolve ce problème et qui soit le plus rapide possible. Ce chapitre s'inspire largement d'une partie du polycopié de l'ENSIMAG de B. Plateau, A. Rasse, J.L. Roch et J.P. Verjus intitulé "Parallélisme" [40] et d'un cours de DEA de l'université Joseph Fourier intitulé "Complexité Parallèle".

Les deux premières sections sont consacrées à la définition des principales classes de la complexité parallèle. Elles s'inspirent de l'article de R.M. Karp et V. Ramachandran [34], qui constitue une introduction très complète à la complexité parallèle. La dernière section présente les techniques algorithmiques fréquemment utilisées pour construire un algorithme parallèle PRAM le plus rapide possible pour un problème pour lequel un algorithme séquentiel existe déjà. Le plan est le suivant :

- Le modèle PRAM est présenté, ainsi que les notions d'algorithme efficace et optimal. L'utilisation d'un compromis séquentiel-parallèle (principe de Brent et technique d'équilibre des travaux) permet parfois de construire un algorithme parallèle optimal à partir d'un autre efficace et d'un algorithme séquentiel optimal.

- La classe NC, qui rassemble les problèmes intrinsèquement parallèles, est introduite sur le modèle PRAM. La classification NC, et la réduction associée, est ensuite définie à partir du modèle booléen, ce qui permet de mieux préciser les problèmes "résistants" à la parallélisation (problèmes P-complets).

- Les techniques de l'algorithmique PRAM sont introduites à partir d'exemples didactiques et sont présentées par ordre de puissance (i.e. de diminution du temps de résolution parallèle) croissante :

  - L'analyse du graphe de précédence permet d'exploiter le parallélisme qui existe dans l'algorithme séquentiel initial.
  - La technique Diviser pour Paralléliser, version restreinte de la technique séquentielle du Diviser pour Régner, permet parfois de mettre directement en évidence un fort degré de parallélisme.
  - Les techniques précédentes restent assez proches de l'algorithmique séquentielle. Mais le clivage séquentiel-parallèle est important : pour aller plus vite, il est souvent nécessaire de faire des calculs redondants, ce qui est paradoxal en algorithmique séquentielle. La puissance de

l'introduction de redondance est montrée sur les algorithmes classiques de tri par insertion et de l'addition d'entiers.

- En revenant sur la caractérisation générale d'un problème polynômial (à partir du problème P-complet de référence), nous verrons que l'utilisation de la technique de contraction algébrique permet d'aller encore plus loin dans l'analyse du graphe de précédence d'un algorithme séquentiel. Pour cela, il est nécessaire de considérer non seulement les précédences mais aussi les propriétés mathématiques des opérations effectuées (et notamment commutativité, associativité et distributivité).

En algorithmique parallèle, comme en algorithmique séquentiel, il n'existe pas de méthode "miracle" qui permette de développer un bon algorithme de manière automatique. Cependant, les techniques présentées dans ce chapitre s'avèrent souvent utiles dans la recherche de nouveaux algorithmes parallèles [47].

1.2 Le modèle PRAM

1.2.1 Introduction

De façon à pouvoir évaluer précisément la qualité d'un algorithme parallèle et le comparer à un algorithme séquentiel résolvant le même problème il est fondamental de définir un modèle de calcul parallèle permettant de mesurer quantitativement un algorithme.Une fois ce modèle choisi, il est intéressant de classifier les problèmes selon cette mesure. Il y a ici une analogie avec l'algorithmique séquentielle et la classification en classes P (problèmes solubles en temps polynômial sur une machine de Turing déterministe) et NP (problèmes solubles en temps polynômial sur une machine de Turing non-déterministe).

Par la suite A_n désigne un algorithme (séquentiel ou parallèle) calculant la solution d'un problème P_n (instance d'un problème général P ayant $n^{O(1)}$ entrées). Comme exemples de problème P_n, on peut considérer le produit de deux entiers de n bits, de deux matrices nxn à coefficients flottants, de deux matrices nxn à coefficients entiers de n bits, etc. Pour mesurer la qualité de l'algorithme A_n, il est nécessaire de dégager des critères de mesure. Il apparaît naturel de considérer le temps d'exécution de l'algorithme, c'est-à-dire le temps qui s'écoule entre le lancement de l'exécution de l'algorithme et la fin de l'exécution de l'algorithme. Ce critère intuitif n'est cependant pas le seul à prendre en compte, puisqu'un algorithme, pour être exécuté, a besoin d'un support matériel (lié au modèle de calcul) que le temps ne décrit pas.

Dans le cadre du calcul séquentiel, le modèle considéré est la machine de Turing (et ses variantes). De manière pratique, le modèle RAM (Random Access Machine) est une adaptation de ce modèle théorique, mieux adaptée à la description d'algorithmes sur une machine séquentielle [1].

De nombreux modèles sont proposés pour le calcul parallèle *synchrone*, liés à la variété des architectures parallèles (circuits, machines à mémoire partagée ou distribuée, machines hiérarchiques etc.). Il y a cependant deux modèles théoriques qui prédominent [13]. Le premier (**PRAM**) est basé sur des processeurs en relation via une mémoire partagée : ce modèle est "relativement" proche des machines parallèles ; différentes variantes (comme les XRAM) sont proposées pour mieux prendre en compte les caractéristiques des ordinateurs parallèles [15]. Le deuxième (**circuit**) considère des circuits constitués de portes logiques reliées entre elles selon un graphe de connexion.
Nous avons choisi ici de présenter surtout le modèle PRAM, qui est le plus utilisé en algorithmique parallèle et qui peut être vu comme une généralisation du modèle séquentiel RAM. Le modèle circuit, plus théorique, sera introduit pour définir plus précisément le mécanisme des réductions.

1.2.2 Présentation du modèle PRAM

Le modèle PRAM correspond à différentes unités de calcul synchrones (du modèle RAM), en nombre infini, qui communiquent via une mémoire partagée.

Une PRAM consiste en (une définition plus formelle peut être trouvée dans [4]):

- un ensemble illimité de processeurs indicés (chacun connaissant son indice et possédant son propre compteur ordinal),
- une mémoire globale partagée infinie,
- un programme fini, qui consiste en une séquence finie d'instructions étiquetées (soit une lecture ou une écriture en mémoire, soit un branchement conditionnel dans le programme à une étiquette, soit un calcul).

Le fonctionnement est le suivant. A l'initialisation tous les processeurs initialisent leur propre compteur ordinal. Puis, à chaque pas, toutes les unités exécutent l'instruction correspondant à leur compteur ordinal. La notion de "pas" correspond à l'hypothèse synchrone avec durée unitaire des opérations.

Ce modèle s'adapte bien aux architectures parallèles dites à mémoire partagée, c'est-à-dire où tous les processeurs accèdent à une même mémoire. "Accéder" aux données veut alors dire "lire en mémoire" et "retourner" un résultat veut dire modifier une variable en mémoire.

Considérons un programme fini qui ne contient pas de tests sauf portant sur l'indice n du problème en entrée (on parlera par la suite de *programme sans boucle*). L'exécution de ce programme, pour une taille d'instance n donnée mais indépendamment des autres valeurs en entrées, peut être décrite par un ensemble d'opérations (chaque opération est de durée unitaire) et un graphe de précédence. Lors de l'exécution de ce programme sur une PRAM, au premier pas de calcul

seront exécutées toutes les opérations qui n'ont pas d'arc (de précédence) entrant, puis au second pas toutes les opérations qui suivent (au sens du graphe) celles qui ont été exécutées au premier pas et ainsi de suite. On parle de *pas* de calcul ou d'*étape* de calcul.

De nombreux algorithmes parallèles sont évalués sur le modèle PRAM, qui peut être vu comme une abstraction des machines parallèles à mémoire partagée -ou distribuée avec un réseau de connexion complètement connecté- (bien que le cadre théorique du calcul synchrone soit peu vérifié en pratique). Cependant, ce modèle reste relativement imprécis et il est souvent nécessaire de préciser certaines caractéristiques importantes, notamment le mode selon lequel sont réglés les conflits de lecture et écriture par plusieurs processeurs à une même adresse dans la mémoire commune. On distingue ainsi trois sous-modèles (classés par ordre de puissance croissante) :

- EREW (Exclusive Read Exclusive Write, lecture et écriture exclusives) : deux processeurs différents ne peuvent ni lire ni écrire à une même étape sur une même case mémoire.

- CREW (Concurrent Read Exclusive Write, lecture concurrente et écriture exclusive) : des processeurs différents peuvent lire à une même étape le contenu d'une même case mémoire, mais ne peuvent pas écrire simultanément sur une même case.

- CRCW (Concurrent Read Concurrent Write, lecture et écriture concurrentes) : à une même étape, différents processeurs peuvent lire ou écrire sur une même case. Dans le cas d'écritures concurrentes, plusieurs modes peuvent être utilisés pour préciser comment est réglé le conflit d'écriture :

  - COMMUNE-CRCW : les écritures concurrentes ne sont valides que si tous les processeurs concurrents à l'écriture à une même étape sur une même case écrivent la même valeur (sinon, il y a erreur).
  - ARBITRAIRE-CRCW : lorsque plusieurs processeurs veulent écrire à une même étape sur une même case, une valeur au hasard parmi les différentes valeurs proposées est écrite.
  - PRIORITAIRE-CRCW : chaque processeur a un numéro unique, et lorsque plusieurs processeurs veulent écrire à une même étape sur une même case, c'est celui qui a le plus petit numéro qui impose sa valeur.

Ces différents sous-modèles de PRAM sont relativement proches : il existe, comme nous le verrons plus loin, des résultats permettant de passer d'un modèle à un autre.

1.2.3 Travail d'un algorithme parallèle.

Une fois le modéle théorique défini, il est maintenant possible de définir les critères de mesure de la qualité, et notamment l'optimalité, d'un algorithme parallèle. Sur le modèle séquentiel RAM, deux caractéristiques (l'une temporelle, l'autre matérielle) sont considérées pour évaluer la qualité d'un algorithme séquentiel A_n résolvant P_n:

- le *temps*, noté $T_s(A_n)$, défini comme le nombre d'opérations effectuées sur des données bornées (ex: opérations flottantes, opérations sur des entiers machines, lecture ou écriture en mémoire d'un mot machine...). Pour les problèmes "faisables", ce temps séquentiel est un polynôme en n (classe P).

- l'*espace mémoire*, noté $S(A_n)$ (S pour *space*) défini comme le nombre de places en mémoire nécessaires à l'exécution de l'algorithme.

Par analogie, dans le cadre du calcul parallèle, la qualité d'un algorithme parallèle A_n résolvant sur une PRAM le problème P_n est basée sur deux caractéristiques, l'une matérielle, l'autre temporelle:

- la *surface*, notée $H(A_n)$ (H pour *hardware*), définie comme le nombre de processeurs utilisés lors du pas de calcul qui nécessite le plus de processeurs $H(A_n)$ est en général un polynôme en n.

- le *temps*, noté $T_{//}(A_n)$, qui est le nombre de pas nécessaires à l'exécution de l'algorithme avec $H(A_n)$ processeurs.

A partir de ces deux critères, différentes quantités peuvent être introduites, qui seront utilisées par la suite.

Convention : Dans la suite, nous ne considèrerons les évaluations temporelles et matérielles qu'au niveau de leur ordre. Par abus de langage, les dénominations comme "efficace" ou "optimal" correspondent donc aux dénominations "asymptotiquement efficace" ou "asymptotiquement optimal" à cause des résultats donnés en O, donc connus à une constante multiplicative près.

Travail. On appelle *travail* de l'algorithme parallèle A_n la quantité notée $W(A_n)$ definie par:

$$W(A_n) = H(A_n).T_{//}(A_n)$$

Le travail d'un algorithme séquentiel correspond donc au temps séquentiel $T_s(A_n)$ de cet algorithme. Intuitivement, le travail d'un algorithme est l'aire d'un rectangle ayant pour côtés d'une part le nombre de processeurs qu'il utilise, et d'autre part son temps d'exécution. Dans le cas d'un algorithme séquentiel, il n'y a qu'un seul processeur. Si l'on considère le graphe de précédence de l'algorithme, dessiné

de telle façon que les opérations exécutées sur la PRAM au premier pas soient sur une première ligne, puis celles exécutées au i-ème pas sur la i-ème ligne, ce graphe s'inscrit dans ce rectangle[1].

Algorithmes efficaces et optimaux. Parmi tous les problèmes, il est intéressant de déterminer les problèmes qui peuvent être résolus beaucoup plus rapidement en parallèle qu'en séquentiel. Un algorithme séquentiel s'exécute au moins en temps linéaire (sinon, il y a des entrées inutiles). Il apparaît donc naturel de considérer les problèmes qui peuvent être traités en parallèle en temps poly-logarithmique ($\log^{O(1)} n$), caractéristique qui est reprise dans la classe NC, donc plus rapidement que tout algorithme séquentiel.

- Un algorithme parallèle est dit *efficace* si son temps d'exécution est poly-logarithmique et si son travail est le temps du meilleur algorithme séquentiel connu multiplié par un facteur poly-logarithmique. Un algorithme efficace permet donc d'obtenir un temps de résolution impossible à atteindre en séquentiel, avec un travail plus important mais raisonnable par rapport au meilleur algorithme séquentiel.

- Un algorithme parallèle est dit *optimal* s'il est efficace et que son travail est du même ordre que le travail du meilleur algorithme séquentiel connu.

Un algorithme parallèle optimal est donc particulièrement intéressant : non seulement il est très rapide en parallèle, mais sa simulation séquentielle n'est pas plus coûteuse (asymptotiquement tout au moins) que ce qui peut être fait de mieux en séquentiel.

Remarque. Il ne faut pas confondre la notion d'algorithme efficace qui est une qualification d'un type d'algorithme et l'efficacité qui est une grandeur calculable pour tout algorithme. L'efficacité d'un algorithme efficace est égale à l'inverse d'un poly-logarithmique alors que celle d'un algorithme optimal est constante.

1.2.4 Parallèle versus séquentiel - Principe de Brent et applications.

Un algorithme parallèle peut apparaître plus général qu'un algorithme séquentiel : il est toujours possible de simuler le comportement de plusieurs processeurs synchrones sur un seul, alors qu'il n'est en général pas possible d'exécuter un algorithme séquentiel sur plusieurs processeurs (en tirant bien sûr parti de la présence des différents processeurs).

1. Il faut noter que dans l'algorithme parallèle, les $H(A_n)$ processeurs nécessaires de la PRAM exécutent à chaque instant une instruction (soit de l'algorithme proprement dit, soit vide).

Plus précisément, on appelle simulation séquentielle d'un algorithme parallèle l'exécution sur un seul processeur des opérations - même vides - de l'algorithme parallèle. Pour ce faire, le processeur exécute d'abord toutes les opérations du premier pas de calcul de l'algorithme parallèle, puis toutes les opérations du deuxième pas, etc. Par extension, étant donné un algorithme parallèle dont l'exécution est décrite sur $H(A_n)$ processeurs, on appelle simulation de cet algorithme sur $m < H(A_n)$ processeurs l'exécution sur les m processeurs des opérations de l'algorithme. Cette simulation se fait de façon analogue : on divise les opérations de chaque pas de calcul en m groupes d'opérations, chaque groupe étant à la charge de l'un des m processeurs. Les m processeurs exécutent séquentiellement les opérations d'un groupe, puis séquentiellement les groupes de chaque étape. On remarque que cette façon de faire peut induire pour les m processeurs des pas de calcul vide supplémentaires si m ne divise pas $H(A_n)$, mais ce nombre est au maximum de l'ordre de $T_{//}(A_n)$.

Le travail de l'algorithme parallèle A_n est du même ordre que le temps d'exécution de sa simulation séquentielle ou que le travail de sa simulation sur $m < H(A_n)$ processeurs. Cet énoncé est connu sous le nom de *principe de Brent* (en version restreinte, une version plus précise est donnée dans [15]). Compte tenu de ce principe, le travail d'un algorithme parallèle A_n est un invariant (en ordre au moins) lorsqu'on utilise un nombre de processeurs inférieur à $H(A_n)$.

Considérons $A_n^{(s)}$ le meilleur algorithme séquentiel connu résolvant P_n, et soit $A_n^{(p)}$ un algorithme parallèle résolvant P_n. Alors, le principe de Brent peut également s'énoncer ainsi : le travail de l'algorithme parallèle $A_n^{(p)}$ est supérieur à celui de $A_n^{(s)}$: $W(A_n^{(s)}) \leq W(A_n^{(p)})$, et pour tout m : $1 \leq m \leq H(A_n^{(p)})$, l'algorithme $A_n^{(p)}$ peut être exécuté (simulé) en temps $O\left(m.T_{//}(A_n^{(p)})\right)$ avec $O\left(\frac{H(A_n^{(p)})}{m}\right)$ unités de calcul. C'est pourquoi la complexité de $A_n^{(p)}$ sera dans la suite notée (notation inspirée de [34]) :

$$O_{//}\left(T_{//}(A_n^{(p)}), H(A_n^{(p)})\right)$$

Le principe de Brent permet donc de regrouper des opérations effectuées par un algorithme parallèle pour les exécuter séquentiellement. Lorsque l'on dispose d'un algorithme parallèle efficace mais non optimal, il est alors possible de regrouper des opérations de cet algorithme (en diminuant le nombre de processeurs nécessaires au calcul), pour calculer leur résultat par un algorithme séquentiel optimal. Cette technique est connue sous le nom d'équilibre des travaux [27]. Nous la présentons sur l'exemple du produit itéré.

Rendre optimal un algorithme efficace : équilibre des travaux. Le produit itéré se calcule séquentiellement en $T_s(n) = O(n)$ (avec une surface $S(n) = O(1)$) : cet algorithme est clairement optimal, puisqu'il faut déjà n étapes pour lire les entrées. Un premier algorithme parallèle est naturel : il suffit de grouper les entrées dans l'ordre deux par deux, et d'effectuer les opérations selon un arbre

binaire équilibré. Le coût de cet algorithme est $O_{//}(\log n, n)$ sur une PRAM-EREW. Cet algorithme est efficace. Mais il n'est pas optimal, son travail *O(n.log* n) étant en ordre supérieur au temps séquentiel. Il est cependant possible de le transformer pour obtenir un algorithme optimal, en faisant un bon compromis entre l'algorithme efficace et le meilleur algorithme séquentiel.

Considérons pour cela l'intérprétation du travail d'un algorithme parallèle comme étant la surface d'un rectangle ayant pour côtés d'une part le nombre de processeurs, et d'autre part le temps d'exécution. Lorsqu'un algorithme n'est pas optimal, cette surface n'est pas du même ordre de grandeur que le travail de l'algorithme séquentiel : ceci provient des instructions vides exécutées par les processeurs à certains moments de l'exécution. L'idée de base dans la démarche d'"équilibre des travaux" est la suivante : on cherche à réduire la surface du rectangle en occupant à tout moment les processeurs. Pour cela, il faut modifier certaines phases de l'algorithme en tirant parti d'un "bon" algorithme séquentiel : à certaines phases de l'exécution, l'algorithme parallèle est remplacé par l'exécution en parallèle par chaque processeur de l'algorithme séquentiel. On parvient ainsi à réduire la surface du rectangle en supprimant des parties correspondant à des instructions vides. Le temps de l'algorithme parallèle reste du même ordre, mais le nombre de processeurs utilisés est plus faible.

Dans l'exemple du produit itéré, en comparant les algorithmes, on remarque que le travail parallèle est d'un facteur $O(\log n)$ supérieur au travail séquentiel. Nous allons essayer de garder ce temps (en ordre de grandeur) en réduisant le nombre de processeurs. L'obtention d'un algorithme optimal nécessite donc de n'utiliser que $O\left(\frac{n}{\log n}\right)$ processeurs. Cette diminution du nombre de processeurs peut être réalisée en groupant différentes opérations sur un même processeur, sur lequel sera alors exécuté -de manière optimale- l'algorithme séquentiel.

Groupons les n entrées en $O\left(\frac{n}{\log n}\right)$ groupes ayant chacun $O(\log n)$ éléments. A chaque groupe nous associons une tâche, qui calcule le produit itéré de son groupe par l'algorithme séquentiel. Comme il y a $O(\log n)$ éléments dans chaque groupe, le temps de cette étape est $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs.

Puis, le produit itéré des $O\left(\frac{n}{\log n}\right)$ produits partiels ainsi obtenus peut être calculé par l'algorithme parallèle initial en temps $O(\log n)$, en utilisant encore $O\left(\frac{n}{\log n}\right)$ processeurs. La complexité parallèle de ce nouvel algorithme est alors : $O_{//}\left(\log n, \frac{n}{\log n}\right)$, et son travail est $O(n)$, c'est-à-dire du même ordre que le travail de l'algorithme séquentiel : il est donc *optimal*.

1.3 La classification NC

1.3.1 La classe NC.

Une question de base en calcul parallèle est de déterminer les problèmes intrinsèquement parallèles, c'est-à-dire qui peuvent être résolus beaucoup plus rapidement avec plusieurs processeurs plutôt qu'avec un seul [13] [34].

La classe NC, formalisée par Nicholas Pippenger [39] (et nommée par Cook NC pour "Nick's class" [13]), est la classe des problèmes qui peuvent être résolus en temps poly-logarithmique (c'est-à-dire résolues plus rapidement qu'il ne faut de temps pour lire séquentiellement leurs entrées) sur une machine parallèle ayant un nombre polynomial de processeurs, autrement dit de surface raisonnable. NC peut donc être caractérisée par:

$$NC = \Big\{ \text{ problèmes } P / \exists A = (A_n) \text{ famille d'algorithmes : } A_n \text{ résout } P_n \text{ en coût } O_{//}\left(\log^{O(1)} n, n^{O(1)}\right) \Big\}$$

Une propriété fondamentale de NC est d'être **résistante** : elle reste la même quel que soit le modèle parallèle (PRAM -CREW)-, circuits, etc.

Il est facile de voir que NC est un sous-ensemble de la classe P des fonctions qui peuvent être calculées séquentiellement en temps polynomial. On a donc $P \supseteq NC$ mais l'inclusion stricte reste à démontrer.

Sous-classes de NC De façon à distinguer plus précisément les problèmes à l'intérieur de NC, NC est partitionnée en sous-classes. On distingue ainsi dans le cadre du modèle PRAM :

- $EREW^k$: classe des problèmes qui peuvent être résolus en temps $O(\log^k n)$ avec un nombre polynomial de processeurs d'une EREW-PRAM.
- $CREW^k$: classe des problèmes qui peuvent être résolus en temps $O(\log^k n)$ avec un nombre polynomial de processeurs d'une CREW-PRAM.
- $CRCW^k$: classe des problèmes qui peuvent être résolus en temps $O(\log^k n)$ avec un nombre polynomial de processeurs d'une CRCW-PRAM (le choix du sous-modèle n'influe pas sur cette définition).

Ces différents sous-modèles de PRAM sont cependant très proches, et laissent invariante la propriété d'apartenance à NC d'un algorithme donné. Plus précisément, on peut montrer qu'un algorithme qui s'exécute en temps $T(n)$ avec $H(n)$ processeurs sur une PRIORITAIRE-PRAM (la plus puissante des PRAM) peut être simulé en temps $T(n).O(\log H(n))$ avec $H(n)$ processeurs d'une EREW-PRAM (la moins puissante). Au niveau des CRCW-PRAM, un algorithme s'exécutant en temps T(n) avec H(n) processeurs d'une PRIORITAIRE-PRAM peut être simulé en temps T(n) avec $H(n)^2$ processeurs d'une COMMUNE-PRAM.

Exemple. Considérons le problème du OU-booléen de n bits : ce problème est du type produit itéré. Sur une EREW-PRAM, la solution proposée précédemment permet de montrer que ce problème peut être résolu en temps $O(\log n)$ avec $O(n)$ processeurs[2]. Sur une CRCW-PRAM, ce problème peut être cependant résolu en temps constant : il suffit de disposer d'une case mémoire *résultat* initialisée à la valeur *faux*; à chaque bit en entrée est associé un processeur, qui écrit *vrai* dans la case résultat si le bit qui lui est associé est à la valeur *vrai*, et rien sinon. En une étape le OU-booléen de n bits est donc calculé avec n processeurs d'une CRCW-PRAM. Il est à noter que cet algorithme fonctionne quel que soit le mode (COMMUNE, ARBITRAIRE ou PRIORITAIRE) selon lequel s'effectuent les écritures concurrentes.

Ces remarques entraînent une méthodologie sur le choix de la "bonne" PRAM pour la définition d'un algorithme parallèle pour un problème donné (la plus faible EREW étant l'idéale) : on choisira en premier lieu le sous-modèle qui est le mieux adapté aux besoins de l'algorithme. Il sera toujours possible par une simulation (directe comme ci-dessus ou plus intelligente, avec modification de l'algorithme) de construire à partir de ce premier algorithme d'autres algorithmes sur des sous-modèles plus faibles, avec une perte en temps "relativement" faible : le facteur $O(\log n)$ induit par la simulation directe garantit qu'un algorithme efficace sur un sous-modèle donné restera efficace sur les autres sous-modèles.

La classe NC peut alors être vue comme l'union de toutes les sous-classes précédentes, et on a les inclusions suivantes (pour k ≥ 1) :

$$EREW^k \subset CREW^k \subset CRCW^k \subset EREW^{k+1} \subset NC$$

Remarque. Classes probabilistes. Dans le domaine de la complexité parallèle, les classes probabilistes (et notamment RNC pour les algorithmes Monte-Carlo et ZNC pour les algorithmes Las Vegas) sont particulièrement importantes. (cf [34] pour une introduction). Ainsi le calcul du rang d'une matrice a d'abord été montré comme étant dans RNC [20] avant d'être montré dans NC^2 [37]. Plus récemment, les formes de Smith ou de Jordan ont eu le même sort [28] [43].

1.3.2 Modèle booléen et NC-Réduction

De façon à pouvoir classer les problèmes par ordre de difficulté à l'intérieur de NC, et préciser où peut se trouver la différence entre P et NC, une relation d'ordre entre les problèmes est définie dans le cadre du modèle booléen [7] : la NC-réductibilité.

2. Ce temps est même montré comme étant une borne inférieure pour ce problème sur une CREW PRAM (la complexité temporelle est $\Omega(\log n)$) [14].

Le modèle booléen

Dans ce modèle, une machine parallèle est une famille *uniforme* $(B_n)_{n\in I\!N}$ de graphes booléens orientés et acycliques (DAG) telle que B_n a $n^{O(1)}$ entrées. Un nœud du circuit est ici une porte logique effectuant une opération booléenne (et, ou, négation). On distingue essentiellement deux sous-modèles, selon que le nombre d'entrées d'une porte (fan-in) est borné (i.e. vaut 2 ici) ou non borné. Le nombre de sorties d'une porte (fan-out) est quant à lui non borné[3].

Les noeuds d'entrée (respectivement de sortie) ont un fan-in (respectivement fan-out) de 0. L'uniformité permet de limiter la complexité architecturale du circuit [48]. On utilise le plus souvent la "log-uniformité" qui signifie que la description du circuit B_n (pour $n \in I\!N$) peut être calculée sur une machine de Turing avec un espace logarithmique.

La surface $H(n)$ est ici définie comme le nombre de nœuds du circuit B_n, et le temps comme sa profondeur.

Dans ce modèle on distingue les sous-classes NC^k (respectivement AC^k) pour les circuits dont les portes ont un fan-in borné (resp. non borné) et qui sont de profondeur $O(\log^k n)$ et de taille $n^{O(1)}$. On a alors les relations suivantes [34] :

$$NC^k = EREW^k \subseteq CREW^k \subseteq CRCW^k = AC^k \subseteq NC^{k+1}$$

La classe NC est alors définie comme l'union de toutes les classes NC^k :

$$NC = \bigcup_{k=0}^{\infty} NC^k$$

Remarque : circuits arithmétiques. Des extensions du modèle booléen [53] permettent de considérer que les portes du circuit peuvent faire en temps unité des opérations sur un espace donné E (par exemple les rationnels, ou les polynômes à coefficients rationnels). Les classes de complexité correspondantes sont alors notées NC^k_E pour préciser que les opérations de base considérées sont des opérations sur E. Par exemple, le produit de n entiers de n bits appartient à $NC^1_{I\!N}$ (produit itéré) mais le même algorithme ne permet que de prouver l'appartenance à NC^2 (la multiplication de deux entiers de n bits appartenant à NC^1).

NC-réductibilité et problèmes P-complets.

Une fois le modéle booléen défini, il est maintenant possible de classifier les problèmes selon leur complexité, grâce à une relation d'ordre : la NC^1-réductibilité [13]. On dit qu'une fonction (ou un problème) f est NC^1-réductible à une fonction g (ce qui est note $f \leq_{NC^1} g$) s'il existe une famille uniforme de circuits qui calcule f en temps logarithmique ($O(\log n)$), et dont les nœuds sont soit des

3. Un circuit de fan-in borné et de fan-out non borné peut en effet être transformé en un circuit de même surface et de même temps -en ordre- qui soit de fan-in et de fan-out borné [25].

portes booléennes, soit des oracles permettant de calculer g. Un oracle pour g est ici un nœud ayant r entrées $(e_1, \ldots, e_r)$ et t sorties $(s_1, \ldots, s_t)$ et qui calcule le résultat $(s_1, \ldots, s_t) = g(e_1, \ldots, e_r)$. La profondeur d'un tel nœud est assimilée à $\log(rt)$.
Il est clair que la relation de NC^1-réductibilité est réflexive et transitive et que NC^k est close par NC^1-réductibilité.

Soit E une classe de problèmes. On dira qu'une fonction (un problème) f est $NC^1 - dur$ pour l'ensemble E (ou E-dur) si et seulement si :

$$\forall g \in E : g \leq_{NC^1} f$$

f est dit complet pour E (E-complet) si f est E-dur et si $f \in E$.

Nous avons vu que $NC \subseteq P$. L'inclusion stricte restant conjecturelle, on peut se demander quels sont les problèmes complets pour P, qui sont ceux contenant -ou susceptibles de contenir- le moins de parallélisme intrinsèque.

Le problème P-complet de référence (l'analogue de la satisfaisabilité pour la complexité séquentielle et la classe NP) est le MCVP (monotone circuit value problem) [24] : " Etant donné une séquence de n équations booléennes du type $e_1 = 0$, $e_2 = 1$ et $e_k = e_i \wedge e_j$ ou $e_k = e_i \vee e_j$ pour $1 \leq i \leq j < k \leq n$, calculer la valeur de e_n [4]."

1.4 Techniques de base de l'algorithmique PRAM

Après cette présentation du modèle PRAM et de la classification NC, nous allons maintenant dégager les principales techniques qui permettent de développer, à partir d'un algorithme séquentiel connu, de bons algorithmes parallèles sur ce modèle (qui permettent de prouver, si possible, l'appartenance à NC du problème).

1.4.1 Etudier le graphe de précédence.

Une première technique pour extraire le parallélisme d'un problème donné (pour lequel on dispose déjà d'un algorithme séquentiel), est de considérer le graphe de précédence des opérations effectuées dans l'algorithme séquentiel. La profondeur de ce graphe permet de donner une borne supérieure sur le temps d'un algorithme parallèle résolvant le même problème, le nombre de processeurs nécessaires à l'obtention de ce temps étant borné par la "largeur" de ce graphe.

Considérons comme exemple la résolution d'un système linéaire triangulaire. Soit A une matrice nxn triangulaire inférieure inversible, à coefficients dans un

4. Tout problème s'exécutant en temps polynômial sur une machine de Turing déterministe peut être NC^1-réduit à ce problème [34], ce qui prouve, étant clairement dans P, qu'il est P-complet.

corps K, et b un vecteur de K^n. Le problème est de calculer l'unique vecteur x de K^n tel que : $A\ x = b$.

La parallélisation de ce problème est caractéristique de nombreux problèmes d'algèbre linéaire : la parallélisation des algorithmes séquentiels de factorisation (Gauss, Householder, Givens...) met en jeu les mêmes techniques.

La solution séquentielle s'exprime très facilement à partir de l'itération :

$$Pour\ \ i = 1 \ldots n : x_i := \frac{b_i - \sum_{k=1}^{i-1} a_{i,k}.x_k}{a_{i,i}}$$

Cet algorithme a une complexité $O(n^2)$ et est donc optimal (les $\frac{n^2}{2}$ coefficients de la matrice A devant être examinés).

Mais sa parallélisation semble difficile : il y a une dépendance très forte entre le calcul de x_i et celui de x_{i-1}. Néanmoins, pour chaque indice i, le calcul à effectuer est du type produit itéré, et cette phase peut être réalisé en temps $O_{//}\left(\log i, \frac{i}{\log i}\right)$. Les n phases s'effectuant séquentiellement, on obtient comme complexité globale sur une PRAM-CREW :

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

Pour trouver ce résultat, tous les *log i* sont majorés par *log n* et la mise en séquence implique l'addition des temps. Le nombre de processeurs nécessaire est égal au nombre maximum de processeurs nécessaires pour chacune des phases.

La parallélisation peut être améliorée si l'on regarde précisément le graphe de précédence des tâches de l'algorithme séquentiel. En effet, dès que x_i est calculé, il est possible de calculer en parallèle pour $k{=}i{+}1 \ldots n$ toutes les produits : $a_{k,i}.\ x_i$, si l'on suppose que x_i peut être lu en parallèle par les processeurs chargés des calculs de x_k ($k{=}i{+}1 \ldots n$). En terme d'étapes, le schéma de calcul est alors le suivant (les calculs effectués en parallèle sur des processeurs différents sont séparés par "//") :

étape 1 : $x_1 := \frac{b_1}{a_{1,1}}$

étape 2 : $x_2 := \frac{b_2 - a_{2,1}.x_1}{a_{1,1}}$ // $t_3 := b_3 - a_{3,1}.x_1$ // $t_4 := b_4 - a_{4,1}.x_1$ // ...

étape 3 : $x_3 := \frac{t_3 - a_{3,2}.x_2}{a_{3,3}}$ // $t_4 := t_4 - a_{4,2}.x_2$ // $t_5 := t_5 - a_{5,2}.x_2$ // ...

étape 4 : $x_4 := \frac{t_4 - a_{4,3}.x_3}{a_{4,4}}$ // $t_5 := t_5 - a_{5,3}.x_3$ // $t_6 := t_6 - a_{6,3}.x_3$ // ...

...

A chaque nouvelle étape (toutes les étapes sont effectuées en temps constant avec n processeurs), une nouvelle composante de x est calculée. Le vecteur x peut ainsi être calculé en temps *O(n)* sur une CREW-PRAM de n processeurs. En ordonnançant les calculs différemment, la contrainte CREW peut être allégée en EREW: il suffit de pipeliner le parcours de x_1, puis de x_2, etc. Le nombre d'étapes est doublé, mais la complexité asymptotique reste la même.

L'algorithme donné ici présente l'avantage d'être de même travail en ordre que le meilleur algorithme séquentiel connu. Mais il ne permet pas de décider de

l'appartenance à NC de ce problème car sa complexité en temps est en $O(n)$. Nous verrons dans la prochaine partie que par un tout autre algorithme, ce problème peut être en fait calculé très rapidement en temps $O(\log^2 n)$ sur une PRAM. Le problème considéré appartient donc bien en fait à NC, mais même une bonne parallélisation de l'algorithme séquentiel, comme celle donnée ici, ne permet pas de le démontrer.

1.4.2 Diviser pour paralléliser

Une autre technique de parallélisation consiste à essayer de construire la solution d'un problème P_n à partir d'instances indépendantes (i.e. pouvant être traitées en parallèle) plus petites du même problème. Cette technique s'apparente à la technique séquentielle du "Diviser pour Régner", avec la restriction que les sous-instances doivent pouvoir être traitées indépendamment. Le schéma algorithmique est le suivant :

1. Réduction du problème à des instances indépendantes plus petites.
2. Résolution parallèle récursive des sous-instances, en appliquant récursivement la même technique à chacune de sous instances.
3. Construction de la solution du problème initial à partir des solutions de chacune des sous-instances.

Pour illustrer cette technique, considérons l'exemple du calcul des préfixes pour une loi associative, qui est une extension du produit itéré. Etant données n entrées $a_1, \ldots, a_n$ d'un ensemble E muni d'un loi associative $*$, le problème est de calculer les préfixes $\pi_k = a_1 * \ldots * a_k$, $k = 1, \ldots, n$. La solution séquentielle triviale conduit à un algorithme optimal en temps n; mais le graphe de précédence de cet algorithme est de profondeur n et ne permet pas une parallélisation fine. Il est cependant facile d'appliquer la technique "diviser pour paralléliser":

1. Réduction du problème : on considère en parallèle les deux sous-séquences de taille $n/2$: $S_1 = (a_1, \ldots, a_{\frac{n}{2}})$ et $S_2 = (a_{\frac{n}{2}+1}, \ldots, a_n)$.
2. Résolution des sous-instances : la récursivité fait le travail. On obtient ainsi les préfixes $(\pi_1, \ldots, \pi_{\frac{n}{2}})$ pour s_1 et $(\mu_{\frac{n}{2}+1}, \ldots, \mu_n)$ pour S_2.
3. Construction de la solution du problème initial : les préfixes manquants sont obtenus en calculant : $\pi_k = \pi_{\frac{n}{2}} * \mu_k$ pour $k = \frac{n}{2} + 1, \ldots, n$.

Cet algorithme peut s'exécuter trivialement avec une complexité $O_{//}(\log n, n)$ sur une PRAM CREW (du fait de la lecture concurrente de $\pi_{\frac{n}{2}}$ dans la phase de construction).
Il est cependant possible de raffiner cet algorithme pour qu'il puisse s'exécuter

sur une PRAM EREW avec la même complexité : la phase de construction peut être simplifiée en compliquant la phase de réduction de la façon suivante :

1. Réduction du problème : ceci peut être fait en calculant en parallèle les produits des entrées groupées par 2. On calcule donc les quantités (n est supposé être une puissance de 2) $b_k = a_{2k-1} * a_{2k}$, $k = 1, \ldots, n/2$. Le calcul des préfixes des $n/2$ éléments b_k constitue bien une sous-instance du même problème.

2. Résolution des sous-instances : la récursivité fait le travail.

3. Construction de la solution du problème initial : on a déja obtenu tous les préfixes π_{2k}. Les préfixes d'indice impair peuvent être obtenus en parallèle en temps 1 en calculant $\pi_{2k+1} = \pi_{2k} * a_{2k+1}$.

On obtient ainsi un algorithme de même complexité avec la contrainte EREW. Pour rendre optimal cet algorithme efficace, la technique d'équilibre des travaux peut être appliquée, en groupant les éléments de la séquence intiale en $\frac{n}{\log n}$ groupes de $\log n$ éléments. Les préfixes de chacun des groupes peuvent alors être obtenus en temps $\log n$ par application de l'algorithme séquentiel en parallèle pour chaque groupe. On obtient ainsi assez facilement un algorithme de complexité $O(\log n, \frac{n}{\log n})$ sur une PRAM EREW.

1.4.3 Casser les précédences par la redondance

Les deux techniques précédentes s'inspiraient fortement des techniques séquentielles. Une autre manière d'apporter du parallélisme dans un algorithme consiste à effectuer des calculs redondants, de façon à diminuer les dépendances de données, donc le temps. Cette démarche est contraire à celle utilisée en séquentiel où, lorsque deux calculs se ressemblent, on essaie de factoriser les termes communs pour ne les calculer qu'une fois. L'introduction de redondance est donc une approche souvent intéressante pour diminuer le temps de traitement parallèle, même si elle conduit, au premier abord tout au moins, à des algorithmes non optimaux . La duplication de certains calculs évités en séquentiel fait que le travail parallèle est souvent plus important. Nous allons expliciter cet apport de redondance sur deux exemples : l'addition d'entiers et le tri.

Addition d'entiers. On considère deux entiers A et B de n bits représentés par la séquence de leurs chiffres (poids forts en tête):

$$A = [a_{n-1}, a_{n-2}, \ldots, a_0] = \sum_{i=0}^{n-1} a_i 2^i \text{ avec } a_i \in \{0,1\} \text{ pour tout } i$$
$$B = [b_{n-1}, b_{n-2}, \ldots, b_0] = \sum_{i=0}^{n-1} b_i 2^i \text{ avec } b_i \in \{0,1\} \text{ pour tout } i$$

et on cherche la séquence de $(n+1)$ bits : $[r_n, r_{n-1}, \ldots, r_0]$ représentant l'entier $R = A + B$.

On peut remarquer la propriété suivante : si X et Y sont deux entiers de k bits au plus, alors $X+Y$ et $X+Y+1$ sont deux entiers de $k+1$ bits au plus.

L'algorithme séquentiel consiste alors à additionner deux à deux les chiffres de A et B, poids faibles d'abord, en faisant propager les éventuelles retenues (une retenue est soit nulle, soit égale à 1), ce qui peut être exprimé par la récurrence suivante :

$$\begin{cases} c_{-1} & = & 0 \\ r_k & = & (a_k + b_k + c_{k-1}) \mathrm{m}od2 \\ c_k & = & (a_k + b_k + c_{k-1}) \mathrm{d}iv2 \end{cases}$$

où *div* et *mod* désignent respectivement les opérations de division euclidienne et de calcul de reste (avec reste positif).

De cette relation de récurrence, il est très difficile d'extraire du parallélisme. La propagation de retenue rend cet algorithme très séquentiel : un nouveau chiffre r_k du résultat ne peut être calculé que si l'on connaît la retenue c_{k-1} précédente.

Pour éviter cette propagation et introduire du parallélisme, l'idée est d'utiliser l'approche diviser pour paralléliser en calculant a priori deux résultats possibles : pour cela, on partitionne A et B en séparant poids forts (A_H et B_H) et poids faibles (A_L et B_L). De même, on calcule le résultat R en séparant les poids forts (R_H) et les poids faibles (R_L). Deux calculs sont possibles pour R_H : l'un en supposant que la retenue entrante pour le calcul de $A_H + B_H$ est nulle, l'autre en supposant qu'elle est égale à 1. La sélection du bon résultat pourra être faite a posteriori, en regardant la retenue sortant effectivement du calcul de $A_L + B_L$. L'algorithme est donc le suivant :

- réduction : en temps constant séparer en parallèle A_H et A_L, B_H et B_L.
- résolution : calculer par un appel récursif en parrallèle A_H+B_H, A_H+B_H+1 et $A_L + B_L$.
- fusion : si A_L+B_L génère une retenue fusionnerA_L+B_L et A_H+B_H+1 sinon fusionnerA_L+B_L et A_H+B_H. L'opération de fusion (simple concaténation) se fait en temps constant.

Il y a clairement ici redondance : les chiffres de poids fort de A et B sont additionnés deux fois. Mais cette redondance est intéressante puisqu'elle permet de rendre indépendants les trois appels à l'adition. Le temps parallèle de ce nouvel algorithme sur une EREW-PRAM (il est facile d'éliminer la concurrence au niveau des lectures des poids forts en les dupliquant) est alors :

$$T(n) = T\left(\frac{n}{2}\right) + 1 = O(\log n)$$

avec un nombre de processeurs :

$$H(n) = 3H\left(\frac{n}{2}\right) + n = O\left(n^{\log_2 3}\right) = O(n^{1,58})$$

L'introduction de la redondance permet donc d'obtenir un temps parallèle très intéressant en $O(\log n)$.

Par contre le travail de cet algorithme - $O(n^{1,58} \log n)$ - est important devant la complexité séquentielle du problème - $O(n)$-. L'algorithme parallèle obtenu est loin d'être *efficace*. Il permet cependant de montrer que l'addition d'entiers peut être résolu très rapidement en parallèle et appartient à la classe NC^1. Nous verrons avec la prochaine technique comment obtenir un algorithme optimal pour ce problème.

Tri par sélection. Il est à noter que l'application de la technique "diviser pour paralléliser" réduit le problème du tri à celui de la fusion (bitonique, pair-impair, ... [3]). Ici, pour illustrer le travail de parallélisation par la redondance, nous considérons l'agorithme trivial du tri par insertion.

On se donne n éléments e_k (k=*0...n-1*) d'un ensemble E totalement ordonné, et on voudrait obtenir les n éléments triés dans un tableau. Nous supposerons ici les éléments distincts, mais il est facile d'étendre cet algorithme au cas général.

La méthode séquentielle immédiate, même si ce n'est pas la plus efficace, est celle du tri par insertion : à chaque étape on prend un nouvel élément que l'on essaie de classer parmi les précédents éléments que l'on a déjà triés. Exploitée sous cette forme en parallèle, cette méthode ne permet pas d'obtenir un temps parallèle inférieur à n .

Cependant l'idée de classer un élément parmi d'autres semble prometteuse : si l'on considère un élément e_i donné, il peut être facilement comparé à tous les autres éléments : le nombre d'éléments qui lui sont inférieurs indique la position de l'élément dans le tableau trié.

La comparaison de deux éléments étant effectuée en temps constant, l'algorithme se déroule en deux phases :

- 1 - chaque élément e_k est comparé aux n-1 autres éléments. Pour cela, à chaque élément e_k est associé un groupe G_k de n-1 processeurs $G_{k,i}$ ($i \neq k$) : chaque processeur $G_{k,i}$ compare e_k à un autre élément e_i. Le résultat de cette comparaison est 1 si e_k est strictement plus grand que e_i, et 0 sinon. La complexité de cette étape est donc $O_{//}(1,\ n^2)$ sur une EREW-PRAM (il suffit de dupliquer chaque entrée n fois en complexité $O_{//}(1, n^2)$ pour éliminer la concurrence au niveau des lectures).

- 2 - dans chaque groupe G_k on effectue la somme des n-1 valeurs trouvées : on obtient ainsi la position de l'élément e_k dans la séquence triée.

La phase 2 se réduit donc à l'addition de n bits. Cette opération peut être réalisée par un algorithme est du type produit itéré, avec comme opération de base la somme de deux nombres ayant au plus $\log n$ bits. D'après ce qui a été vu précédemment, cette somme est de complexité $O_{//}(\log\log n, \log n)$ sur une EREW-PRAM. Compte-tenu qu'une somme itérée avec des opérations en temps

constant peut être réalisée en temps $\log n$ avec $O(\frac{n}{\log n})$ processeurs, le coût de cette étape est donc $O_{//}\left(\log n \log\log n, \frac{n^2}{\log n}\right)$ pour tous les groupes.

Néanmoins il est possible de calculer la phase 2 en complexité $O_{//}(\log n, n^2)$ en utilisant la technique dite du *2 pour 3*. En effet, l'utilisation du produit itéré dans la phase 2 ignore le fait que les entrées sont sur 1 bit seulement : la complexité serait la même si le problème était l'addition de n nombres de n bits. Or, additionner 3 nombres de n bits peut se ramener à additionner 2 nombres de $n+1$ bits. Soient

$$a = [a_{n-1}, a_{n-2}, \ldots, a_0] \quad b = [b_{n-1}, b_{n-2}, \ldots, b_0] \text{ et } \quad c = [c_{n-1}, c_{n-2}, \ldots, c_0]$$

trois entiers de n bits dont on désire calculer la somme.
Pour tout i, $a_i + b_i + c_i$ est un entier de deux bits : soit u_i son bit de poids fort et v_i son bit de poids faible, et soient u et v les deux entiers de $n+1$ bits :

$$u = [u_{n-1}, u_{n-2}, \ldots, u_0, 0] \text{ et } \quad v = [0, v_{n-1}, v_{n-2}, \ldots, v_0]$$

Alors on a:

$$a + b + c = u + v$$

Additionner n nombres de 1 bit se ramène donc à additionner $\frac{2n}{3}$ nombres de 2 bits, et en réitérant cette réduction, de coût $O(1)$, $\log_{\frac{3}{2}} n$ fois, deux nombres de $\log_{\frac{3}{2}} n + 1$ bits. Après une réduction de temps $\log n$, le problème se ramène donc à une addition qui peut se faire en temps $\log\log n$. Le coût de cette phase 2 est donc $O_{//}(\log n, n)$ sur une EREW PRAM, pour chaque groupe, soit $O_{//}(\log n, n^2)$ pour tous les groupes.

La complexité globale de cette parallélisation de l'algorithme de tri par insertion est $O_{//}(\log n, n^2)$ sur une EREW-PRAM : la parallélisation est efficace et peut être rendue optimale par un équilibre des travaux. Finalement, le temps de l'algorithme est particulièrement intéressant, mais le nombre de processeurs est très important et fait que cet algorithme, tout en montrant l'appartenance du tri à la classe *NC*, n'est pas efficace par rapport aux algorithmes séquentiels optimaux en $O(n \log n)$. Il existe cependant un algorithme optimal de complexité $O_{//}(\log\ n,\ n)$ [12].

1.4.4 Paralléliser un graphe algébrique de précédence.

Nous avons vu que tout problème dans P pouvait être réduit au probleme MCVP, qui est P-complet. Toutes les techniques permettant d'évaluer rapidement en parallèle des instances du MCVP pourront donc être appliquées à n'importe quel problème polynômial (pour autant que l'on puisse construire "facilement" les instances du MCVP correspondant au problème que l'on cherche à paralléliser)[42].

De manière générale, une instance du MCVP se présente comme un programme arithmétique sans boucle, de longueur n, (i.e. un graphe de précédence comportant n nœuds) ne comportant que des affectations ou des opérations booléennes $\wedge$ ou $\vee$.

Différentes techniques ont été proposées pour évaluer rapidement des programmes sans boucles dans des structures algébriques. Considérons un ensemble E muni d'une loi associative : si le DAG correspondant au programme peut être transformé en un arbre ayant $n^{O(1)}$ nœuds, l'algorithme du produit itéré permettra de l'évaluer en temps $O(\log n)$ avec $O\left(\frac{n}{\log n}\right)$ processeurs d'une CREW PRAM. Ce résultat peut s'étendre à des expressions plus générales [11] [22].

Considérons en exemple l'addition de 2 entiers de n bits, introduit précédemment. Le calcul du résultat se réduit au calcul des retenues c_i : une fois celles-ci obtenues, il est facile de calculer les bits r_i du résultat en temps constant. Les équations booléennes qui caractérisent ces retenues c_i sont les suivantes :

$$c_i = (a_i \wedge b_i) \vee (c_{i-1} \wedge (a_i \vee b_i))$$

En exprimant c_i en fonction des entrées a_k et b_k $(k \leq i)$, le calcul de c_i se ramène alors à un OU booléen de i bits, qui peut être évalué en temps $\log n$ sur une CREW-PRAM (et en temps constant sur une CRCW-PRAM).

Mais si le nombre de nœuds de l'arbre correspondant à l'expansion du circuit est exponentiel, l'évaluation ne pourra se faire en parallèle qu'en temps linéaire . Comme exemple, on peut considérer le programme suivant : $x_i := x_{i-1} + x_{i-1}$ pour $i = 1, \ldots, n$ avec comme entrée x_0 un entier dans le monoïde $(I\!N, +)$. Pour pallier à ce problème, il est cependant possible d'utiliser la puissance de la structure algébrique de l'ensemble dans lequel est défini le problème. Pour l'exemple précédent, il est clair que le programme calcul $2^n.x$; l'existence de la multiplication, distributive par rapport à l'addition, peut permettre de ramener le problème à un produit itéré pour calculer 2^n. Plus précisément, une technique d'évaluation parallèle tirant parti à la fois de l'associativité et de la distributivité d'une loi par rapport à l'autre a été exhibée, lorsque la structure algébrique correspondant aux opérations du programme sans boucle est upn semi-anneau commutatif (i.e. un ensemble muni de deux lois + et $\times$ associatives, commutatives et possédant chacune un élément neutre, $\times$ étant distributive par rapport à +).
Le résultat est le suivant [30] : tout programme sans-boucle de n noeuds dans un semi-anneau peut être évalué en temps $\log n \log(n.d(n))$ avec $M(n)$ processeurs d'une CREW-PRAM, où $M(n)$ nombre de processeurs nécessaires pour effectuer un produit de matrices en temps $\log n$ - $M(n) < n^3$-. $d(n)$ est le degré -au sens mathématique- du programme (par rapport à l'opération $\times$) : c'est le maximum du degré des nœuds du circuit correspondant au programme : le degré d'une entrée est 1, celui d'un nœud + le maximum des degrés de ses opérandes, et celui d'un nœud $\times$ la somme des degrés de ses opérandes.

Considérons en exemple la résolution du système linéaire triangulaire $Ax = b$ introduit précédemment. Le graphe de précédence décrit constitue bien un pro-

gramme sans boucle. Le degré de ce programme est le degré de x_n. Or le degré de x_k est le degré de $x_{k-1} + O(1)$, pour tout k. Le degré du programme est donc $O(n)$. On en déduit que la résolution d'un système linéaire peut être réalisée en temps $O(\log^2 n)$ sur une PRAM CREW avec $M(n)$ processeurs. Ce problème appartient donc à NC^2.

Remarque : cas des treillis. Des extensions de ce résultat à la structure algébrique de treillis ont été données [42]. Cette structure est intéressante, car elle permet de mieux prendre en compte la structure des booléens, cadre du MCVP.

Chercher une expression algébrique équilibrée du problème. Le résultat précédent montre que la recherche d'une expression algébrique "équilibrée" des sorties en fonction des entrées permet d'obtenir des algorithmes parallèles fins. Encore faut-il pouvoir trouver cette expression.

Cette recherche peut être illustrée par l'exemple de la résolution de systèmes triangulaires. En fait, résoudre en parallèle un système triangulaire ou n systèmes prend le même temps : au lieu de considérer le problème de la résolution d'un seul système, nous allons donc considérer le problème du calcul de l'inverse d'une matrice triangulaire. L'introduction de redondance permet ici une meilleure caractérisation algébrique du résultat, dans le corps des matrices inversibles.

La matrice inverse est définie comme la transposée de la matrice des cofacteurs : les cofacteurs de A sont définis comme le rapport de deux déterminants d'ordre n. Or, il est clair qu'un déterminant correspond à une expression algébrique équilibrée des coefficients de la matrice : il suffit de développer le déterminant par ligne pour s'en rendre compte. Malheureusement, en développant un déterminant d'ordre n, on obtient $(n!)$ termes à sommer : avec l'algorithme du produit itéré, on déduit donc un algorithme de temps $(n.log\ n)$ avec un nombre exponentiel de processeurs... ce n'est pas la bonne méthode ! Mais il nous reste l'espoir de trouver une expression algébrique plus simple...

Cette caractérisation algébrique du résultat (l'inverse de A) en fonction de A est cependant facile à obtenir lorsque A est triangulaire; il suffit de décomposer A en une matrice D, formée à partir de la diagonale de A, et la matrice T formée par les opposés des autres éléments :

$$A = D - T$$

Soit D^{-1} l'inverse de D et $U = D^{-1}\ T$. On a alors : $A = D\ (\ I\ -\ U)$

U est une matrice nilpotente (triangulaire à diagonale nulle), et par suite, $U^k = 0$ pour $k \geq n$. La matrice inverse de A peut donc s'écrire :

$$A^{-1} = (D(I-U))^{-1} = \left(\sum_{k=0}^{n-1} U^k\right) D^{-1}$$

Le calcul des n premières puissances de U peut être effectué en utilisant un algorithme de type préfixe parallèle, avec comme loi associative le produit de deux matrices. Il suffit d'effectuer ensuite la somme de ces puissances (de type somme itérée) et de multiplier cette somme par la matrice D^{-1} pour obtenir la matrice A^{-1}. On obtient ainsi un algorithme de complexité $O_{//}\left(\log^2 n, nM(n)\right)$ sur une EREW PRAM.

Cet algorithme explicite la relation algébrique qui est utilisée de manière implicite lorsque l'on utilise l'évaluation parallèle dans un semi-anneau. Les meilleurs algorithmes connus aujourd'hui pour des matrices quelconques dans un anneau sont basées sur des relations analogues (via le polynôme caractéristique ou le polynôme minimal)

1.5 Conclusion.

Les études de complexité théorique permettent de mieux cerner le parallélisme intrinsèque à un problème. De nombreux problèmes restent à classer : un exemple typique est le pgcd d'entiers [44], pour lequel le meilleur algorithme parallèle connu aujourd'hui améliore seulement d'un facteur poly-logarithmique le meilleur algorithme séquentiel [32]. Les techniques précédentes apportent des renseignements sur le problème lui-même; leur utilisation se situe donc en amont de la parallélisation sur une machine parallèle, pour laquelle non seulement le nombre de processeurs est fini mais les caractéristiques de l'architecture (notamment si elle est distribuée) sont déterminantes dans le développement d'algorithmes.

Chapitre 2

Algorithmes de travail optimal et probabilistes

2.1 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 [1]. 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 [46].

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 2.2, les outils introduits dans [50] 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, des algorithmes numériques qui donnent une approximation de la solution d'un problème. Notamment, les algorithmes numériques de type Monte-Carlo, qui fournissent une approximation aussi précise que désirée à partir de tirages aléatoires, ne sont pas des algorithmes probabilistes. La cible des algorithmes probabilistes est en général les problèmes

1. Ce chapitre est construit comme une suite au chapitre [45], et s'appuie sur les notions qui y sont présentées.

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 [34, 26, 10], nous introduisons les différents types d'algorithme probabilistes tout d'abord dans le cadre séquentiel (§2.3), puis parallèle (§2.4). Les paragraphes suivants sont consacrés à la présentation de trois techniques de base pour la construction d'algorithmes probabilistes :

- diviser pour paralléliser probabiliste (§2.5),
- introduction d'aléatoire par homomorphisme sur les entiers (§2.6),
- introduction d'aléatoire à partir d'une caractérisation algébrique (§2.7).

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 [18].

2.2 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 [35] [50].

2.2.1 Algorithmes d'accélération polynomiale

Soit $\mathcal{P}_n$ un problème, et soit $T_s(n)$ le temps du meilleur algorithme séquentiel résolvant ce problème. Soit $A^{(p)}$ 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 $A^{(p)}$ est dit d'*accélération polynomiale* [34] (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 $A^{(p)}$. 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 [45], un algorithme parallèle ne peut pas être *plus* rapide théoriquement qu'un algorithme séquentiel.

L'*inefficacité* de $A^{(p)}$ est dite :

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

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

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

2.2.2 Illustration : inversion de matrice

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

En parallèle, il existe des algorithmes d'inversion de matrice dans $\mathcal{NC}^2$ depuis longtemps [16]; mais ces algorithmes ne sont pas de travail optimal. Le meilleur algorithme dans NC^2 est de travail $n^{\omega+\frac{n}{2}}$ [38].

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 [51] – version "Divide & Conquer" de Gauss, de complexité n^ω, $\omega \simeq 2,38$ – fournit un

algorithme de complexité $O_{//}(n, n^{\omega-1})$. 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 [31].

2.3 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 $f(x)$.

2.3.1 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, alors que la deuxième fournira 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, par une augmentation soit du temps avec plusieurs exécutions séquentielles, soit du nombre de processeurs avec plusieurs exécutions parallèles.

2.3.2 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.

2.3.3 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 2.5.2.

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 [10].

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.

2.3.4 Une définition plus formelle

Nous reprenons ici la définition proposée dans [34] 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 $x\mathcal{R}y$. 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 $x\mathcal{R}y$,

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

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 $x\mathcal{R}y$.

Si, sur une entrée x, il existe un élément y vérifiant $x\mathcal{R}y$, 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 $x\mathcal{R}y$, 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)[2].

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 $X2$ 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").

2.4 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 [45], 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.

2.4.1 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

2. 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" [34].

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" [3] pour les algorithmes Las Vegas, et par "R" [4] 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.

2.4.2 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 $\mathcal{NC}^k$ (et $\mathcal{AC}^k$) 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 $\mathcal{RNC}^k$ est l'ensemble des problèmes qui peuvent être résolus par une famille log-uniforme (cf [45]) 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 $\frac{1}{2}$).

2.5 Diviser pour paralléliser probabiliste

La technique "Diviser Pour Régner" parallèle [45] 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 inté-

3. "Z" pour "Zero-error"
4. "R" pour "Random"

ressante si elle permet une découpe et une reconstruction très rapides (idéalement de temps parallèle constant).

2.5.1 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 [40]. 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[5].

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

Pourtant, l'algorithme de quicksort présenté au paragraphe 2.3.3 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, sans changer le travail de l'algorithme. Nous obtiendrons ainsi (2.5.2) 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 [26]

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

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

5. Il est à noter que la fusion bitonique peut être rendue optimale [5], mais le temps parallèle reste $O(\log^2 n)$.

2.5.2 Quicksort parallèle

L'algorithme 2.3.3 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.

2.5.3 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 [18]), la probabilité de l'union de n événements est plus petite que la somme des probabilités de ces événements. On en déduit que :

$$\text{Prob}_{\text{échec}}(t) \leq n\text{Prob}^{(e)}_{\text{échec}}(t) \tag{2.1}$$

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

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

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 :

$$Prob\left(n_{j+1} \leq \frac{3n_j}{4}\right) \geq \frac{1}{2} \tag{2.2}$$

Après $\log_{\frac{4}{3}} n$ étapes de partition réussie, 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 moins de $\log_{\frac{4}{3}} n$ partitions réussies.
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 $p = \frac{1}{2}$ (2.2).
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 :

$$\text{Prob}^{(e)}_{\text{échec}}(t) = \text{Prob}(X_t \leq \log_{\frac{4}{3}} n) \tag{2.3}$$

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

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

Cette probabilité peut être majorée en utilisant la borne de Chernoff [18], qui permet de quantifier les événements rares – théories des grandes déviations – :

$$\text{Prob}\left(X_t \leq (1-\epsilon)pt\right) \leq e^{-\frac{\epsilon^2 tp}{2}} \tag{2.4}$$

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 (2.3) on obtient :

$$\text{Prob}\left(X_t \leq \log_{\frac{4}{3}} n\right) \leq e^{-\frac{\epsilon^2 K \log_{\frac{4}{3}} n}{4}}. \tag{2.5}$$

Comme $e^{\log_{\frac{4}{3}} n} = n^{\frac{1}{\log \frac{4}{3}}}$, il reste d'après (2.1) à déterminer K tel que :

$$\left(\frac{K-2}{K}\right)^2 \frac{K}{\log \frac{4}{3}} > 1. \tag{2.6}$$

$K = 4$ 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.

2.5.4 Tri parallèle probabiliste optimal

Pour accélérer la phase de partition dans l'algorithme précédent, il est possible de choisir directement plusieurs pivots de façon à obtenir, en une étape de partition, plusieurs seaux qui peuvent être triés ensuite en parallèle. C'est le principe de l'algorithme suivant proposé par Reishuk [41], qui peut être vu comme une généralisation de l'algorithme précédent.

1. Tirer en parallèle au hasard $\sqrt{n} - 1$ éléments dans le tableau de n éléments T à trier. On pose $k = \sqrt{n}$.

2. Trier ces éléments (en utilisant un algorithme de tri par sélection par exemple). Soient $p_0 = -\infty, p_1, \ldots, p_{k-1}, p_k = +\infty$ les pivots ainsi obtenus.

3. Deux pivots consécutifs (p_i, p_{i+1}) définissent alors un seau B_i, $0 \leq i < k$. Le tableau à trier est alors restructuré, en plaçant chacun de ses éléments dans le seau qui lui correspond.

4. Le tri peut alors être terminé en appliquant le même algorithme récursivement à chacun des seaux B_i.

La complexité de cet algorithme est $O_{//}(\log n, n)$. Nous donnons ci-dessous une justification intuitive de ce coût. L'analyse détaillée fait appel aux mêmes outils probabilistes que la démonstration donnée pour l'algorithme précédent de quicksort parallèle. Une démonstration précise de ce coût peut être trouvée dans [41].

Nous supposerons donc ici pour simplifier l'analyse que le nombre d'éléments dans chacun des seaux B_i est de l'ordre de $\sqrt{n}$, c'est à dire que les pivots sont

bien choisis (ce qui correspond au cas moyen). Le coût de chacune des étapes est le suivant :

- Tri par sélection des $\sqrt{n}$ pivots : $O_{//}(\log n, n)$.
- Formation des seaux B_i. En structurant les pivots dans un arbre équilibré, le calcul de l'indice I_k du seau B_{I_k} correspondant à un élément $x_k, 0 \leq k < n$ quelconque du tableau se ramène en séquentiel à une recherche dans un arbre équilibré. Une telle recherche nécessite un temps $\log \sqrt{n}$ en séquentiel pour chaque élément. Pour tous les éléments en parallèle, on peut donc calculer le tableau $I_k, 0 \leq k < n$ tel que l'élement x_k est dans le seau B_{I_k} avec un coût $O_{//}(\log n, n)$.
  Il reste alors à compacter les seaux. Par une technique analogue à celle utilisée pour l'algorithme de quicksort précédent (utilisation de préfixes somme), cette opération peut être effectuée avec un coût $O_{//}(\log n, n)$.

Le coût temporel de cet algorithme récursif est alors :

$$T(n) \leq T(\sqrt{n}) + \alpha \log n = O(\log n)$$

et le nombre de processeurs nécessaires :

$$H(n) = Max(n, sqrtnH(\sqrt{n})) = n.$$

En conclusion, on obtient donc un algorithme de complexité parallèle $O_{//}(\log n, n)$. Cet algorithme est optimal en travail.

Remarque. La faible complexité de communication de cet algorithme – O(n) – et son travail très proche de celui du quicksort séquentiel (qui est le meilleur algorithme pratique) font que cet algorithme permet d'obtenir de très bonnes performances expérimentales lors de l'implémentation sur un support distribué [19]. Il est cependant alors nécessaire d'une part de stopper la découpe récursive dès que la taille des seaux est inférieure à un seuil de granularité et de mettre en œuvre une stratégie de régulation de la charge (ordonnancement en-ligne), les seaux étant de taille variable et le temps de l'algorithme de tri séquentiel imprévisible.

2.6 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[6] – défini dans un ensemble E, une méthode consiste à transformer le problème initial

6. 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.

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 [42], 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 [53].
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 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.

2.6.1 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 [2]. En séquentiel, le premier algorithme optimal de complexité $O(n+m)$ est dû à Boyer et Moore [9], 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 $\mathcal{NC}^1$, 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 [33] [2]. Des extensions de cet algorithme pour des problèmes de filtrage multi-dimensionnel sont présentés dans [26].

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 $\mathcal{E}$

$$\mathcal{E} = \{M \in \mathcal{M}_{2,2} / \det(M) = 1\}$$

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 :

$$\begin{cases} \Phi(\epsilon) = I \\ \Phi(a \wedge b) = \Phi(a)\Phi(b) \quad \forall (a,b) \in E^2 \end{cases} \tag{2.7}$$

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 :

$$\Phi('0') = \begin{pmatrix} 1 & 0 \\ 1 & 1 \end{pmatrix} \qquad \Phi('1') = \begin{pmatrix} 1 & 1 \\ 0 & 1 \end{pmatrix} \tag{2.8}$$

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 $2^{\text{ième}}$ ligne de $\Phi(s)$ par la somme de ses 2 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 $\Phi(x_1, \ldots, x_m) = M_X$
2. Calcul de $A_k = \Phi('y_0 \ldots y'_k)$, $0 \leq k \leq n$ (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 $k = 0, \ldots, n-m$, 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 :

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

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 comparant les chaînes de caractères.

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 :

$$N_k \bmod p = 0,$$

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 :

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

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

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, la vérification déterministe prend une complexité $O_{//}\left(\log m, \frac{nm}{\log m}\right)$, 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.

2.7 Test de nullité d'un polynôme

De nombreux problèmes peuvent être ramenés au problème de décidabilité de la nullité d'une expression algébrique [53], 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 [42] 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 [49] (dont une démonstration peut être trouvée dans [26]), 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 [49]. *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 :

$$Prob(P(\alpha_1, \ldots, \alpha_n) = 0) \leq \frac{d}{c} \tag{2.9}$$

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 [26] et en calcul algébrique [8]. Deux exemples en algèbre linéaire sont présentés ci-après : la vérification d'un produit de matrices et le calcul du rang d'une matrice.

2.7.1 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[7]

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

La propriété (2.9) permet de construire un algorithme de Monte-Carlo de complexité arithmétique

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

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

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

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

$$(AB - C)x = 0 \quad \forall x \in R^n. \tag{2.10}$$

Posons $x = [x_1, \ldots, x_n]^t$ et $P(x) = ABx - Cx$. L'équation (2.10) 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 (2.9), 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 donc une probabilité d'échec inférieure à $\frac{1}{2^k}$.

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

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 [8] 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.

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, le polynôme $\delta_r(L, C)$ ne peut 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 (2.9), 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^2_K, on en déduit que le calcul du rang dans un corps K quelconque[8] appartient à REREW^2_K.

8. **Cas où K est de cardinal fini.** Si K est de cardinal fini, [8], 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 suffisante [23].

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.

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 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 $O(M(n))$.

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_F^2 (algorithme déterministe de Mulmuley [37]), 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 [23], 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 [29] pour ce problème ont été trouvés avant des algorithmes déterministes [47]. Des algorithmes probabilistes efficaces sont connus [23].

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 [47, 21].

2.8 Conclusion

Nous avons introduit dans ce chapitres 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 [6] et en combinatoire [26].

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 [36] pour rendre déterministe un algorithme probabiliste, sans augmenter d'un facteur plus que poly-logarithmique le travail de l'algorithme.

Bibliographie

[1] Aho (A.), Hopcroft (J.) et Ullman (J.). – *The design and analysis of computer algorithms.* – Addison-Wesley, 1974.

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

[3] Akl (S.). – *Parallel Sorting Algorithms.* – Academic Press, 1985.

[4] Balcázar (J.), Diaz (J.) et Gabarró (J.). – *Structural Complexity II.* – Springer-Verlag, 1989.

[5] 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.

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

[7] Borodin (A.). – On relating time and space to size and depth. *SIAM Journal of Computing*, vol. 6, 1977, pp. 733–744.

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

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

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

[11] Brent (R.). – The parallel evaluation of general arithmetic expressions. *J. ACM*, vol. 21, 1974, pp. 201–206.

[12] Cole (R.). – Parallel merge sort. *SIAM J. Computing*, vol. 17, 1988, pp. 770–785.

[13] Cook (S.). – A taxonomy of problems with fast parallel algorithms. *Information and Control*, vol. 64, 1985, pp. 2–22.

[14] Cook (S.), Dwork (C.) et Reischuk (R.). – Upper and lower time bounds for parallel random access machines with simultaneous writes. *SIAM J. Computing*, vol. 15, 1986, pp. 87–97.

[15] Cosnard (M.) et Trystram (D.). – *Algorithmes et Architectures Parallèles.* – Interéditions, 1993.

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

[17] Culler (D.), Karp (R.), Patterson (D.), Sahay (A.), Schausser (K.), Santos (E.), Subramonian (R.) et von Eicken (T.). – LogP: Towards a realistic model of parallel computation. *In : proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pp. 1–12.

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

[19] Doreille (M.), Cavalheiro (G.) et Roch (J.-L.). – Régulation dynamique en athapascan : exemple d'un tri parallèle probabiliste optimal. *In : RenPar'8*, pp. 181–184. – Bordeaux, France., mai 1996.

[20] Gathen (J. v. z.). – Algebraic complexity theory. *Ann. Rev. Comput. Sci.*, vol. 3, 1988, pp. 317–347.

[21] 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.

[22] Gibbons (A.) et Rytter (W.). – *Efficient parallel algorithms.* – Cambridge University Press, 1988.

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

[24] Goldschlager (L.). – The monoton and planar circuit value problems are log space complete for P. *SIGACT News*, vol. 9, 1977, pp. 25–29.

[25] Hoover (H.), Klawe (M.) et Pippenger (N.). – Bounding fan-out in logical networks. *J. ACM*, vol. 31, 1984, pp. 13–18.

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

[27] Kaltofen (E.). – *Parallel algebraic computing design - Tutorial ISSAC 89, Portland.* – Rapport technique, Rensselaer Polytechnic Institute, 1989.

[28] Kaltofen (E.), Krishnamoorthy (M.) et Saunders (B.). – Fast parallel computation of Hermite and Smith forms of polynomials matrices. *SIAM J. Alg. Disc. Meth.*, vol. 8 4, pp 683-690, 1987.

[29] 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.

[30] Kaltofen (E.), Miller (G.) et Ramachandran (V.). – Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Computing*, vol. 17, 1988, pp. 687–695.

[31] 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.

[32] Kannan (R.), Miller (G.) et Rudolph (L.). – Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM J. Comput.*, vol. 16 1, 1987.

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

[34] 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.

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

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

[37] 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.

[38] Pan (V.). – Complexity Of Computations with Matrices and Polynomials. *SIAM Review*, vol. 34, n° 2, June 1992, pp. 225–262.

[39] Pippenger (N.). – On simultaneous ressource bounds. *In : 20 th. Annual IEEE Foundations of Computer Science*, pp. 307–311.

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

[41] Reischuk (R.). – Probabilistic parallel algorithms for sorting and selection. *SIAM J. Computing*, vol. 14, n° 2, 1985, pp. 396–409.

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

[43] Roch (J.) et Villard (G.). – Fast parallel Jordan normal form computation. To appear in Parallel Processing Letters.

[44] Roch (J.) et Villard (G.). – Parallel gcd and lattice basis reduction. *In : CONPAR 92, Lyon France.*

[45] 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.

[46] 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.

[47] 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.*

[48] Ruzzo (W.). – On uniform circuit complexity. *J. Computer and System Sciences*, vol. 22, 3, 1981, pp. 365–383.

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

[50] 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.

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

[52] Valiant (L.). – A bridging model for parallel computation. *Communication ACM*, vol. 33, 1990, pp. 103–111.

[53] 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.