Complexité Parallèle
et
Algorithmique PRAM

Edité dans l'ouvrage Algorithmes Parallèles : Analyse et Conception, *Hermès, eds. C. Roucairol et al., chap. 5, p. 105–126, 1994.*

J.L. Roch
jlroch@imag.fr

LMC-IMAG, 46 Av. F. Viallet F38031 Grenoble Cedex.

Le but de ce chapitre 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" [22] 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 [19], 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

⋆. Je remercie Gilles Villard et Nathalie Revol pour leurs remarques pertinentes et correxion attantive.

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 [28].

1 Le modèle PRAM

1.1 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 [7]. 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 [9]. 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 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 [3]):

- 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.

1.3 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 [2].

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.

2. 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).

– 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.

Remarques. 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.
Lorsque l'on considère un nombre fini de processeurs (cas des machines parallèles), un algorithme parallèle de travail optimal, même si sa complexité temporelle n'est pas poly-logaritmique, peut être très intéressant en pratique, si il peut premettre d'obtenir une efficacité expérimentale minorée par une constante indépendante du nombre de processeurs (ce qui garantit l'extensibilté -*scalability*- pour des instances de problèmes suffisament grandes). Pour caractériser de tels algorithmes, la definition d' "algorithme extensible avec une taille quasi-constante" a été introduite [27] pour un algorithme ayant une surface (nombre de processeurs) supérieure à $\frac{n}{\log^{O(1)} n}$, et un travail de l'ordre de $T_s(A_s)$ (c'est à dire de *travail optimal*). Les techniques présentées dans ce chapitre sont adaptées à la conception de tels algorithmes, même si l'on ne se limitera dans la suite qu'à l'utilisation d'un nombre polynômial de processeurs, et que l'on s'attachera à l'obtention d'algorithmes de complexité temporelle poly-logaritmique.

1.4 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).

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 [9]). 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 [19]) :

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

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 [15]. 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.*

2 La classification NC

2.1 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 [7] [19].

La classe NC, formalisée par Nicholas Pippenger [21] (et nommée par Cook NC pour "Nick's class" [7]), 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:

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

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[3]. 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

3. Ce temps est même montré comme étant une borne inférieure pour ce problème sur une CREW PRAM (la complexité temporelle est $\Omega(\log n)$) [8].

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 [19] pour une introduction). Ainsi le calcul du rang d'une matrice a d'abord été montré comme étant dans RNC [11] avant d'être montré dans NC^2 [20]. Plus récemment, les formes de Smith ou de Jordan ont eu le même sort [16] [24].

2.2 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 [4] : la NC-réductibilité.

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 [4].

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 [26]. 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 [19] :

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

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 [10] 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).

4. 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 [14].

***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 [7]. 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 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) [13] : " 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 [5]."

3 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).

3.1 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 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).

5. Tout problème s'exécutant en temps polynômial sur une machine de Turing déterministe peut être NC^1-réduit à ce problème [19], ce qui prouve, étant clairement dans P, qu'il est P-complet.

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...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...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.

3.2 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.

3.3 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 :

$$\begin{cases} c_{-1} = & 0 \\ r_k & = (a_k + b_k + c_{k-1}) \mathrm{m} od 2 \\ c_k & = (a_k + b_k + c_{k-1}) \mathrm{d} iv 2 \end{cases}$$

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 fusionner$A_L + B_L$ et $A_H + B_H + 1$ sinon fusionner$A_L + B_L$ 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, ...[2]). 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 $O_{//}\left(\log n, n^2\right)$ 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 $O_{//}(log\ n,\ n)$ [6].

3.4 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)[23].

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 [5] [12].

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 [17] : 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 programme 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 [23]. 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 : $A = D\ (\ I\text{ - }U)$

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)

4 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 [25], pour lequel le meilleur algorithme parallèle connu aujourd'hui améliore seulement d'un facteur poly-logarithmique le meilleur algorithme séquentiel [18]. 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.

REFERENCES

1. A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The design and analysis of computer algorithms.* Addison-Wesley, 1974.

2. S.G. Akl. *Parallel Sorting Algorithms*. Academic Press, 1985.
3. J.L. Balcázar, J. Diaz, and J. Gabarró. *Structural Complexity II*. Springer-Verlag, 1989.
4. A. Borodin. On relating time and space to size and depth. *SIAM Journal of Computing*, 6:733–744, 1977.
5. R.P. Brent. The parallel evaluation of general arithmetic expressions. *J. ACM*, 21:201–206, 1974.
6. R. Cole. Parallel merge sort. *SIAM J. Computing*, 17:770–785, 1988.
7. S.A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.
8. S.A. Cook, C. Dwork, and R. Reischuk. Upper and lower time bounds for parallel random access machines with simultaneous writes. *SIAM J. Computing*, 15:87–97, 1986.
9. M. Cosnard and D. Trystram. *Algorithmes et Architectures Parallèles*. Interéditions, 1993.
10. J. von zur Gathen. Parallel arithmetic computations: a survey. In *Proc. 12th Int. Symp. Math. Found. Comput. Sci., Bratislava*, pages 93–112. Springer-Verlag LNCS 233, 1986.
11. J. von zur Gathen. Algebraic complexity theory. *Ann. Rev. Comput. Sci.*, 3:317–347, 1988.
12. A.M. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1988.
13. L.M. Goldschlager. The monoton and planar circuit value problems are log space complete for P. *SIGACT News*, 9:25–29, 1977.
14. H.J. Hoover, M.M. Klawe, and N.J. Pippenger. Bounding fan-out in logical networks. *J. ACM*, 31:13–18, 1984.
15. E. Kaltofen. Parallel algebraic computing design - *Tutorial ISSAC 89*, Portland. Technical report, Rensselaer Polytechnic Institute, 1989.
16. E. Kaltofen, M.S. Krishnamoorthy, and B.D. Saunders. Fast parallel computation of Hermite and Smith forms of polynomials matrices. *SIAM J. Alg. Disc. Meth.*, 8 4, pp 683-690, 1987.
17. E. Kaltofen, G.L. Miller, and V. Ramachandran. Efficient parallel evaluation of straight-line code and arithmetic circuits. *SIAM J. Computing*, 17:687–695, 1988.
18. R. Kannan, G. Miller, and L. Rudolph. Sublinear parallel algorithm for computing the greatest common divisor of two integers. *SIAM J. Comput.*, 16 1, 1987.
19. R.M. Karp and V. Ramachandran. Parallel algorithms for shared-memory machines. In J. van Leuwen, editor, *Algorithms and Complexity*, pages 869–932. Elsevier, 1990.
20. K. Mulmuley. A fast parallel algorithm to compute the rank of a matrix over an arbitrary field. *Combinatorica*, 7(1):101–104, 1987.
21. N. Pippenger. On simultaneous ressource bounds. In *20 th. Annual IEEE Foundations of Computer Science*, pages 307–311, 1979.
22. B. Plateau, A. Rasse, J.L. Roch, and J.P. Verjus. *Parallélisme*. Polycopiés ENSIMAG, 1991.
23. N. Revol. Evaluation parallèle de circuits arithmétiques. Technical Report Pré-rapport de thèse, IMAG, 1993.
24. J.L. Roch and G. Villard. Fast parallel Jordan normal form computation. Submitted to Parallel Processing Letters (april 93).
25. J.L. Roch and G. Villard. Parallel gcd and lattice basis reduction. In *CONPAR 92, Lyon France*, LNCS 634, September 1992.
26. W.L. Ruzzo. On uniform circuit complexity. *J. Computer and System Sciences*, 22, 3:365–383, 1981.
27. M. Snir. Scalable parallel computers and scalable parallel codes: From theory to practice. In *Parallel Architectures and their Efficient Use, Paderborn, Germany, 1992, Lect. N. Comp. Sc. 678 pp 176-184*, 1992.
28. G. Villard and J.L. Roch. Fast parallel algorithms for matrix normal forms. Technical Report RT, IMAG Grenoble F, 1993.