Complexite de l'evaluation parallele des circuits arithmetiques

Nathalie Revol

THESE

présentée par

Nathalie REVOL

pour obtenir le titre de DOCTEUR

de l'**INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE**

(Arrêté ministériel du 30 Mars 1992)

Spécialité : **Mathématiques Appliquées**

Complexité de l'évaluation parallèle de circuits arithmétiques

---

Date de soutenance : 31 août 1994

Composition du jury :

Président	François	ROBERT
Rapporteurs	François	BACCELLI
	Bruno	GAUJAL
	Jean-Michel	MULLER
Examinateurs	Jean	DELLA DORA
	Jean-Louis	ROCH
Invité	Paul	JACQUET

Thèse préparée au sein du
Laboratoire de Modélisation et de Calcul

2

Complexité de l'évaluation parallèle de circuits arithmétiques

Nathalie Revol

Thèse de Mathématiques Appliquées de l'Institut National Polytechnique de Grenoble
préparée au Laboratoire de Modélisation et Calcul
sous la direction de Jean-Louis Roch et Jean Della Dora

Voici venu le temps de mettre un point final à cette thèse en remerciant tous ceux qui ont contribué à son élaboration.

Tout d'abord, je remercie ceux qui ont composé mon jury. Merci à toi, François, qui n'as pas seulement accepté de le présider, mais qui as bien voulu être mon tuteur en enseignement ; tu as effectivement rempli ton rôle et tu m'as toujours encadrée et conseillée avec « paternalisme » (sic).

Je vous remercie également, François et Bruno, qui avez accepté avec une grande gentillesse d'être les rapporteurs de cette thèse et qui n'avez pas hésité à consacrer une part importante de votre temps à la lire et la relire, la comprendre et la corriger.

Que mon autre rapporteur soit remercié aussi, non seulement pour avoir relu ma thèse pendant ses vacances, mais également pour le temps que nous avons passé à travailler ensemble au sein du groupe « Arithmétique des Ordinateurs ». Merci particulièrement pour toutes les fois où tu as accepté de te pencher sur mes problèmes. Merci également pour ton humour. Ô Jean-Michel, que le grand Manitou te protège et te récompense !

Paul Jacquet a également accepté de participer à ce jury. Je l'en remercie et cela s'ajoutera à l'excellent souvenir que je garde du temps où j'étais son élève.

Jean Della Dora mérite aussi ma reconnaissance pour avoir été mon directeur de thèse, pour m'avoir accordé sa confiance pour enseigner les TD et le projet d'Équations Différentielles Ordinaires sur son cours, pour sa bonne humeur et pour quelques discussions philosophiques sur la recherche du bonheur...

Jean-Louis, tu as droit à une mention spéciale, puisque c'est toi qui m'as encadrée dans ce travail de recherche. Merci pour ton enthousiasme (et le mot est faible), pour être une source féconde et inépuisable d'idées, pour les bonnes pistes que tu m'as indiquées et pour tous les autres moments de travail et de discussions moins professionnelles mais tout aussi animés et intéressants.

Merci encore à Gilles pour diverses causeries et pour des fous-rires partagés (après quelques facéties), à Jean-Marc pour ses relectures attentives et ses conseils judicieux, au grand Jacques pour nos discussions sur des thèmes variés et son assiduité natatoire, à Joëlle, autre fidèle compagne de piscine, mais aussi dépanneuse système. Merci aussi à Françoise pour ses œufs à la neige incomparables ainsi que pour une expédition mémorable aux Karellis, à Denis pour m'avoir fait découvrir le yoga et à tous ceux que j'oublie.

Côté thésards, merci aux deux malheureux qui ont réussi à partager mon bureau, Philippe qui a apparemment bien survécu et Alain qui est là depuis trop peu de temps pour que les effets de cette « cohabitation » soient visibles, mais qui a déjà durement été mis à contribution lors de la relecture de ce mémoire. Merci aussi à mes autres relecteurs idéaux, rapides et fiables, Alexandre et Séverine, Thierry, Cécile. Merci aux autres frangins, Laurent et Pascal, pour tous les bons moments passés ensemble. Merci enfin aux autres étudiants, qu'ils soient des gens de calcul formel comme Laurent, Isabelle, Clemens, Inès ou Evelyne, ou de parallélisme comme Frédéric, Yannick ou Christophe. Merci aussi aux Lyonnais, à Michel d'abord pour les longues heures passées à débugger mes premiers balbutiements de C, à Xavier et Christophe, à Marc et aux autres. Merci à Mathias pour le travail réalisé en enseignement.

Le dernier, mais pas le moindre, que je voudrais remercier est Thierry. Tu as partagé avec moi tout ce temps de thèse (et même plus), tu as su être à mes côtés, m'écouter (le pauvre), me conseiller, me consoler et même me faire une grosse frayeur, mais heureusement tout cela est désormais de l'histoire ancienne...

Sommaire

Introduction

En parallélisme, utiliser efficacement les ordinateurs dédiés requiert la construction d'algorithmes spécifiques. Un cadre général pour l'algorithmique parallèle est le modèle *PRAM*. Ce modèle, bien que théorique, est toujours valide, d'une part parce qu'il permet de séparer les problèmes très parallèles des autres et d'autre part parce que les schémas algorithmiques associés à ce modèle sont utilisés et validés pour l'écriture d'algorithmes sur des machines parallèles réelles. Par exemple, le schéma « diviser pour paralléliser » est utilisé dans *Athapascan* (le langage des Apaches, développé au Laboratoire de Modélisation et Calcul et au Laboratoire de Génie Informatique de l'IMAG) afin d'adapter la granularité du problème au grain de la machine et donc de permettre une portabilité des programmes.

Le modèle *PRAM* n'est certes pas la panacée en matière de parallélisation effective ; par exemple, seuls les algorithmes efficaces et optimaux (ou asymptotiquement efficaces et optimaux) peuvent être candidats à un usage pratique, parce qu'ils effectuent un nombre d'opérations comparable au nombre d'opérations d'un algorithme séquentiel résolvant le même problème : les machines actuelles ne sont pas surdimensionnées, en nombre de processeurs, par rapport au problème à traiter. Par ailleurs, il existe des problèmes qui admettent une solution parallèle théorique excellente, mais cette solution ne s'étend pas aux machines à mémoire distribuée, comme le problème de l'indiçage des éléments d'une liste.

Les algorithmes parallèles théoriques n'offrent donc pas des solutions idéales pour la parallélisation sur machine réelle ; cependant, ils ont prouvé et prouvent encore leur intérêt, ne serait-ce que par leur indépendance par rapport à toute machine, qui les rend indémodables. Bien plus, certains schémas classiques en algorithmique *PRAM*, de type réduction et autres opérations globales, permettent de construire des programmes efficaces (même si l'identification de ces motifs dans un programme est parfois difficile). La meilleure preuve en est leur intégration comme « fonctions intrinsèques » dans les langages parallèles modernes : la réduction est une fonction *intrinsic* de HPF (*High Performance Fortran*), l'extension de Fortran pour le parallélisme en cours de définition et appelée à devenir un standard.

Parvenir à repérer, dans un algorithme donné, ces schémas dont le parallélisme peut être efficacement exploité, peut être source de nouveaux algorithmes parallèles pratiques plus performants. Dans ce domaine, l'exemple de l'addition d'entiers par l'algorithme de Brent et Kung [BK82] est caractéristique : cet algorithme repose sur la mise en évidence d'une réduction dans ce problème, séquentiel de prime abord. De manière générale, de nombreux

algorithmes $PRAM$ sont basés sur une même idée : l'anticipation. Que ce soit pour le produit itéré, l'indiçage de liste ou le saut de pointeur, c'est toujours cette idée d'anticipation, liée dans les cas cités à l'associativité d'une opération, qui est exploitée. Elle peut s'étendre à des structures algébriques plus complexes que celles ne possédant qu'une seule loi associative. Des techniques ont été proposées pour les semi-anneaux, le cas des treillis est traité dans ce mémoire.

Le modèle d'algorithme sous-jacent est celui des circuits arithmétiques, introduit par von zur Gathen dans le cadre de la complexité parallèle. Historiquement, le problème de l'évaluation des expressions a été examiné en premier, dès 1974 [Bre74], ensuite le cas des programmes sans boucle dans les semi-anneaux ou dioïdes a été étudié à partir de 1986 [MRK86]. La première partie de ce mémoire présente tous ces algorithmes. Nous proposons des adaptations pour les structures de corps, souvent utilisés aussi bien en calcul formel qu'en calcul numérique. Nous nous intéressons ensuite à la structure de treillis. L'intérêt de cette dernière est double : d'une part il y a de nombreuses applications (en arithmétique par exemple) et d'autre part l'évaluation de circuits booléens est le problème de référence en complexité parallèle. (Ce problème, P-complet, est l'analogue du problème de la satisfaisabilité, NP-complet, pour la complexité séquentielle).

Cette thèse est structurée de la façon suivante :
un chapitre introductif met en place les notions de base en complexité séquentielle (problèmes P-complets en particulier), en algorithmique parallèle et en complexité parallèle, puis définit les structures algébriques utilisées : monoïde, groupe, semi-anneau ou dioïde, anneau, corps, treillis. Nous nous limiterons à l'étude des expressions dans des corps et à celle des circuits (programmes sans test) dans des semi-anneaux unitaires intègres et des treillis distributifs.

Dans un deuxième chapitre, les différents algorithmes d'évaluation pour les expressions sont présentés de manière synthétique et sont replacés dans un canevas général.

Le troisième chapitre est consacré à l'évaluation de circuits dans des semi-anneaux ou dioïdes. Nous rappelons l'algorithme pour les semi-anneaux de Miller, Ramachandran et Kaltofen, puis nous proposons une extension pour les corps et montrons son application à la résolution de systèmes linéaires triangulaires. Par ailleurs, d'autres extensions de ces algorithmes (corps, anneaux finis non commutatifs, $(\mathbb{R}, min, max, +)$) sont présentées.

Le quatrième chapitre concerne le cas des circuits dans des treillis distributifs. Les algorithmes précédemment proposés pour évaluer ces circuits sont de simples adaptations de l'algorithme pour les dioïdes. Nous proposons un nouvel algorithme dédié à la structure de treillis distributif, dont nous déterminons la complexité. Dans tous les cas, ce nouvel algorithme améliore les résultats obtenus précédemment.

Le cinquième chapitre présente quelques applications (calcul formel, réseaux de Petri temporisés) et des simulations (arithmétique). La définition d'un simulateur est importante, parce que les bornes de complexité qui sont données sont toujours au pire. Cette simulation permet, sur un cas plus spécifique, d'affiner et de vérifier ces bornes. Il n'y a pas, à notre connaissance, d'autres implémentations effectuées dans le cas des circuits, mais seulement des travaux sur les expressions.

Enfin, le dernier chapitre traite d'un prototype de compilateur basé sur ces techniques.

Un accent particulier est mis sur les produits de matrices (qui constituent l'opération centrale de l'algorithme en terme de coût, temporel comme en nombre de processeurs), d'un point de vue aussi bien théorique que pratique. Une autre application possible est l'aide à la recherche de schéma de type réduction en parallélisation automatique : ces opérations ne peuvent être décelées que par une analyse de la sémantique des opérations.

Ce travail de thèse rassemble les différents travaux sur le thème de l'évaluation parallèle de programmes sans boucle. Notre contribution essentielle est l'algorithme, présenté au chapitre 4, d'évaluation de ces programmes quand les opérations s'effectuent dans des treillis, qui a une meilleure complexité que les algorithmes précédemment connus pour ce problème.

Chapitre 1

Problème - Définitions

Résumé

Ce chapitre introductif pose le problème de l'évaluation parallèle des expressions et des circuits arithmétiques. Il contient également les définitions utiles pour la suite : du côté complexité séquentielle, problèmes P-complets ; modèle de parallélisme – $PRAM$ et circuits booléens – ainsi que la classification associée des programmes parallèles et la classe NC ; les principes de base de l'algorithmique parallèle : étudier le graphe de dépendance, diviser pour paralléliser, équilibrer les travaux, introduire de la redondance et penser parallèle. Enfin, les structures algébriques de monoïde, groupe, semi-anneau, anneau, corps et treillis sont définies.

1.1 Position du problème

Dans cette thèse, le problème central est la recherche d'algorithmes parallèles rapides afin d'évaluer certains programmes séquentiels. La classe de programmes séquentiels traitée est celle des expressions et des programmes sans boucle. Pour les expressions, il existe des algorithmes parallèles très performants qui les évaluent. Quant aux programmes sans boucle, il s'agit de programmes où les seuls tests autorisés portent sur la taille des entrées uniquement et jamais sur la valeur de celles-ci. Actuellement, les algorithmes d'évaluation parallèle de ces programmes ont des performances très dépendantes du programme à évaluer. De fait, évaluer un tel programme est un problème connu pour être (assez) difficile en séquentiel et difficile en parallèle, puisqu'il s'agit d'un problème P-complet et qu'il est conjecturé que les problèmes P-complets n'admettent pas de solution parallèle rapide. Le choix de cette classe de programmes (expressions et programmes sans boucle) est motivé par le désir de représenter une assez grande classe de programmes, sans s'imposer de problèmes insurmontables : les tests quelconques – qui sont une contrainte de séquentialité puisqu'il faut avoir calculé la valeur testée avant de savoir sur quelle branche du programme s'engager – constituent un problème que l'on ne sait pas résoudre de façon exacte (à moins d'explorer en parallèle toutes les branches possibles, mais la taille du programme est alors exponentielle, ce qui sera exclu par les modèles de complexité choisis).

Savoir évaluer rapidement en parallèle un programme signifie qu'il est possible de produire un programme parallèle calculant le même résultat, en construisant le graphe de tâches de l'algorithme d'évaluation appliqué à ce programme (cf. chapitre 6). Il faut cependant que cette compilation ne soit pas dépendante d'une machine parallèle spécifique. Les modèles de machines parallèles abstraites, tels que le modèle $PRAM$ et le modèle des circuits booléens, sont assez généraux. Le modèle des circuits arithmétiques, qui sera utilisé pour représenter les programmes sans boucle, généralise le modèle des circuits booléens. La compilation présentée dans ce mémoire reprend, de façon automatique, quelques-uns des paradigmes de la programmation parallèle, comme l'introduction de redondance (calculs superflus en séquentiel) afin de briser des dépendances entre tâches, ou comme l'identification de schémas connus pour appliquer leur solution parallèle rapide. Enfin, il reste à savoir comment mesurer la qualité d'un programme parallèle. Les critères usuels sont les suivants : d'une part on espère que le programme parallèle calculera beaucoup plus vite que les programmes séquentiels qui résolvent le même problème, d'autre part le nombre de processeurs ne doit pas être irréaliste. En complexité parallèle, on estime qu'un temps polylogarithmique en la taille des entrées est satisfaisant et on limite le nombre de processeurs à être polynomial en la taille des entrées également, même si à l'heure actuelle les machines parallèles existantes sont loin d'offrir de telles ressources. On impose enfin aux programmes parallèles d'être faciles à décrire ; dans ce mémoire, cela sous-entend qu'il doit être possible, pour toute taille possible n des entrées, de construire en temps polynomial en n un programme parallèle résolvant le problème à n entrées.

Les techniques présentées ici utilisent un algorithme séquentiel résolvant le problème ainsi que des informations supplémentaires sur ce problème ou, plus précisément, sur une modélisation de ce problème conduisant à cette résolution séquentielle, à savoir la sémantique des opérations effectuées. Par sémantique, nous entendons les propriétés algébriques de ces opérations et la structure algébrique qu'elles utilisent. L'associativité des opérations par exemple est une propriété fondamentale et déjà utilisée pour construire explicitement des algorithmes parallèles, comme le calcul de préfixe. L'utilisation de ces propriétés permet d'anticiper certains calculs, en remplaçant par exemple $2 + (3 + x)$ par $(2 + 3) + x = 5 + x$, et également de mettre en évidence certains schémas, comme celui du préfixe, qui ne sont pas détectables facilement, comme dans le problème de l'addition de deux nombres à n bits.

Une dernière remarque sur les techniques de compilation du chapitre 6, qui répond à une critique possible sur le temps de compilation assez important et sur le fait qu'il faille compiler un programme pour chaque taille possible des entrées, concerne leur application à la compilation de circuits VLSI : dans ce domaine, le problème est de concevoir un circuit le plus rapide possible, dédié à un seul calcul et pour lequel la taille des entrées est fixée (par exemple, un additionneur flottant 64 bits). Le temps de conception n'est pas limité (dans la mesure où il reste raisonnable, c'est-à-dire polynomial), seul le résultat importe. C'est typiquement ce que sait faire notre compilateur.

Dans ce chapitre, nous rappelons les définitions usuelles en complexité séquentielle, puis nous définissons précisément le modèle de machine parallèle utilisé, ainsi que les quantités

mesurant les performances d'un algorithme. Une classification des problèmes parallèles sera présentée à l'aide des classes NC^k et NC, qui seront définies dans la prochaine section ; cette classification sera comparée à la classification analogue des problèmes séquentiels, la classe P. Les principes généraux de l'algorithmique parallèle seront ensuite présentés et illustrés par les problèmes classiques du calcul des préfixes et de l'indiçage des éléments d'une liste. Ces deux algorithmes seront réutilisés par la suite. Un dernier exemple sera présenté pour illustrer la différence qui peut exister entre un algorithme séquentiel classique, celui de calcul du rang d'une matrice par l'élimination de Gauss, et l'algorithme parallèle de Chistov qui résout le même problème. Enfin, les définitions des structures algébriques utilisées seront rappelées : monoïde, groupe, semi-anneau et anneau unitaires intègres, corps et treillis. Ce chapitre s'inspire des cours de parallélisme de B. Plateau, A. Rasse, J.L. Roch et J.P. Verjus [PRRV91] et de J.L. Roch[Roc94].

1.2 Modèle séquentiel

Nous allons tout d'abord rappeler la classe P des problèmes séquentiels envisagés et, parmi ces problèmes, en isoler quelques-uns particulièrement représentatifs, les problèmes P-complets, auxquels tout autre problème appartenant à P peut se réduire.

1.2.1 Définitions : P, problèmes P-complets

Commençons par définir la classe des problèmes qui peuvent être résolus en séquentiel avec des ressources raisonnables. Le modèle de machine séquentielle le plus simple et le plus couramment adopté est le modèle des machines de Turing à une seule bande (il est suivi de près par le modèle RAM). Il s'agit d'une machine disposant d'une bande de longueur infinie, découpée en cases ou cellules, chaque cellule pouvant contenir un symbole d'un alphabet fini, et d'une tête de lecture/écriture se déplaçant sur cette bande. Cette tête est gérée par un automate à nombre fini d'états, qui, selon l'état courant dans lequel elle se trouve et le symbole écrit dans la case qu'elle est en train de lire, écrit un symbole, effectue éventuellement un déplacement d'une case sur la droite ou sur la gauche et change d'état. L'espace mémoire est le nombre de cases utilisées et le temps est le nombre de changements d'états.
Une machine de Turing est supposée répondre uniquement à des problèmes de décision, *i.e.* les seules réponses sont « oui » ou « non ». Cependant, il est possible de transformer le calcul d'une fonction $f : \Sigma^n \to \Sigma^n$, où Σ est un ensemble fini et où f résout un problème $\mathcal{P}$, en un problème de décision. Par abus de langage, on dira que la machine de Turing résout le problème $\mathcal{P}$. On distingue deux catégories de machine de Turing : celle où chaque couple (état courant, symbole lu), également appelé configuration, conduit à au plus un nouvel état, et les autres, *i.e.* celles où une configuration peut avoir plusieurs successeurs. Une machine de Turing appartenant à la première catégorie est dite déterministe, elle est non déterministe sinon. Tous les outils nécessaires à la définition des classes de complexité séquentielle sont maintenant réunis.

Définition 1 *La classe P est l'ensemble des problèmes résolus en temps polynomial par une machine de Turing déterministe.*

Remarque
Si le temps est polynomial, l'espace utilisé sera également au plus polynomial, puisqu'à chaque pas de temps, le déplacement est au plus d'une case.

Dans cette classe de problèmes, la relation d'ordre indiquant qu'un problème est plus « difficile » qu'un autre est la réductibilité en espace logarithmique.

Définition 2 *Un problème de décision A est réductible en espace logarithmique à un problème B si et seulement s'il existe une fonction f calculable en espace logarithmique sur une machine de Turing déterministe, telle que*

$$[x \text{ est la réponse au problème } A \Longleftrightarrow f(x) \text{ est la réponse au problème } B]\,.$$

On note ceci $A \leq B$.

Remarque
La notion de NC^1-réductibilité, qui sera définie au §1.3.5 et qui permet de comparer des algorithmes parallèles, aurait pu être utilisée, mais ce n'est pas la notion classique en complexité séquentielle ; les démonstrations utilisent la notion de réductibilité en espace logarithmique.

On omettra désormais le terme « en espace logarithmique ». Il existe des problèmes plus difficiles que tous ceux de P ; ce sont les problèmes auxquels tout problème de P peut se réduire.

Définition 3 *Un problème $\mathcal{P}$ est P-dur si et seulement si $\forall \mathcal{Q} \in P$, $\mathcal{Q}$ est réductible à $\mathcal{P}$.*

Théorème 1 *La classe P est close par réduction.*

Ce théorème signifie que P contient tous les problèmes qui se réduisent à un problème de P.

Les problèmes les plus intéressants pour étudier les relations entre P et d'autres classes de complexité sont les problèmes P-durs appartenant à P. Ils servent de représentants caractéristiques de P.

Définition 4 *Un problème $\mathcal{P}$ est P-complet si et seulement s'il appartient à P et il est P-dur.*

Remarque
Pour prouver qu'un problème $\mathcal{P}$ est P-complet, on montre qu'il appartient à P et qu'un problème P-complet déjà connu se réduit à $\mathcal{P}$.

1.2.2 Exemples de problèmes P-complets

Le CVP

Parmi ces problèmes représentatifs de P, certains vont jouer un rôle important dans cette thèse. Le premier qu'il faut mentionner est celui de la valeur d'un circuit (ou *Circuit Value Problem*) : il s'agit de déterminer le résultat d'un circuit booléen donné en entrée.

Définition 5 *Le problème de la valeur d'un circuit ou Circuit Value Problem, noté CVP, est le problème dont*

- *l'entrée est la donnée d'un circuit booléen à n entrées, une sortie et un nombre de portes polynomial en n, ainsi qu'un jeu de valeurs pour les entrées ;*
- *la sortie est la valeur du nœud de sortie (ou, pour avoir un problème de décision, « oui » si la valeur de sortie est égale à Vrai, « non » sinon).*

Théorème 2 *Le CVP est un problème P-complet [Lad75].*

DÉMONSTRATION

- Un circuit booléen, étant un programme sans boucle, dont les portes sont de *fan-in* égal à 2 et possédant un nombre polynomial de nœuds, est évalué en temps polynomial. Le CVP appartient à P.
- Montrons que le CVP est P-complet. Pour cela, réduisons un programme quelconque écrit pour une machine de Turing, qui s'exécute en temps polynomial, *i.e.* qui appartient à P, en un circuit booléen. Calculer le résultat du programme reviendra alors à déterminer la valeur du circuit booléen.
  Soit x une entrée de la machine de Turing. La machine de Turing peut être caractérisée (pour cette entrée x) à l'étape i par sa position sur la bande et par le symbole qui s'y trouve.
  - *calcul de la position.* Le tableau booléen $T = (T_{i,j})$ défini par $T_{i,j} = Vrai$ si et seulement si à l'étape i, la tête de la machine de Turing se trouve sur la j^e case de la bande, peut être calculé par des fonctions booléennes simples. Si $i = 0$ ou i est supérieur à la durée du calcul, ou si j est supérieur au plus grand numéro de case utilisée ou inférieur au plus petit numéro de case, la valeur de $T_{i,j}$ est une constante. Pour les autres valeurs de T, la remarque permettant de conclure est que $T_{i,j}$ ne dépend que du contenu de la case j ou des cases adjacentes $j-1$ et $j+1$ à l'étape $i-1$, *i.e.* $T_{i,j}$ ne dépend que de $T_{i-1,j-1}$, $T_{i-1,j}$ et de $T_{i-1,j+1}$. Chaque fonction booléenne qui calcule $T_{i,j}$ n'a que trois entrées, elle peut être codée en utilisant un espace logarithmique.
  - *calcul du symbole.* Une machine de Turing utilise un alphabet fini. Tout symbole de cet alphabet peut donc être codé en binaire avec une longueur finie m. De même que précédemment, on définit un tableau $S = (S_{i,j,k})$ avec i le numéro de

l'étape, j le numéro de la case, k représente le k^e bit du codage du symbole qui se trouve sur la case j à l'étape i. La même observation que précédemment permet d'affirmer que les fonctions qui calculent les $S_{i,j,k}$ ne dépendent que de $3m$ valeurs : $S_{i-1,j-1,1} \ldots S_{i-1,j-1,m}$, $S_{i-1,j,1} \ldots S_{i-1,j,m}$ et $S_{i-1,j+1,1} \ldots S_{i-1,j+1,m}$. Elles peuvent être codées en utilisant un espace logarithmique.

Puisque le programme s'exécute en temps polynomial sur la machine de Turing (et donc il utilise un espace polynomial), il y a un nombre polynomial de telles fonctions qui calculent l'état de la machine, ce qui nous permet de conclure que tout programme d'une machine de Turing se réduit, en espace logarithmique, à un circuit booléen de taille polynomiale.

□

Le $MCVP$

Le problème de la valeur d'un circuit monotone est un sous-cas du CVP, où les portes NON sont interdites. Le qualificatif de monotone provient du fait que les seules portes autorisées, ET et OU, sont des fonctions monotones croissantes (si on prend pour convention Vrai > Faux).

Définition 6 *Le problème du calcul de la valeur d'un circuit monotone, ou Monotone Circuit Value Problem, noté MCVP, est le problème dont*

- *l'entrée est la donnée d'un circuit booléen monotone à n entrées, une sortie et un nombre de portes polynomial en n, ainsi qu'un jeu de valeurs pour les entrées ;*
- *la sortie est la valeur du nœud de sortie.*

Théorème 3 *Le $MCVP$ est un problème P-complet [Gol77].*

DÉMONSTRATION

- Tout d'abord, le $MCVP$ appartient à P, puisqu'il se réduit trivialement au CVP et que P est close par réduction.
- Pour montrer que le CVP est réductible au $MCVP$, on construit un circuit monotone m qui calcule la même valeur que le circuit en entrée c du CVP. On suppose d'abord que le circuit monotone équivalent m dispose en entrée des mêmes variables que le circuit c : $(a_i)_{1 \leq i \leq n}$ et de leur négation ; la procédure de réduction peut le réaliser en espace logarithmique (il faut bien lire un codage de n). Ensuite, le circuit c est en quelque sorte dupliqué en une copie qui calcule la négation de chacune des portes de c, à l'aide des lois de De Morgan :

$$\begin{aligned} \neg(a \vee b) &= (\neg a) \wedge (\neg b), \\ \neg(a \wedge b) &= (\neg a) \vee (\neg b). \end{aligned}$$

La transformation est la suivante : pour chaque porte de $g_i \leftarrow g_j \vee g_k$, deux portes, h_i correspondant à g_i et $\bar{h}_i$ correspondant à $\neg g_i$, sont créées, *i.e.*

$$\begin{array}{rcl} h_i & \leftarrow & h_j \vee h_k, \\ \bar{h}_i & \leftarrow & \bar{h}_j \wedge \bar{h}_k, \end{array}$$

et de même, pour chaque porte $g_i \leftarrow g_j \wedge g_k$, deux portes, h_i et $\bar{h}_i$, sont créées :

$$\begin{array}{rcl} h_i & \leftarrow & h_j \wedge h_k, \\ \bar{h}_i & \leftarrow & \bar{h}_j \vee \bar{h}_k. \end{array}$$

Enfin, pour chaque porte $g_i \leftarrow \neg g_j$, les portes h_i et $\bar{h}_i$ sont :

$$\begin{array}{rcl} h_i & \leftarrow & \bar{h}_j, \\ \bar{h}_i & \leftarrow & h_j. \end{array}$$

Si g_p est la porte de sortie du circuit c, la porte h_p sera la porte de sortie du circuit m correspondant.
Chaque porte g_i est transformée indépendamment des autres et nécessite pour cela un espace logarithmique, pour stocker les indices i, j, k écrits en binaire. Le CVP se réduit donc bien au $MCVP$.

□

Ces transformations sont illustrées sur la figure 1.1.

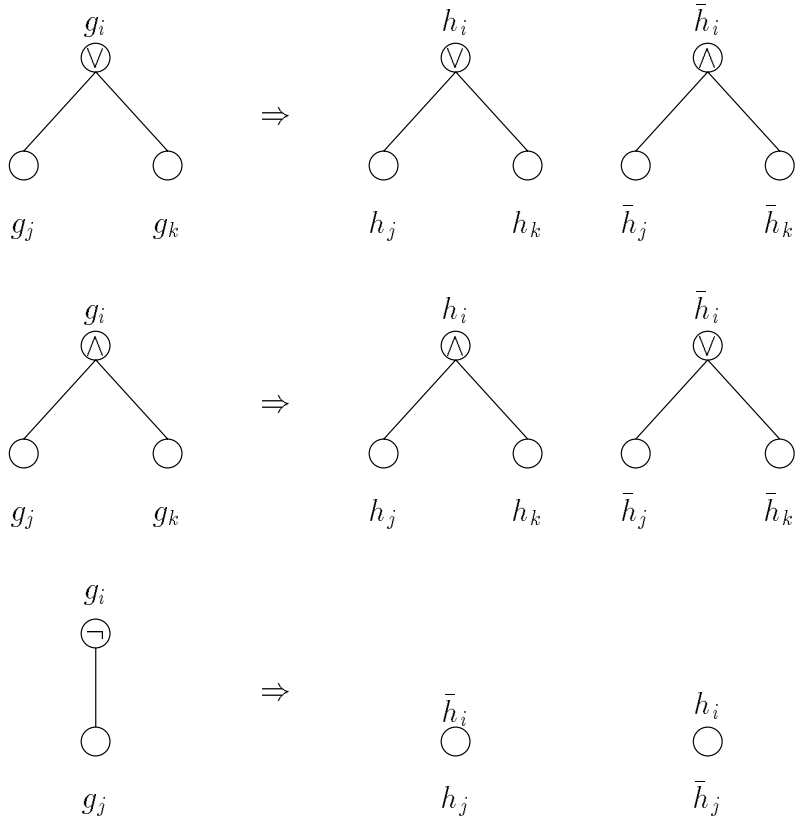


FIG. 1.1 - *Transformation d'un circuit en circuit monotone.*

Ensemble indépendant lexicographiquement maximal

Le dernier exemple détaillé de problème P-complet, qui servira d'exemple et de pire cas à l'algorithme de contraction sur les treillis, est le problème du calcul de l'ensemble lexicographiquement maximal de sommets indépendants d'un graphe.

Définition 7 *Soit $G = (V, E)$ un graphe non orienté, avec $V = \{v_1, v_2, \ldots v_n\}$ totalement ordonné par l'ordre lexicographique $v_1 > v_2 > \ldots > v_n$. Le problème du calcul de l'ensemble lexicographiquement maximal de sommets indépendants, ou Lexicographic Maximal Independent Set Problem, noté $LMISP$, est l'ensemble $LMIS$, maximal pour l'ordre lexicographique, des sommets indépendants de G, i.e. le sous-graphe engendré par cet ensemble n'a pas d'arête.*

Théorème 4 *[Coo81, GR88a, KR90] Le $LMISP$ est P-complet.*

DÉMONSTRATION

- Ici encore, le $LMISP$, écrit sous la forme booléenne du §5.2.2, est une instance du CVP ; il s'y réduit donc trivialement, ce qui prouve son appartenance à P.

- Montrons maintenant comment le $MCVP$, problème P-complet, se réduit au $LMISP$. La construction est un peu « parachutée », une preuve par récurrence sur le nombre

de sommets prouve qu'elle est correcte. Soit un $MCVP$:

$$\begin{array}{lll} g_0 & \leftarrow & 0, \\ g_1 & \leftarrow & 1, \\ g_i & \leftarrow & g_j \vee g_k,\ 2 \leq i \leq n,\ j < i,\ k<i \\ \text{ou} & & \\ g_i & \leftarrow & g_j \wedge g_k,\ 2 \leq i \leq n,\ j<i,\ k<i. \end{array}$$

Le résultat de la réduction est un graphe $G = (V, E)$ avec $V = \{v_1, \ldots v_n\} \bigcup \{w_1, \ldots w_n\}$, $\{v_i\}$ correspondant aux portes du $MCVP$ dont la valeur est vraie et $\{w_i\}$ aux portes dont la valeur est fausse. On définit l'ordre total de V par

$$\forall i, j,\ \text{si } i<j,\ \begin{array}{lll} v_i & < & v_j,\ w_j, \\ w_i & < & v_j,\ w_j. \end{array}$$

L'ensemble E est initialisé à $E \leftarrow \{(v_i, w_i)\}$.
Il reste maintenant à ordonner v_i et w_i, et à construire l'ensemble E des arêtes de G.

$$\begin{array}{rcl} w_0 & < & v_0, \\ v_1 & < & w_1, \\ \text{si } g_i & \leftarrow & g_j \vee g_k, \\ w_i & < & v_i, \\ E & \leftarrow & E \bigcup \{(v_j, w_i), (v_k, w_i)\}, \\ \text{si } g_i & \leftarrow & g_j \wedge g_k, \\ v_i & < & w_i, \\ E & \leftarrow & E \bigcup \{(w_j, v_i), (w_k, v_i)\}. \end{array}$$

Chaque transformation utilisant seulement les indices d'un nombre constant de nœuds, elle nécessite un espace de travail logarithmique en n.
Le $LMIS$ contiendra un nœud v_i si et seulement si g_i vaut Vrai et un nœud w_i sinon.

□

Exemple

$$\begin{array}{lll} g_0 & \leftarrow & 0, \\ g_1 & \leftarrow & 1, \\ g_2 & \leftarrow & g_0 \wedge g_1, \\ g_3 & \leftarrow & g_0 \vee g_1. \end{array}$$

Le graphe correspondant est représenté figure 1.2.

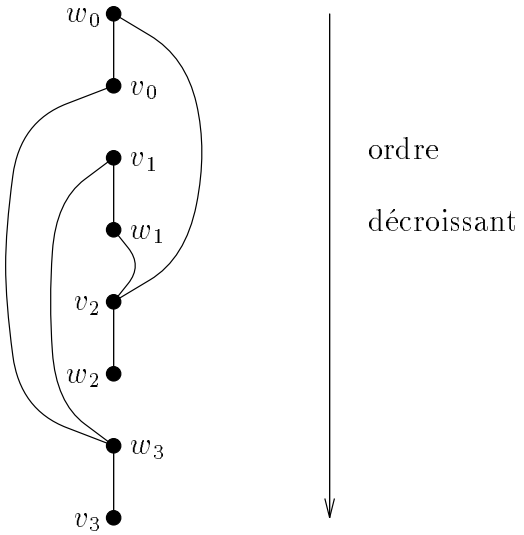


FIG. 1.2 - *Le* $LMIS = \{w_0, v_1, w_2, v_3\}$.

1.2.3 D'autres exemples de problèmes P-complets

Pour mémoire, citons d'autres problèmes P-complets, sans détailler la preuve de leur P-complétude. (Pour plus de détails sur les problèmes précédents, cf. [GR88a, KR90] et pour un exposé plus complet de la complexité, cf. [BDG90, Pap94]). Le problème du calcul de la valeur d'un circuit planaire (*i.e.* qui admet une représentation dans un plan sans que les arêtes ne se croisent) est un problème P-complet. Un problème analogue au $LMISP$ appartient à cette catégorie, celui de déterminer une clique lexicographiquement maximale, *i.e.* un ensemble de sommets qui engendrent un sous-graphe complet du graphe donné en entrée.

Dans un autre registre, le problème de la programmation linéaire est peut-être l'exemple le plus connu de problème P-complet. Il s'énonce de la façon suivante :
soient une matrice et deux vecteurs à composantes entières

$$\begin{array}{l} A = (a_{i,j}) \in \mathcal{M}_{m,n}(\mathbb{Z}), \\ D = (d_1, \ldots d_n) \in \mathbb{Z}^n \text{ qui définit les contraintes}, \\ C = (c_1, \ldots c_n) \in \mathbb{Z}^n \text{ qui définit la fonction de coût}, \\ b \in \mathbb{Z} \end{array}$$

et dont le résultat est « Vrai » si et seulement si $\exists X \in \mathbb{Q}^n /$

$$\begin{array}{lcl} {}^tC.X & \geq & b, \\ (A.X)_i & \leq & d_i,\ 1 \leq i \leq m. \end{array}$$

1.3 Parallélisme : modèle et définitions

1.3.1 Le modèle *PRAM*

Un modèle de machine parallèle est le modèle $PRAM$ (Parallel RAM), défini à partir du modèle RAM (Random Access Machine) de machine séquentielle. Ce modèle RAM est une adaptation du modèle de la machine de Turing, plus facile à utiliser en pratique pour décrire des algorithmes. Le modèle $PRAM$ est un modèle de machine parallèle synchrone, où les processeurs communiquent via une mémoire partagée. Il correspond assez bien à la notion intuitive de machine parallèle. Ce modèle a été adapté afin de prendre en compte de manière plus fine les caractéristiques des ordinateurs parallèles (par exemple, les machines à réseau d'interconnexion sont modélisées par les $XRAM$, où X est une description du réseau d'interconnexion [CT93]). Développer des algorithmes parallèles étant une tâche parfois complexe, il est plus facile de ne pas tenir compte des contraintes d'interconnexion des processeurs dans un premier temps. Nous utiliserons donc le modèle $PRAM$ pour la description et la complexité des algorithmes.

Définition 8 *[BDG90]*
Une machine $PRAM$ *consiste en un ensemble indicé infini de processeurs (de type RAM), une mémoire globale infinie et un programme fini.*

- *Chaque processeur dispose d'une mémoire locale infinie, de son propre compteur ordinal et d'un drapeau indiquant s'il est actif ou non.*

- *Un* programme *est une suite d'instructions étiquetées, qui peuvent être : ne rien faire, une lecture ou une écriture en mémoire (locale ou globale), un calcul arithmétique (addition de deux valeurs, ou division entière par 2 de la valeur d'une des cases) ou un branchement conditionnel à une étiquette (puisque les instructions sont étiquetées). La multiplication ne fait pas partie des instructions d'une PRAM pour éviter le grossissement de la taille des données ; ceci permet de prendre pour hypothèse de travail le fait que chaque instruction peut être exécutée en une unité de temps. Cet ensemble d'instructions est celui d'une RAM classique. L'instruction de parallélisme est l'instruction* `fork` *: si un processeur p_i exécute l'instruction* `fork(m)`*, le processeur inactif de plus petit numéro commencera un calcul à l'étiquette* `m`*.*
  Le déroulement d'un programme est totalement synchrone. Au début du programme, tous les processeurs initialisent leur compteur ordinal. À chaque pas de temps, chaque processeur effectue l'instruction correspondant à son compteur ordinal.

Le temps t d'exécution de ce programme est le nombre de pas du programme et son nombre de processeurs est le nombre maximal de processeurs actifs simultanément.

Il existe des variantes classiques du modèle $PRAM$ pour préciser comment s'effectuent les accès à la mémoire[1] :

- le modèle $EREW-PRAM$ (Exclusive Read Exclusive Write), où les lectures comme les écritures sont exclusives (lecture simultanée d'une même case mémoire interdite et

1. Un processeur peut écrire dans une case de la mémoire globale en même temps qu'un processeur la lit. Par convention, l'écriture s'effectue « après » la lecture, *i.e.* la lecture délivre l'ancienne valeur.

écriture simultanée interdite).

- le modèle $CREW-PRAM$ (Concurrent Read Exclusive Write), où plusieurs processeurs peuvent lire en même temps la même case de la mémoire globale. En revanche, les écritures concurrentes sont interdites.

- le modèle $CRCW-PRAM$ (Concurrent Read Concurrent Write), où les lectures concurrentes sont autorisées, ainsi que les écritures concurrentes. Il faut alors préciser comment se règlent les écritures concurrentes. Pour cela, il y a trois sous-modèles, présentés par puissance croissante :

  - $COMMUNE\ CRCW-PRAM$: une écriture concurrente n'est effectuée que si tous les processeurs essaient d'écrire la même valeur.
  - $ARBITRAIRE\ CRCW-PRAM$: un processeur au hasard parmi tous les processeurs qui essaient d'écrire impose sa valeur.
  - $PRIORITAIRE CRCW–PRAM$: le processeur qui a le plus petit numéro parmi tous les processeurs qui essaient d'écrire impose sa valeur.

Ces sous-modèles du modèle $PRAM$ sont assez proches et il est possible de simuler l'exécution d'un programme conçu pour un modèle sur un autre avec un surcoût limité : un programme qui s'exécute sur une $PRIORITAIRE\ CRCW-PRAM$ (le modèle le plus puissant), en temps t avec p processeurs, peut être exécuté sur une $EREW–PRAM$ (le moins puissant) en temps $t\mathcal{O}(\log p)$ avec p processeurs. Par ailleurs, toutes les $CRCW-PRAM$ sont équivalentes pour ce qui est du temps parallèle : le programme précédent peut ainsi s'exécuter sur une $COMMUNE\ CRCW-PRAM$ en temps t avec p^2 processeurs. Ces résultats ont pour conséquence que, lors de la conception d'un algorithme[2] parallèle, on n'a pas à se soucier du modèle dans un premier temps, puisqu'il est toujours possible de passer de l'un à l'autre. Ensuite, une modification « à la main » du premier algorithme peut permettre de construire un algorithme pour un modèle moins puissant.

1.3.2 Algorithmes parallèles efficaces et optimaux

À partir du modèle $PRAM$, deux quantités définissent les performances d'un programme parallèle. La première est évidemment le temps : soit un problème $\mathcal{P}$, qui est une « famille » de problèmes $(\mathcal{P}_n)_{n\in\mathbb{N}}$, avec $\mathcal{P}_n$ l'instance du problème $\mathcal{P}$ qui a $n^{\mathcal{O}(1)}$ entrées. $\mathcal{P}$ est résolu par un algorithme parallèle A qui est une famille $(A_n)_{n\in\mathbb{N}}$ d'algorithmes, A_n résolvant $\mathcal{P}_n$. Le temps séquentiel de résolution de $\mathcal{P}_n$ est au moins linéaire en $n^{\mathcal{O}(1)}$, la taille des entrées car il faut au moins le temps de les lire (les entrées sont supposées être toutes utiles). On espère donc que $t_{//}(A_n)$, le temps de l'algorithme parallèle, sera beaucoup plus petit : polylogarithmique en la taille des entrées, *i.e.* $t_{//}(A_n) = \mathcal{O}(\log^{\mathcal{O}(1)})n$. Une deuxième grandeur est importante : l'algorithme parallèle doit utiliser une ressource matérielle raisonnable, *i.e.* un

2. Dans cette thèse, le terme d'algorithme désigne plutôt le principe abstrait, la méthode employée pour résoudre un problème, et celui de programme l'écriture effective sur machine. Cependant cette distinction ne sera pas toujours scrupuleusement respectée.

nombre de processeurs $p(A_n)$ raisonnable. « Raisonnable » signifie, en complexité, « polynomial en la taille des entrées », $p_{//}(A_n) = \mathcal{O}(n^{\mathcal{O}(1)})$. Ces deux grandeurs associées définissent la *complexité* d'un algorithme.

Remarque
Pour les programmes séquentiels, deux grandeurs analogues sont utilisées : l'espace mémoire d'une part, qui matérialise la longueur des dépendances entre les calculs et correspond donc au temps d'un algorithme parallèle ; le temps d'autre part, qui est le nombre total d'opérations réalisées. Le temps séquentiel, lui, est lié au nombre de processeurs d'un programme parallèle.

Souvent, « l'activité » d'un programme parallèle est représentée par un diagramme bidimensionnel où, sur un premier axe, est porté le nombre de processeurs et sur l'autre, le temps : on coche les cases (i, j) correspondant à un processeur j actif à l'étape i. Ce diagramme s'inscrit dans un rectangle ayant pour côtés le nombre de processeurs $p_{//}(A_n)$ et le temps parallèle $t_{//}(A_n)$. L'aire de ce rectangle est appelée *travail de l'algorithme*. Elle sera supérieure au temps séquentiel de l'algorithme, du fait de l'inactivité de certains processeurs.

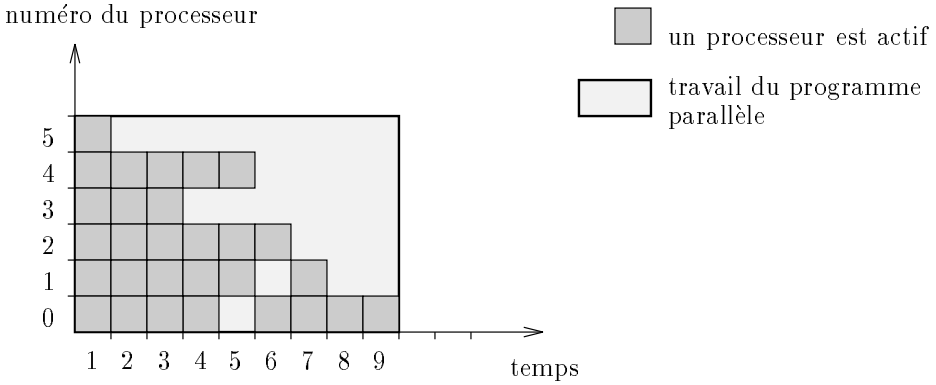


FIG. 1.3 - *Diagramme schématisant l'activité d'un programme parallèle.*

Définition 9 *Le travail $W(A_n)$ d'un algorithme est le produit du temps de son exécution parallèle et du nombre de processeurs utilisés*

$$W(A_n) = t_{//}(A_n) * p_{//}(A_n).$$

Il correspond à la quantité totale de ressources consommées par l'algorithme.

Pour un algorithme parallèle rapide ($t_{//}(A_n) = \log^{\mathcal{O}(1)} n, p_{//}(A_n) = n^{\mathcal{O}(1)}$), il est intéressant d'effectuer une comparaison avec les algorithmes séquentiels résolvant le même problème. Ceci conduit à la définition d'algorithme efficace et d'algorithme optimal.

Définition 10 *Un algorithme parallèle $A = (A_n)$ est* (asymptotiquement) efficace *si et seulement si son temps d'exécution est polylogarithmique*

$$t_{//}(A_n) = \log^{\mathcal{O}(1)} n$$

et son travail est le produit du temps $t(n)$ du meilleur algorithme séquentiel connu qui résout le même problème par un facteur de surcoût polylogarithmique

$$W(A_n) = t(n) * \log^{\mathcal{O}(1)} n.$$

Un algorithme parallèle $A = (A_n)$ est (asymptotiquement) optimal *si et seulement si son temps d'exécution est polylogarithmique*

$$t_{//}(A_n) = \log^{\mathcal{O}(1)} n$$

et son travail est du même ordre que le temps $t(n)$ du meilleur algorithme séquentiel connu qui résout le même problème

$$W(A_n) = \mathcal{O}(t(n)).$$

Remarque

- Attention, la notion d'algorithme efficace est différente de l'efficacité d'un programme parallèle, qui est une valeur mesurable. L'efficacité d'un programme est utilisée en parallélisation sur machines « réelles », elle vaut :

$$e = \frac{t_{séq}}{t_{//} * p_{//}}.$$

- Dans tout ce qui suit, les complexités temporelles et matérielles seront indiquées en ordre de grandeur, avec des $\mathcal{O}$. Pour cette raison, les notions d'algorithme efficace et d'algorithme optimal définies ici correspondent en fait aux notions d'asymptotiquement efficace et d'asymptotiquement optimal. Le mot « asymptotiquement » sera toujours en ellipse.

- La convention de notation suivante est adoptée: pour un algorithme parallèle $A = (A_n)$, qui s'exécute en temps $t_{//}(A_n)$ avec $p_{//}(A_n)$ processeurs sur une machine M ($M = EREW-PRAM$, $CREW-PRAM$ ou $CRCW-PRAM$), sa complexité sera notée $M(p_{//}(n), t_{//}(n))$. Cette notation est inspirée de Karp et Ramachandran [KR90].

1.3.3 Modèle booléen

Le modèle booléen est un modèle de machine parallèle beaucoup plus simple, mais plus éloigné de la notion intuitive de machine parallèle. Il sera utilisé au chapitre 4, lors de l'évaluation de circuits arithmétiques sur des treillis.

Définition 11 *Dans le modèle booléen, une machine parallèle est un ensemble de circuits booléens, composés de portes logiques ET, OU et NON.*
Plus formellement, une machine parallèle est une famille uniforme *$B = (B_n)_{n\in\mathbb{N}}$ de graphes booléens orientés et sans circuit telle que B_n possède $n^{\mathcal{O}(1)}$ entrées et sorties. Un nœud du circuit est ici une porte logique effectuant une opération booléenne (et, ou, négation).*

L'uniformité permet de limiter la complexité architecturale du circuit [Ruz81]. On utilise le plus souvent la « log-uniformité », qui signifie que la description du circuit B_n (pour $n \in \mathbb{N}$) peut être calculée sur une machine de Turing avec un espace logarithmique. Cette machine de Turing dispose en outre d'une bande de sortie de longueur non précisée[3] sur laquelle elle décrit le circuit parallèle. Il lui serait difficile sinon de décrire un circuit ayant un nombre polynomial de portes. La contrainte d'espace logarithmique ne porte donc que sur la longueur de la bande de travail.

On distingue essentiellement deux sous-modèles, selon que le nombre d'entrées d'une porte (*fan-in*) est borné (par convention, il est fixé à 2) 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.

Le nombre de processeurs est ici défini comme le nombre de nœuds du circuit B_n et le temps comme sa profondeur, *i.e.* le plus long chemin entre une porte d'entrée et une porte de sortie.

Il s'avère que l'utilisation de cette log-uniformité ne permet pas de trancher au sujet de l'inclusion stricte ou de l'égalité de NC et de P (cf. §1.3.7). Il paraît légitime de relaxer la condition de log-uniformité en une condition plus souple, pour tenter de cerner les liens entre la classe obtenue avec cette nouvelle uniformité et P. On peut penser à gravir l'échelle des fonctions logarithmiques, polynomiales puis exponentielles. Considérons tout d'abord le cas de l'exp-uniformité : soit f une fonction non calculable, qui, pour une valeur de n donnée, répond soit toujours « Vrai », soit toujours « Faux » quelle que soit la valeur des entrées, il y a 2^n valeurs possibles pour les entrées et une machine de Turing déterministe peut très bien, en espace exponentiel, tabuler la table de vérité de f, déterminer son écriture booléenne et renvoyer un circuit de taille constante. Cette notion d'uniformité n'est donc pas une notion satisfaisante et elle n'est jamais utilisée. En particulier, pour les circuits envisagés dans ce travail, elle n'entrera pas en ligne de compte.

Reste maintenant la notion de P-uniformité, *i.e.* de description en temps polynomial des circuits. Cette notion signifie que la compilation d'un circuit en un autre circuit en temps polynomial, comme celle présentée au chapitre 6, est autorisée. Von zur Gathen a utilisé cette notion pour comparer le modèle arithmétique, sur des corps finis de différentes caractéristiques par exemple, au modèle booléen [GS89a]. Les classes de complexité utilisées sont alors les classes (P-uniforme)NC^k et (P-uniforme)NC.

Définition 12 *(P-uniforme)NC^k, $k \geq 0$, est l'ensemble des problèmes résolus par une famille de circuits $C = (C_n)$ de taille $n^{\mathcal{O}(1)}$, de profondeur $\mathcal{O}(\log^k n)$ et décrits en temps*

3. Comme le temps est polynomial, la machine ne pourra pas parcourir plus qu'une longueur polynomiale de cette bande de sortie.

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 de grandeur – qui soit de *fan-in* et de *fan-out* bornés [HKP84].

polynomial par une machine de Turing déterministe.

$$(P\text{-}uniforme)NC = \bigcup_{k=0}^{+\infty} (P\text{-}uniforme)NC^k.$$

Dans cette thèse, la notion d'uniformité désignera la log-uniformité, le recours à d'autres notions d'uniformité sera explicitement mentionné. Cependant, cette notion signifie que, pour un circuit de profondeur $\mathcal{O}(\log n)$, la machine de Turing qui construit le circuit est plus puissante, en termes de problèmes qu'elle peut traiter, que le circuit construit. Pour éviter cela, la notion rigoureuse d'uniformité qui doit être employée est la U_{E^*}-uniformité : un circuit est U_{E^*}-uniforme si un langage défini à partir de son programme, son langage de connexion étendu, est reconnu par une machine de Turing alternative, en temps proportionnel à sa profondeur et en espace proportionnel à son nombre de portes.

Définition 13 *[Ruz81]*
Le langage de connexion étendu *d'une famille de circuits $C = (C_n)$ est l'ensemble des chaînes de la forme $<n, g, p, t>$, avec $n \in \mathbb{N}$, $g \in \mathbb{N}$ un numéro de porte (g comme gate), $p \in \{L, R\}^*$ et la longueur de p est inférieure au nombre de portes de C_n (p comme path, avec L pour Left et R pour Right), et $t \in \{x, \vee, \wedge, \neg\}$ (t comme type, x désignant une feuille), telles que :*

- *$p = \varepsilon$ et la porte numéro g de C_n est du type t ;*
- *$p \neq \varepsilon$ et la porte de C_n atteinte à partir de la porte g en suivant le chemin p est du type t.*

Définition 14 *[Ruz81]*
Une famille de circuits $C = (C_n)$, de taille (ou nombre de portes) P_n et de profondeur (ou temps parallèle) T_n, est U_{E^*}-uniforme si et seulement si son langage de connexion étendu est reconnu à l'aide d'une machine de Turing alternative en temps $\mathcal{O}(T_n)$ à l'aide d'un espace mémoire $\mathcal{O}(\log P_n)$.

1.3.4 Classes de complexité

Maintenant que les notions de machine et de programme parallèles sont explicitées ainsi que celles de performances d'un programme, il est possible de classifier les problèmes. Les problèmes $\mathcal{P} = (\mathcal{P}_n)$, dits parallèles, sont ceux qui admettent un temps de résolution parallèle polylogarithmique, en utilisant un nombre polynomial de processeurs. Ces problèmes constituent la classe NC, comme « Nick's Class », du prénom de Nicholas Pippenger qui établit cette définition. La classification au sein de NC s'effectue sur le degré du polynôme en $\log n$ qui correspond au temps. Selon le modèle de machine choisi, différentes classifications sont obtenues, les plus usitées étant les classes NC^k et AC^k.

Pour les modèles $PRAM$, on définit ainsi les classes (pour $k \geq 1$)

$EREW^k$	des problèmes qui s'exécutent en temps $\mathcal{O}(\log^k n)$ sur une $EREW-PRAM$, ce qui se note $EREW(n^{\mathcal{O}(1)}, \log^k n)$
$CREW^k$	des problèmes dont la complexité parallèle est $CREW(n^{\mathcal{O}(1)}, \log^k n)$
$CRCW^k$	des problèmes dont la complexité parallèle est $CRCW(n^{\mathcal{O}(1)}, \log^k n)$

Comme il est possible de simuler n'importe laquelle de ces machines sur une autre, avec éventuellement une perte de temps logarithmique, il est possible d'établir les relations suivantes entre ces classes :

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

Il est possible de définir le même type de classes à l'intérieur du modèle booléen.

- La classe NC^k $(k \geq 0)$ est l'ensemble des problèmes résolus par des familles U_{E^*}-uniformes de circuits de *fan-in* borné et de profondeur $\mathcal{O}(\log^k n)$.
- La classe AC^k $(k \geq 0)$ est l'ensemble des problèmes résolus par des familles uniformes de circuits de *fan-in* non borné et de profondeur $\mathcal{O}(\log^k n)$.

Remarque
[Ruz81]
Pour $k \geq 2$, les classes NC^k définies à l'aide de la U_{E^*}-uniformité ou de la log-uniformité sont identiques. De façon informelle, ces classes sont les mêmes pour la plupart des notions classiques d'uniformité, tant que le temps de construction des circuits appartenant à ces classes est limité par un logarithme de leur taille. NC reste la même quelle que soit la notion d'uniformité employée, NC est dite *résistante*.

Les classes NC^k sont naturellement incluses dans les classes AC^k et les classes AC^k sont elles-mêmes incluses dans les classes NC^{k+1} :

$$NC^k \subset AC^k \subset NC^{k+1}.$$

Enfin, les classes de complexité définies à l'aide des deux modèles de machines parallèles peuvent être comparées, pour $k \geq 2$:

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

La classe NC est maintenant définie rigoureusement comme l'union de toutes ces classes :

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

1.3.5 Comparaison de deux problèmes

Soient deux problèmes $\mathcal{P} = (\mathcal{P}_n)$ et $\mathcal{Q} = (\mathcal{Q}_n)$. Le problème $\mathcal{P}$ est NC^1-réductible à $\mathcal{Q}$ si, quand on sait résoudre $\mathcal{Q}$, il est possible de résoudre $\mathcal{P}$ en temps logarithmique, en utilisant des « portes oracles » qui résolvent $\mathcal{Q}$.

Définition 15 *[Coo85]*
Une porte oracle pour $\mathcal{Q}$ est un nœud qui a pour entrées $i_1, \ldots i_r$ et pour sorties $o_1, \ldots o_s$, dont les valeurs sont $(o_1, \ldots o_s) = \mathcal{Q}(i_1, \ldots i_r)$ et dont la profondeur compte pour $\lceil \log(r+s) \rceil$ dans la profondeur du circuit où elle est employée.

Définition 16 *[Coo85]*
$\mathcal{P}$ est NC^1-réductible à $\mathcal{Q}$: $\mathcal{P} \leq^{NC^1} \mathcal{Q}$, si et seulement s'il existe une famille uniforme $C = (C_n)$ de circuits qui résout $\mathcal{P}$, de profondeur $\mathcal{O}(\log n)$ et qui contient des portes oracles pour $\mathcal{Q}$.

La relation $\leq^{NC^1}$ est réflexive et transitive. La clôture C^* d'une classe de complexité C est l'ensemble des problèmes $\mathcal{P}$ tels que $\exists \mathcal{Q} \in C$, $\mathcal{P} \leq^{NC^1} \mathcal{Q}$. La classe C est fermée par NC^1-réduction si et seulement si $C^* = C$.

Théorème 5 *[Coo85]*
La classe NC^k est fermée par NC^1-réduction, $\forall k \geq 1$.

DÉMONSTRATION
Soient $\mathcal{P}$ et $\mathcal{Q}$ deux problèmes, $\mathcal{P} \leq^{NC^1} \mathcal{Q}$ et $\mathcal{Q} \in NC^1$. Soit $A = (A_n)$, la famille de circuits qui réalise la réduction de $\mathcal{P}$ en $\mathcal{Q}$ et soit $B = (B_n)$, la famille de circuits de profondeur $\mathcal{O}(\log^k n)$ qui résout $\mathcal{Q}$.
Une famille de circuits $C = (C_n) \in NC^k$ qui résout $\mathcal{P}$ est construite en remplaçant chaque porte oracle pour $\mathcal{Q}_m$, dans A_n, par le circuit B_m correspondant. Le nombre de portes de C_n est polynomial. Il reste à vérifier que sa profondeur est $d(C_n) = \mathcal{O}(\log^k n)$: soit un chemin γ dans C_n, si γ rencontre les instances $B_{m_1}, B_{m_2}, \ldots$ des circuits qui ont été substitués aux portes oracles dans A_n, alors la longueur de γ est au plus :

$$\begin{aligned} \sum d(B_{m_i}) + \log n &= \mathcal{O}(\textstyle\sum \log^k m_i) + \mathcal{O}(\log n) \\ &= \mathcal{O}((\textstyle\sum \log m_i)^k) + \mathcal{O}(\log n) \\ &= \mathcal{O}(\log^k n). \end{aligned}$$

En effet, la convention choisie pour déterminer la profondeur de A_n, dans la définition de la NC^1-réductibilité, implique que $\sum \log m_i = \mathcal{O}(\log n)$. □

Il existe des problèmes auxquels tous les problèmes d'une classe sont réductibles, *i.e.* des problèmes plus durs que tous ceux de cette classe.

Définition 17 *Un problème $\mathcal{P}$ est NC^1-dur pour une classe C si et seulement si $\forall \mathcal{Q} \in C$, $\mathcal{Q} \leq^{NC^1} \mathcal{P}$.*
Un problème $\mathcal{P}$ est NC^1-complet pour une classe C si et seulement si $\mathcal{P} \in C$ et $\mathcal{P}$ est NC^1-dur pour C.

Les problèmes NC^1-complets pour une classe sont les problèmes les plus difficiles de cette classe.

Théorème 6 *Le problème de l'évaluation d'une formule booléenne est NC^1-complet pour NC^1 [Bus87, Bus93].*

L'évaluation en temps logarithmique d'une expression fera l'objet du chapitre 2.

1.3.6 Modèles arithmétique et arithmétique booléen

Le modèle arithmétique a été introduit par von zur Gathen [Gat86, Gat88]. Il s'agit des familles uniformes de circuits arithmétiques qui ont $n^{\mathcal{O}(1)}$ entrées. Les nœuds de ces circuits sont des portes arithmétiques, effectuant des opérations arithmétiques sur une structure algébrique donnée S. Par hypothèse, chaque opération est effectuée en une unité de temps. La structure peut être l'ensemble des rationnels, des polynômes à coefficients rationnels, des matrices à coefficients polynomiaux...et même des réels. Nous redéfinirons ces circuits arithmétiques au chapitre 3, afin d'adapter leur définition à nos besoins.
Le modèle arithmétique booléen est une extension du modèle arithmétique, dans lequel les circuits ont des portes arithmétiques, des portes de test d'égalité à 0 : $a \stackrel{?}{=} 0$ qui prennent en entrée un élément arithmétique et qui renvoient un booléen, des portes logiques ET, OU, NON et des portes de sélection qui prennent en entrée deux éléments arithmétiques a et b et un booléen c et qui, suivant la valeur du booléen c, renvoient a ou b : $sel(a,b,c) = a$ si $c = Vrai$ et b sinon.

Remarque

Si le modèle choisi est le modèle arithmétique, il est possible de définir des classes d'équivalence analogues à NC^k et à NC. Pour les distinguer, on les indice par le nom de la structure S : NC^k_S et NC_S.
Ces classes sont différentes ; par exemple, l'addition de deux n-uplets d'entiers à n bits s'effectue en une unité de temps dans $\mathbb{N}$; elle appartient à $NC^0_{\mathbb{N}}$, mais elle appartient à NC^1 quand les opérations sont des opérations logiques.

Les circuits arithmétiques sur des corps finis de cardinal p par exemple peuvent être simulés par des circuits booléens dont la taille et la profondeur sont multipliées par un facteur $\log^{\mathcal{O}(1)} p$ [GS89a]. Le modèle booléen apparaît même supérieur au modèle arithmétique, si naturellement, pour pouvoir comparer ce qui est comparable, on se limite aux circuits arithmétiques simulés par des circuits booléens de taille polynomiale en la longueur des entrées et des sorties, c'est-à-dire que n'entrent pas en considération les circuits arithmétiques dont le codage binaire est de longueur exponentielle ou infinie. Sur $\mathbb{Q}$ par exemple, avec un circuit booléen de taille polynomiale et de profondeur polylogarithmique, il est possible de décider, étant donnés deux entiers a et $b \neq 0$ de longueur $\leq n$, si $\frac{a}{b} \in \mathbb{Z}$ ou non. Pour cela, on effectue la division euclidienne de a par b. Ce problème appartient à NC^2 [Rei86, KR90] et admet un circuit non log-uniforme, mais seulement uniforme en espace polynomial, de taille $n^{\mathcal{O}(1)}$ et de profondeur $\mathcal{O}(\log n)$ [BCH86]. En revanche, il n'existe pas de circuit arithmétique à opérations dans $\mathbb{Q}$ de taille polynomiale en n qui réponde à ce problème, quelle que soit la notion d'uniformité choisie, même si on autorise de plus des portes de test $a \stackrel{?}{=} 0$. La seule solution aujourd'hui connue est de construire un circuit contenant la réponse pour tous les couples (a,b) possibles ; ce circuit est de taille exponentielle.

1.3.7 Problèmes séquentiels et problèmes parallèles

La théorie de la complexité parallèle s'attache non seulement à classifier les problèmes selon leur degré de parallélisme, mais également à établir des liens entre les différentes classes

de complexité, ce dernier thème étant loin d'être complet. Dans ce domaine, une question intéressante est de savoir quels sont les problèmes admettant une solution séquentielle qui peuvent être traités rapidement en parallèle. Telle quelle, cette question est mal posée. Tout d'abord, il est impératif que ces problèmes soient résolus en séquentiel avec des ressources raisonnables, en l'occurrence qu'ils appartiennent à la classe P, la classe des problèmes parallèles considérée étant la classe NC. Savoir quels sont les problèmes admettant une solution séquentielle raisonnable qui ont une solution parallèle rapide revient à établir les liens entre P et NC. On sait que P contient NC : une simulation séquentielle d'un algorithme parallèle de temps polylogarithmique utilisant un nombre polynomial de processeurs s'effectue en temps polynomial. En 1985, Cook a conjecturé une inclusion stricte de NC dans P [Coo85], mais ceci n'admet jusqu'à présent ni confirmation ni démenti. La même question reste sans réponse si on affaiblit certaines hypothèses, par exemple si on autorise des descriptions P-uniformes (définies au §1.3.3) des programmes parallèles.
Cette thèse n'a pas la prétention de résoudre ce problème, ni même de s'y attaquer. Cependant, les techniques de transformation d'un programme séquentiel sans boucle présentées aux chapitres 3 et 4 s'insèrent dans le cadre des transformations P-uniformes. Elles ont permis d'établir que si F est un corps et si DP_F note l'ensemble des fonctions polynomiales de degré polynomial en n sur $F[x_1, x_2, \ldots x_n]$, alors

$$P_F \bigcap DP_F = (P\text{-uniforme})NC_F{}^2 \bigcap DP_F.$$

Elles contribuent à établir des majorants non triviaux de la complexité parallèle d'un algorithme, comme les résultats du chapitre 5. Bien sûr, ces techniques ne sont pas en mesure de fournir des majorants pour un problème donné dans toute sa généralité et si l'algorithme séquentiel fourni pour le résoudre ne s'y prête pas, sa transformation par l'outil de compilation du chapitre 6 ne donnera pas un programme très parallèle.

1.4 Algorithmique parallèle

Maintenant que le modèle de machines et celui des programmes parallèles sont définis et que la comparaison de deux programmes parallèles est rendue possible par l'introduction de la classe NC et de ses sous-classes, nous montrons comment on peut construire des algorithmes parallèles « à la main ». En ce domaine, il n'existe pas réellement de méthodologie, simplement quelques principes et quelques schémas connus auxquels on essaie de se ramener.

1.4.1 Étude du graphe de précédence

Une idée naturelle est de prendre pour base un algorithme séquentiel et de déterminer quelles sont les tâches indépendantes. Une représentation de l'algorithme par son graphe de précédence permet d'établir clairement les contraintes de séquentialité et les possibilités de parallélisme. Sa profondeur fournit une estimation du temps parallèle de l'algorithme parallèle que l'on peut dériver de cet algorithme, sa largeur indiquant le nombre de tâches exécutables simultanément et donc une borne sur le nombre de processeurs requis.

Prenons l'exemple de la résolution d'un système triangulaire inférieur $Ax = b$ par l'algorithme de descente triangulaire :

$$x_i = \frac{b_i - \sum_{j=1}^{i-1} A_{i,j} * x_j}{A_{i,i}},\ i = 1 \ldots n.$$

Il s'agit d'un algorithme séquentiel optimal : sa complexité est linéaire en la taille des données (n^2 à cause de la matrice A). Comme le calcul de x_i utilise les valeurs des x_j précédents, $1 \leq j < i$, il semble que le seul parallélisme consiste à découper cet algorithme en n phases séquentielles, une pour chaque x_i. Le calcul d'un x_i fait intervenir un calcul dont la solution parallèle est connue (cf. §1.4.3), le calcul d'un produit itéré, qui a pour complexité $EREW(\frac{i}{\log i}, \log i)$. La complexité globale de cet algorithme est alors

$$EREW(\frac{n}{\log n}, n \log n).$$

Cet algorithme est un bon algorithme, puisque son travail est $\mathcal{O}(n^2)$; cependant il n'est pas très parallèle.

On peut remarquer que le calcul de chaque somme $\Sigma_i = \sum_{j=1}^{i-1} A_{i,j} * x_j$ peut être effectuée au fur et à mesure du calcul des x_j : dès que x_j est calculé, les sommes partielles Σ_i, $i > j$ accumulent $A_{i,j} * x_j$ en parallèle. À chaque étape de calcul de x_i, il reste donc un nombre constant d'opérations à effectuer :

$$x_i = \frac{b_i - \Sigma_i}{A_{i,i}}.$$

Le temps parallèle de cet algorithme est $\mathcal{O}(n)$. Le nombre de processeurs est également $\mathcal{O}(n)$, puisqu'il faut un processeur par accumulateur Σ_i. Ce nouvel algorithme est encore intéressant, plus rapide que le précédent, mais il ne permet pas de conclure sur l'appartenance à NC de ce problème, *i.e.* on ne sait pas encore si ce problème admet une solution parallèle rapide. En utilisant une autre approche, Csanki a proposé un algorithme d'inversion de matrices en temps $\mathcal{O}(\log^2 n)$ dans le cas des corps de caractéristique finie [Csa76] et Chistov a généralisé ce résultat en présentant un algorithme (exposé au §1.4.6) avec la même complexité temporelle valable pour les corps de caractéristique quelconque [Chi85]. Ce problème est donc dans NC. Au chapitre 3, §3.5, une construction automatique (et P-uniforme) d'un programme parallèle de résolution de systèmes triangulaires de temps $\mathcal{O}(\log^2 n)$ sera exposée.

1.4.2 Diviser pour paralléliser

Il s'agit d'une restriction du paradigme « diviser pour régner » de l'algorithmique séquentielle, qui consiste à scinder un problème en plusieurs sous-problèmes de taille moindre, qui seront traités de la même manière. En algorithmique parallèle, une contrainte supplémentaire s'impose, à savoir que les sous-problèmes doivent être traités indépendamment. Le principe algorithmique consiste en un découpage du problème en sous-problèmes indépendants, une résolution parallèle (récursive) de ces sous-problèmes, puis la construction du résultat final à

partir des solutions aux sous-problèmes. Le temps parallèle d'un tel algorithme est de façon générale :

$$T_{//}(n) = T_{découpage}(n) + T_{//}(\frac{n}{2}) + T_{fusion}(n).$$

L'exemple type dans ce domaine est le calcul des préfixes : soient n valeurs $a_1, a_2, \ldots a_n$ et $+$ une opération associative, calculer $\pi_i = \sum_{j=1}^{i} a_j$, $1 \leq i \leq n$. Le graphe de dépendance est de profondeur n, il n'est d'aucune utilité dans ce cas. Ce problème peut se découper en deux sous-problèmes de taille $\frac{n}{2}$: le calcul des préfixes $a_1, \ldots a_{\frac{n}{2}} \rightarrow \pi_1, \ldots \pi_{\frac{n}{2}}$ et un autre calcul de préfixes sur les données $b_1 = a_{\frac{n}{2}+1}, \ldots b_{\frac{n}{2}} = a_n \rightarrow \rho_1, \ldots \rho_{\frac{n}{2}}$ qui sont effectués en parallèle, puis une dernière étape où les préfixes $\pi_{\frac{n}{2}+1}, \ldots \pi_n$ sont obtenus en additionnant en parallèle chaque ρ_i à $\pi_{\frac{n}{2}}$.

La complexité temporelle $T_{//}(n)$ de cet algorithme est :

$$T_{//}(n) = T_{//}(\frac{n}{2}) + 1 = \mathcal{O}(\log n).$$

Le nombre de processeurs utilisés $P_{//}(n)$ est

$$P_{//}(n) = \max(\mathcal{O}(n), 2P_{//}(\frac{n}{2})) = \mathcal{O}(n).$$

Comme $\pi_{\frac{n}{2}}$ est utilisé par tous les processeurs lors de la dernière phase, cet algorithme nécessite une $CREW{-}PRAM$. Par abus de langage, on dira que cet algorithme est $CREW$. Cet algorithme de calcul des préfixes a pour complexité

$$CREW(n, \log n).$$

Il est possible de modifier cet algorithme pour qu'il puisse s'exécuter sur une $EREW-PRAM$. Pour cela, la phase de réduction du problème en sous-problèmes est rendue plus complexe : on calcule en parallèle les sommes $b_k = a_{2k-1} + a_{2k}$, $1 \leq k \leq \frac{n}{2}$. Le sous-problème est maintenant le calcul des préfixes sur les b_k. Il est résolu (récursivement) en parallèle. Les résultats sont les préfixes d'indice pair : π_{2k}. Les préfixes d'indice impair sont obtenus en effectuant en parallèle les sommes $\pi_{2k} + a_{2k+1}$. Il n'y a plus de lecture concurrente, ce nouvel algorithme est donc $EREW$. Sa complexité est :

$$EREW(n, \log n).$$

Il est efficace. Il est possible de le rendre optimal par la technique d'équilibrage des travaux.

1.4.3 Équilibrer les travaux

Étant donné un algorithme parallèle qui utilise p processeurs, il est possible de l'exécuter sur $q \leq p$ processeurs de la façon suivante : les q processeurs exécutent les opérations du premier pas de calcul en se répartissant équitablement les tâches, même vides ; par exemple, le processeur i effectue les opérations des processeurs $i\frac{p}{q}, \ldots (i+1)\frac{p}{q} - 1$, ou bien celles des processeurs j tels que $j \, mod \, q = i$. Il faut un temps $\mathcal{O}(\frac{p}{q})$. De la même façon, les opérations du

deuxième pas sont effectuées, puis celles des pas suivants... Cette répartition peut nécessiter la mise à plat de l'algorithme initial ; le nouveau programme sera alors décrit par une machine de Turing utilisant un espace polynomial et ne sera donc plus uniforme. C'est pour cette raison que le passage d'un algorithme efficace à un algorithme optimal, quand il est possible, est souvent réalisé par une description explicite du nouveau programme, cette description étant uniforme.

On a donc un algorithme parallèle utilisant moins de processeurs et, à la limite, si $q = 1$, une simulation séquentielle d'un algorithme parallèle. On prouve ainsi l'inclusion $NC \subset P$. Si l'algorithme parallèle original s'exécute en temps T, la simulation sur $q \leq p$ processeurs s'exécute en temps $\mathcal{O}(\frac{p}{q}T)$. Si $q = 1$, le temps d'une simulation séquentielle est égal au travail de l'algorithme parallèle. Pour $q \leq p$ quelconque, le travail de l'algorithme parallèle sur q processeurs reste du même ordre que le travail de l'algorithme pour p processeurs.

Ce mécanisme de simulation est connu sous le nom de principe de Brent. Il permet de regrouper les opérations effectuées par un algorithme parallèle pour les calculer séquentiellement. Lorsqu'un algorithme efficace, mais non optimal, est connu, ce principe est souvent appliqué pour réduire le nombre de processeurs tout en conservant la même complexité temporelle, de façon à rendre l'algorithme optimal. Cette technique s'appelle équilibrage des travaux [Kal89]. Illustrons-la sur l'exemple des préfixes, pour lequel le dernier algorithme obtenu est $EREW$ et efficace, mais pas optimal.

Si la répartition des tâches est du type « le processeur i effectue les opérations des processeurs $i\frac{p}{q}, \ldots (i+1)\frac{p}{q} - 1$ », avec $q = \frac{n}{\log n}$ processeurs[5], cela revient à faire calculer en séquentiel, à chaque processeur i, les préfixes correspondant à

$$b^i_1, \ldots b^i_{\log n} \leftarrow a_{i \log n}, \ldots a_{(i+1)\log n - 1}.$$

en temps $\mathcal{O}(\log n)$.

Ensuite, le calcul de préfixe du §1.4.2 s'applique sur les $b^i_{\log n}$, $1 \leq i \leq \frac{n}{\log n}$, avec une complexité $EREW(\frac{n}{\log n}, \log(\frac{n}{\log n})) = EREW(\frac{n}{\log n}, \log n)$; enfin, on additionne en séquentiel chacun de ces préfixes $\rho_i = \sum_{j=1}^{i \log n} a_j$ par les préfixes $b^{i+1}_1, \ldots b^{i+1}_{\log n}$ calculés lors de la première phase par le processeur $i+1$ afin d'obtenir tous les préfixes, en temps $\mathcal{O}(\log n)$, avec $\mathcal{O}(\frac{n}{\log n})$ processeurs seulement. La complexité de cet algorithme est

$$EREW(\frac{n}{\log n}, \log n).$$

C'est un algorithme optimal [LF80].

Remarque
L'hypothèse d'associativité de l'opération + est essentielle ici. Les numériciens arguent parfois de la non-associativité des opérations flottantes sur les ordinateurs, en citant des

5. n est supposé être à la fois une puissance de 2 et divisible par $\log n$ pour simplifier les formules.

exemples comme $10^n + [(-10^n) + 1] \simeq 10^n + (-10^n) = 0$ si n est assez grand, alors que $[10^n + (-10^n)] + 1 = 0 + 1 = 0$. Cependant nier l'associativité des opérations, pour préserver la stabilité numérique d'un programme, revient à rendre l'ordre séquentiel des opérations intangible, et toute tentative de parallélisation est alors vouée à l'échec. Or des procédures de préfixe (*réduction* ou *scan*) existent dans les nouveaux langages parallèles, ce qui signifie qu'elles correspondent à un compromis acceptable entre le parallélisme d'un programme et sa stabilité numérique.

1.4.4 Introduire de la redondance

Une idée intéressante en parallélisme est l'introduction de calculs redondants pour permettre de briser des dépendances. Cette idée est une hérésie en algorithmique séquentielle, où les compilateurs s'efforcent plutôt de factoriser des parties de code communes pour ne les calculer qu'une seule fois. Ajouter des calculs redondants peut faire craindre de supprimer tout espoir d'optimalité pour l'algorithme. Or, rien n'est moins sûr puisque la notion d'optimalité utilisée ici est en ordre de grandeur, multiplier par deux le nombre de calculs par exemple ne change rien à l'optimalité d'un algorithme. En outre, l'introduction de redondance peut être considérée comme une étape dans la démarche de parallélisation : elle peut conduire à des algorithmes exempts de redondance et très parallèles.

L'exemple de l'addition de deux entiers de n bits illustrera cette idée. Soient

$$\begin{array}{rcccl} A & = & (a_{n-1}, a_{n-2}, \ldots a_0) & = & \sum_{i=0}^{n-1} a_i 2^i, \\ B & = & (b_{n-1}, b_{n-2}, \ldots b_0) & = & \sum_{i=0}^{n-1} b_i 2^i \end{array}$$

deux entiers, écrits en binaire poids fort à gauche, dont on veut connaître la somme. Le graphe de dépendance de ce problème a une profondeur $\mathcal{O}(n)$, du fait des propagations de retenues. Une découpe de ce problème en une somme de $A_L + B_L$ et $A_H + B_H$ avec $A_L = (a_{\frac{n}{2}-1}, \ldots a_0)$, $B_L = (b_{\frac{n}{2}-1}, \ldots b_0)$, $A_H = (a_{n-1}, \ldots a_{\frac{n}{2}})$ et $B_H = (b_{n-1}, \ldots b_{\frac{n}{2}})$, n'est pas possible : on ne connaît pas la valeur de la retenue générée par le calcul de $A_L + B_L$. On sait cependant qu'elle ne peut prendre que deux valeurs, 0 ou 1. Les calculs redondants qui vont être effectués vont être $A_H + B_H$ et $A_H + B_H + 1$ en parallèle avec $A_L + B_L$, on sélectionnera le calcul utile selon la valeur de la retenue générée par $A_L + B_L$ et le résultat sera obtenu en recollant les deux valeurs, par une simple concaténation.

Algorithme 1 *Addition de deux entiers à n bits*

- Séparer A en A_H et A_L, B en B_H et B_L
- Calculer (récursivement) en parallèle
  $A_L + B_L$,
  $A_H + B_H$,
  $A_H + B_H + 1$.

- Si $A_L + B_L$ a produit une retenue, alors
  le résultat est la concaténation de $A_H + B_H + 1$ et de $A_L + B_L$,

sinon

c'est la concaténation de $A_H + B_H$ et de $A_L + B_L$.

fin

Calculons la complexité de cet algorithme.

$$\begin{aligned} T_{//}(n) &= T_{//}(\tfrac{n}{2}) + 1 \\ &= \mathcal{O}(\log n), \end{aligned}$$

$$\begin{aligned} P_{//}(n) &= \max(n, 3P_{//}(\tfrac{n}{2})) \\ &= \mathcal{O}(n^{\log_2 3}) \\ &= \mathcal{O}(n^{1.58}). \end{aligned}$$

Cette complexité temporelle est très intéressante et permet de conclure à l'appartenance à la classe NC^1 du problème de l'addition. Cependant, l'algorithme n'est pas efficace.

Brent et Kung [BK82] ont proposé un algorithme optimal d'addition de deux nombres de n bits, dont le principe est de reconnaître un problème de préfixe. Les équations booléennes correspondant à l'addition sont, si $r = r_n, \ldots r_0$ est le résultat et $c = c_{n-1}, \ldots c_0$ le vecteur des retenues qui existent après l'addition de $a_i, \ldots a_0$ et de $b_i, \ldots b_0$,

$$\begin{aligned} c_0 &= 0, \\ c_i &= (a_i \wedge b_i) \vee ((a_i \oplus b_i) \wedge c_{i-1}), \end{aligned}$$

$$\begin{aligned} r_i &= a_i \oplus b_i \oplus c_{i-1}, \\ r_n &= c_{n-1}. \end{aligned}$$

Quand tous les c_i sont calculés, les r_i peuvent tous être calculés en parallèle sans problème. Concentrons-nous donc sur les c_i.
c_i dépend d'une composante $g_i = a_i \wedge b_i$ qui est une condition de génération de retenue, et de $p_i = a_i \oplus b_i$ qui est une condition de propagation de retenue précédemment créée. Tous les g_i et tous les p_i peuvent être calculés en parallèle au début de l'algorithme.
L'idée de Brent et Kung est de reconnaître dans le calcul des c_i un problème de préfixe utilisant les couples (g_i, p_i) et l'opération associative $\odot$ définie par :

$$(g, p) \odot (g', p') = (g \vee (p \wedge g'), p \wedge p').$$

On a alors

$$c_i = \bigodot_{j=1}^{i} (g_j, p_j).$$

On applique l'algorithme optimal de calcul des préfixes du §1.4.3 pour calculer les c_i. Il est possible de calculer, grâce au principe d'équilibrage des travaux, les g_i, les p_i et les r_i en temps $\mathcal{O}(\log n)$ avec $\mathcal{O}(\frac{n}{\log n})$ processeurs. Cet algorithme d'addition a donc pour complexité

$$EREW(\frac{n}{\log n}, \log n).$$

Au chapitre 5, §5.2.1, des résultats expérimentaux seront présentés, dans lesquels l'addition est effectuée en temps $\mathcal{O}(\log n)$, avec $\mathcal{O}(n)$ processeurs, grâce à une parallélisation automatique de l'algorithme séquentiel.

1.4.5 Le saut de pointeur

Le problème considéré ici est l'indiçage d'une liste (*list ranking*). Il s'agit d'une variante du problème des préfixes, où les éléments sont rangés dans une liste simplement chaînée, plutôt que dans un tableau, et où l'on veut effectuer toutes les accumulations partielles de ces éléments à l'aide d'une loi associative notée +. Ces accumulations partielles seront ici $\sum_{j=i}^{n} a_j$, puisque l'élément en i^e position ne pointe que sur les éléments en plus grande position. Si chaque élément a pour valeur 1 et que l'opération est l'addition, ce problème est celui du calcul du rang de chaque élément dans la liste, en partant de la fin, ce qui justifie son nom d'indiçage de liste.

Chaque élément de la liste connaît sa valeur et un pointeur vers son successeur dans la liste. La liste est supposée se terminer par un élément artificiel qui pointe sur lui-même et qui a pour valeur 0, 0 désignant le neutre de la loi associative. La technique utilisée ici consiste à faire sauter les pointeurs par-dessus un nombre croissant d'éléments : chaque élément de la liste additionne sa valeur avec celle de son successeur puis pointe vers le successeur de son successeur. Le pointeur saute maintenant deux cases. On recommence avec, pour chaque élément, un nombre deux fois plus petit d'éléments après lui dans sa liste chaînée. (La structure de liste chaînée est conservée, mais il y a désormais deux listes). Comme la longueur de chaque liste est divisée par 2, on appliquera récursivement cette idée $\log n$ fois. L'algorithme est le suivant :

Algorithme 2 *Indiçage de liste CREW*

faire $\log n$ fois
    faire en parallèle pour chaque élément
        accumulateur $\leftarrow$ accumulateur + valeur(successeur)
        successeur $\leftarrow$ successeur (successeur).

fin

Cet algorithme est *CREW* puisque de plus en plus de processeurs vont lire les données du dernier élément de la liste. Pour chaque processeur, il est possible de tester s'il pointe sur un élément de la liste qui a fini son calcul ; il s'arrête alors, en pointant sur lui-même.

Algorithme 3 *Indiçage de liste EREW*

faire $\log n$ fois
    faire en parallèle pour chaque élément i
        sauvegarde $\leftarrow$ successeur
        si (sauvegarde = i) alors
            { l'élément pointe sur lui-même, *i.e.* il a fini son calcul }
            ne rien faire
        sinon
            accumulateur $\leftarrow$ accumulateur + valeur(successeur)

successeur ← successeur (successeur).
si (sauvegarde = successeur) alors
{ le successeur pointe sur lui-même, c'est qu'il a fini son calcul }
sauvegarde ← i

fin

Enfin, il est possible de rendre cet algorithme optimal, en utilisant $\frac{n}{\log n}$ processeurs [CV88].

1.4.6 Inversion de matrice et calcul de déterminant

Ce dernier paragraphe est destiné à mettre en évidence la différence entre un algorithme parallèle rapide et un algorithme destiné à résoudre le même problème en séquentiel. Si l'algorithme de Chistov qui est présenté ici est simulé en séquentiel, il s'exécutera en un temps bien supérieur à celui de n'importe lequel des algorithmes séquentiels classiques qui résolvent ce problème. Cependant il permet de conclure à l'appartenance à NC^2 du problème de l'inversion d'une matrice, alors que « penser séquentiel » avait conduit à un algorithme au temps parallèle linéaire pour le problème *a priori* plus simple de la résolution d'un système triangulaire inférieur.

Ce paragraphe s'intitule « Inversion de matrice et calcul de déterminant » parce que ces deux problèmes sont équivalents. En effet, si on sait calculer le déterminant d'une matrice $n \times n$ inversible à coefficients dans un corps de caractéristique quelconque, on sait également calculer A^{-1}, l'inverse d'une matrice A $n \times n$ inversible, à l'aide de $\det(XI - A)$, en utilisant le théorème de Cayley-Hamilton : si on note $P_A(X) = \det(XI - A)$, alors

$$\begin{array}{rlcl} & P_A(X) & = & X^n + \ldots + a_1 X + \det(A), \\ & P_A(A) & = & 0, \\ & P_A(A) & = & A * (A^{n-1} + \ldots + a_1 I + \det(A) A^{-1}), \\ \Rightarrow & A^{-1} & = & -\frac{A^{n-1} + \ldots + a_1 I}{\det(A)}. \end{array}$$

Réciproquement, si on sait inverser une matrice, on sait calculer son déterminant : si on note A_i la sous-matrice carrée $i \times i$ de A obtenue en ne gardant que les i dernières lignes et colonnes de A (la sous-matrice « en bas à droite » de A), on a la formule

$$\prod_{i=1}^{n} (A_i^{-1})_{1,1} = \frac{1}{\det(A)}.$$

Chistov [Chi85] a montré comment calculer en temps $\mathcal{O}(\log^2 n)$ avec un nombre polynomial de processeurs le déterminant d'une matrice $n \times n$. La dernière égalité permet d'écrire

$$\frac{1}{\det(I - xA)} = \prod_k ((I_k - xA_k)^{-1})_{1,1}.$$

On utilise des développements en série formelle pour inverser $I_k - xA_k$:

$$(I_k - xA_k)^{-1} = \sum_{i=0}^{+\infty} x^i A_k^i.$$

Comme le déterminant $\det(I - xA)$ est un polynôme de degré n, après avoir calculé l'inverse du premier coefficient (en haut à gauche) grâce à un nouveau développement en série, on peut tronquer toutes ces séries à l'ordre n pour obtenir le résultat exact.

Il reste à calculer les puissances de matrices $(A_k^i)_{0 \leq i \leq n}$, ce que l'on sait faire avec un produit itéré où l'opération associative est le produit de matrices. Comme chaque produit de matrices a une complexité temporelle $\mathcal{O}(\log n)$ et utilise un nombre polynomial de processeurs, on obtient un temps total $\mathcal{O}(\log^2 n)$ et un nombre de processeurs polynomial.

Cette section a offert un rapide tour d'horizon des grands types de programmes parallèles les plus courants. Les algorithmes de calcul de préfixe et d'indiçage de liste seront utilisés pour résoudre des sous-problèmes à plusieurs reprises par la suite.

Au cours de cette section, certaines techniques de parallélisation se sont appuyées sur les propriétés algébriques des opérations mises en jeu par les programmes. Nous rappelons les définitions de ces structures afin de pouvoir les utiliser plus tard.

1.5 Structures algébriques

Afin de faciliter la compréhension de la suite de ce travail, les définitions des structures algébriques utilisées par la suite sont rappelées dans cette dernière section. Ces définitions se trouvent dans tout livre d'algèbre de DEUG ou de classes préparatoires. Nous nous sommes basés sur les livres d'Arnaudiès et Fraysse [AF87], ainsi que sur celui de Carvallo en ce qui concerne les treillis [Car66]

1.5.1 Monoïdes et groupes

Commençons par les structures ayant une seule loi, appelée addition et notée $+$. La structure la plus simple qui sera envisagée est le monoïde.

Définition 18 *Un monoïde est un triplet $(E, +, 0)$, où*

- *E est un ensemble ;*
- *0 est un élément de E ;*
- *$+$ est une loi de composition interne sur E,*
  - *associative : $\forall (a, b, c) \in E^3,\ (a + b) + c = a + (b + c)$;*
  - *0 est élément neutre pour $+$: $\forall a \in E,\ a + 0 = 0 + a = a$.*

Le monoïde est dit commutatif ou abélien si la loi $+$ est de plus commutative : $\forall (a, b) \in E^2,\ a + b = b + a$

La condition d'associativité a déjà été implicitement utilisée dans l'algorithme parallèle du préfixe 1.4.3, pour regrouper les calculs. L'existence d'un élément neutre montrera son importance pour tous les algorithmes d'évaluation, que ce soit des expressions ou des circuits

arithmétiques dans les semi-anneaux ou dans les treillis.

Exemples

- $(\mathbb{N}, +, 0)$ est un monoïde commutatif ;
- $(\mathbb{N}, *, 1)$ est un monoïde commutatif ;
- si E est un ensemble et $\mathcal{P}(E)$ est l'ensemble des parties de E, alors $(\mathcal{P}(E), \cup, \emptyset)$ et $(\mathcal{P}(E), \cap, E)$ sont des monoïdes commutatifs ;
- si E est un ensemble et $\mathcal{F}(E)$ est l'ensemble des fonctions de E dans E, $(\mathcal{F}(E), \circ, I_d)$ est un monoïde non commutatif.

Si chaque élément de E admet un symétrique, *i.e.* si $\forall a \in E$, $\exists \bar{a} \in E \,/\, a + \bar{a} = \bar{a} + a = 0$, la structure est un groupe.

Définition 19 *Un groupe est un triplet $(E, +, 0)$, où*

- *$(E, +, 0)$ est un monoïde ;*
- *tout élément de E admet un symétrique.*

Le groupe est dit commutatif ou abélien si la loi $+$ est de plus commutative.

Exemples

- $(\mathbb{N}, +, 0)$ n'est pas un groupe ;
- $(\mathbb{N}, *, 1)$ n'est pas un groupe ;
- $(\mathbb{Z}, +, 0)$ est un groupe commutatif ($\mathbb{Z}$ est d'ailleurs construit pour être le groupe engendré par $\mathbb{N}$) ;
- si E est un ensemble, $\mathcal{P}(E)$ est l'ensemble des parties de E et Δ est la différence symétrique : $A \Delta B = (A \setminus B) \cup (B \setminus A) = (A \cup B) \setminus (A \cap B)$, alors $(\mathcal{P}(E), \Delta, \emptyset)$ est un groupe commutatif. Chaque élément est son propre symétrique.

1.5.2 Semi-anneaux, dioïdes et anneaux

Passons maintenant aux structures munies d'une deuxième loi, appelée multiplication et notée $*$.

Définition 20 *Un semi-anneau est un quadruplet $(E, +, *, 0)$, tel que*

- *E est un ensemble ;*

- *0 est un élément de E ;*
- *$(E,+,0)$ est un monoïde commutatif ;*
- *$*$ est associative ;*
- *0 est un élément absorbant pour $*$;*
- *$*$ est distributive par rapport à $+$: $\forall(a,b,c) \in E^3$, $(a+b)*c = (a*c)+(b*c)$ et $c*(a+b) = (c*a)+(c*b)$.*

Le semi-anneau est dit commutatif si la loi $*$ est de plus commutative.

Définition 21 *Un semi-anneau unitaire est un quintuplet $(E, +, *, 0, 1)$, tel que $(E, +, *, 0)$ est un semi-anneau et la loi $*$ admet un élément neutre noté 1, i.e. $(E, *, 1)$ est un monoïde.*

Définition 22 *Un semi-anneau unitaire $(E, +, *, 0, 1)$ est intègre si*

- *il est non nul (un semi-anneau nul est un anneau dont l'ensemble de base est réduit à un seul élément, 0, et $0+0=0$, $0*0=0$) ;*
- *la multiplication est commutative ;*
- *il ne contient pas de diviseur de zéro[6], i.e. $\forall(a,b) \in E^2$, si $a \neq 0$ et $b \neq 0$, alors $a*b \neq 0$.*

Par convention, $*$ est prioritaire par rapport à $+$, *i.e.* $a*b+c$ s'interprète comme $(a*b)+c$.

Dans ce qui suit, seuls les semi-anneaux unitaires seront utilisés et l'adjectif unitaire pourra être omis.

Exemples

- $(\mathbb{N}, +, *, 0, 1)$ est un semi-anneau commutatif unitaire intègre ;
- $(\mathbb{Z}, +, *, 0, 1)$ est un semi-anneau commutatif unitaire intègre ;
- $(\mathbb{R}, +, *, 0, 1)$ est un semi-anneau commutatif unitaire intègre ;
- si E est un ensemble, $\mathcal{P}(E)$ est l'ensemble des parties de E, alors $(\mathcal{P}(E), \cup, \cap, \emptyset, E)$ est un semi-anneau commutatif unitaire non intègre. ;
- $(\mathcal{P}(\{a,b,\ldots z\}^*), \cup, ., \emptyset, \{\varepsilon\})$ est un semi-anneau non commutatif unitaire, avec
  - $\{a,b,\ldots z\}^*$ est l'ensemble des mots formés à l'aide de l'alphabet (usuel) $a, b, \ldots z$;

6. Un diviseur de 0 est un élément $a \neq 0 \in E$ tel que $\exists x \neq 0 \in E$, $a*x = 0$.

- $\cup$ est encore l'union ensembliste, ;
- . est une extension de la concaténation de chaînes : $A.B$ est l'ensemble contenant les concaténations de toutes les chaînes de A avec toutes les chaînes de B ;
- ε est la chaîne vide (qui ne comporte aucune lettre).

Définition 23 *Un dioïde est un quintuplet $(E, +, *, 0, 1)$, tel que*

- *$(E, +, *, 0)$ est un semi-anneau unitaire ;*
- *l'addition est idempotente : $\forall\, a \in E,\ a + a = a$.*

Le dioïde est dit commutatif si la loi $*$ est de plus commutative.

Exemples

- Le dernier exemple de semi-anneau est un dioïde ;
- $([0, 1], \max, \min, 0, 1)$ est un semi-anneau commutatif unitaire ;
- $(\mathbb{R}, \max, +, -\infty, 0)$ et son équivalent avec les réels positifs, $(\mathbb{R}^+, \max, *, 0, 1)$ sont des semi-anneaux commutatifs unitaires.

L'anneau est une structure très ressemblante à celle de semi-anneau, mais l'addition est une loi de groupe.

Définition 24 *Un anneau est un quintuplet $(E, +, *, 0)$ où*

- *$(E, +, 0)$ est un groupe commutatif ;*
- *$(E, +, *, 0)$ est un semi-anneau.*

Un anneau est dit commutatif si la loi $*$ est de plus commutative.

Définition 25 *Un anneau $(E, +, *, 0, 1)$ est unitaire si le semi-anneau sous-jacent est unitaire.*

Définition 26 *Un anneau unitaire $(E, +, *, 0, 1)$ est intègre si le semi-anneau unitaire sous-jacent est intègre.*

Nous n'utiliserons que les anneaux unitaires et pour cette raison il se peut que nous omettions le qualificatif « unitaire ».

Exemples

- $(\mathbb{Z}, +, *, 0, 1)$ est un anneau commutatif unitaire intègre ;

- $(\mathbb{R}, +, *, 0, 1)$ est un anneau commutatif unitaire intègre ;
- $(\mathbb{C}^n, +, *, (0,\dots 0), (1,\dots 1))$ avec $(x_1,\dots x_n) * (y_1,\dots y_n) = (x_1y_1,\dots x_ny_n)$ est un anneau commutatif unitaire non intègre ;
- si $(A, +, *, 0, 1)$ est un anneau et si X est un ensemble, alors $(A^X, \oplus, \otimes, \dot{0}, \dot{1})$ est un anneau, qui est commutatif si et seulement si A est commutatif, avec
  - A^X l'ensemble des fonctions de X dans A ;
  - $\oplus : f \oplus g : \begin{array}{ccc} X & \to & A \\ x & \mapsto & f(x)+g(x) \end{array}$;
  - $\otimes : f \otimes g : \begin{array}{ccc} X & \to & A \\ x & \mapsto & f(x)*g(x) \end{array}$;
  - $\dot{0} : \begin{array}{ccc} X & \to & A \\ x & \mapsto & 0 \end{array}$;
  - $\dot{1} : \begin{array}{ccc} X & \to & A \\ x & \mapsto & 1 \end{array}$;
- si $(A, +, *, 0, 1)$ est un anneau commutatif, alors l'ensemble des applications linéaires de $A \times A$ dans $A \times A$, noté $\mathcal{M}_2(A)$, est également un anneau, qui n'est pas commutatif, mais qui est unitaire. Il contient des diviseurs de zéro.

1.5.3 Corps

Dernière catégorie de structure dans la lignée des précédentes : les corps. Il s'agit des structures où tout élément non nul admet un inverse pour $*$.

Définition 27 *Un corps est un quintuplet $(E, +, *, 0, 1)$ qui vérifie les propriétés suivantes :*

- *$(E, +, *, 0, 1)$ est un anneau unitaire ;*
- *$(E \setminus \{0\}, *, 1)$ est un groupe : $\forall x \in E \setminus \{0\}, \exists x^{-1} / \ x * x^{-1} = 1$.*

Un corps est dit commutatif si la loi $*$ est de plus commutative.

En particulier un corps n'a pas de diviseur de zéro.

Nous noterons parfois un corps par F, comme mnémonique du terme anglais pour corps : *field*.

Exemples

- $(\mathbb{Z}/p\mathbb{Z}, +, *, 0, 1)$ est un corps commutatif si p est un nombre premier ;

- $(\mathbb{Q},\ +,\ *, 0,\ 1)$ est un corps commutatif. (Il a d'ailleurs été construit pour être le corps engendré par $\mathbb{Z}$) ;
- $(\mathbb{C}^n,\ +,\ *,\ (0,\ldots 0),\ (1,\ldots 1))$ est un corps commutatif ;
- $(\mathbb{Q}(\sqrt{2}), +, *, 0, 1)$, $(\mathbb{Q}(i), +, *, 0, 1)$ et toutes les extensions de la forme $(\mathbb{Q}(\alpha_1, \ldots \alpha_n), +, *, 0, 1)$ sont des corps commutatifs ;
- $(F[\alpha],\ +,\ *,\ 0,\ 1)$ est un corps, où $(F,\ +,\ *,\ 0,\ 1)$ est un corps et où α est algébrique sur F, puisqu'alors $F[\alpha] = F(\alpha)$;
- $(A,\ +,\ *,\ 0,\ 1)$, avec $A = \{\alpha \in \mathbb{C} \ /\ \ \alpha$ est algébrique sur $\mathbb{Q}\}$, est un corps ; c'est la clôture algébrique de $\mathbb{Q}$.

1.5.4 Treillis

Ce dernier type de structure est assez différent des précédents, en ce sens qu'il n'y a pas d'opération prioritaire. Un treillis admet deux définitions équivalentes, l'une qui le caractérise par un ordre partiel et l'autre qui est algébrique. Nous n'utiliserons que la définition algébrique, mais il est parfois utile de penser à un treillis en terme d'ensemble ordonné pour faciliter la compréhension.

Nous appellerons souvent l'ensemble de base L, pour le terme anglais *lattice*. Pour distinguer les opérations d'un treillis de celles d'un monoïde, d'un groupe, d'un semi-anneau, d'un anneau ou d'un corps, l'addition sera représentée par $\oplus$ et la multiplication par $\otimes$.

Définition 28 Définition algébrique
Un treillis $(L,\ \oplus,\ \otimes)$ est un ensemble L muni de deux lois de composition internes, $\oplus$ et $\otimes$:

- *elles sont commutatives : $\forall a, b \in L,\ a \oplus b = b \oplus a$ et $a \otimes b = b \otimes a$;*
- *elles sont associatives : $\forall a, b, c \in L,\ a \oplus (b \oplus c) = (a \oplus b) \oplus c$ et $a \otimes (b \otimes c) = (a \otimes b) \otimes c$;*
- *elles vérifient la loi d'absorption : $\forall a, b \in L,\ (a \oplus b) \otimes a = (a \otimes b) \oplus a = a$.*

De la loi d'absorption, on peut déduire l'idempotence de $\oplus$ et de $\otimes$: $\forall a \in L,\ a \oplus a = a \otimes a = a$

Définition à l'aide d'un ensemble partiellement ordonné
Un treillis est un ensemble partiellement ordonné tel que, pour tout couple d'éléments $(a,\ b)$, il existe un plus petit majorant de $\{a,\ b\}$, noté $a \oplus b$, et un plus grand minorant de $\{a,\ b\}$, noté $a \otimes b$.

Définition 29 *Un treillis distributif est un treillis $(L,\ \oplus,\ \otimes)$ tel que l'une des lois est distributive par rapport à l'autre.*

En effet, si $\otimes$ est distributive par rapport à $\oplus$, alors $\oplus$ sera distributive par rapport à $\otimes$.

Définition 30 *Un treillis complémenté est un treillis* $(L, \oplus, \otimes, \neg, \epsilon, e)$ *tel que* ϵ *est le plus petit élément du treillis et* e *est le plus grand élément.* e *est alors élément neutre pour* $\otimes$ *et* ϵ *est élément neutre* $\oplus$*. La loi* $\neg$ *est une loi unaire, telle que* $\forall x \in L, \neg x \otimes x = \epsilon$ *et* $\neg x \oplus x = e$*.*

On peut en déduire les lois de De Morgan :

$$\begin{aligned} \neg(a \oplus b) &= (\neg a) \otimes (\neg b), \\ \neg(a \otimes b) &= (\neg a) \oplus (\neg b). \end{aligned}$$

Définition 31 *Un treillis de Boole est un treillis distributif complémenté.*

Remarque
Dans ce travail, nous nous restreindrons d'une part aux treillis distributifs, et d'autre part aux treillis qui ont un plus grand élément e et un plus petit élément ϵ. Cette dernière condition n'est pas réellement contraignante, puisqu'il est toujours possible d'ajouter deux éléments artificiels qui conviennent, les lois s'étendant « naturellement » à ces deux éléments : $\forall x \in \overline{L}$, $e \oplus x = e$ et $\epsilon \otimes x = \epsilon$ grâce à la loi d'absorption.

Exemples

- $([0, 1], \min, \max, 1, 0)$ est un treillis distributif ;
- $(\overline{\mathbb{R}}, \max, \min, -\infty, +\infty)$ est un treillis distributif ;
- $(\mathbb{N}, pgcd, ppcm, +\infty, 1)$ est un treillis; l'ordre sur $\mathbb{N}$ est l'ordre induit par la relation de divisibilité ;
- l'algèbre de Boole $(\{Vrai, Faux\}, \wedge, \vee, Vrai, Faux)$ est en fait un treillis distributif ;
- certaines logiques excluant le principe du tiers exclu comme axiome de base ont un ensemble de valeurs de vérité qui forment un treillis dont $Vrai$ est le plus grand élément et $Faux$ le plus petit. En particulier, les mathématiques constructives sont basées sur de telles logiques.

1.6 Conclusion

Ce premier chapitre nous a permis de replacer le problème étudié dans cette thèse dans un cadre général. Il sert également de boîte à outils pour la suite du mémoire. En particulier, il contient les définitions de base de l'algorithmique parallèle, comme les définitions de machine et de programme parallèles, et celles des classes de complexité qui permettent d'évaluer la qualité d'un programme parallèle et de le comparer à d'autres programmes. Enfin, les grands principes de l'algorithmique parallèle, comme « diviser pour paralléliser » et l'introduction de redondance, ont été présentés et illustrés par des problèmes dont l'importance apparaîtra par la suite. Ce chapitre se termine par la liste des définitions des structures algébriques utiles, dont les propriétés s'avéreront fondamentales pour les techniques de parallélisation qui constituent le cœur de ce travail.

Première partie

Évaluation parallèle de programmes sans boucle

Chapitre 2

Évaluation parallèle des expressions

Résumé

Dans ce chapitre, nous présentons un algorithme efficace d'évaluation parallèle d'expressions arithmétiques, puis une amélioration le rendant optimal. Quelques variantes sont également présentées, qui permettent de calculer dans des ensembles finis ou d'évaluer toutes les sous-expressions d'une expression, avec la même complexité. Cet algorithme est ensuite appliqué à la reconnaissance de langages hors-contexte parenthésés : il s'agit d'une version parallèle de l'algorithme de Cocke, Younger et Kasami, qui est déterministe dans le cas des langages parenthésés. Ces algorithmes d'évaluation utilisent une représentation arborescente des expressions. Dans le but de présenter une solution complète au problème de l'évaluation parallèle des expressions, l'algorithme de construction parallèle de l'arbre correspondant à une expression – dû à Bar-On et Vishkin – est développé ainsi que celui de numérotation parallèle des feuilles. Enfin, nous rappelons les complexités obtenues pour une parallélisation sur machines à mémoire distribuée.

Les expressions arithmétiques forment un modèle de programmes assez simple. Le problème de l'évaluation parallèle des expressions arithmétiques, qui est NC_1-complet [Bus87], est l'un des problèmes traités dans ce mémoire, avec celui de l'évaluation des circuits arithmétiques, qui est un problème P-complet. Les expressions arithmétiques, également appelées formules [Bus87, BCGR92] seront définies au début de ce chapitre. Elles sont formées à partir de valeurs et d'opérations arithmétiques ($+$, $*$, $-$, $/$) sur un corps ; cette structure algébrique présente l'avantage d'être assez générale, de posséder les propriétés des monoïdes, des groupes, des semi-anneaux et des anneaux commutatifs unitaires intègres et par conséquent de permettre de déduire facilement, à partir d'un algorithme d'évaluation d'expressions sur un corps, un algorithme pour n'importe laquelle de ces structures, si elle est commutative. Pour un groupe par exemple, un nouvel algorithme d'évaluation est obtenu en supprimant tout ce qui concerne $*$ et $/$ dans l'algorithme d'évaluation pour les corps. Le problème du traitement des divisions par zéro se règle un peu brutalement, en essayant d'effectuer cette division et en retournant une erreur (comme l'algorithme évalue toutes les opérations de l'expression, cf. §2.4.3, cette division ne pourra pas être éludée).

Une parallélisation gloutonne, où une opération est évaluée dès que ses opérandes sont évalués, ne donnant pas de bons résultats, Brent, en 1974 [Bre74], propose d'utiliser les propriétés algébriques des opérations mises en jeu, telles que l'associativité, la commutativité et surtout la distributivité, pour établir des bornes supérieures théoriques. Son algorithme nécessite un pré-traitement de l'expression, pour la découper en sous-expressions de complexités comparables. En 1985, Miller et Reif [MR85] ont proposé un algorithme d'évaluation parallèle des expressions arithmétiques, sans pré-traitement, qui utilise les idées de Brent. Cet algorithme a alors donné naissance, après simplification puis divers raffinements et reformulations, à un algorithme efficace et simple d'évaluation parallèle dynamique des expressions arithmétiques. Le même algorithme a été proposé, avec différentes variantes, par Kosaraju et Delcher [KD88], Abrahamson, Dadoun, Kirkpatrick et Przytycka [ADKP89], Cole et Vishkin [CV88] et Gibbons et Rytter [GR86, GR88b]. Le premier algorithme, proposé par Miller et Reif [MR85], est efficace : il permet d'évaluer le résultat d'une expression arithmétique sur une $CREW-PRAM$ à n processeurs en temps $\mathcal{O}(\log n)$, si n est le nombre d'opérations et d'opérandes. Il a ensuite été montré que l'hypothèse « lecture concurrente » était superflue dans cet algorithme [ADKP89], ce qui donne une complexité $EREW(n, \log n)$ pour ce problème. Une modification de cet algorithme fournit un nouvel algorithme qui permet en deux fois plus de temps seulement d'évaluer le résultat de toutes les sous-expressions [GR86]. Enfin, une application du principe de Brent, associée à la technique d'équilibre des travaux, conduit à un algorithme optimal de complexité $EREW(\frac{n}{\log n}, \log n)$ [GR88b].

Dans un souci didactique, nous présenterons tout d'abord l'algorithme le plus simple pour les expressions. Différentes variantes du même algorithme seront ensuite esquissées, soit pour calculer dans un ensemble fini non commutatif, soit pour déterminer tous les résultats partiels, *i.e.* le résultat de toutes les opérations apparaissant dans l'expression, soit enfin pour obtenir un algorithme optimal. Nous montrerons ensuite les résultats de cet algorithme appliqué à divers problèmes, en particulier à la reconnaissance de langages hors-contexte : il s'agit d'une parallélisation de l'algorithme (rendu déterministe) de Cocke, Younger et Kasami. Les solutions aux problèmes préliminaires de la construction parallèle d'un arbre à partir d'une expression (algorithme de Bar-On et Vishkin [BOV85]) ou de la numérotation des feuilles de l'arbre seront exposées, afin de compléter la présentation de l'évaluation dynamique des expressions arithmétiques. Des résultats de complexité pour machines à mémoire distribuée seront ensuite rappelés ; en général, les communications induisent un surcoût (multiplicatif) de la taille du diamètre du réseau d'interconnexion.

Cependant, les expressions arithmétiques ne permettent de représenter qu'une classe assez limitée de programmes. C'est ce que nous illustrerons dans un dernier paragraphe qui nous servira de transition vers l'évaluation dynamique de programmes arithmétiques plus généraux, les programmes sans boucle.

2.1 Position du problème

Rappelons la définition d'une expression arithmétique et sa représentation sous forme d'arbre, avant de travailler avec ces objets.

2.1.1 Expressions et arbres : définitions

De façon générale, la structure algébrique utilisée dans ce chapitre sera un corps $(F, +, *, 0, 1)$. Une expression arithmétique est une formule mathématique usuelle, une alternance d'éléments de F et d'opérations $+$, $-$, $*$ ou $/$. L'expression est supposée être complètement parenthésée, pour éviter toute ambiguïté dans le choix de sa représentation arborescente. De façon plus formelle, on peut définir une expression à l'aide de la grammaire suivante, *valeur* étant un symbole terminal représentant un élément quelconque du corps F :

$$\begin{aligned} expr \quad \leftarrow \quad & (expr + expr) \\ & \mid (expr - expr) \\ & \mid (expr * expr) \\ & \mid (expr / expr) \\ & \mid (valeur). \end{aligned}$$

Une expression admet une représentation sous forme d'arbre orienté binaire, où chaque nœud a zéro ou deux fils. Définissons ces arbres.

Définition 32 *[Ber73]*

- *Un arbre est un graphe non orienté connexe sans cycle.*
- *Un arbre orienté (ou arborescence) est un graphe orienté possédant un sommet particulier appelé racine et tel que pour tout sommet, il existe un unique chemin liant ce sommet à la racine.*

Définition 33 *Un arbre orienté est binaire si tout nœud a zéro ou deux fils*[1].

Dans ce chapitre, seuls les arbres binaires orientés seront utilisés.

2.1.2 Représentation arborescente des expressions

On obtient l'arbre orienté binaire associé à une expression en construisant l'arbre syntaxique à l'aide de la grammaire précédente : les valeurs sont les feuilles de l'arbre, les nœuds internes (non feuilles) sont les opérations, chaque opération a pour fils ses deux opérandes. La racine est la dernière opération effectuée par l'expression, appelée *résultat* de l'expression. Dans ce qui suit, les expressions seront représentées uniquement par des arbres. Le problème de la construction parallèle d'un arbre à partir d'une expression donnée sous forme de tableau a été résolu par Bar-On et Vishkin en 1985 [BOV85]. Nous présenterons leur technique à la fin de ce chapitre.

Rappelons quelques propriétés élémentaires des arbres, utiles par la suite.

Propriété 1 *Un arbre à n sommets a $n-1$ arcs. Un arbre orienté binaire à f feuilles a $2f-1$ sommets au total.*

1. La définition donnée ici est une restriction de la définition générale : un arbre binaire général est un arbre où chaque nœud a zéro, un ou deux fils.

Il est également intéressant de connaître les opérations qui peuvent être effectuées rapidement en parallèle sur les arbres. On sait construire un circuit eulérien d'un arbre binaire à n nœuds en temps constant avec n processeurs ou en temps $\mathcal{O}(\log n)$ avec $\frac{n}{\log n}$ processeurs, sur une $EREW{-}PRAM$ [TV85]. Pour cela, chaque arête (non orientée) est remplacée par deux arcs orientés opposés et le circuit eulérien est un circuit qui passe une fois et une seule par tous ces arcs (ou une fois dans chaque sens par chaque arête). À l'aide de ce circuit eulérien, on peut numéroter de façon pré-ordonnée ou post-ordonnée les nœuds de l'arbre, calculer les niveaux des nœuds ou le nombre de descendants de chaque nœud, *i.e.* le nombre de nœuds dans le sous-arbre dont il est racine, avec la même complexité, en utilisant un algorithme d'indiçage de liste simple, de complexité $EREW(\frac{n}{\log n}, \log n)$.

Par la suite, nous abrégerons « arbre orienté binaire » par « arbre ».

2.2 Algorithme d'évaluation des expressions

2.2.1 Position du problème

Le problème est de savoir comment évaluer rapidement en parallèle une expression donnée sous forme d'arbre. Une évaluation parallèle gloutonne, qui évalue une opération dès que la valeur de ses opérandes est connue, s'avère particulièrement inefficace dans le cas du produit itéré par exemple (il s'agit de calculer $\sum_{i=1}^{n} v_i$) si l'expression est donnée sous la forme $v_1 + (v_2 + \ldots + (v_{n-1} + v_n)\ldots)$ (figure 2.1). Le temps d'évaluation est égal à la profondeur de l'arbre, en $\mathcal{O}(n)$.

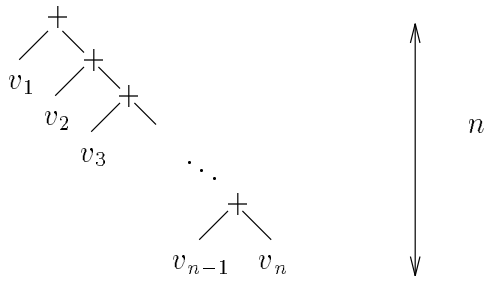


FIG. 2.1 - *Produit itéré* $v_1 + (v_2 + \ldots + (v_{n-1} + v_n)\ldots)$ *: l'arbre est ici déséquilibré.*

Dans ce cas précis, en utilisant l'associativité de $+$, on aurait pu écrire l'expression sous la forme $(\ldots((v_1 + v_2) + (v_3 + v_4)) + \ldots + (v_{n-1} + v_n)\ldots)$; cette expression est représentée par un arbre équilibré (figure 2.2), de profondeur $\log n$. Le temps d'évaluation devient alors $\mathcal{O}(\log n)$.

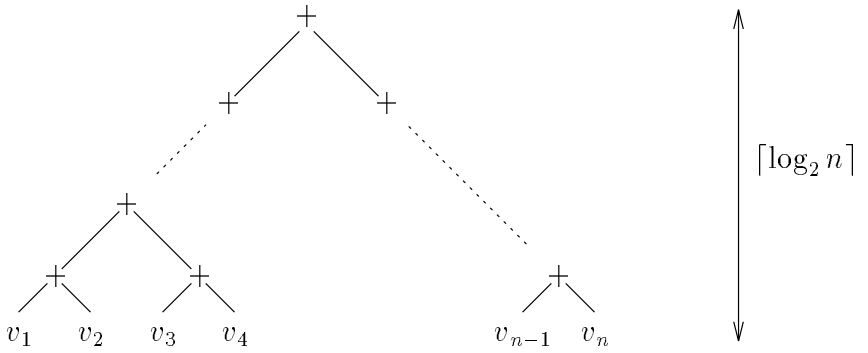


FIG. 2.2 - *Produit itéré* $(\ldots((v_1+v_2)+(v_3+v_4))+\ldots+(v_{n-1}+v_n)\ldots)$ *: l'arbre est ici équilibré.*

Si maintenant, on considère un « peigne » où l'on alterne les deux opérations + et $*$ (à dessein, pour empêcher tout rééquilibrage) (figure 2.3), on ne peut pas construire un arbre équilibré équivalent et le temps d'évaluation par un algorithme glouton est encore $\mathcal{O}(n)$.

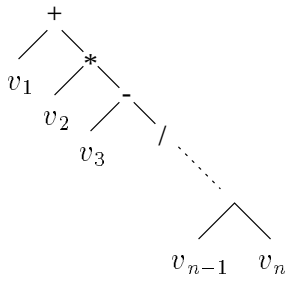


FIG. 2.3 - *Un peigne déséquilibré.*

L'idée de Brent, en 1974, a été de considérer que le résultat d'une expression peut s'écrire sous la forme d'une fraction rationnelle de numérateur et de dénominateur affines

$$\frac{av_i+b}{cv_i+d},$$

où l'on considère la feuille v_i comme une indéterminée, et de montrer que l'on peut calculer – récursivement, en utilisant les autres feuilles – les coefficients a, b, c et d en temps $\mathcal{O}(\log n)$. Les travaux ultérieurs ont porté sur le découpage de l'expression, c'est-à-dire sur la détermination de la feuille qui conduit à des calculs de même importance pour chacun des coefficients a, b, c et d. L'idée de Brent, appliquée telle quelle, conduisait à un découpage en $\mathcal{O}(\log n)$ à chaque étape, ce qui donnait un algorithme d'évaluation en temps $\mathcal{O}(\log^2 n)$. Buss, Cook, Gupta et Ramachandran [BCGR92] ont proposé un nouvel algorithme permettant de précalculer toutes les découpes en temps $\mathcal{O}(\log n)$ avant d'appliquer une phase d'évaluation, ce qui donnait un temps d'évaluation parallèle de $\mathcal{O}(\log n)$.

Il n'y a pas eu d'avancée notable dans la décennie qui a suivi le résultat de Brent. En 1985, Miller et Reif [MR85] ont eu l'idée de considérer chaque opération comme étant une fonction du type $\frac{ax+b}{cx+d}$ de son opérande x. De plus, ces fonctions étant stables par addition et multiplication par une constante, et par composition, on peut alors « anticiper » des calculs en attendant que la valeur de x soit connue en « propageant » cette fonction.

C'est ce mécanisme qui est employé de façon systématique et qui permet d'évaluer n'importe quel arbre en temps logarithmique. Définissons algorithmiquement la procédure de *ratissage* (l'anticipation).

2.2.2 Fonctions de pondération des arcs

La structure algébrique utilisée dans ce chapitre est un corps $(F, +, *, 0, 1)$. Dans toute la suite, à chaque arc $v - -v'$ de l'arbre, avec v' fils ou opérande de v, est associée une fonction d'arc [2],

$$f(x) = \frac{a * x + b}{c * x + d},$$

représentée par le quadruplet $(a, b, c, d) \in F^4$. La valeur du nœud v, $val(v)$, est alors liée à la valeur $f(val(v'))$, plutôt qu'à la valeur de v' (figure 2.4) : si v est un opérateur $\diamond$, avec $\diamond$ représentant $+$ ou $*$, v' a pour valeur α et v'' a pour valeur β, alors v vaudra $f'(\alpha) \diamond f''(\beta)$.

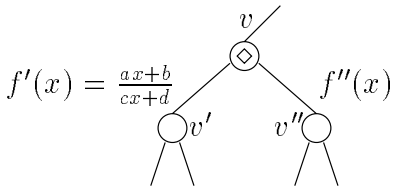


FIG. 2.4 - *Fonction d'arc.*

À l'initialisation, c'est-à-dire lors de la construction de la représentation arborescente de l'expression, on associe à chaque arc la fonction *Identité :* $\frac{1*x+0}{0*x+1}$, représentée par le quadruplet $(1, 0, 0, 1)$. C'est ici qu'apparaît l'importance de l'existence d'un élément neutre pour la multiplication, *i.e.* de l'hypothèse semi-anneau ou anneau unitaire.

Un autre avantage des fonctions d'arcs proposées est de simplifier les arbres, en considérant que les nœuds opérateurs ne sont que de type $+$ ou $*$. À cette fin, on remplace tout nœud $-$ par un nœud $+$ ayant les mêmes fils et on pondère l'arc vers le fils droit par la fonction $-x = \frac{-1*x+0}{0*x+1}$; de la même façon, on remplace tout nœud / par un nœud $*$ ayant les mêmes fils, en pondérant l'arc vers le fils droit par la fonction $\frac{1}{x} = \frac{0*x+1}{1*x+0}$.

2. Convention de notation : on appelle les nœuds « v » comme « vertex » et on désigne les éléments du corps F par des lettres grecques ou les premières lettres de l'alphabet latin.

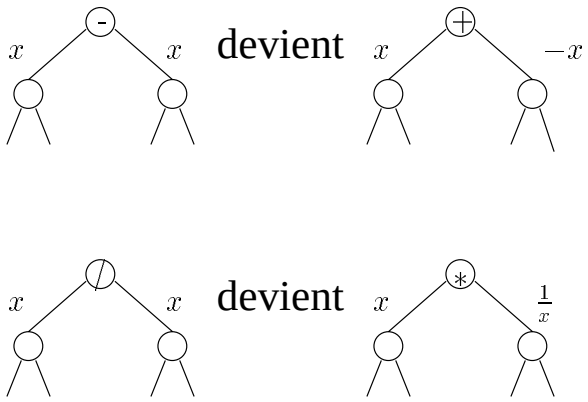


FIG. 2.5 - *Suppression des opérations non commutatives.*

À l'avenir, nous supposerons donc que les seuls nœuds de l'arbre sont des nœuds + et des nœuds $*$, alors que les fonctions d'arcs sont des fractions rationnelles de numérateur et de dénominateur affines.

Remarque
Si l'expression initiale est à évaluer dans un semi-anneau commutatif unitaire intègre ou dans un treillis, il n'y aura ni soustraction ni division. Les fonctions d'arcs seront donc des fonctions affines de la forme $a * x + b$. Dans un anneau commutatif unitaire intègre, il y aura des soustractions, mais les fonctions d'arcs seront également affines. Si la structure de base est un monoïde commutatif ou un groupe, avec une opération +, les fonctions d'arcs seront de la forme $x + a$.

2.2.3 L'opération de *ratissage*

Soient w un nœud interne de l'arbre et v un de ses fils, par exemple son fils gauche. Quand l'un des fils de v connaît sa valeur, w ne dépend (pour ce qui est de son opérande gauche) que de la valeur de l'autre fils de v, valeur supposée non encore connue. v et son fils évalué ne sont plus d'aucune utilité et ils peuvent être supprimés de l'arbre. Le *ratissage* effectue cette opération, en mettant à jour les fonctions d'arcs pour préserver la valeur de w : $f'(x) = f(g(\alpha) \diamond h(x))$, avec les notations de la figure 2.6. On vérifie aisément que la nouvelle fonction d'arc $f'(x)$ est encore de la forme $\frac{ax+b}{cx+d}$. En effet, l'ensemble des fonctions d'arcs est stable par composition, par addition d'un élément du corps et par multiplication par un élément du corps. Voici ce que donne cette opération dans le cas du *ratissage* d'un

nœud $*$:

$$f'(x) = f(g(\alpha) * h(x)) = \frac{ax+b}{cx+d},$$

$$f'(x) = \frac{a_f(\frac{a_g\alpha+b_g}{c_g\alpha+d_g} * \frac{a_h x+b_h}{c_h x+d_h})+b_f}{c_f(\frac{a_g\alpha+b_g}{c_g\alpha+d_g} * \frac{a_h x+b_h}{c_h x+d_h})+d_f},$$

$$f'(x) = \frac{a_f(a_g\alpha+b_g)(a_h x+b_h)+b_f(c_g\alpha+d_g)(c_h x+d_h)}{c_f(a_g\alpha+b_g)(a_h x+b_h)+d_f(c_g\alpha+d_g)(c_h x+d_h)},$$

$$\text{avec} \begin{cases} a &= a_f(a_g\alpha a_h + b_g a_h) + b_f(c_g\alpha c_h + d_g c_h), \\ b &= a_f(a_g\alpha b_h + b_g b_h) + b_f(c_g\alpha d_h + d_g d_h), \\ c &= c_f(a_g\alpha a_h + b_g a_h) + d_f(c_g\alpha c_h + d_g c_h), \\ d &= c_f(a_g\alpha b_h + b_g b_h) + d_f(c_g\alpha d_h + d_g d_h). \end{cases}$$

De façon algorithmique générale, l'opération de *ratissage* d'un nœud v par son père w s'écrit :

Algorithme 4 *Ratissage, première version*

en parallèle
si v, fils gauche, a un fils qui est une feuille, alors
calculer la nouvelle fonction de w vers l'autre fils de v,
déconnecter w de v
connecter w au fils de v qui n'est pas une feuille

si v, fils droit, a un fils qui est une feuille, alors
idem

fin

Cet algorithme présente cependant un inconvénient : il transforme implicitement l'arbre initial en un graphe orienté sans circuit, où les nœuds ont un degré entrant quelconque (figure 2.6) et donc l'objet manipulé n'est plus un arbre. Comme le nœud v ainsi que son fils feuille ne sont plus utiles, un nœud ratissé et sa feuille sont détruits.

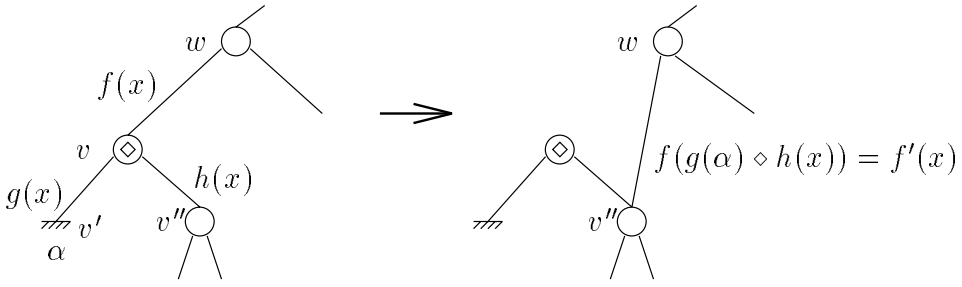


FIG. 2.6 - *Finalement v a un degré entrant nul et v'' a un degré entrant égal à 2.*

On remarque que si *ratissage* est effectué simultanément sur tous les nœuds, il y aura des conflits d'accès mémoire ; par exemple, si un nœud v a ses deux fils α et β qui sont des feuilles, son père w essayera de se connecter simultanément à β en ratissant v grâce à α et à α en ratissant v grâce à β (figure 2.7).

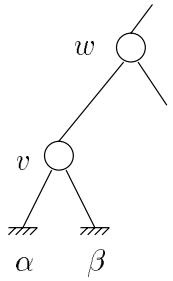


FIG. 2.7 - *Comment ratisser v ?*

Ce problème est résolu en supprimant la ligne « en parallèle » dans *ratissage*, donc en traitant séquentiellement les fils gauches puis les fils droits. On suppose que l'algorithme est synchrone. Toutes les sources de conflit (accès concurrents à des ressources communes) ne sont pas encore éliminées car si un nœud v a un fils feuille et un fils v'' qui a un fils feuille, l'arc $v--v''$ sera utilisé pour deux *ratissages* (figure 2.8). Il sera utilisé par w, le père de v, (pour le *ratissage* de son fils v), en même temps qu'il sera détruit par v lors du *ratissage* de son fils v''.

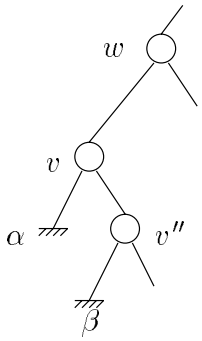


FIG. 2.8 - *Ratissage simultané de v et v''.*

De façon à remédier à ces problèmes, les feuilles de l'arbre sont numérotées consécutivement de gauche à droite avant toute opération de *ratissage* : lors du *ratissage*, seuls les pères des feuilles de numéro impair sont ratissés. Une telle numérotation des feuilles s'obtient en fait facilement à partir d'une opération d'indiçage de liste. Dans un premier temps, on construit un circuit eulérien de l'arbre, qui permet de parcourir tous les nœuds et de rencontrer les feuilles de gauche à droite. On affecte alors un poids entier à chaque nœud, qui vaut 1 pour les feuilles et 0 pour les nœuds internes, la numérotation des feuilles désirée est obtenue par

un calcul d'indiçage de liste en utilisant ces poids (cf. section 2.6.2 pour une description plus détaillée). Après cela, il n'est pas très difficile de numéroter toutes les feuilles, sauf la première et la dernière. On peut les ranger dans un tableau en temps constant, ce qui permet d'y accéder en temps constant lors du *ratissage* et lors de la première phase de l'algorithme optimal de la section 2.4.

Pour renuméroter les feuilles en temps constant après chaque *ratissage*, il suffit de remarquer que toutes les feuilles de numéro impair ont été supprimées, alors que toutes les feuilles de numéro pair sont conservées, puisque l'opération de *ratissage* détruit seulement la feuille de numéro impair et son père. Une division par 2 du numéro des feuilles restantes permet donc de renuméroter les feuilles.

Ces modifications de l'algorithme de *ratissage* permettent d'éviter tout conflit d'accès mémoire. Pour le démontrer, on identifie l'ensemble des objets (nœuds et arcs) affectés par le *ratissage* d'une feuille et on vérifie que les *ratissages* simultanés des pères de deux feuilles différentes utilisent deux ensembles disjoints. Il n'y a désormais plus de conflit d'accès mémoire, si on considère que l'accès concurrent à deux arcs incidents à un même nœud est possible. Pour les besoins de la démonstration, on définit *impliqué*(v) l'ensemble (hétéroclite) des nœuds et des arcs impliqués par le *ratissage* de v.

$$\text{impliqué}(v) = \left\{ \begin{array}{l} v, \\ \text{fils feuille de } v, \\ \text{arc(père de } v - -v), \\ \text{arc}(v - - \text{fils feuille de } v), \\ \text{arc}(v - - \text{ autre fils de } v) \end{array} \right\}.$$

On vérifie qu'il n'y a pas de problème lors du *ratissage* des pères w_1 et w_2 de deux feuilles gauches (resp. droites) v_1 et v_2 de numéro impair : puisque l'arbre est *binaire* et que v_1 et v_2 sont des feuilles gauches distinctes, leurs pères w_1 et w_2 sont distincts et non adjacents. Si w_1 et w_2 étaient adjacents – ce qui poserait problème, cf. figure 2.8 –, par exemple si w_1 était le père de w_2, on aurait la situation schématisée figure 2.9.

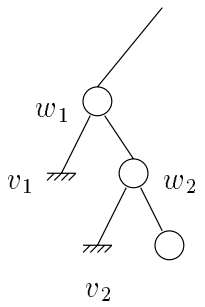


FIG. 2.9 - *v_1 et v_2 sont des feuilles consécutives.*

Or les feuilles sont numérotées consécutivement de gauche à droite, donc il est impossible que v_1 et v_2 aient des numéros consécutifs impairs. On a donc *impliqué*$(v_1) \bigcap$ *impliqué*$(v_2) = \emptyset$.

Même si w_1 et w_2 sont frères, leur *ratissage* modifiera des arcs distincts issus de leur père commun, ce qui ne pose pas problème.

Comme cela a déjà été mentionné, le *ratissage* des pères des feuilles gauches de numéro impair ne détruit que ces feuilles. On peut appliquer le même argument sur les feuilles droites de numéro impair pour finir de démontrer que l'opération de *ratissage* peut s'effectuer sur une $EREW-PRAM$. La durée d'une opération de *ratissage* est le temps nécessaire pour additionner une valeur à une fonction d'arc, ou pour la multiplier par une valeur, puis pour composer deux fonctions d'arcs ; cela se réalise en temps constant, il suffit pour s'en convaincre d'examiner les formules de composition de fractions rationnelles de numérateur et de dénominateur affines.

2.2.4 Algorithme de contraction

La procédure fondamentale de l'algorithme d'évaluation parallèle des expressions est maintenant définie de façon satisfaisante. La seule inconnue de cet algorithme est désormais le nombre d'étapes de *ratissage* à effectuer afin d'évaluer la racine de l'arbre. Le décompte du nombre de feuilles de l'arbre à chaque étape permet de répondre. Puisque le nombre de nœuds d'un arbre est $2f-1$ si le nombre de feuilles est f (propriété 1, §2.1.2), la décroissance du nombre de feuilles correspond directement à la décroissance du nombre total de nœuds de l'arbre. On a vu que l'opération de *ratissage* supprimait toutes les feuilles de numéro impair et aucune feuille de numéro pair. On en déduit que le nombre de feuilles numérotées est divisé par 2 par une application de *ratissage* sur l'arbre. Pour réduire l'arbre à la racine et ses deux fils feuilles, $\mathcal{O}(\log n)$ passages dans la boucle sont donc suffisants. Il ne reste plus qu'à évaluer la racine. Comme le temps d'un *ratissage* est constant, on en déduit que l'algorithme d'évaluation d'arbres a une complexité $EREW(n, \log n)$.

Le principe de Brent avec un nombre de processeurs $\frac{n}{\log n}$ nous conduit à une complexité (non log-uniforme) optimale $EREW(\frac{n}{\log n}, \log n)$. En effet, la complexité séquentielle est minorée par n : il faut bien lire chaque opérande, et chaque opérateur pour calculer sa valeur ! Elle vaut d'ailleurs n. Nous verrons à la section 2.4 un algorithme effectif et uniforme permettant d'atteindre cette complexité parallèle optimale.

Le théorème suivant récapitule ces résultats.

Théorème 7 *Toute expression arithmétique sur un corps commutatif F peut être évaluée en temps $\mathcal{O}(\log n)$ avec $\frac{n}{\log n}$ processeurs.*

Corollaire 1 *Toute expression arithmétique sur un monoïde commutatif, un groupe, un semi-anneau ou un anneau commutatif unitaire intègre peut être évaluée en temps $\mathcal{O}(\log n)$ avec $\frac{n}{\log n}$ processeurs.*

DÉMONSTRATION
La remarque 2.2.2 signale que dans le cas d'un semi-anneau ou d'un anneau, les fonctions d'arcs sont des fonctions affines. Dans le cas d'un groupe ou d'un monoïde, les fonctions d'arcs sont de la forme $x + a$. □

Voici maintenant l'algorithme de contraction d'expressions [KD88], illustré par un exemple.

Algorithme 5 *Contraction d'arbre*

faire $\lceil \log n \rceil$ fois
    ratisser les pères de toutes les feuilles gauches de numéro impair
    ratisser les pères de toutes les feuilles droites de numéro impair
    diviser le numéro des feuilles restantes par 2
Évaluer la racine.

fin

Remarque

- Chaque étape de *ratissage* s'effectue en temps constant, qui est 26 fois le temps d'une opération[3] (supposée en temps unitaire) dans le corps de base. Le temps exact de l'algorithme est donc 26 fois le temps d'une évaluation séquentielle de l'expression. Pour de petits exemples, le temps parallèle, qui est $26 \log n + o(\log n)$, peut être supérieur au temps d'une évaluation gloutonne. Cependant, dès que l'on a des exemples assez importants à traiter, $26 \log n$ est très inférieur à n (la valeur de n pour laquelle il y a égalité se situe aux alentours de 200).

- Si cet algorithme est utilisé comme compilateur (cf. chapitre 5) pour transformer un arbre quelconque en arbre équilibré, le programme parallèle obtenu est SIMD : il est synchrone et chaque tâche consiste en une composition de fonctions d'arcs.

2.3 Un exemple

L'expression $2*2+(3+2)*(5*3*2+4*(0*7+(4+3)*3))$, ou plutôt $((2*2)+((3+2)*(((5*3)*6)+(4*((0*7)+((4+3)*3))))))$ puisque les expressions sont complètement parenthésées, est représentée par l'arbre de la figure 2.10 qui comporte 12 nœuds. Les feuilles sont numérotées consécutivement de gauche à droite, sauf la première et la dernière.

3. Cela peut être vérifié sur la formule donnée pour le *ratissage* d'un nœud $*$; la formule pour le *ratissage* d'un nœud $+$ est similaire.

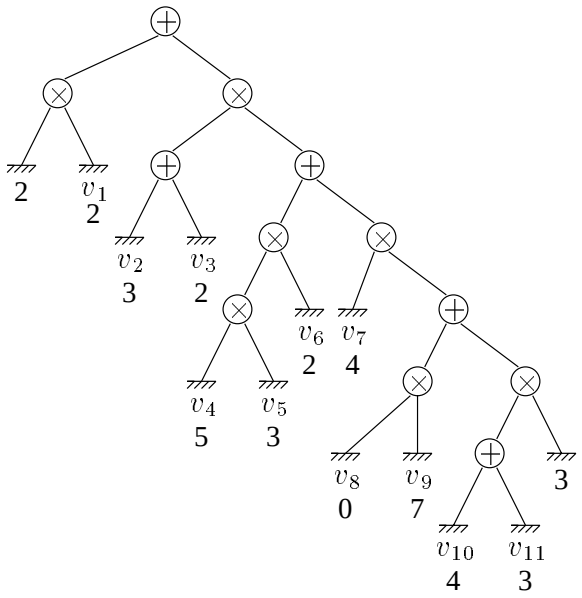


FIG. 2.10 - *Exemple.*

Étape 1 : le père de la seule feuille gauche de numéro impair, la feuille numéro 7 (dont le grand-père est grisé) est ratissé : figure 2.11. La feuille 7 a été supprimée et on note l'apparition de la fonction $4x$ sur le nouvel arc entre son grand-père et son frère. Les pères des feuilles droites de numéro impair, *i.e.* les feuilles numéro 1, 3, 5, 9 et 11 (dont les grand-pères sont grisés), sont alors ratissés : figure 2.12. Après cette étape de *ratissage*, il ne reste que des feuilles de numéro pair. La renumérotation des feuilles est obtenue en divisant leur numéro par 2 : figure 2.13.

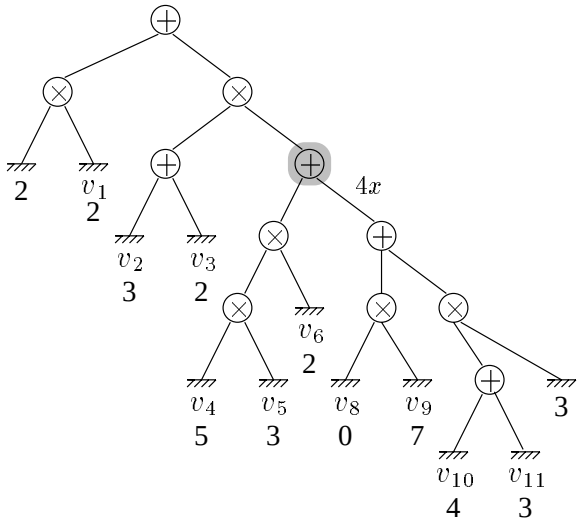


FIG. 2.11 - *Étape 1 : ratissage des pères des feuilles gauches de numéro impair.*

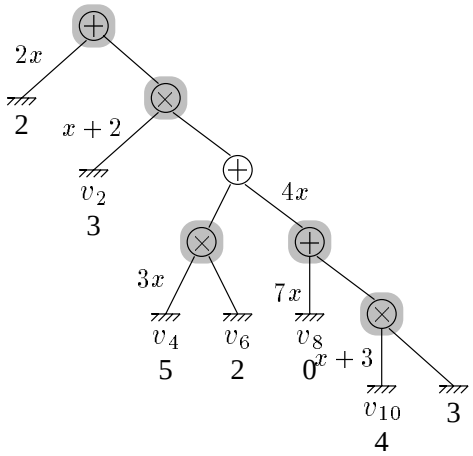


FIG. 2.12 - *Étape 1 : ratissage des pères des feuilles droites de numéro impair.*

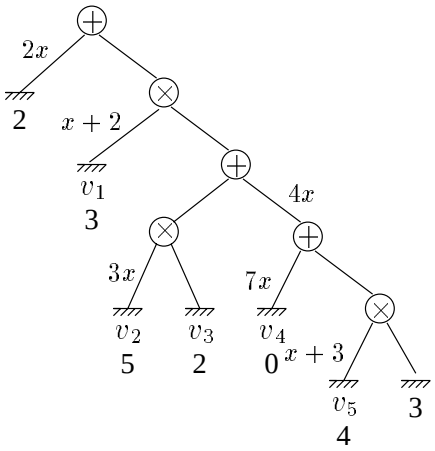


FIG. 2.13 - *Étape 1 : renumérotation.*

Étape 2 : après la renumérotation des feuilles, une nouvelle étape de *ratissage* commence. Les pères des feuilles gauches de numéro impair, les feuilles 1 et 5, sont ratissés : figure 2.14. Il ne reste qu'une seule feuille droite de numéro impair, la feuille 3, dont le père est alors ratissé : figure 2.15. À la fin de cette nouvelle phase de *ratissage*, les feuilles sont renumérotées : figure 2.16.

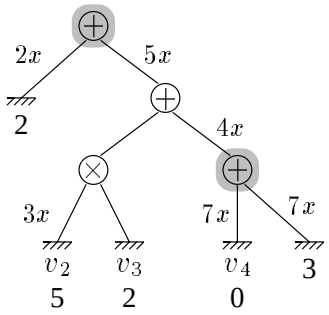


FIG. 2.14 - *Étape 2 : ratissage des pères des feuilles gauches de numéro impair.*

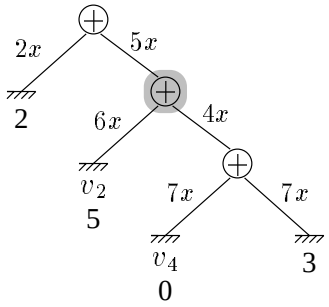


FIG. 2.15 - *Étape 2 : ratissage des pères des feuilles droites de numéro impair.*

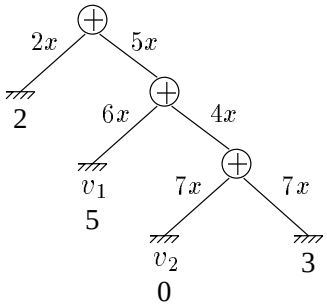


FIG. 2.16 - *Étape 2 : renumérotation.*

Étape 3 : le père de la feuille gauche de numéro impair numéro 1 est ratissé : figure 2.17. Comme il n'y a pas de feuilles droites de numéro impair, le *ratissage* se termine par une renumérotation : figure 2.18.

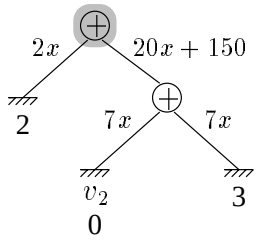


FIG. 2.17 - *Étape 3: ratissage des pères des feuilles gauches de numéro impair.*

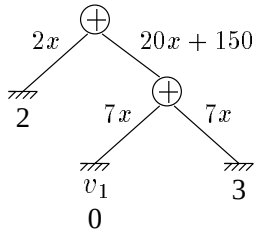


FIG. 2.18 - *Étape 3: pas de feuilles droites de numéro impair, renumérotation.*

Étape 4 : la dernière étape de *ratissage* se résume à ratisser le père de la dernière feuille numérotée de l'arbre : figure 2.19.

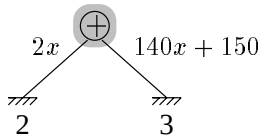


FIG. 2.19 - *Étape 4: ratissage des pères des feuilles gauches de numéro impair.*

Étape 5 : il ne reste plus que la racine et les deux feuilles extrêmes, on évalue la racine : figure 2.20.

FIG. 2.20 - *5 étapes ont suffi pour évaluer l'arbre.*

Cette présentation de l'algorithme est inspirée de Kosaraju et Delcher [KD88]. Elle ressemble à celle d'Abrahamson, Dadoun, Kirkpatrick et Przytycka [ADKP89]. Cet algorithme

a également été proposé par Gibbons et Rytter en 1988, qui expliquent comment utiliser effectivement $\frac{n}{\log n}$ processeurs (cf. section 2.4). Chez Miller et Reif [MR85] se trouve une version un peu différente du premier algorithme, où l'opération de *ratissage* est décomposée en deux temps, le premier où l'on détache la feuille de l'arbre, et le second où son père, devenu unaire, se fait « compresser ». D'autres précurseurs, Cole et Vishkin [CV88], ont proposé essentiellement le même algorithme que Miller et Reif, mais en le noyant sous des définitions et des découpages algorithmiques complexes (travail par niveaux centroïdes, application d'une décomposition centroïde accélérée) et apparemment devenus inutiles depuis, puisque leurs résultats sont les mêmes. Nous ne détaillerons pas cette version de l'algorithme.

2.4 Variantes

Nous allons maintenant présenter quelques variantes de cet algorithme, qui permettent de calculer dans des structures algébriques différentes, comme des structures non commutatives ou des ensembles finis, ou qui évaluent toutes les sous-expressions avec la même complexité. Enfin, nous présenterons un algorithme optimal effectif, qui ne nécessite que $\frac{n}{\log n}$ processeurs pour évaluer une expression en temps $\log n$.

2.4.1 Calculs dans des structures non commutatives

Dans le cas des expressions sur un corps, nous avons vu que l'évaluation était basée sur un mécanisme de composition des fonctions d'arcs, qui constituent une représentation fine des dépendances algébriques entre les nœuds.

Si les calculs s'effectuent dans une structure où la multiplication n'est pas commutative, mais où elle reste associative et distributive par rapport à l'addition et où elle conserve un élément neutre, il reste possible d'évaluer une expression avec la même complexité que précédemment. Pour cela, il faut tenir compte du fait qu'une feuille est fils droit ou fils gauche lors du *ratissage* d'un nœud $*$, c'est-à-dire que pour un fils gauche (resp. droit), la composition de fonctions d'arcs utilisera la valeur de cette feuille comme opérande gauche (resp. droit). De plus, il apparaît que les fonctions d'arcs doivent respecter la non-commutativité de la multiplication. La forme générale qu'elles prennent est $f(x) = \frac{a*x*b+c}{d*x*e+f}$, avec une constante multiplicative à gauche de x et une à droite. L'ensemble de ces fonctions d'arcs est stable par composition et par addition et multiplication, aussi bien à gauche qu'à droite, par une valeur.

Si maintenant les hypothèses de commutativité à la fois de la multiplication et de l'addition sont levées, c'est-à-dire si on ne conserve que les propriétés d'associativité, celle de distributivité de la multiplication par rapport à l'addition, ainsi que l'existence de neutre pour chacune de ces opérations, toute expression arithmétique sur un monoïde, un groupe, un semi-anneau ou un anneau unitaire intègre ou un corps peut encore être évaluée en temps $\mathcal{O}(\log n)$, avec $\frac{n}{\log n}$ processeurs. Les fonctions d'arcs sont de la forme $f(x) = \frac{a+b*x*c+d}{e+f*x*g+h}$ pour un corps, ou $a + b * x * c + d$ pour un semi-anneau ou un anneau et le *ratissage* tient compte du fait que la feuille utilisée est un fils droit ou un fils gauche.

2.4.2 Calculs dans un ensemble fini

Supposons maintenant que les calculs aient lieu dans un ensemble fini [GR86, ADKP89]. Dans un tel ensemble, les opérations seront remplacées par des fonctions binaires quelconques et les fonctions sur un ensemble fini (quelle que soit leur arité) sont représentables par une table de valeurs de taille finie. Par exemple, une fonction unaire : $F \to F$ (comme nos fonctions d'arcs précédentes) peut être représentée par une table qui consiste en une seule ligne (figure 2.21) et une fonction binaire : $F \times F \to F$ peut être représentée par une table bidimensionnelle (figure 2.22).

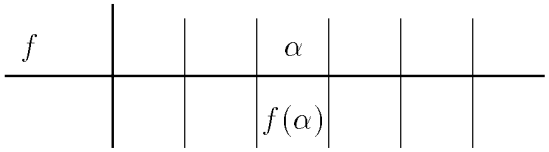


FIG. 2.21 - *Table d'une fonction unaire.*

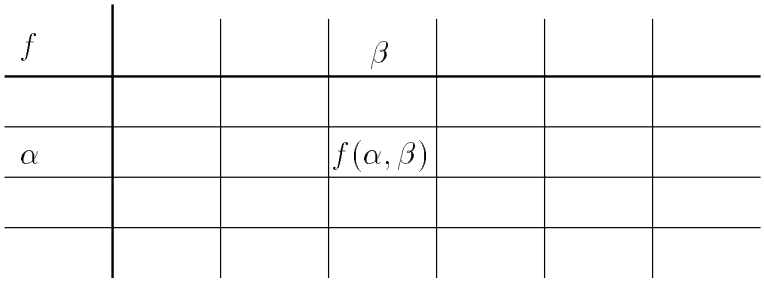


FIG. 2.22 - *Table d'une fonction binaire.*

Le mécanisme d'anticipation de l'algorithme de contraction de la section 2.2 peut s'appliquer, si on utilise encore des fonctions binaires en guise d'opérateurs et des fonctions unaires comme fonctions d'arcs. En effet, ce qui importe dans ce mécanisme est d'une part que les fonctions d'arcs soient simples (comme les fonctions unaires) et d'autre part, que l'on sache déterminer facilement le résultat de la composition d'une fonction binaire par une fonction constante et une fonction unaire. Or, dans un ensemble fini, ces compositions peuvent être effectuées en temps constant à l'aide des tables de valeurs de ces fonctions.

Supposons que l'on ait une expression, dont les opérateurs sont des fonctions binaires, représentées par leur table de valeurs qui est de taille finie puisque l'ensemble est lui-même de cardinal fini. Les fonctions d'arcs seront encore des fonctions unaires. Dans ce cas, on stocke sur chaque arc la table de sa fonction d'arc.

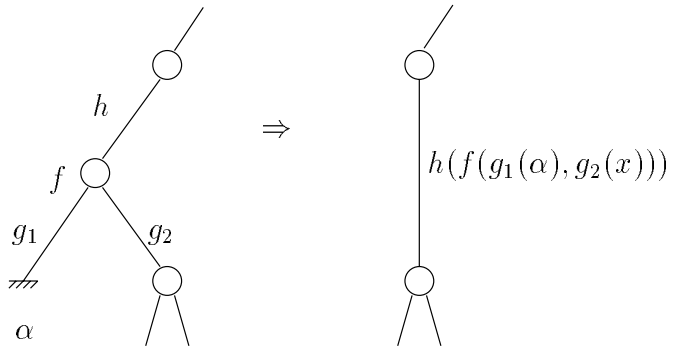


FIG. 2.23 - *Ratissage dans un ensemble fini.*

Par exemple, dans le schéma 2.23, h, g_1, g_2 sont unaires, f est binaire et on conserve pour $f(g_1(\alpha), g_2(x))$ la ligne du tableau bidimensionnel de f représentée figure 2.24.

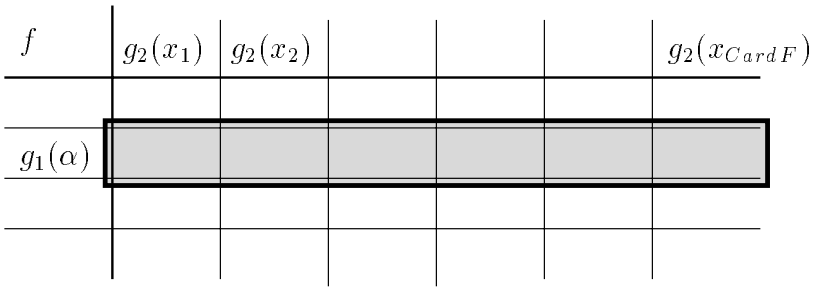


FIG. 2.24 - *la ligne correspondante de la table de f.*

On peut même généraliser ceci au cas où l'ensemble des fonctions est indicé et où une fonction peut être évaluée en temps constant, les compositions du type $h(f(g_1(\alpha), g_2(x)))$ pouvant être remplacées par des calculs d'indices effectués en temps également constant [ADKP89].

2.4.3 Évaluation de toutes les sous-expressions

On peut avoir besoin du résultat de toutes les opérations effectuées dans une expression (*i.e.* résultats partiels) et pas seulement de la valeur de la racine. C'est ce qui se produit pour les calculs de type préfixe. L'arbre correspondant ne fournit que le résultat du problème de produit itéré associé (figure 2.1).

L'opération de *ratissage* de la section 2.2 supprime des nœuds de l'arbre pour préserver la structure arborescente de l'expression. Pour pouvoir calculer tous les résultats partiels, il est impératif de ne pas supprimer ces nœuds, mais simplement de les déconnecter de leur père et de laisser intacts leurs arcs pondérés vers leurs fils. Chaque nœud ratissé mémorise

de plus un pointeur vers son père, ainsi que la fonction d'arc au moment du *ratissage*. Cela permet de calculer le résultat de chaque nœud, en rétablissant les pointeurs d'un nœud vers ses fils et son père dès que les fils sont évalués. Une phase de « descente », symétrique de la phase de contraction de la racine, réalise cela. Cette phase s'appelle phase d'**expansion**.

Plus précisément, on traite les nœuds dans l'ordre inverse de la remontée : si un nœud a été ratissé à l'étape i, sa valeur pourra être calculée à l'étape $(\log n) - i$ de la procédure d'expansion et il rétablira les pointeurs vers ses fils et son père pour se retrouver dans le même état qu'avant son *ratissage*. On peut utiliser la même astuce de renumérotation des feuilles (à l'envers) pour écrire l'algorithme d'expansion. Un raisonnement simple, par induction sur le numéro i de l'étape, permet de montrer qu'un nœud ratissé à l'étape i de la contraction aura ses deux fils évalués au début de l'étape $(\log n) - i$ de l'expansion et qu'il pourra alors calculer sa valeur.

Comme cette procédure est absolument symétrique de la procédure de contraction, sa complexité est la même.

Théorème 8 *Le calcul de tous les résultats partiels d'une expression a pour complexité* $EREW(\frac{n}{\log n}, \log n)$.

2.4.4 Algorithme optimal d'évaluation d'expressions

Le principe de Brent permet de calculer le temps parallèle d'exécution d'un algorithme prévu pour p processeurs, quand on ne dispose que de q processeurs, avec $q \leq p$.

Principe de Brent [Bre73, CT93] (Rappel)
Tout algorithme parallèle qui s'exécute en π phases, où chaque phase i nécessite p_i processeurs ($p = \max(p_i)$) et s'exécute en t_i unités de temps, peut être implanté sur $q \leq p$ processeurs et s'exécuter en

$$\sum_{i=1}^{\pi} \lceil \frac{t_i p_i}{q} \rceil + \pi \text{ unités de temps.}$$

Ici, $t_i = 1$, $p_i = \frac{n}{2^i}, \pi = \log n$. On suppose que n est à la fois une puissance de 2 et divisible par $\log n$ (sinon il faut surcharger les formules avec des parties entières supérieures). Si on prend $q = \frac{n}{\log n}$, on obtient un temps d'exécution de

$$\sum_{i=1}^{\log n} \frac{\frac{n}{2^i}}{\frac{n}{\log n}} + \log n = \log n \sum_{i=1}^{\log n} \frac{1}{2^i} + \log n \leq 3 \log n.$$

Gibbons et Rytter ont proposé l'algorithme suivant pour $p \leq n$ processeurs, qui est une version du principe de Brent, effective et uniforme (l'équilibrage des travaux s'effectue par des calculs d'indices et non pas en écrivant toutes les tâches de l'étape i, puis en les « repliant » sur les processeurs disponibles). On peut remarquer qu'avant l'étape $i+1$, le nombre de feuilles est $\frac{n}{2^i}$. Ils partitionnent ces feuilles en p segments de longueur $\lceil \frac{n}{2^i p} \rceil$, en temps constant puisque les feuilles sont numérotées, le dernier processeur a $\frac{n}{2^i} - (p-1)\lceil \frac{n}{2^i p} \rceil$ feuilles. Ce découpage s'effectue en temps constant puisque les feuilles sont rangées dans un tableau (cf. section 2.2.4), de façon uniforme. Chaque processeur effectue alors séquentiellement le

ratissage correspondant aux feuilles qu'il possède, donc en temps $\lceil \frac{n}{2^i p} \rceil$. Quand il reste moins de $2p$ feuilles, on applique l'algorithme 2.2.4.

Si $p = \frac{n}{\log n}$ et si on note $\lceil \log \frac{n}{p} \rceil = k$, on effectue k étapes modifiées, dont le temps d'exécution cumulé est

$$\sum_{i=0}^{k} \lceil \frac{n}{p} \rceil \frac{1}{2^i} = \lceil \frac{n}{p} \rceil (1 + \frac{1}{2} + \ldots + \frac{1}{2^k}) = \mathcal{O}(\frac{n}{p}) = \mathcal{O}(\log n),$$

puis l'algorithme 2.2.4 avec $2p$ feuilles, en temps $\mathcal{O}(\log p) = \mathcal{O}(\log n)$. L'algorithme restant applicable sur une $EREW-PRAM$, sa complexité est donc

$$EREW(\frac{n}{\log n}, \log n).$$

Il s'agit d'un algorithme optimal.

Toutes les variantes de l'algorithme autorisent la même réduction du nombre de processeurs et sont donc rendues optimales de la même manière.

2.4.5 Conclusion

On a vu qu'il est possible d'évaluer une expression à n opérations et opérandes sur un semi-anneau, un anneau, un corps ou une structure algébrique quelconque de cardinal fini, ainsi que toutes ses sous-expressions, en parallèle avec une complexité optimale $EREW(\frac{n}{\log n}, \log n)$.

2.5 Application : langages hors-contexte parenthésés

Les applications de cet algorithme sont variées : il permet de déterminer différentes caractéristiques des arbres, telles que l'isomorphisme de deux arbres (ce qui permet ensuite de repérer et d'éliminer des sous-expressions communes dans du code, en vue de son optimisation) ou de résoudre le problème associé de l'étiquetage canonique d'arbres, de façon non déterministe [GR88b].

On peut également travailler sur des co-graphes, donnés par leur arbre syntaxique, pour calculer leur nombre chromatique, qui est également la taille de leur plus grande clique, et déterminer une clique de taille maximale [ADKP89]. Un co-graphe est un graphe défini inductivement : un graphe composé d'un seul sommet est un co-graphe, l'union de deux co-graphes est un co-graphe, le complément d'un co-graphe est un co-graphe. L'arbre syntaxique d'un co-graphe a pour feuilles les sommets du graphe et pour nœuds internes des nœuds union ou complément-union. Une clique est un sous-graphe complet.

Une autre application est la reconnaissance de langages hors-contexte parenthésés. L'algorithme d'évaluation correspond à celui de Cocke, Younger et Kasami, rendu déterministe : en effet, le parenthésage indique quel sera l'unique arbre syntaxique possible.

Algorithme de Cocke-Younger-Kasami [AU72]
On suppose que l'on a une grammaire, sous forme normale de Chomsky, et une chaîne w à

reconnaître. Une grammaire est un quadruplet $G = (N, T, R, S)$. T est l'alphabet du langage, ou ensemble des symboles terminaux. C'est l'ensemble des lettres qui composent les mots du langage. Les terminaux sont notés par des minuscules. N est un ensemble de symboles dits non-terminaux, ce qui signifie qu'ils n'apparaissent pas dans les mots du langage mais qu'ils servent seulement à écrire les règles de la grammaire. Les éléments de N sont représentés par des majuscules. R est l'ensemble des règles de la grammaire. Pour une grammaire sous forme normale de Chomsky, les règles sont de la forme $A \to BC$, avec A, B, C non-terminaux, ou $A \to a$ avec a terminal. Enfin S, l'axiome de la grammaire, est un élément distingué de N : c'est le non-terminal d'où débutent toutes les dérivations, *i.e.* pour tout mot w du langage, $S \to^+ w$.

L'algorithme de Cocke-Younger-Kasami effectue une analyse ascendante : on part de la chaîne à reconnaître w et on essaie de remonter jusqu'à S. Si w est de longueur n, l'algorithme nécessite n étapes séquentielles ; à l'étape j – de durée $\mathcal{O}(n-j)$ – on construit les ensembles de non-terminaux qui permettent de dériver chaque sous-chaîne de longueur j de la chaîne initiale w. La première étape remplace chaque terminal a par l'ensemble des non-terminaux A qui permettent de dériver a, *i.e.* tels que $A \to a \in R$. À l'étape j, on essaie de combiner deux à deux tous les non-terminaux qui ont permis de dériver une sous-chaîne de longueur k avec ceux qui permettent de dériver la sous-chaîne de longueur $j-k$ qui la suit dans w. À la $n^{\text{ème}}$ étape, il reste un seul ensemble inclus dans N, puisque w n'admet qu'une seule sous-chaîne de longueur n, elle-même. Si S appartient à cet ensemble, alors il existe une dérivation $S \to^+ w$, *i.e.* w appartient au langage.

Algorithme 6 *Cocke-Younger-Kasami*

Entrée : $G = (N, T, R, S)$
$w = a_1 a_2 \dots a_n \in T^+$
Sortie : τ un tableau $n \times n$ à éléments dans $\mathcal{P}(N)$ (l'ensemble des parties de N), tel que $\tau_{i,j} \ni A$ si et seulement si $A \to^+ a_i \dots a_{i+j-1}$, $1 \le j \le n$, $1 \le i \le n-j+1$
{ La chaîne w sera reconnue si $S \in \tau_{1,n}$ }
Algorithme :

$\tau_{i,1} = \{A / A \to a_i \in R\}$

pour $j = 2$ à n faire
{ On a calculé $\tau_{i,j'}$ avec $1 \le j' < j$, $1 \le i \le n - j' + 1$ }

pour $i = 1$ à $n - j + 1$ faire
$\tau_{i,j} = \{A / \exists k, 1 \le k < j,\ A \to BC \in R,\ B \in \tau_{i,k},\ C \in \tau_{i+k,j-k}\}$

ce que nous noterons
$\tau_{i,j} = \bigcup_{k=1}^{j-1} \tau_{i,k} * \tau_{i+k,j-k}$
avec, si $S_1 \subset N \setminus T$ et $S_2 \subset N \setminus T$
$S_1 * S_2 = \{A/A \to BC \in R,\ B \in S_1,\ C \in S_2\}$.

fin

Si on le représente sous forme de graphe, on obtient un graphe orienté sans circuit ayant pour opérations $*$ et $\bigcup$ (section 3.7.2). Cela est dû au fait que la grammaire est ambiguë. Pour lever les ambiguïtés, on utilise des grammaires complètement parenthésées. C'est d'ailleurs ce type de grammaire que l'on a utilisé pour définir les expressions (§2.1.1). Un langage parenthésé est un langage pour lequel toutes les règles sont de la forme $A \to (BC)$ avec A, B, C des non terminaux, ou $A \to (a)$ avec A non terminal et a terminal. Si on effectue uniquement la première étape de l'algorithme de Cocke-Younger-Kasami, en remplaçant la concaténation de lettres par l'opération $*$, on obtient une expression dont les n opérandes sont des parties de N et où le seul opérateur est $*$. Le résultat est un calcul dans un ensemble fini $\mathcal{P}(N)$, avec la fonction binaire $*$ qui peut être tabulée, et la version de l'algorithme d'évaluation d'expressions donnée à la section 2.4 s'applique.

L'algorithme est le suivant :

- **Pré-traitement :** on effectue, en parallèle et en temps constant avec n processeurs, en temps $\mathcal{O}(\log n)$ avec $\frac{n}{\log n}$ processeurs, un pré-traitement qui consiste à remplacer chaque sous-chaîne « (a_i) » de $w = a_1 \ldots a_i \ldots a_n$ – où a_i est un terminal qui n'est pas une parenthèse – par « X_i » $= \{A/A \to (a_i) \in R\}$; puis, chaque fois que deux ensembles sont adjacents, *i.e.* on a « $X_i X_{i+1}$ » dans la chaîne ainsi obtenue, on les sépare par $*$: « $X_i * X_{i+1}$ ». De plus, on remplace chaque occurrence de « $(X_j($ » par « $(X_j * ($ » et chaque occurrence de « $)X_k)$ » par « $) * X_k)$ ».

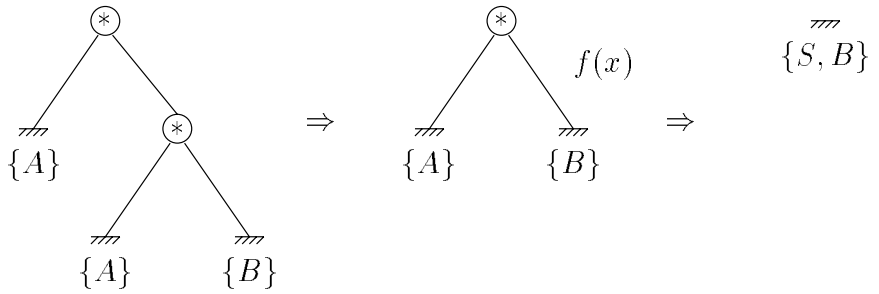
- **Évaluation parallèle :** on est alors ramené à l'évaluation d'une expression sur une structure de cardinal fini. Ce problème se résout en temps $\log n$ avec $\frac{n}{\log n}$ processeurs.

- **Conclusion :** on sait si la chaîne a été reconnue en testant, en temps constant, si S appartient au résultat de l'évaluation.

Exemple
La grammaire est $G = (\{S, A, B\}, \{a, b\}, R, S)$ avec

$$R = \left\{ \begin{array}{lcl} S & \leftarrow & (AB) \\ A & \leftarrow & (a) \\ B & \leftarrow & (AB)|(b) \end{array} \right\}.$$

La chaîne à reconnaître, $w = ((a)((a)(b)))$, est d'abord transformée en $(\{A\}(\{A\}\{B\}))$, puis en $(\{A\} * (\{A\} * \{B\}))$, avant d'être évaluée par l'algorithme 2.2.4 (figure 2.25).



avec x	$\emptyset$	$\{S\}$	$\{A\}$	$\{B\}$	$\{S,A\}$	$\{S,B\}$	$\{A,B\}$	$\{S,A,B\}$
$f(x)$	$\emptyset$	$\emptyset$	$\emptyset$	$\{S,B\}$	$\emptyset$	$\{S,B\}$	$\{S,B\}$	$\{S,B\}$

FIG. 2.25 - *Exemple.*

2.6 Compléments pratiques

Pour être (presque) complet sur ce problème, il reste à préciser comment construire la représentation arborescente d'une expression en parallèle, puisque c'est rarement sous cette forme qu'une expression est fournie. Bar-On et Vishkin [BOV85] ont proposé un algorithme effectuant cette transformation avec la même complexité que l'algorithme d'évaluation. Nous montrerons également comment numéroter les feuilles, à partir d'un circuit eulérien construit par l'algorithme de Tarjan et Vishkin [TV85]. Nous finirons par les résultats de parallélisation sur machine à mémoire distribuée.

2.6.1 Mise sous forme arborescente

On s'intéresse à une expression non complètement parenthésée sur un corps F, avec les précédences classiques sur les opérations. L'arbre correspondant à l'expression doit être tel que son évaluation directe fournisse le même résultat que l'expression initiale. Les problèmes liés à la construction d'un tel arbre ont deux origines : tout d'abord, tel quel, le problème est ambigu ; plusieurs arbres sont possibles, comme les arbres des figures 2.1 et 2.2 pour l'expression $v_1 + v_2 + \ldots + v_n$. Un parenthésage artificiel préliminaire lèvera cette ambiguïté, en ne conservant entre les mêmes parenthèses que des opérateurs de même précédence ($+$ et $-$, ou $*$ et $/$) ; une telle expression est appelée *expression simple*. Le second problème, également lié à l'ambiguïté de la grammaire, est de savoir comment traiter des conventions d'associativité adoptées pour les opérateurs non associatifs $-$ et $/$, à savoir $a - b - c = (a - b) - c$, *i.e.* l'associativité est à gauche. Il faut donc interdire une écriture du type $a - b - c = a - (b - c)$ (associativité à droite).

Dans un premier temps, l'algorithme de Bar-On et Vishkin [BOV85] effectue un parenthésage de l'expression en tenant compte des règles de précédence, mais pas des règles d'associativité. Ce parenthésage permet d'identifier des sous-expressions, qui correspondront à des sous-arbres, ces sous-arbres étant uniques, modulo les règles d'associativité choisies (associativité à gauche). L'appariement des parenthèses permet une description logique de l'arbre et également de supprimer les parenthèses inutiles créées par le parenthésage préliminaire. Enfin, l'arbre est construit, le problème essentiel par rapport à la description précédemment obtenue étant de prendre en compte les règles d'associativité à gauche. Cet algorithme permet de démontrer de façon constructive le théorème suivant :

Théorème 9 *Il est possible de construire l'arbre correspondant à une expression (donnée sous forme de tableau) en parallèle en temps* $\mathcal{O}(\log n)$ *avec* $\mathcal{O}(\frac{n}{\log n})$ *processeurs sur une* $CREW-PRAM$.

DÉMONSTRATION
L'expression est supposée rangée dans un tableau, chaque case du tableau contenant une parenthèse, une valeur ou une opération. Ce tableau est de longueur n. L'algorithme qui suit va tout d'abord (en temps constant avec n processeurs, ou en temps $\log n$ avec $\frac{n}{\log n}$ processeurs) parenthéser complètement l'expression, puis le cœur de l'algorithme consistera à établir les correspondances entre parenthèses ouvrantes et fermantes ; ensuite, il sera facile de construire (en temps $\mathcal{O}(\log n)$ avec $\frac{n}{\log n}$ processeurs) la représentation arborescente de l'expression donnée en entrée.

Le schéma de l'algorithme est le suivant :

Algorithme 7 *Construction parallèle de l'arbre*

Passage de l'expression initiale à une expression simple : parenthéser l'expression

Construction de l'arbre logique : apparier les parenthèses et supprimer les parenthèses inutiles

Construction de l'arbre, en prenant en compte les associativités

fin

Première étape : passage à une expression simple, par parenthésage

Pour assurer que $*$ et $/$ sont prioritaires par rapport à $+$ et $-$, on parenthèse l'expression de telle sorte que plusieurs opérateurs se trouvent entre les mêmes parenthèses si et seulement s'ils ont la même priorité.

On insère des parenthèses dans l'expression :

1. Pour chaque opération $+$ ou $-$, on insère deux parenthèses ouvrantes à sa droite et deux parenthèses fermantes à sa gauche : $a + b \rightarrow a)) + ((b$.

2. Pour chaque opération $*$ ou $/$, on insère une parenthèse ouvrante à sa droite et une parenthèse fermante à sa gauche : $g * h \rightarrow g) * (h$.

3. Pour chaque parenthèse ouvrante initiale, on insère deux parenthèses ouvrantes à sa droite et pour chaque parenthèse fermante initiale, on insère deux parenthèses fermantes à sa gauche. Ces parenthèses sont notées par des crochets dans l'exemple qui suit, pour plus de clarté.

4. On ajoute également deux parenthèses ouvrantes au début de la chaîne et deux parenthèses fermantes à la fin.

Le fait d'ajouter deux parenthèses autour de $+$ et $-$ et une seule autour de $*$ et $/$ permet d'assurer le respect des précédences. L'exemple qui suit permettra de s'en convaincre.

Exemple

$a + b + c * d/e - f - g * h * [i + j]$

$a)) + ((b)) + ((c * d/e)) - ((f)) - ((g * h * [i)) + ((j]$

$a)) + ((b)) + ((c) * (d)/(e)) - ((f)) - ((g) * (h) * ([i)) + ((j]$

$a)) + ((b)) + ((c) * (d)/(e)) - ((f)) - ((g) * (h) * ([((i)) + ((j))]$

$((a)) + ((b)) + ((c) * (d)/(e)) - ((f)) - ((g) * (h) * ([((i)) + ((j))]))$

FIG. 2.26 - *Exemple de parenthésage de l'expression* $a + b + c * d/e - f - g * h * [i + j]$.

On a décomposé ce parenthésage en quatre phases successives pour illustrer séparément chacune des actions énumérées ci-dessus, mais cela ne doit pas faire oublier qu'elles s'exécutent en parallèle dans l'algorithme.

Pour effectuer ce parenthésage, chaque élément du tableau initial est remplacé par un pointeur sur une liste contenant cet élément et les parenthèses éventuellement ajoutées. Chaque liste est de longueur au plus 5. La nouvelle expression est donc de longueur $\mathcal{O}(n)$. Il n'est pas nécessaire de ranger la nouvelle expression dans un tableau, tout élément sera repéré par la position de la liste qui le contient dans le tableau initial et par sa position dans cette liste. Cette première étape est bien réalisée en temps constant avec $\mathcal{O}(n)$ processeurs.

Deuxième étape : passage à un arbre logique, par appariement des parenthèses

On applique l'algorithme d'appariement des parenthèses, qui est assez long à décrire et qui, pour cette raison, se trouve placé en exergue après l'algorithme proprement dit de construction de l'arbre. Pour cela, on découpe le tableau initial en $\frac{n}{\log n}$ segments de longueur $\log n$ et on applique l'algorithme d'appariement des parenthèses. Chaque processeur traite $\log n$ éléments de l'expression initiale, soit au plus $5 \log n$ éléments de la nouvelle expression. Il parcourt son segment – en parcourant les listes chaînées qui remplacent ses éléments – et effectue l'algorithme d'appariement. À la fin de cet algorithme, une parenthèse ouvrante (resp. fermante) connaît la position de la parenthèse fermante (resp. ouvrante) qui lui correspond par sa position dans le tableau initial et sa position dans la liste située à cette position du tableau.
La complexité de l'algorithme d'appariement des parenthèses est $CREW(\frac{n}{\log n}, \log n)$ si l'expression a n éléments. Comme ici elle en a au plus $5n$, parenthèses comprises, la complexité de cette étape est $CREW(\frac{n}{\log n}, \log n)$.

Remarque
Suppression de parenthèses inutiles
Cette étape est facultative. Elle permet de supprimer des parenthèses inutiles, de remplacer $((a+b))$ par $(a+b)$ et $(a)+b$ par $a+b$ par exemple. Une parenthèse fermante se détruit (ou se remplace par un caractère blanc ou nul) s'il y a une autre parenthèse fermante à sa gauche et que leurs parenthèses ouvrantes correspondantes sont voisines aussi. Une parenthèse ouvrante se détruit selon la même règle. Une paire de parenthèses se détruit si elle contient seulement une valeur.
Ces suppressions de parenthèses ne font intervenir que des informations locales. Cette étape est réalisée en temps constant avec $\mathcal{O}(n)$ processeurs, ou en temps $\log n$ avec $\mathcal{O}(\frac{n}{\log n})$ processeurs.

Troisième étape : construction de l'arbre

Il reste à déterminer, pour chaque sous-expression délimitée par des parenthèses, l'arbre correspondant ; pour cela, il faut repérer la racine de cette sous-expression, puis, pour chaque opérateur, son fils gauche et son fils droit. Dans un premier temps, seules les parenthèses

donnent lieu à un traitement. Une paire de parenthèses délimite une sous-expression, dont la représentation arborescente admet pour racine l'opérateur le plus à droite (pour être compatible avec l'interprétation usuelle des opérations $-$ et $/$, on convient que les opérateurs sont associatifs à gauche, *i.e.* $a-b-c-d = (((a-b)-c)-d)$. Chaque parenthèse, ouvrante ou fermante, détermine cet opérateur racine de la sous-expression qu'elle délimite. Cet opérateur est trouvé en temps constant : il est situé à gauche du dernier opérande. À gauche de la parenthèse fermante en cours de traitement, on trouve :

- une valeur. L'opérateur cherché est à gauche de cette valeur : $\ldots \diamond a)$, $\diamond$ est la racine cherchée.

- une parenthèse fermante. Le dernier opérande est une sous-expression, $\diamond$ l'opérateur cherché est à gauche de la parenthèse ouvrante correspondante : $\ldots \diamond (\ldots))$.

On procède de même pour une parenthèse ouvrante. Chaque processeur détermine, pour chaque parenthèse ouvrante ou fermante, la racine de la sous-expression qu'elle délimite.

Dans un deuxième temps, seuls les opérateurs donnent lieu à un traitement. Chaque opérateur recherche ses deux fils. Comme on a adopté pour convention l'associativité à gauche pour les opérations, cela signifie que l'opérateur à gauche de l'opérateur en cours de traitement est son fils gauche et que l'opérande à sa droite est son fils droit dans l'arbre. Pour trouver son fils gauche, l'opérateur teste si, à sa gauche, il y a une valeur ou une sous-expression ; il teste également s'il est le premier opérateur de sa sous-expression (le plus à gauche) ou s'il y a un autre opérateur susceptible de devenir son fils gauche. À gauche de l'opérateur $\diamond$ en cours de traitement, on trouve :

- une valeur a. À gauche de cette valeur, il y a :
  - une parenthèse ouvrante : le fils gauche est a : $(a \diamond \ldots$
  - un opérateur $\bullet$. Le fils gauche de $\diamond$ est $\bullet$: $\ldots \bullet a \diamond \ldots$

- une parenthèse fermante. À gauche de la parenthèse ouvrante correspondante, il y a :
  - une parenthèse ouvrante. Le fils gauche de $\diamond$ est $\bullet$, la racine de la sous-expression à sa gauche : $((\ldots \bullet \ldots) \diamond \ldots$
  - un opérateur $\bullet$. Le fils gauche de $\diamond$ est $\bullet$: $\ldots \bullet (\ldots) \diamond \ldots$

Quant au fils droit, il s'agit simplement de la racine de l'opérande droit : s'il y a une valeur à droite de l'opérateur, elle devient son fils droit, sinon il y a une sous-expression entre parenthèses et la racine de cette sous-expression devient le fils droit cherché.

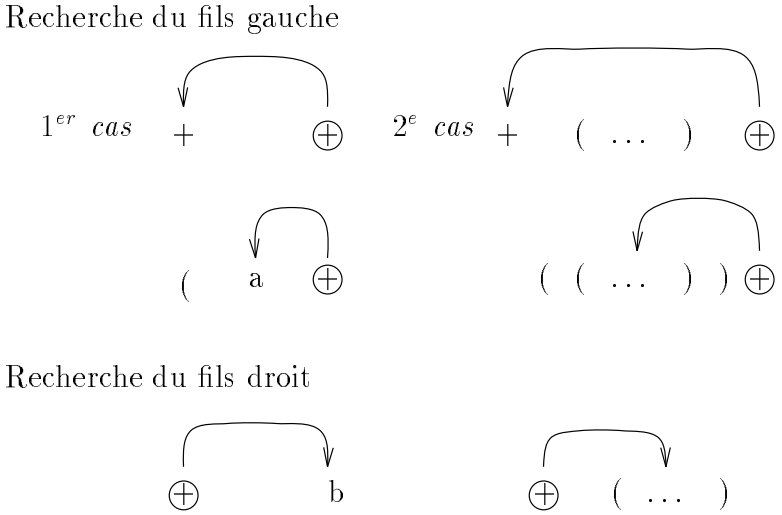


FIG. 2.27 - *Recherche des fils gauche et droit.*

Pour que l'algorithme soit optimal, on affecte à chaque processeur un segment de longueur $\log n$ du tableau initial, ce qui correspond à au plus $5 \log n$ éléments de l'expression parenthésée, pour qu'il les traite en séquentiel. Le traitement d'une parenthèse ou d'un opérateur s'effectue en temps constant. Chaque processeur ayant $\mathcal{O}(\log n)$ éléments à traiter, le temps d'exécution de cette étape est $\mathcal{O}(\log n)$, avec $\mathcal{O}(\frac{n}{\log n})$ processeurs.

Exemple

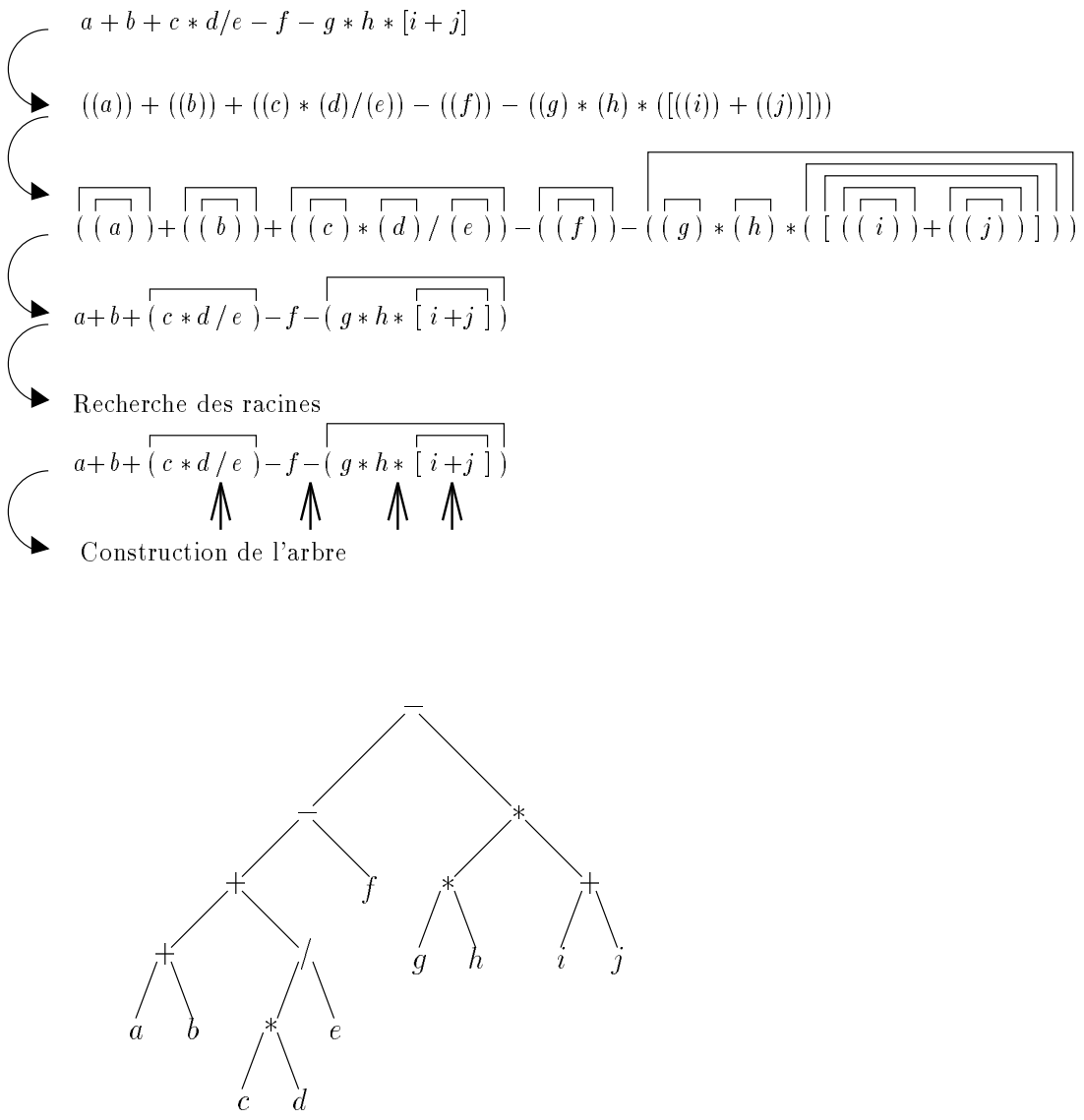


FIG. 2.28 - *Construction de l'arbre correspondant à l'expression* $a+b+c*d/e-f-g*h*[i+j]$.

Sur la figure 2.28, les racines de chaque sous-expression sont repérées par des flèches.

Appariement des parenthèses

Il nous reste à voir comment on calcule, pour chaque parenthèse, la position de la parenthèse associée.

On partage le tableau en $\frac{n}{\log n}$ segments de longueur $\mathcal{O}(\log n)$, chaque processeur traite un segment. Cela permet d'obtenir un algorithme optimal, en utilisant seulement $\mathcal{O}(\frac{n}{\log n})$ processeurs. Dans un premier temps, chaque processeur trouve les paires de parenthèses incluses dans son segment en temps $\mathcal{O}(\log n)$. On néglige maintenant ces parenthèses appariées, ainsi que les opérateurs et les valeurs.
Chaque processeur est alors en possession d'une séquence de parenthèses du type «)) ...)(... ((», que l'on appellera sa sous-suite. Il cherche la parenthèse fermante correspondant à la parenthèse ouvrante la plus à gauche de sa sous-suite (ou la plus englobante) et la parenthèse ouvrante correspondant à sa parenthèse fermante la plus à droite. Pour déterminer les autres correspondances, il cherchera ensuite parmi les voisines des parenthèses déjà appariées : si, dans (()), les parenthèses extrêmes sont mises en correspondance, la deuxième parenthèse ouvrante trouvera sa parenthèse fermante à gauche de celle appariée à la première parenthèse ouvrante.

Pour apparier les parenthèses extrêmes (les plus englobantes), on calcule le niveau d'imbrication de chaque parenthèse. La correspondante d'une parenthèse ouvrante, par exemple, est la parenthèse fermante, située à droite, la plus proche qui a le même niveau d'imbrication, d'où l'intérêt de connaître ces niveaux. On les calcule grâce à un produit itéré : à chaque parenthèse ouvrante, on associe la valeur $+1$ et à chaque parenthèse fermante, la valeur -1. Si la parenthèse en position i est une parenthèse ouvrante, son niveau d'imbrication est $L[i] = \sum_{j=1}^{i} val[j]$, sinon c'est $L[i] + 1$.
Pour réaliser ce préfixe parallèle, on a construit un arbre binaire équilibré, dont les feuilles sont les sous-suites. On l'utilise à nouveau pour déterminer les correspondances. Pour chaque nœud interne de l'arbre, on calcule deux quantités, MIN_O et MIN_F, qui sont les minima sur ses descendants des valeurs $L[i]$ pour les i parenthèses ouvrantes et les i parenthèses fermantes respectivement. Il suffit d'une phase ascendante de calcul. Ces quantités serviront de « panneaux indicateurs » pour savoir où se trouve une parenthèse, ouvrante ou fermante, d'un niveau donné. Il reste maintenant à trouver, pour une parenthèse fermante la plus à droite dans sa sous-suite, la parenthèse ouvrante qui lui correspond. Si cette parenthèse fermante est en position i, on cherche le plus grand $j<i$ tel que $L[j] = L[i]+1$. Si, contrairement à l'usage établi dans la nature, on représente les arbres avec la racine en haut et les feuilles en bas, on commence par « grimper » dans l'arbre, en partant de la feuille i, tant que dans le sous-arbre en dessous il n'y a pas de parenthèse ouvrante de même niveau d'imbrication, *i.e.* jusqu'à ce que l'on atteigne le fils droit d'un nœud dont le fils gauche k vérifie $MIN_O(k) \leq L[i] + 1$. On passe en k, puis on dégringole dans l'arbre, en se dirigeant vers la parenthèse ouvrante de même niveau, c'est-à-dire en allant à droite, si $MIN_O \leq L[i]+1$ sur le fils droit, à gauche sinon. On finit par atterrir sur la sous-suite contenant la parenthèse ouvrante correspondant à notre parenthèse fermante. On marque ces parenthèses. On procède de même pour la parenthèse ouvrante la plus à gauche. Cette promenade dans l'arbre prend un temps $\mathcal{O}(\log n)$ et chaque processeur a au plus deux parenthèses à traiter.

Il reste alors à trouver les parenthèses correspondant aux autres parenthèses de la sous-suite. On part de la parenthèse fermante la plus à droite et on établit les correspondances pour les parenthèses fermantes à sa gauche, en les parcourant de droite à gauche. Les pa-

renthèses ouvrantes recherchées se trouvent à droite de la correspondante de la parenthèse fermante « extrême ». Quand on rencontre une parenthèse fermante déjà marquée, on cherche alors à droite de sa parenthèse ouvrante correspondante. Un processeur s'arrête quand il a fini de traiter les parenthèses fermantes de sa sous-suite. Il procède de même pour les parenthèses ouvrantes. L'appariement de chaque parenthèse non extrême est effectué en temps constant. Comme chaque processeur a $\mathcal{O}(\log n)$ parenthèses à traiter, cela prend un temps $\mathcal{O}(\log n)$. □

Exemple
Sur cet exemple (figure 2.29), les parenthèses appartenant à l'expression initiale sont représentées par des crochets ([et]), pour être distinguées des parenthèses ajoutées par l'algorithme.

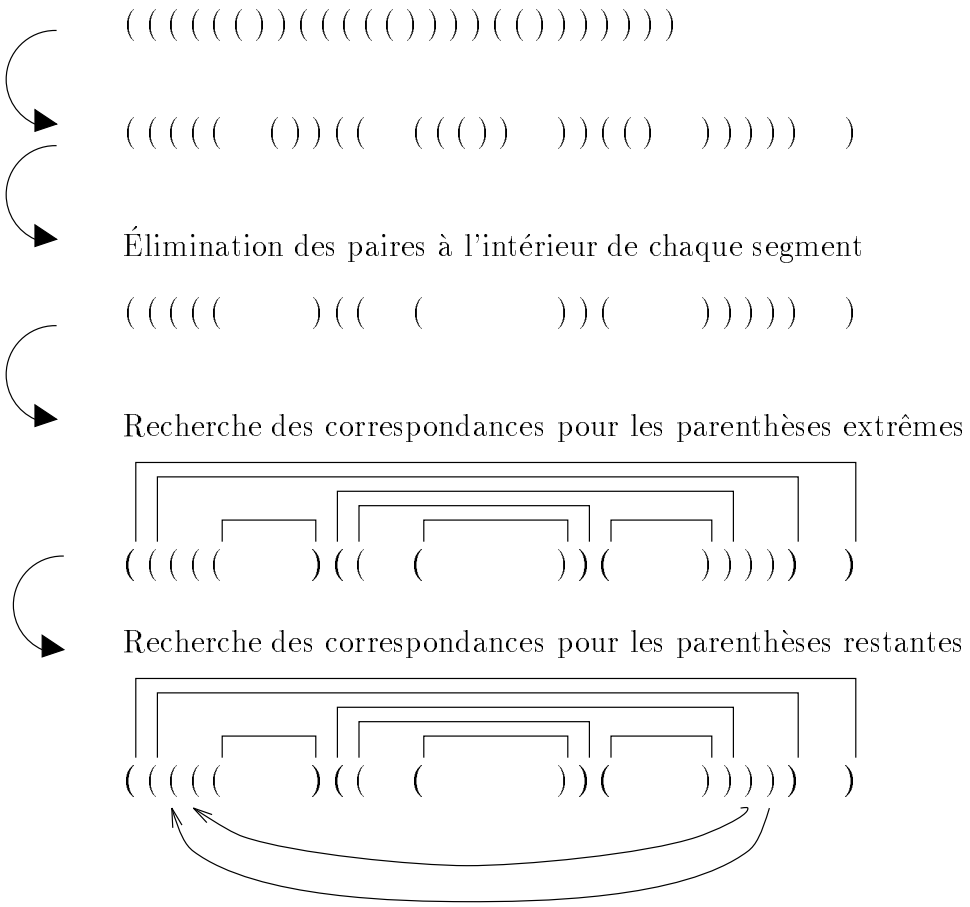


FIG. 2.29 - *Un exemple avec 26 parenthèses, soit 6 segments de longueur ≤ 5.*

On vient de voir un algorithme de construction parallèle de l'arbre correspondant à une expression, si cette expression est donnée sous forme d'un tableau.
Si le tableau est de longueur n, on peut construire l'arbre en temps $\mathcal{O}(\log n)$ avec $\mathcal{O}(\frac{n}{\log n})$ pro-

cesseurs. On vérifie facilement que cet algorithme n'entraîne jamais d'écritures concurrentes. Cependant, il peut y avoir des lectures concurrentes d'une même donnée, par exemple lors de la recherche dans l'arbre des correspondances pour les parenthèses extrêmes. L'algorithme de Bar-On et Vishkin a donc pour complexité

$$CREW(\frac{n}{\log n}, \log n).$$

2.6.2 Numérotation des feuilles

Il ne nous reste plus qu'à montrer comment on numérote les feuilles consécutivement de gauche à droite, pour compléter la présentation de cet algorithme d'évaluation d'expressions arithmétiques.

Pour cela, on construit un circuit eulérien de l'arbre, supposé non orienté ici [TV85] : on remplace chaque arête par deux arcs anti-parallèles et le circuit parcourt une fois et une seule chaque arc. Quand il arrive, par l'arc le connectant à son père, sur un nœud interne, la première fois, il se dirige vers le fils gauche, la deuxième, vers le fils droit et la troisième fois, il retourne vers le père. S'il arrive sur la racine, comme il n'existe pas de père et qu'il s'agit d'un circuit, il retourne toujours vers les fils. Si enfin, il arrive sur une feuille, il repart vers le père de cette feuille. Par convention, le point de départ de ce circuit est la racine, et le parcours du circuit commence en se dirigeant vers le fils gauche. Pour éviter de rencontrer trois fois chaque nœud interne, on remplace chaque nœud interne v, de père u et de fils w_1 et w_2, par trois copies de lui-même, v_1, v_2 et v_3 : v_1 a pour arc entrant l'arc issu de u et pour arc sortant l'arc dirigé vers w_1 ; v_2 a pour arc entrant l'arc issu de w_1 et pour arc sortant l'arc dirigé vers w_2 ; enfin v_3 a pour arc entrant l'arc issu de w_2 et pour arc sortant l'arc dirigé vers u. La racine n'a que deux copies et les feuilles ne sont pas copiées. Cette transformation est illustrée par le schéma 2.30. Elle s'effectue en temps constant avec $\mathcal{O}(n)$ processeurs, ou en temps $\mathcal{O}(\log n)$ avec $\mathcal{O}(\frac{n}{\log n})$ processeurs.

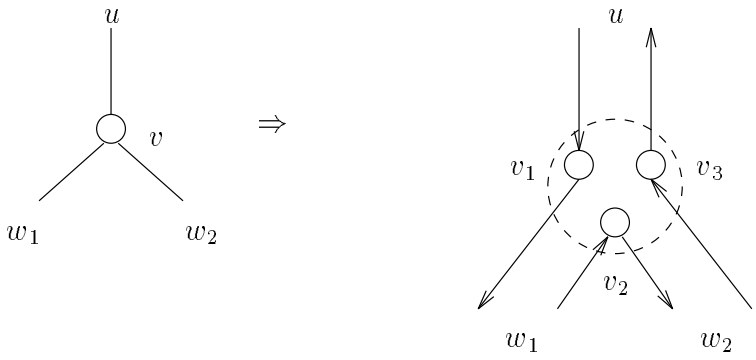


FIG. 2.30 - *Duplication des sommets et des arcs.*

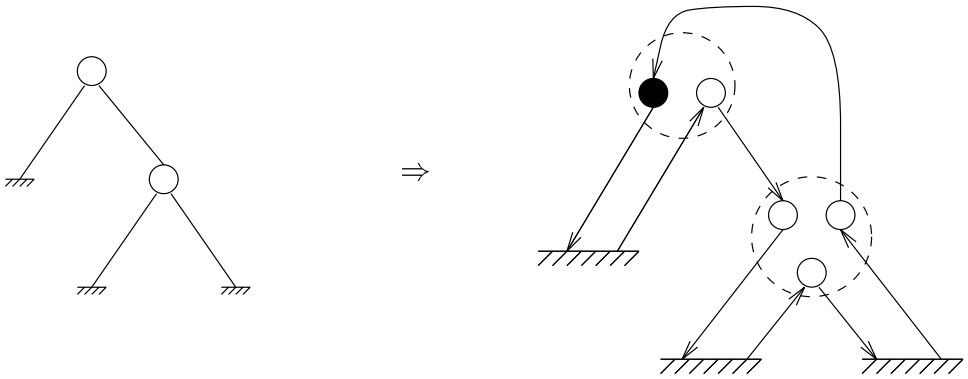


FIG. 2.31 - *Un petit exemple.*

À partir de ce circuit eulérien on numérote les feuilles : on affecte à chaque feuille la valeur 1 et à chaque nœud interne, la valeur 0. Grâce à un indiçage de liste, chaque feuille connaît ensuite son numéro. Pour éviter de numéroter la première feuille, on supprime le numéro de la feuille numéro 1 et on soustrait 1 aux numéros des autres feuilles. Pour éviter de numéroter la dernière feuille, soit on connaît le nombre de feuilles et celle qui a le numéro correspondant supprime son numéro, soit on effectue un nouvel indiçage de liste – ou un produit itéré – avec l'opération max, afin de déterminer le plus grand numéro et le supprimer.

2.6.3 Réalisations pratiques

Il y a eu peu de travaux sur l'implantation effective de cet algorithme théorique sur une machine parallèle à mémoire distribuée, même en supposant que celle-ci a n processeurs [Bau92].

En fait, comme d'une part, on ne sait pas placer un arbre quelconque sur une architecture donnée et que d'autre part, les communications qui auront lieu ne respectent pas cette localité, il est difficile d'établir des résultats de complexité. Par exemple, l'information la plus précise que l'on possède sur les communications est qu'à l'étape i, elles ne feront intervenir que des processeurs à distance au plus 2^i. Même en supposant que l'on ait une sorte d'hypercube enrichi, où chaque processeur de numéro p a pour voisins les processeurs de numéro $p \pm 2^i$, $0 \leq i < \log n$, on n'arrive pas à borner le temps d'exécution plus finement que par

$$\sum_{i=1}^{\log n} \lfloor \frac{i-1}{2} \rfloor \simeq \frac{\log^2 n}{4}.$$

On a donc intérêt à appliquer des résultats très généraux sur les routages de permutations quelconques dans le réseau choisi [Bau92] :

$\mathcal{O}(\log n)$	avec une forte probabilité avec un algorithme probabiliste pour un hypercube ou un CCC[4][Val90],
$\mathcal{O}(\log n \log\log n)$	avec un algorithme déterministe pour les mêmes architectures [CP90],
$\mathcal{O}(\sqrt{n})$	pour une grille $\sqrt{n} \times \sqrt{n}$ [GS89b].

À chaque étape, les communications peuvent être assimilées à une permutation des données entre les processeurs ; on obtient alors comme bornes :

$\mathcal{O}(\log^2 n)$	avec une forte probabilité avec un algorithme probabiliste pour un hypercube ou un CCC,
$\mathcal{O}(\log^2 n \log\log n)$	avec un algorithme déterministe pour les mêmes architectures,
$\mathcal{O}(\log n\sqrt{n})$	pour une grille $\sqrt{n} \times \sqrt{n}$.

Si on veut ensuite étudier le cas où l'on n'a que p processeurs, on constate que les communications sont encore plus imprévisibles et par conséquent, les résultats plus durs à établir. On obtient un surcoût dû aux communications, à chaque étape, de $\mathcal{O}(\frac{n}{p}\log p)$, ce qui fait au total $\mathcal{O}(\frac{n}{p}\log n \log p)$.
Si $n = \Omega(p^{1+\varepsilon})$, Ryu et JàJà [RJ90] décrivent un algorithme en temps optimal $\mathcal{O}(\frac{n}{p})$ sur un hypercube.

2.7 Bilan

Nous avons présenté l'algorithme de Kosaraju et Delcher, ou d'Abrahamson, Dadoun, Kirkpatrick et Przytycka, d'évaluation parallèle d'expressions arithmétiques. Nous avons également détaillé une version optimale de cet algorithme, due à Gibbons et Rytter, dont la complexité est $EREW(\frac{n}{\log n}, \log n)$, si l'expression effectue n opérations. L'existence d'algorithmes en temps sous-logarithmique pour des machines $CRCW-PRAM$ est une question ouverte.
Nous avons ensuite vu que, dans le cadre de la reconnaissance de langages hors-contexte parenthésés, une parallélisation de l'algorithme de Cocke, Younger et Kasami pouvait être considérée comme l'évaluation parallèle d'une expression avec des opérations bien choisies. Ce problème se résout donc avec la même complexité $EREW(\frac{n}{\log n}, \log n)$, si n est la longueur de la chaîne à reconnaître.

L'algorithme de Bar-On et Vishkin permet de construire l'arbre correspondant à une expression fournie sous forme de tableau avec une complexité $CREW(\frac{n}{\log n}, \log n)$. Il faut relaxer les hypothèses portant sur la machine, c'est-à-dire lever l'hypothèse lecture exclusive et passer à une machine $CREW-PRAM$.

4. Cycle Connected Cube : hypercube dans lequel chaque processeur est remplacé par un anneau de $\log n$ processeurs ; le diamètre du réseau ainsi obtenu est encore $\log n$, alors que le degré est constant et égal à 3, n désignant le nombre de processeurs.

Des études portant sur l'implantation de ces algorithmes sur des machines à mémoire distribuée à n processeurs mettent en évidence la difficulté que cela représente, du fait de schémas de communications inconnus. Il s'avère qu'un surcoût dû à ces communications de l'ordre du diamètre du réseau est à prévoir : $\mathcal{O}(\log n)$ pour un hypercube ou un CCC, $\mathcal{O}(\sqrt{n})$ pour une grille. Si le nombre de processeurs est réduit, les complexités sont encore plus difficiles à prévoir.

2.8 Limitations

Les expressions arithmétiques forment une classe de programmes certes simples, mais trop simples pour exprimer les programmes les plus courants. Prenons l'exemple de la résolution de systèmes triangulaires (inférieurs) par la technique classique dite de descente triangulaire :

Algorithme 8 *Résolution de systèmes triangulaires inférieurs*

pour $i = 1$ à n faire
  $x[k] = b[k]$
  pour $j = 1$ à $i - 1$ faire
    $x[k] \leftarrow x[k] - A[k,j] * x[j]$

  $x[k] \leftarrow \frac{x[k]}{A[k,k]}$

fin

On peut transformer ce programme en une forêt de n arbres, chacun fournissant un élément du vecteur x résultat de $Ax = b$. Comptons le nombre de nœuds n_k de l'arbre permettant de calculer $x[k]$, par induction sur k.

$$\begin{array}{rcl@{\qquad}rcl} x[1] & = & \frac{b[1]}{A[1,1]} & \Rightarrow & n_1 & = & 3, \\ x[k] & = & \frac{b[k]-\sum_{j=1}^{k-1} A[k,j]*x[j]}{A[k,k]} & \Rightarrow & n_k & = & 4 + 3(k-1) - 1 + \sum_{j=1}^{k-1} n_j, \end{array}$$

$$\begin{array}{rcl} \text{notons} \quad s_k & = & \sum_{j=1}^{k-1} n_j, \\ s_k & = & 3k + 2s_{k-1}, \\ s_k & = & 6 * 2^k - 3k - 6, \\ \\ n_k & = & s_k - s_{k-1} = 3 * 2^k - 3. \end{array}$$

On obtient des arbres avec un nombre exponentiel de nœuds, que l'on peut donc évaluer en temps linéaire avec un nombre exponentiel de processeurs ! Il est clair que ce n'est pas très raisonnable.

On constate donc que le modèle des expressions arithmétiques forme une classe de programmes trop restreinte par rapport à celle que l'on utilise couramment ; on décide alors de se tourner vers le modèle plus puissant des circuits arithmétiques.

Chapitre 3

Évaluation de circuits arithmétiques

Résumé

Dans ce chapitre, la définition de la nouvelle classe de programmes que nous utilisons, la classe des circuits arithmétiques, sera donnée. L'algorithme d'évaluation pour les expressions peut être appliqué aux circuits arithmétiques, mais il ne donne pas toujours de bons résultats. Une solution plus performante à ce problème d'évaluation des circuits arithmétiques à opérations dans un semi-anneau est l'algorithme de Miller, Ramachandran et Kaltofen. La section 3.4 contient la preuve de sa complexité, qui est $CREW(M(n), \log n \log(nd))$, si n est le nombre de nœuds du circuit et d son degré, et si $M(n)$ est le nombre de processeurs nécessaires pour multiplier deux matrices $n \times n$ en temps $\mathcal{O}(\log n)$ sur une $CREW-PRAM$. Le problème de l'extension de cet algorithme au cas des corps sera ensuite étudié. Le calcul du déterminant et de l'inverse d'une matrice illustreront ces résultats.

À la fin du chapitre 2, l'exemple de la résolution de systèmes triangulaires a mis en évidence la faille du modèle des expressions arithmétiques : l'impossibilité de réutiliser le résultat d'un calcul. Pour obvier à cet inconvénient, l'utilisation multiple d'un même résultat est autorisée. Les programmes ainsi obtenus forment la classe des circuits arithmétiques, *i.e.* la classe des programmes représentables par des DAGs (*directed acyclic graphs* ou graphes orientés sans circuit). La définition des circuits arithmétiques est introduite par von zur Gathen [Gat86, Gat88]. Cette classe est encore assez restreinte, puisqu'elle n'autorise pas les tests.

L'évaluation parallèle dynamique d'un programme sans test est un problème fondamental : en effet, nous verrons dans le chapitre 4 que le problème, dans le cas des booléens, est P-complet. La présence de tests (dont la valeur dépend des entrées) empêche l'anticipation des calculs. Il n'est pas possible de calculer simultanément l'évaluation de la partie « alors » et de la partie « sinon » liées au test sans entraîner un grossissement exponentiel de la taille du circuit (sauf bien sûr lorsque le nombre de tests est borné par une constante indépendante de la taille du circuit). Un algorithme naïf d'évaluation des circuits arithmétiques s'obtient en adaptant l'algorithme conçu pour les expressions. On verra les faiblesses de cette approche. En 1986, Miller, Ramachandran et Kaltofen [MRK86] ont proposé un algorithme d'évaluation des circuits arithmétiques à opérations dans un semi-anneau, qui a pour com-

plexité $CREW(M(n), \log(n)\log(nd))$, avec n le nombre de nœuds du circuit, d le degré du circuit, qui est proche du degré algébrique de la fonction calculée par le circuit, et $M(n)$, le nombre de processeurs nécessaires pour effectuer le produit de deux matrices $n \times n$ en temps $\mathcal{O}(\log n)$ dans ce semi-anneau, $M(n) \leq \mathcal{O}(n^3)$. La preuve de cette complexité s'effectue par récurrence sur la taille du circuit (son nombre de nœuds). Elle est détaillée car, au chapitre 4, la complexité de l'algorithme pour les treillis sera démontrée suivant un schéma analogue. Des extensions possibles au cas d'opérations dans des corps seront présentées. Quelques exemples classiques en complexité parallèle serviront d'illustrations pour cet algorithme.

3.1 Circuits arithmétiques

Ce modèle a été introduit et étudié par von zur Gathen [Gat86, Gat88]. Un circuit est une représentation sous forme de graphe orienté d'un programme sans boucle. Un programme sans boucle sur un semi-anneau $(S, +, *, 0, 1)$ est une succession d'affectations de la forme :

$$\begin{array}{lll} v_i & \leftarrow & val \in S \\ \text{ou} & & \\ v_i & \leftarrow & v_j + v_k, \quad 1 \leq j,k<i, \quad 1 \leq i \leq n \\ \text{ou} & & \\ v_i & \leftarrow & v_j * v_k, \quad 1 \leq j,k<i, \quad 1 \leq i \leq n. \end{array}$$

L'existence du programme sous-jacent est souvent oubliée, et les circuits arithmétiques servent directement de modèle de programme (cf. chapitre 1).

Définition 34 *Un circuit arithmétique sur un semi-anneau est un graphe orienté sans circuit ou DAG (directed acyclic graph), dont les nœuds (ou les sommets, en termes exacts de la théorie des graphes) sont étiquetés comme étant des nœuds d'entrée (valeur appartenant au semi-anneau ou variable), des nœuds + ou des nœuds $*$, + et $*$ étant les opérations d'addition et de multiplication du semi-anneau. Les arcs sont orientés des nœuds opérateurs vers leurs opérandes.*

Nous adoptons la convention inverse de la convention d'orientation usuelle, mais ceci nous simplifie l'écriture (et la programmation) des algorithmes. L'orientation des arcs sera rarement représentée sur les figures. Un nœud de sortie (c'est-à-dire un nœud dont le résultat n'est utilisé dans aucun calcul, ou nœud sans père) se retrouve donc étrangement avec un degré entrant nul : c'est un nœud qui contiendra un résultat du programme. Pour la même raison d'inversion de convention, un nœud d'entrée (c'est-à-dire une valeur, ou nœud sans fils) a un degré sortant égal à zéro. Les nœuds $*$ ont un degré sortant égal à 2 et les nœuds + ont un degré sortant ≥ 1. Ces choix se justifient grâce à la remarque suivante : quand le point de vue adopté est celui du calcul formel par exemple, il s'avère que l'addition de n entiers de n bits a la même complexité parallèle que l'addition de deux entiers de n bits. L'opération la plus pénalisante est donc la multiplication. l'algorithme – aujourd'hui connu – utilisé. Comme ce DAG sera représenté par sa matrice d'adjacence, les multi-arcs sont

interdits. Ce n'est pas réellement restrictif, puisque l'on peut toujours intercaler un nœud + unaire (figure 3.1).

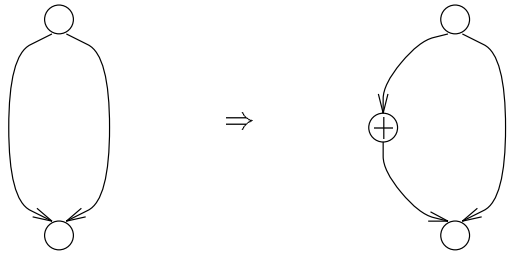


FIG. 3.1 - *On insère un nœud + unaire pour éviter un multi-arc.*

Pour des raisons exposées plus loin, qui tiennent à l'algorithme d'évaluation, il est également interdit à deux nœuds $*$ d'être adjacents. Ici encore, il suffit d'intercaler un nœud + unaire entre les deux. L'introduction de ces nœuds artificiels n'entraîne pas un grossissement exagéré du circuit initial ; il y en a au plus $\mathcal{O}(n)$.

Avant d'aller plus loin, définissons la taille d'un circuit et sa profondeur.

Définition 35 *La taille d'un circuit arithmétique est son nombre de nœuds.*

Définition 36 *La profondeur d'un circuit arithmétique est la longueur du plus long chemin de ce graphe.*

Un circuit arithmétique est donc une représentation d'un programme sans boucle (*straight-line code*). À titre d'exemple, le programme sans boucle suivant :

$$\begin{aligned} v_1 &\leftarrow x_1 + x_2 \\ v_2 &\leftarrow v_1 + x_3 \\ v_3 &\leftarrow x_2 * x_4 \\ v_4 &\leftarrow v_3 * x_5 \\ v_5 &\leftarrow v_1 + v_2 \\ v_6 &\leftarrow v_3 + v_5 \\ v_7 &\leftarrow v_4 + v_6 \end{aligned}$$

est représenté par le circuit arithmétique de la figure 3.2.

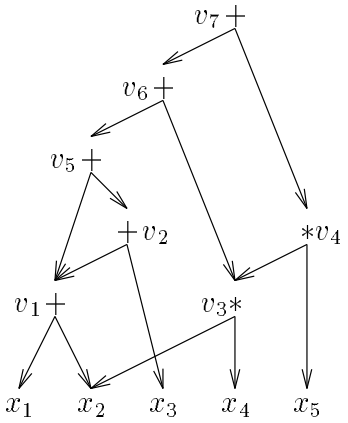


FIG. 3.2 - *Un exemple de circuit arithmétique.*

Plus généralement, considérons un algorithme $\mathcal{A}$ qui résout un problème $\mathcal{P}$ en n'effectuant que des opérations arithmétiques sur une structure F, ou des opérations entières liées à la taille n des entrées du problème. Si $\mathcal{A}$ ne contient pas d'autres branchements que ceux portant éventuellement sur n (comme par exemple une boucle *pour* $i = 1$ *à* n), l'instance de $\mathcal{A}$ permettant de résoudre le problème pour une taille fixée n des entrées peut être représentée par un programme sans boucle, de taille liée à n, et donc par un circuit arithmétique sur F, indépendant de la valeur des entrées. À titre d'exemple, le produit de deux matrices de tailles $n_1 \times n_2$ et $n_2 \times n_3$, par l'algorithme standard de produit de matrices peut être décrit par un circuit de taille $n_1 n_2 n_3$. La figure 3.3 présente le circuit arithmétique qui effectue le produit d'une matrice 2×3, $A = (a_{i,j})$, par une matrice 3×2, $B = (b_{j,k})$.

La figure 3.4 présente le circuit correspondant à l'instanciation de l'algorithme usuel de descente triangulaire pour la résolution de systèmes triangulaires inférieurs 4×4. Sur cet exemple, les opérations appartiennent à un corps. Nous verrons au §3.5 comment traiter ce cas.

3.2 Un algorithme de type contraction d'arbre

L'algorithme conçu pour les arbres s'applique pour les circuits. Il faut modifier la définition des circuits arithmétiques : l'interdiction à deux nœuds $*$ d'être adjacents peut être levée, en revanche il faut imposer aux nœuds $+$ d'être binaires pour pouvoir les ratisser.
Le premier algorithme d'évaluation de circuits arithmétiques est alors le suivant.

Algorithme 9 *Évaluation de circuits de type contraction d'arbre*

tant qu'il reste des nœuds non évalués faire
en parallèle pour tous les nœuds faire
si le fils gauche d'un des fils est une feuille, le ratisser
si le fils droit d'un des fils est une feuille, le ratisser

fin

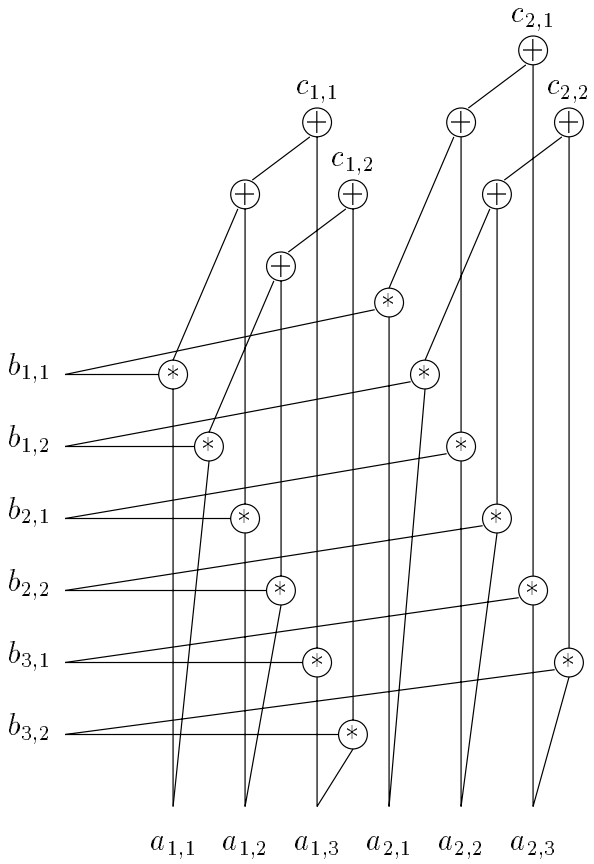


FIG. 3.3 - *Produit de matrices* 2×3 *par* 3×2.

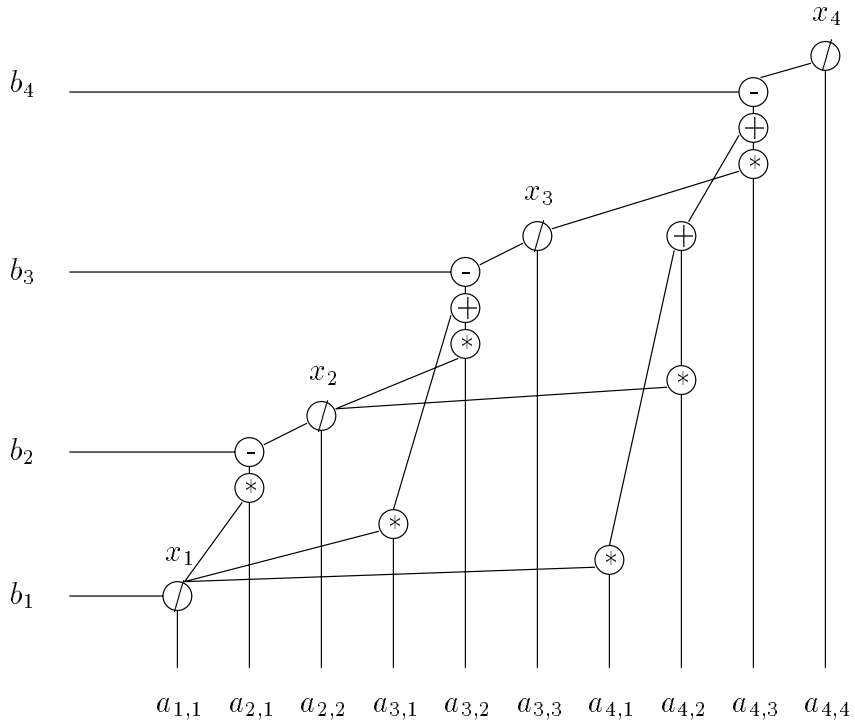


FIG. 3.4 - *Résolution d'un système triangulaire* 4×4.

Cet algorithme ne permet pas d'évaluer rapidement le circuit de la figure 3.5.

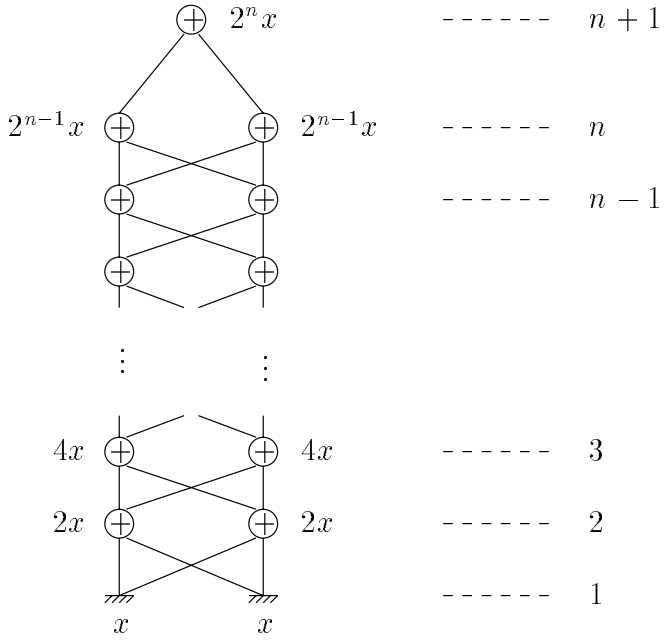


FIG. 3.5 - *Circuit évalué en temps n avec l'algorithme de type contraction d'arbre.*

Ce circuit a $2n+1$ nœuds et calcule $2^n x$. Si on applique l'algorithme 9, on supprime deux « niveaux » à chaque ratissage ; il faut par conséquent appliquer $\lceil \frac{n}{2} \rceil$ fois *ratissage* pour évaluer complètement ce circuit. Au paragraphe suivant, l'algorithme 13 permettra d'évaluer ce circuit en temps $\mathcal{O}(\log^2 n)$.

3.3 L'algorithme de Miller, Ramachandran et Kaltofen

Miller, Ramachandran et Kaltofen [MRK86] ont proposé un algorithme qui utilise l'opération de ratissage, fondamentale pour anticiper des calculs et donc gagner du temps ; ils ont cependant restreint ce ratissage, en ce sens que seuls les opérateurs $*$ peuvent être ratissés. Pour traiter les opérateurs $+$, ils ont eu l'idée de regrouper des opérateurs $+$ séquentiels en un seul opérateur $+$ d'arité variable. Il est à noter que les nœuds $*$ n'ayant pas de fils $*$, ils ne ratissent jamais leurs fils. L'intérêt qu'il y a à interdire à un nœud $*$ de ratisser un de ses fils est d'éviter une situation où un multi-arc est créé entre ce nœud et son fils, ce qui revient à pondérer un arc simple par une fonction polynomiale et à perdre les propriétés de composition en temps constant de ces fonctions (figure 3.6).

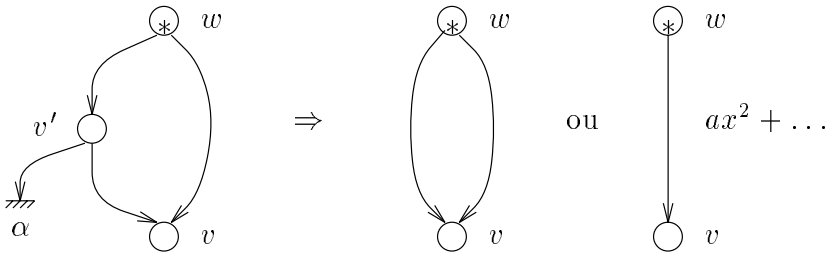


FIG. 3.6 - *Ratisser les nœuds $*$ pose problème.*

Une conséquence de ce ratissage des seuls nœuds $*$ est que les fonctions d'arcs se simplifient encore et deviennent linéaires, d'affines qu'elles étaient. Elles peuvent être représentées par un simple coefficient, que l'on stocke dans la matrice d'adjacence du DAG, U, avec $U_{v,w}$ le coefficient de l'arc d'origine v et d'extrémité w.

3.3.1 *Shunt*

On définit la procédure de ratissage qui, dans le cas des circuits arithmétiques, s'appelle *Shunt* [MRK86], par les formules suivantes (bien que U_{v,w_1} et U_{v,w_2} soient égaux à 1, ils sont mentionnés pour plus d'homogénéité) :

si v est un nœud $*$ qui a pour fils w_1 et w_2,
si w_1 est une feuille, alors
pour tous les pères u de v, $U_{u,w_2} \leftarrow U_{u,w_2} + U_{u,v} * U_{v,w_1} * U_{v,w_2} * val(w_1)$,
i.e. $U_{u,w_2} \leftarrow U_{u,w_2} + U_{u,v} * val(w_1)$
et $U_{u,v} \leftarrow 0$.

En effet, les nœuds $*$ ne « shuntant » jamais leurs fils, leurs arcs vers leurs fils restent pondérés par la fonction *Identité* tout au long de l'algorithme. On représente ce *Shunt* sur la figure 3.7.

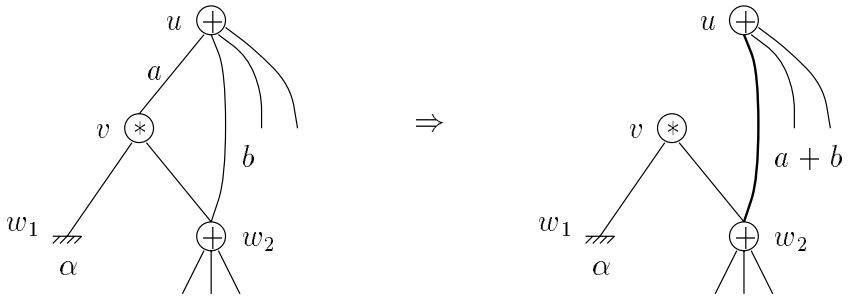


FIG. 3.7 - *Shunt.*

Cette procédure ne modifie pas les nœuds du graphe, elle supprime certains arcs et éventuellement en crée, s'il n'y en avait pas entre le père et son petit-fils. Elle modifie les fonctions d'arcs et éventuellement l'arité des nœuds $+$, mais jamais celle des nœuds $*$, puisqu'elle ne touche pas à ces derniers. Comme les nœuds « shuntés » ont exactement deux fils, le nombre de processeurs est majoré par le nombre de couples (nœud $+$, fils $*$) ; ce nombre est majoré par le nombre d'arcs du graphe, qui est au maximum de l'ordre de n^2. Cette majoration grossière nous suffit. La procédure de *Shunt* s'exécute en temps $\mathcal{O}(\log n)$: pour chaque nœud $+$, il y aura une accumulation $U_{u,w_2} \leftarrow U_{u,w_2} + U_{u,v} * val(w_1)$, chaque accumulation s'effectuant en temps $\mathcal{O}(\log n)$ avec $\mathcal{O}(n)$ processeurs (et même $\mathcal{O}(\frac{n}{\log n})$, mais ce raffinement n'est pas utile, les autres procédures étant beaucoup plus gourmandes en processeurs) et il y a moins de $n-1$ telles accumulations. Un majorant, assez large, de la complexité de la procédure *Shunt* est donc $CREW(n^2, \log n)$.

3.3.2 $\acute{E}val_+$ et $\acute{E}val_*$

Pour faire « disparaître » les nœuds $+$, il faut rétablir l'opération d'évaluation d'un nœud dont tous les fils sont évalués, qui remplace un nœud, aussi bien $+$ que $*$, dont tous les fils sont des feuilles, par une feuille. Cette opération est l'opération de base d'une parallélisation gloutonne, déjà mentionnée dans l'introduction du chapitre 2, qui est inefficace si elle est utilisée seule.

On définit les procédures $\acute{E}val_+$ et $\acute{E}val_*$ d'évaluation des nœuds $+$ et $*$ respectivement.

Algorithme 10 $\acute{E}val_+$

si v, un nœud $+$, a m fils tous feuilles (w_i pour $i \in [1, \ldots m]$), alors
    $val(v) \leftarrow \sum_{i=1}^{m} U_{v,w_i} * val(w_i)$
    pour $i = 1$ à m faire
        $U_{v,w_i} \leftarrow 0$
    v devient une feuille

fin

Algorithme 11 $\acute{E}val_*$

si v, un nœud $*$, a ses deux fils w_1 et w_2 feuilles, alors
    $val(v) \leftarrow U_{v,w_1} * val(w_1) * U_{v,w_2} * val(w_2)$
    $U_{v,w_1} \leftarrow 0$
    $U_{v,w_2} \leftarrow 0$
    v devient une feuille

fin

Ces procédures transforment certains nœuds en feuilles et elles suppriment les arcs sortants de ces nœuds évalués, mais elles ne modifient pas le nombre de nœuds. $\acute{E}val_+$ effectue une accumulation par nœud + dont tous les fils sont des feuilles, sa complexité est donc majorée par celle de n calculs de sommes itérées, *i.e.* $CREW(n^2, \log n)$. Ici encore, une majoration assez grossière nous suffit. $\acute{E}val_*$ s'effectue en temps constant avec $\mathcal{O}(n)$ processeurs, sur une $CREW-PRAM$.

3.3.3 *Group*

Enfin, pour se débarrasser du circuit gênant de la figure 3.5, l'apport de Miller, Ramachandran et Kaltofen est de regrouper plusieurs opérateurs + successifs en un seul, comme sur la figure 3.8.

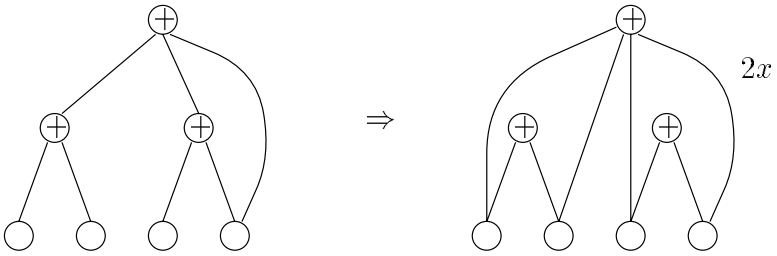


FIG. 3.8 - *Regroupement de nœuds +.*

Cette opération ressemble à un pas de calcul d'une fermeture transitive, qui serait appliquée au sous-graphe engendré par les nœuds +. À cet effet, deux autres matrices sont créées à partir de la matrice d'adjacence U du DAG :

- U^{++} la matrice d'adjacence entre les nœuds +,
  ${U^{++}}_{i,j} = U_{i,j}$ si i est un nœud + et j est un nœud +, 0 sinon ;
- $U^{+\cdot}$ la matrice d'adjacence entre les nœuds + et les autres,
  ${U^{+\cdot}}_{i,j} = U_{i,j}$ si i est un nœud + et j n'est pas un nœud +, 0 sinon.

Ce pas d'un calcul de fermeture transitive est effectué par un produit de matrices usuel : il est clair que la composition de fonctions linéaires équivaut au produit des coefficients qui les représentent. *Group* est défini à l'aide des produits de matrices suivants :

Algorithme 12 *Group*

$$\begin{aligned} U^{++} &\leftarrow U^{++} * U^{++} \\ U^{+\cdot} &\leftarrow U^{++} * U^{+\cdot} + U^{+\cdot} \end{aligned}$$

fin

Cette procédure ne modifie pas les nœuds du circuit, mais seulement la structure des arcs et les fonctions d'arcs. Elle modifie également l'arité des nœuds + mais pas celle des nœuds $*$. Sa complexité est $CREW(M(n), \log n)$, où $M(n)$ est le nombre de processeurs nécessaires pour effectuer un produit de matrices dans un semi-anneau en temps $\mathcal{O}(\log n)$. Cette notation $M(n)$ aura toujours la même signification dans la suite. *A priori*, dans un semi-anneau $M(n) = \mathcal{O}(n^3)$.

Remarque
Lorsqu'un nœud $+$, v, a deux fils $+$ qui ont tous les deux un fils commun w, l'opération *Group* (et la somme itérée qu'elle effectue) permet de rassembler les fonctions d'arcs pour qu'il n'y ait qu'un seul arc entre v et w (cf. figure 3.8).

3.3.4 Algorithme d'évaluation des circuits arithmétiques sur un semi-anneau

L'algorithme de Miller, Ramachandran et Kaltofen [MRK86] combine ces quatre opérations dans l'ordre présenté dans l'algorithme 13. La succession des procédures *Group*, $\acute{E}val_+$, $\acute{E}val_*$ et *Shunt* sera désignée sous le nom de *Phase*.

Algorithme 13 *Algorithme d'évaluation des circuits arithmétiques sur un semi-anneau*

tant qu'il reste un nœud non évalué faire
Phase :
Group
$\acute{E}val_+$
$\acute{E}val_*$
Shunt

fin

On peut montrer, par induction sur la taille du circuit, que chacune de ces opérations et *Phase* également, ne modifie pas la valeur d'un nœud, si la valeur d'un nœud v est définie inductivement par

$$val(v) = \begin{cases} \text{sa valeur} & \text{si } v \text{ est une feuille,} \\ \sum_{w=1}^{n} U_{v,w} * val(w) & \text{si } v \text{ est un nœud } + \text{ et les } w \text{ sont ses fils,} \\ U_{v,w_1} * val(w_1) * U_{v,w_2} * val(w_2) & \\ & \text{si } v \text{ est un nœud } * \text{ dont les fils sont } w_1 \text{ et } w_2 \end{cases}$$

et si les fonctions d'arc sont initialisées à la fonction *Identité*, représentée par le coefficient 1. L'hypothèse « semi-anneau unitaire » intervient ici pour l'existence de ce 1.

Pour le montrer, on utilise les propriétés algébriques de la structure de semi-anneau. Grâce à la commutativité de la multiplication, les fonctions d'arcs sont de la forme $a * x$.

La distributivité de $*$ par rapport à $+$ permet de simplifier $a*x+b*x$ en $(a+b)*x$, lors des opérations *Group* et *Shunt*. La commutativité de l'addition permet de simplifier $a*x+b*y+c*x$ en $(a+c)*x+b*y$, lors du *Group* également. Enfin, l'associativité et la commutativité des deux opérations permet d'échanger l'ordre des calculs sans modifier le résultat, ce qui est réalisé aussi bien par *Shunt* que par *Group*.

La complexité de la procédure *Phase* est déterminée par celle de *Group*, qui est l'opération la plus coûteuse à réaliser, c'est-à-dire $CREW(M(n), \log n)$.

3.3.5 Exemple

Considérons à titre d'exemple le circuit arithmétique dans le semi-anneau $(\mathbb{Z}, +, *, 0, 1)$ unitaire intègre de la figure 3.9. Pour simplifier le déroulement de l'algorithme, nous avons associé aux entrées de l'algorithme des valeurs. Le déroulement de l'algorithme est cependant indépendant des valeurs en entrée.

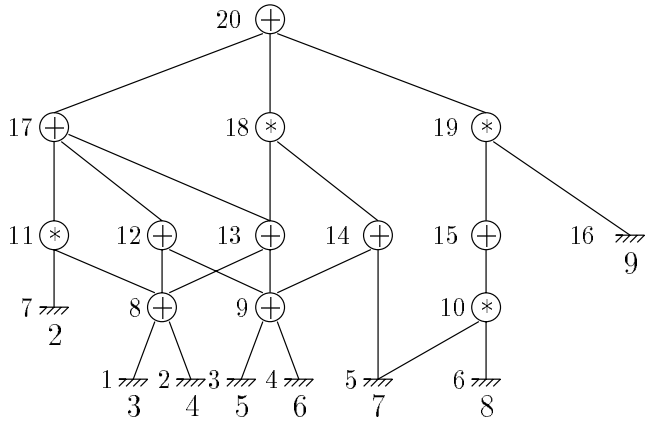


FIG. 3.9 - *Exemple.*

La matrice d'adjacence correspondante est :

$$U = \begin{pmatrix}
. & & & & & & & & & & & & & & & & & & & \\
0 & . & & & & & & & & & & & & & & & & & & \\
0 & 0 & . & & & & & & & & & & & & & & & & & \\
0 & 0 & 0 & . & & & & & & & & & & & & & & & & \\
0 & 0 & 0 & 0 & . & & & & & & & & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & . & & & & & & & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & . & & & & & & & & & & & & & \\
1 & 1 & 0 & 0 & 0 & 0 & 0 & . & & & & & & & & & & & & \\
0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & . & & & & & & & & & & & \\
0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & . & & & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & . & & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & . & & & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & . & & & & & & & \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & . & & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & . & & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & . & & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 0 & 0 & 0 & . & & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & . & & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & . & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & .
\end{pmatrix}.$$

Étape 1 :
La procédure *Group* concerne les nœuds grisés : figure 3.10. $\acute{E}val_+$ évalue les nœuds + dont tous les fils sont des feuilles : figure 3.11. $\acute{E}val_*$ évalue les nœuds $*$ dont les deux fils sont des feuilles : figure 3.12. On applique *Shunt* sur les nœuds $*$ dont un fils est une feuille : figure 3.13.

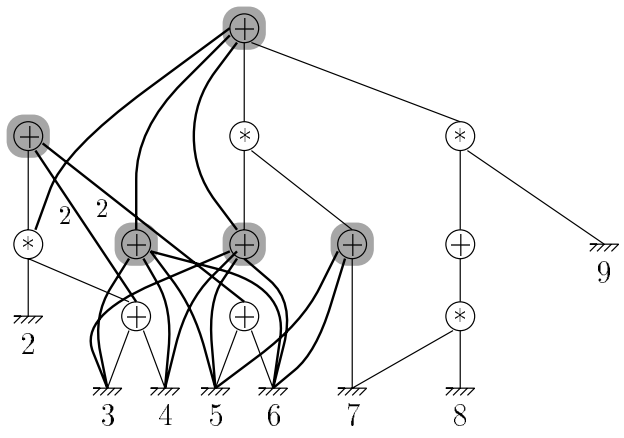


FIG. 3.10 - *Phase 1 : Group.*

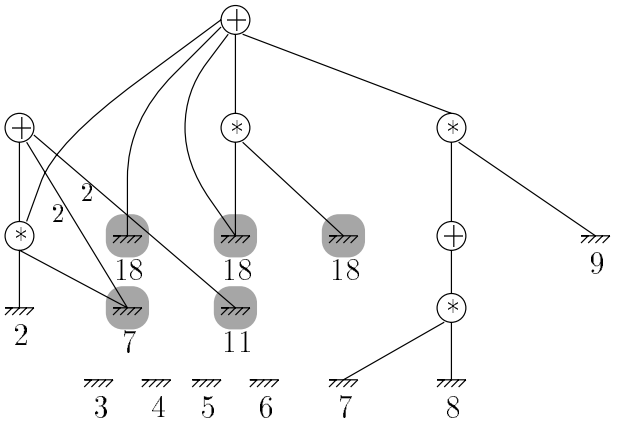


FIG. 3.11 - *Phase 1 : $\acute{E}val_{+}$.*

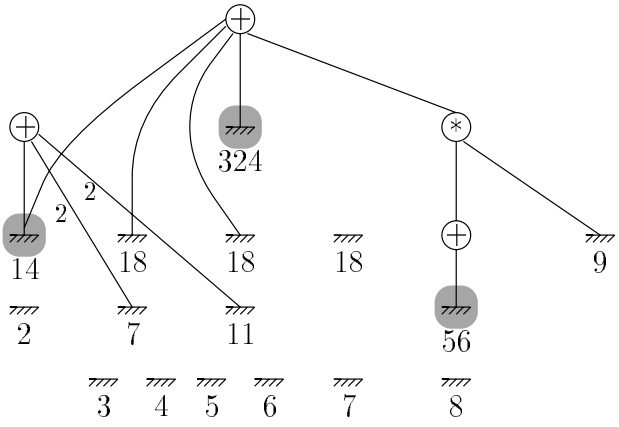


FIG. 3.12 - *Phase 1 : $\acute{E}val_{*}$.*

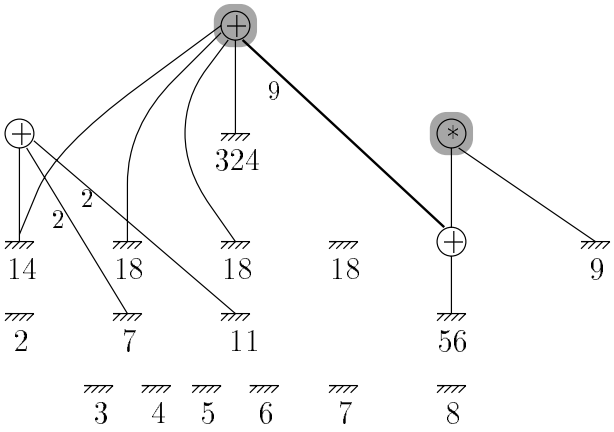


FIG. 3.13 - *Phase 1 : Shunt.*

Étape 2 :
On applique *Phase* une deuxième fois, *Group* ne concerne que le seul nœud de sortie : figure 3.14. On applique $\acute{E}val_+$, les nœuds concernés sont les nœuds grisés : figure 3.15. Le dernier nœud, qui est un nœud $*$, est évalué grâce à $\acute{E}val_*$: figure 3.16.

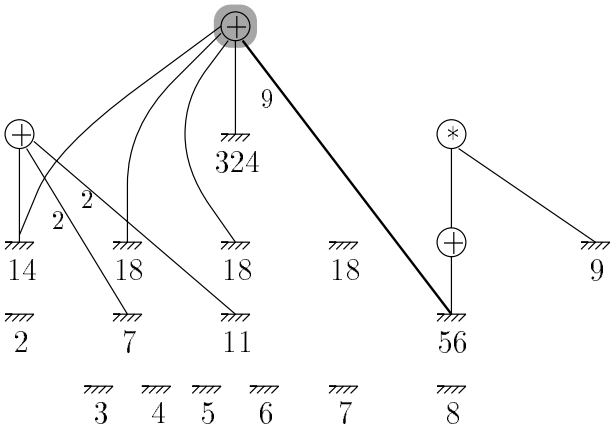


FIG. 3.14 - *Phase 2: Group.*

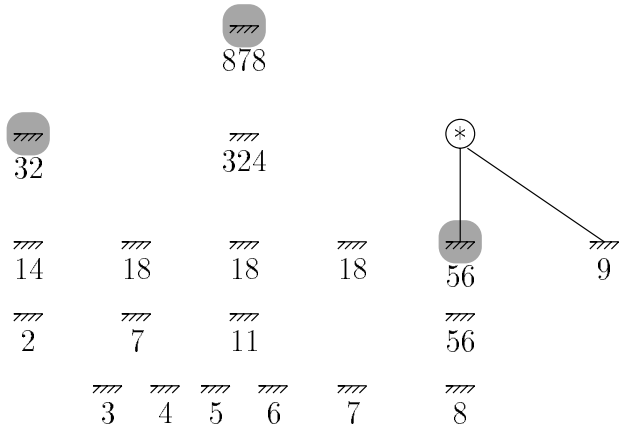


FIG. 3.15 - *Phase 2: $\acute{E}val_+$.*

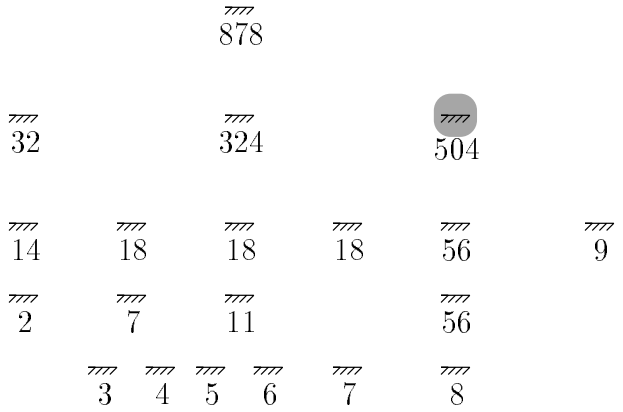


FIG. 3.16 - *Phase 2: $\acute{E}val_*$.*

Il a donc suffi de deux applications de *Phase* pour évaluer ce circuit, alors que la profondeur du circuit initial était 4.

3.4 Complexité

De façon générale, on aimerait connaître une borne sur le nombre d'applications de la procédure *Phase* pour évaluer un circuit. Pour le circuit de la figure 3.5, il suffit de calculer une fermeture transitive complète du graphe, ce qui requiert $\mathcal{O}(\log n)$ produits de matrices, soit $\mathcal{O}(\log n)$ applications de *Phase*. Si on remplace chaque nœud interne de ce circuit par un nœud $*$ et que l'on intercale des nœuds $+$ unaires, le circuit obtenu calcule x^{2^n} ; ce polynôme a un degré d exponentiel. Chaque application de *Phase* remplace le niveau de $*$ le plus bas, ou le plus proche des feuilles, par des feuilles. Il faut donc $\mathcal{O}(n) = \mathcal{O}(\log 2^n) = \mathcal{O}(\log d)$ applications de *Phase*.

3.4.1 Résultats préliminaires

Miller, Ramachandran et Kaltofen ont montré que le nombre d'applications de *Phase* dépend à la fois du nombre de nœuds du circuit et du degré d de la fonction calculée par le circuit ; ce nombre est $\mathcal{O}(\log nd)$. Pour cela, ils définissent une « hauteur » h pour chaque nœud du circuit et une hauteur pour le circuit, qui est le maximum des hauteurs de ses nœuds. Ils montrent que $h \leq \frac{1}{2}n^2d$. Ensuite, par récurrence sur la taille du circuit, ils montrent qu'une application de *Phase* divise cette hauteur par 2. Nous allons détailler leur preuve ici et nous nous inspirerons de ce schéma au chapitre 4 pour la preuve de la complexité de l'algorithme pour les treillis.

Définition 37 *Pour chaque nœud v d'un circuit, sa hauteur est définie inductivement par :*

$$h(v) = \begin{cases} 1 & \text{si } v \text{ est une feuille,} \\ \sum_w h(w \text{ fils de } v) & \\ = h(w_1 \text{ fils gauche de } v) & \\ +h(w_2 \text{ fils droit de } v) & \text{si } v \text{ est un nœud } *, \\ \max_w \begin{pmatrix} h(w \text{ fils feuille}) \\ h(w \text{ fils } *) \\ h(w \text{ fils } +) + \frac{1}{2} \end{pmatrix} & \text{si } v \text{ est un nœud } +. \end{cases}$$

La hauteur h d'un circuit est le maximum des hauteurs de ses nœuds.

Définition 38 *Un fils w d'un nœud $+$ v est* dominant *si $h(v) = h(w)$ si w est une feuille ou un nœud $*$, ou si $h(v) = h(w) + \frac{1}{2}$ et w est un nœud $+$.*

On peut considérer *Phase*, *Group*, $\acute{E}val_+$, $\acute{E}val_*$ et *Shunt* comme des fonctions sur les circuits, qui ne modifient pas les sommets (mais elles changent les étiquettes « nœud + », « nœud $*$ » ou « feuille » de ces sommets) et qui changent la structure des arcs. On parlera donc de l'image d'un circuit ou d'un nœud et de leur antécédent par *Phase*. On notera avec des primes les images des circuits U ou des nœuds v :

$$\begin{aligned} Phase(U) &= U', \\ Phase(v) &= v'. \end{aligned}$$

De plus, pour un nœud v', w' notera de façon générique un de ses fils. Le lemme qui suit est primordial car c'est lui qui permet d'établir un majorant non trivial sur la complexité.

Lemme 1 *Si U est un circuit et U' est son image par Phase, si v' est un nœud de U' qui n'est ni une feuille ni un nœud de sortie et si v est son antécédent, alors $h(v) \geq 2h(v')$.*

DÉMONSTRATION
On le démontre par induction sur la taille (le nombre de nœuds) de $U'_{v'}$ le sous-circuit induit

par v', ou le sous-circuit qui calcule v'. Plus formellement, $U'_{v'}$ est le sous-graphe de U' engendré par $\{x' \in U' \,/\, \exists$ un chemin allant de v' à $x'\}$.

INITIALISATION DE LA RÉCURRENCE

Soit v' un nœud de U', qui n'est ni une feuille ni un nœud de sortie, ou nœud sans père. (On élimine ces deux cas parce que *Phase* peut ne pas les modifier beaucoup, du fait de leur position « extrême » dans le circuit et il n'est pas certain que la hauteur soit divisée par 2 dans ces cas.) Le cas initial est celui où v' n'a que des fils feuilles. Soit v l'antécédent de v', montrons que $h(v) \geq 2h(v')$.

Supposons que v' est un nœud $+$: $h(v') = 1$. v est également un nœud $+$. Montrons que $h(v) \geq 2$, par l'absurde.
Supposons donc que $h(v)<2$: les seules possibilités sont $h(v) = 1$ ou $h(v) = \frac{3}{2}$.

- Si $h(v) = 1$, v est un nœud $+$ dont tous les fils sont des feuilles. *Group* le laisse inchangé, $\acute{E}val_+$ en fait une feuille et par conséquent, v' est une feuille, ce qui est contraire à l'hypothèse.

- Si $h(v) = \frac{3}{2}$, v est alors un nœud $+$ dont le fils dominant w est un nœud $+$ de hauteur 1, *i.e.* w n'a que des fils feuilles. Les autres fils éventuels de v sont des feuilles, ou des nœuds semblables à w (figure 3.17).

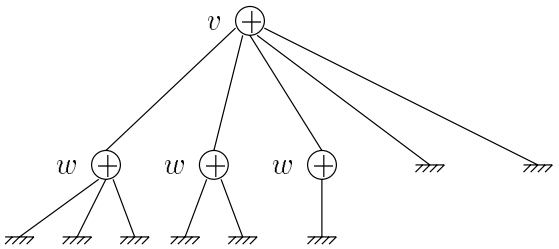


FIG. 3.17 - $h(v) = \frac{3}{2}$.

Après *Group*, v se déconnecte de w (et de tous ses fils $+$) et se connecte à ses petits-fils, qui sont des feuilles. Après $\acute{E}val_+$, v devient une feuille et donc v' est une feuille, ce qui est contraire à l'hypothèse.

On a montré par l'absurde que si v' est un nœud $+$ de hauteur 1, alors $h(v) \geq 2 = 2h(v')$.

Examinons le cas où v' est un nœud $*$ dont les deux fils sont des feuilles. $h(v') = 2$. v, son antécédent, est également un nœud $*$. Montrons, par l'absurde encore, que chacun des fils de v a une hauteur ≥ 2.

Si w, un des fils de v, a une hauteur <2, alors l'étude du cas précédent indique que w est une feuille après $\acute{E}val_+$.

- Si les deux fils de v étaient de hauteur <2, alors, après $\acute{E}val_+$, v a ses deux fils qui sont des feuilles, et après $\acute{E}val_*$, v est lui-même une feuille, ce qui contredit l'hypothèse que v' n'est pas une feuille.

- Si un seul des fils de v est de hauteur <2, après $\acute{E}val_+$, v est un nœud dont l'un des fils est une feuille et v est « shunté ». Après *Shunt*, v' devient un nœud de sortie (il n'a pas de père), ce qui est encore contraire à l'hypothèse émise sur v'.

On a donc montré par l'absurde que si v' est un nœud $*$ de hauteur 2, alors $h(v) \geq 4 = 2h(v')$.

Le cas initial est vérifié.

Cas général

Hypothèse de récurrence au rang k : on suppose que pour tout sous-circuit $U'_{w'}$ de taille $\leq k$ (*i.e.* avec moins de k nœuds) qui calcule w', tel que w' n'est ni une feuille ni un nœud de sortie, la hauteur de w, l'antécédent de w' par *Phase*, est divisée par 2 par une application de *Phase*, *i.e.* $h(w) \geq 2h(w')$.
Montrons que cela est encore vrai pour les sous-circuits de taille $k+1$. Supposons que l'on ait un nœud v', qui n'est ni une feuille ni un nœud de sortie et dont le sous-circuit associé $U'_{v'}$ est de taille $k+1$. Montrons que la hauteur de son antécédent v est divisée par 2 par une application de *Phase*.

Si v' est un nœud $+$, soit w' son fils dominant.

- Si w' est une feuille, v' n'a que des fils feuilles et on se retrouve dans un des cas initiaux déjà traités.

- Si w' est un nœud $*$, $h(v') = h(w')$. v est également un nœud $+$ et $h(v) \geq h(w)$, où w est l'antécédent de w'. On applique l'hypothèse de récurrence à $U'_{w'}$ et on a $h(v) \geq h(w) \geq 2h(w') = 2h(v')$.

- Si w' est un nœud $+$, $h(v') = h(w') + \frac{1}{2}$. v et w sont également des nœuds $+$ et il suffit de montrer qu'il existe un chemin de longueur au moins 2 entre v et w, l'antécédent de w', dans U_v. Si w est un fils de v, *Group* supprime cet arc entre v et w et le seul moyen d'en recréer un (pour qu'il existe un arc entre v' et w' dans $U'_{v'}$) est qu'il y ait un nœud $+$ entre v et w, ou un nœud $*$ à « shunter ». Dans les deux cas, il y a un chemin de longueur au moins 2 entre v et w. Comme chaque nœud sur un chemin rajoute au moins $\frac{1}{2}$ à la hauteur, on a $h(v) \geq h(w) + 1$. On applique l'hypothèse de récurrence à $U'_{w'}$: $h(w) \geq 2h(w')$, ce qui finalement donne

$$h(v) \geq h(w) + 1 \geq 2h(w') + 1 = 2(h(w') + \frac{1}{2}) = 2h(v').$$

Étudions le dernier cas, celui où v' est un nœud $*$. Si ses deux fils w'_1 et w'_2 ne sont pas des feuilles, on applique l'hypothèse de récurrence à $U'_{w'_1}$ et à $U'_{w'_2}$ et on a

$$h(v) \geq h(w_1) + h(w_2) \geq 2h(w'_1) + 2h(w'_2) = 2h(v').$$

Si l'un des deux fils est une feuille, on utilise un résultat démontré pour le cas initial, à savoir que si un nœud $*$ a un fils feuille et n'a pas été « shunté » par *Phase*, alors l'antécédent de ce fils feuille était de hauteur ≥ 2.

Ceci complète notre démonstration. □

Miller, Ramachandran et Kaltofen ont relié la hauteur h d'un circuit à son nombre de nœuds n et à son degré formel, qui ressemble au degré algébrique de la fonction calculée par le circuit. Tout d'abord, ils majorent h par $\frac{1}{2}ad + d$, où a est le nombre d'arcs connectant deux nœuds $+$ et d le degré formel du circuit.

Définition 39 *Le degré formel d'un nœud v est défini inductivement par :*

$$d(v) = \begin{cases} 1 & \text{si } v \text{ est une feuille,} \\ \sum_w d(w \text{ fils de } v) & \text{si } v \text{ est un nœud } *, \\ \max_w(d(w \text{ fils de } v) & \text{si } v \text{ est un nœud } +. \end{cases}$$

Le degré formel d'un circuit est le maximum des degrés de ses nœuds.

Le degré formel d'un circuit n'est pas tout à fait le même que le degré algébrique de la fonction qu'il calcule. Par exemple le circuit qui calcule $x * x + x - x * x$ a un degré formel égal à 2, alors que la fonction calculée est x et a un degré algébrique égal à 1. La distinction entre ces deux degrés permet de traiter correctement ces cas pathologiques.

Lemme 2 *Si U est un circuit arithmétique de degré formel d qui possède a arcs connectant deux nœuds $+$, alors sa hauteur h vérifie*

$$h \leq \frac{1}{2}ad + d.$$

DÉMONSTRATION
On le démontre par récurrence sur la taille de tout sous-circuit U_v, v étant un nœud du circuit U. Le cas initial est celui où v est une feuille : on a $h = 1$, $a = 0$ et $d = 1$, $1 \leq \frac{1}{2} * 0 * 1 + 1$. Supposons que cela soit vrai pour tout sous-circuit de taille $\leq k$. Soit U_v un sous-circuit à $k + 1$ nœuds. Si v a pour fils $v_1, v_2, \ldots v_l$, de hauteur $h_1, h_2, \ldots h_l$, de degré $d_1, d_2, \ldots d_l$ et avec $a_1, a_2, \ldots a_l$ arcs $+-+$, l'hypothèse de récurrence est que

$$h_i \leq \frac{1}{2}a_i d_i + d_i.$$

Si v est un nœud $*$,

$$\begin{array}{rcl} h(v) & = & h(v_1) + h(v_2) \leq (\frac{1}{2}a_1d_1 + d_1) + (\frac{1}{2}a_2d_2 + d_2) \\ & = & \frac{1}{2}(a_1d_1 + a_2d_2) + (d_1 + d_2) \\ & \leq & \frac{1}{2}(a_1d_1 + a_1d_2 + a_2d_1 + a_2d_2) + (d_1 + d_2) \\ & \leq & \frac{1}{2}ad + d. \end{array}$$

Si v est un nœud $+$, soit v_1 son fils dominant.

- Si v_1 est une feuille, $a = 0, h = 1, d = 1$, les valeurs sont les mêmes que pour une feuille.
- Si v_1 est un nœud $*$:

$$h(v) = h(v_1) \leq \frac{1}{2}a_1d_1 + d_1 \leq \frac{1}{2}a_1d + d \leq \frac{1}{2}ad + d.$$

- Si v_1 est un nœud $+$, $a \geq a_1 + 1$,

$$\begin{array}{rcl} h(v) & = & h(v_1) + \frac{1}{2} \leq \frac{1}{2}a_1d_1 + d_1 + \frac{1}{2} \\ & \leq & \frac{1}{2}a_1d + d + \frac{1}{2} \\ & \leq & \frac{1}{2}(a_1 + 1)d + d \\ & \leq & \frac{1}{2}ad + d. \end{array}$$

On a montré, par récurrence sur la taille du sous-circuit, que pour tout sous-circuit U_v, $h_{U_v} \leq \frac{1}{2}a_{U_v}d_{U_v} + d_{U_v}$.

Si on considère maintenant un circuit quelconque U et v le nœud tel que $h_U = h(v)$, alors $h_U = h(v) \leq \frac{1}{2}a_{U_v}d_{U_v} + d_{U_v} \leq \frac{1}{2}a_Ud_U + d_U$. □

3.4.2 Complexité

Comme on peut majorer a par $\mathcal{O}(n^2)$, on a $h = \mathcal{O}(n^2d)$. Le lemme 1 indique que $\mathcal{O}(\log h)$ applications de *Phase* suffisent à évaluer un circuit ; or, $\mathcal{O}(\log h) = \mathcal{O}(\log nd)$ grâce au lemme 2, d'où le théorème :

Théorème 10 *La complexité de l'algorithme 13 est*

$$CREW(M(n), \log n \log nd),$$

n étant le nombre de nœuds du circuit à évaluer et d son degré.

3.4.3 Exemples

Grâce à cet algorithme, on peut estimer théoriquement le temps d'évaluation du circuit de la figure 3.2. Ce circuit a 20 nœuds et un degré égal à 3. Sa hauteur est aussi égale à 3. Il est donc évalué en temps $\lceil \log 3 \rceil = 2$, ce qui confirme le résultat obtenu.

L'algorithme 13, appliqué au circuit correspondant à l'instance du produit de matrices carrées de taille $n \times n$, évalue tous les nœuds de ce circuit en temps $\mathcal{O}(\log^2 n)$ avec $M(n^3)$ processeurs. Pour évaluer ce produit de matrices $n \times n$, il effectue paradoxalement des produits de matrices de taille n^3 !

Si on considère maintenant le problème de la résolution de systèmes triangulaires inférieurs $n \times n$, où les feuilles $a_{i,i}$ sont remplacées par $\frac{1}{a_{i,i}}$ pour faire disparaître les divisions, le circuit correspondant comporte $\mathcal{O}(n^2)$ nœuds. Les formules sont :

$$\begin{aligned} x_1 &= \frac{b_1}{a_{1,1}}, \\ x_k &= \frac{b_1 - \sum_{j=1}^{k-1} a_{k,j} * x_j}{a_{k,k}}. \end{aligned}$$

Calculons le degré de ce circuit :

$$\begin{aligned} d_1 &= 1, \\ d_k &= 2 + d_{k-1} = 2k - 1. \end{aligned}$$

le degré formel du circuit est donc $2n - 1$. Ce circuit est évalué par l'algorithme 13 en temps $\mathcal{O}(\log^2 n)$ avec $M(n^2)$ processeurs, soit environ n^6 processeurs.

3.5 Calcul dans des corps

Quand le circuit est à opérations dans un corps, *i.e.* il a des nœuds $-$ et $/$, les fonctions d'arcs ne peuvent plus être de la forme $a * x$. On peut penser à utiliser des fonctions du type $a * x + \frac{b}{x}$ et à étendre les produits de matrices : la matrice d'adjacence U aurait pour coefficients des fonctions et un produit de deux telles matrices A et B aurait pour élément

$$(A.B)_{i,j} = \sum_{k=1}^{n} a_{i,k}(x) \circ b_{k,j}(x).$$

Il est aisé de vérifier que les fonctions du type $a * x + \frac{b}{x}$, ou même $\frac{a*x+b}{c*x+d}$ ne sont pas stables par composition ou addition.
On ne peut donc appliquer cette idée que si l'on est sûr que chaque nœud du circuit n'intervient que sous une seule des deux formes $a * x$ ou $\frac{b}{x}$ dans le circuit.

Théorème 11 *Si un circuit à n nœuds, de degré formel d, à opérations dans un corps, vérifie la propriété qu'aucun de ses nœuds ne calcule une fraction rationnelle d'un autre nœud du circuit, où la valeur de cet autre nœud apparaît simultanément au numérateur et au dénominateur, alors il peut être évalué avec une complexité $CREW(n^3, \log n \log nd)$.*

DÉMONSTRATION
Si cette condition est vérifiée, le produit de matrices dans un semi-anneau, dont la complexité est $\mathcal{O}(n^3)$, peut encore être utilisé. □

Théorème 12 *Si un circuit à n nœuds, de degré formel d, à opérations dans un corps, vérifie la propriété que le Shunt des nœuds / ne s'effectue qu'en utilisant le fils droit (le diviseur) de ces nœuds, alors il peut être évalué avec une complexité $CREW(n^{2,808}, \log n \log nd)$.*

DÉMONSTRATION
Si cette condition est vérifiée, les seules fonctions d'arcs qui apparaîtront lors de l'évaluation seront des fonctions linéaires. Le produit de matrices dans un anneau, dont la complexité est $\mathcal{O}(n^{2,808})$ [Str69], peut être utilisé. Remplacer $a_{i,k} * b_{k,j}$ par $a_{i,k} \circ b_{k,j}$ ne perturbe pas cette technique. □

Corollaire 2 *Sous les mêmes hypothèses que le théorème précédent, si de plus le corps admet une topologie autorisant les calculs approchés et si les résultats désirés peuvent être des valeurs approchées, alors un circuit à n nœuds, de degré formel d, à opérations dans ce corps, peut être évalué avec une complexité $CREW$ $(n^{2,3755}, \log n \log nd)$.*

DÉMONSTRATION
Dans ce cas, on peut utiliser le meilleur algorithme (approché) connu actuellement pour les produits de matrices, celui de Coppersmith et Winograd [CW90]. □

Par exemple, si on se penche à nouveau sur le problème de la résolution de systèmes triangulaires dans $\mathbb{R}$, il apparaît que les nœuds / seront « shuntés » grâce aux feuilles $a_{k,k}$ et donc les coefficients d'arcs resteront linéaires. Dans ce cas, on peut utiliser le meilleur algorithme (approché) connu actuellement pour les produits de matrices $n \times n$ dans un corps muni d'une topologie adéquate – puisque effectuer des calculs approchés dans $\mathbb{R}$ a un sens. L'algorithme étendu au cas des corps permet donc d'évaluer ce circuit avec une complexité $CREW(n^{4.751}, \log^2 n)$.

Cependant, ces conditions ne sont pas faciles à vérifier et il est préférable d'avoir un algorithme qui traite tous les cas. Ce n'est pas si évident et la seule solution actuelle consiste plutôt à éviter les divisions, en utilisant la technique de Strassen [Str73].

Le résultat obtenu par Strassen est le suivant :

Théorème 13 Élimination des divisions *Soit F un corps et soient $f_i \in F[x_1, \ldots x_m]$ les fonctions polynomiales calculées par les k nœuds $n - k < i \leq n$ d'un circuit à n nœuds avec divisions, dont les feuilles sont les variables $x_1, \ldots x_m$. Si on connaît une borne δ sur le degré algébrique des f_i et que l'on connaît la valeur de tous les nœuds du circuit pour un jeu de valeurs $x_1 \leftarrow a_1 \in F, \ldots x_m \leftarrow a_m \in F$ (qui n'entraîne pas de divisions par zéro), alors on sait construire un circuit sans division qui calcule les f_i, $n - k < i \leq n$, avec $\mathcal{O}(n\delta \log \delta \log\log \delta)$ nœuds et un degré $\leq \delta$.*

L'idée est d'introduire les fonctions auxiliaires en une indéterminée z

$$g_i(y_1, \ldots y_m, z) = f_i(y_1 z + a_1, \ldots y_m z + a_m)$$

et d'écrire les g_i sous forme de série de Laurent en z :

$$g_i(y_1, \ldots y_m, z) = \sum_{j=-l_i}^{+\infty} c_{i,j}(y_1, \ldots y_m) z^j,\ l_i \geq 0,$$

en remplaçant chaque division par un développement en série au voisinage de $z = 0$, du style $(1-z)^{-1} = \sum_{i=0}^{+\infty} z^i$. Les quelques remarques qui suivent permettent de conclure :

- $c_{i,0} = g_i(y_1, \ldots y_m, 0) = f_i(a_1, \ldots a_m)$.
- Si le nœud i, qui calcule f_i, est le diviseur d'un autre nœud, alors $c_{i,0} \neq 0$ (pas de division par zéro). Dans ce cas, on sait remplacer une division par une série.
- Comme les f_i sont des fonctions polynomiales de degré $\leq \delta$, $c_{i,j} = 0$ pour $j<0$ et $j>\delta$.
- Comme les f_i sont des fonctions polynomiales, les $c_{i,j}$ sont des polynômes en $y_1, \ldots y_m$.

Il suffit donc de tronquer les séries à l'ordre $\delta + 1$. Pour récupérer les f_i à partir des g_i, on écrit

$$\begin{aligned} f_i(x_1, \ldots x_m) &= g_i(x_1 - a_1, \ldots x_m - a_m) \\ &= \textstyle\sum_{j=0}^{\delta} c_{i,j}(x_1 - a_1, \ldots x_m - a_m) \\ &= f_i(a_1, \ldots a_m) + \textstyle\sum_{j=1}^{\delta} c_{i,j}(x_1 - a_1, \ldots x_m - a_m). \end{aligned}$$

Il reste alors deux problèmes à régler :

- comment trouver $a_1, \ldots a_n$ pour qu'il n'y ait pas de divisions par zéro, ainsi que les valeurs des nœuds en $a_1, \ldots a_n$?
- comment fait-on si le circuit calcule des fractions rationnelles à la place de polynômes ?

Pour le premier problème, Kaltofen [Kal88] a proposé un algorithme (séquentiel, probabiliste) de type Monte-Carlo, basé sur un lemme de Schwartz, qui est polynomial en temps. Le maximum des degrés des f_i, δ, est déterminé par un algorithme séquentiel (probabiliste de type Monte-Carlo) également. Ces algorithmes sont séquentiels, en particulier parce qu'ils nécessitent l'évaluation du circuit en des valeurs aléatoires et qu'on ne sait pas encore évaluer le circuit en parallèle. En réponse à la seconde question, Kaltofen [Kal88] et Kaltofen et Trager [KT90] ont montré qu'en utilisant des approximants de Padé, on obtient un résultat ressemblant à celui de Strassen :

Théorème 14 *Soient f et $g \in F[x_1 \ldots x_m]$ deux polynômes premiers entre eux de degré inférieur à d, tels que $\frac{f}{g}$ soit calculé par un circuit à n nœuds, alors $\frac{f}{g}$ peut être calculé par un circuit sans division avec $(nd)^{\mathcal{O}(1)}$ nœuds et une profondeur $\mathcal{O}(\log^2(nd))$.*

3.6 Exemple : déterminant et inverse d'une matrice

Au paragraphe 3.5, il a été montré que l'algorithme 13 permet de résoudre un système triangulaire en temps $\mathcal{O}(\log^2 n)$. Si on utilise l'algorithme d'élimination de Gauss pour obtenir ces systèmes triangulaires, on se retrouve avec un circuit avec divisions, auquel on ne peut pas appliquer notre algorithme déterministe étendu ; en effet, dans la formule

$$A[j,k] \leftarrow A[j,k] - \frac{A[i,k] * A[j,i]}{A[i,i]},$$

le terme $A[i,i]$ provient d'une multiplication du même type, faisant intervenir par exemple $A[i-1,i-1]$. Ce même $A[i-1,i-1]$ est apparu comme dénominateur dans le calcul de $A[i,k]$, il apparaît donc à la fois au numérateur et au dénominateur de la fraction rationnelle sur les coefficients de la matrice A qui permet de calculer $A[j,k]$ avec $j>i$, $k>i$.

Seule la méthode de Strassen pour éliminer les divisions s'applique. Il existe une matrice connue pour laquelle ne se produit pas de division par zéro : la matrice *Identité*. On obtient alors un circuit sans division, qui est évalué en temps $\mathcal{O}(\log^2 n)$ avec $M(n^4 \log n \log\log n)$ processeurs [Kal88].

Il est à noter qu'il est possible, avec la même complexité, de calculer le déterminant d'une matrice : c'est le produit des éléments diagonaux de la matrice triangulaire supérieure résultat. Avec 4 fois plus de nœuds et la même complexité, il est également possible d'inverser A : en effet,

$$A^{-1} = \left[\frac{(-1)^{i+j}}{DetA} * \frac{\partial DetA}{\partial x_{i,j}}\right]$$

et il est possible de calculer toutes les dérivées partielles de la fonction calculée par un circuit qui a la même complexité. Pour dériver par rapport à une seule variable, le circuit est modifié pour calculer la dérivée en même temps que la fonction, en appliquant les règles usuelles de dérivation : pour chaque nœud $+\ v$, sa dérivée étant la somme des dérivées des fils de v, on crée un nœud v' qui additionne toutes ces dérivées ; pour chaque nœud $*\ v$ qui a pour fils w_1 et w_2, $v' = w_1' * w_2 + w_1 * w_2'$, on crée le circuit partiel correspondant. Baur et Strassen [BS83, Str90], cités par Kaltofen dans [Kal88], ont montré qu'il est possible de calculer toutes les dérivées partielles avec un circuit ayant 4 fois plus de nœuds, celui-ci étant évalué avec la même complexité que le circuit initial. Il existe donc un circuit de calcul du déterminant et de l'inverse d'une matrice $n \times n$ à $n^{\mathcal{O}(1)}$ nœuds, qui est évalué en temps $\mathcal{O}(\log^2 n)$. Cette complexité temporelle est la même que celle de Chistov [Chi85], mais la transformation décrite ici, appliquée sur le circuit, n'est absolument pas log-uniforme (elle est P-uniforme), contrairement à la construction de Chistov.

3.7 Extensions

Miller et Teng [MT87] ont montré qu'il était possible d'adapter l'algorithme 13 pour traiter des structures plus variées. Les structures $(\mathbb{R}, \min, \max, +)$ ou les semi-anneaux unitaires intègres finis non-commutatifs entrent dans ce cadre.

3.7.1 $(\mathbb{R}, \min,\ \max,\ +)$

Ces opérations apparaissent fréquemment dans les problèmes de programmation dynamique et encore plus en intelligence artificielle, en théorie des jeux, d'autant plus qu'il est possible de remplacer sans problème $+$ par $*$, si l'ensemble de base est $\mathbb{R}^+$ ou tout autre ensemble de nombres positifs (si l'ensemble contient des nombres négatifs, $*$ cesse d'être distributive par rapport à min et à max : $(-1) * \min(2,3) \neq \min((-1)*2, (-1)*3)$).

min ou max est choisi – *a priori* arbitrairement – pour jouer le rôle de l'addition et l'autre comparateur ainsi que $*$ jouent le rôle de la multiplication. L'algorithme 13 ne combinant jamais les opérateurs $*$ entre eux, il est effectivement possible d'utiliser plusieurs « multiplications » différentes sans qu'elles interfèrent. En revanche, les additions étant combinées par l'opération *Group*, cela impose de n'avoir qu'une seule opération d'addition. Les fonctions d'arcs de la forme $\max(a + x, b)$ vérifient toutes les propriétés requises de stabilité par min et composition. Ces modifications conduisent à un nouvel algorithme, dont la complexité est encore $CREW(M(n), \log n \log nd)$, avec la définition du degré d également adaptée pour prendre en compte ces deux multiplications.

3.7.2 Semi-anneaux unitaires intègres finis non commutatifs

On se place dans un semi-anneau $(S, +, *, 0, 1)$ non-commutatif où S est fini. Le problème est de trouver les fonctions d'arcs qui permettent de prendre en compte la non-commutativité. Pour respecter la non-commutativité de l'opération $*$, il faut prendre garde, lors du *Shunt* d'un nœud $*$, à l'origine du fils et utiliser une composition qui respecte la place du fils. Les fonctions d'arcs créées sont du type $\sum_{i=1}^{k} a_i * x * b_i$, avec $k \leq Card(S)^2$: elles sont stables par addition et par composition. Le seul véritable problème vient de l'addition ; comme il n'y a pas plus de $Card(S)^2$ fonctions $a_i * x * b_i$ différentes, on ne peut pas en trouver plus de $Card(S)^2$ dans une somme et toute somme

$$f + g = \sum_{i=1}^{k_f} a_i * x * b_i + \sum_{i=1}^{k_g} c_i * x * d_i$$

s'écrira comme une somme avec $k \leq Card(S)^2$ termes. L'algorithme 13 s'étend donc, moyennant ces adaptations, au cas des semi-anneaux unitaires intègres non commutatifs finis.

Une application de ce résultat est la reconnaissance de langages hors-contexte, donnés par une grammaire sous forme normale de Chomsky éventuellement ambiguë, en temps $\mathcal{O}(\log^2 n)$. Pour cela, il faut reprendre les transformations présentées au chapitre 2 pour convertir l'algorithme de Cocke-Younger-Kasami en une expression. Comme l'hypothèse de non-ambiguïté a été levée, on fait se rencontrer les ensembles de non-terminaux selon tous les parenthésages possibles : si les ensembles de non-terminaux obtenus par le pré-traitement sont $X_1 * X_2 * X_3 \ldots$, le circuit correspondant contiendra les nœuds $(X_1 * X_2) * X_3$ et les nœuds $X_1 * (X_2 * X_3)$, ainsi que le nœud qui réalise l'union de ces ensembles.

L'opération d'union ensembliste joue le rôle de l'addition et l'opération $*$ joue le rôle de la multiplication. Ce circuit a $\mathcal{O}(n^2)$ nœuds et un degré linéaire, ce qui donne la complexité $CREW(M(n^2), \log^2 n)$ annoncée.

3.8 Conclusion

Dans ce chapitre, le modèle des circuits arithmétiques sur les semi-anneaux a été défini. L'algorithme d'évaluation des expressions arithmétiques peut être adapté afin d'évaluer ces circuits, mais il se révèle inefficace sur certains problèmes. Miller, Ramachandran et Kaltofen ont proposé un algorithme d'évaluation des circuits arithmétiques, dans lequel l'opération précédente de ratissage, rebaptisée *Shunt* dans ce contexte, ne s'applique plus qu'aux nœuds $*$. Une opération de regroupement des nœuds $+$ en un seul permet d'accélérer les calculs concernant ces nœuds. Il est démontré, par récurrence sur la taille du circuit, que la complexité de cet algorithme est $CREW(M(n), \log n \log nd)$ avec n le nombre de nœuds du circuit et d son degré.
Une extension au cas des corps est proposée, avec certaines conditions imposées sur les fonctions calculées par le circuit ; si ces conditions ne sont pas remplies, la technique d'élimination des divisions de Strassen permet de transformer le circuit initial avec divisions en un circuit sans division. Enfin, des extensions à d'autres structures algébriques, proposées par Miller et Teng, sont présentées.

Chapitre 4

Évaluation de circuits dans des treillis

Résumé

Dans ce chapitre, un nouvel algorithme est proposé pour l'évaluation parallèle des circuits à opérations dans un treillis distributif. Son efficacité réside dans l'exploitation intensive de la symétrie des opérations $\oplus$ et $\otimes$. Sa complexité est

$$CREW_L(M(n), \min[\log nd, h_a + \log n] \log n),$$

avec n le nombre de nœuds du circuit, d son degré et h_a le nombre maximal d'alternances de $\oplus$ et $\otimes$ sur un chemin du circuit. L'algorithme est explicité, sa complexité est démontrée puis l'exemple du tri par insertion est donné. Une adaptation au cas du treillis distributif complémenté $(\{Vrai, Faux\}, \vee, \wedge, Vrai, Faux, \neg)$ est ensuite discutée.

L'évaluation parallèle de circuits arithmétiques à opérations dans un treillis distributif est traitée dans ce chapitre. La structure de treillis distributif apparaît fréquemment en informatique, que ce soit en conception de circuits ou en arithmétique pour l'algèbre de Boole, ou en théorie des jeux avec les treillis distributifs dont les opérations sont min et max. L'algèbre de Boole est également utilisée comme structure de base dans les études théoriques de complexité et les circuits booléens en particulier constituent l'un des modèles de parallélisme (cf. chapitre 1). De plus, la structure algébrique de treillis distributif est riche de propriétés que les structures précédemment étudiées, comme les semi-anneaux ou les corps, ne possèdent pas : la double distributivité et l'idempotence (qui découle de l'absorption) des opérations jouent un grand rôle dans l'algorithme présenté dans ce chapitre.

Seuls Miller et Teng se sont intéressés au problème de l'évaluation des circuits sur un treillis distributif et ont proposé un algorithme qui consiste à exécuter l'algorithme 13 de Miller, Ramachandran et Kaltofen en alternant, à chaque application de la procédure *Phase*, l'utilisation de l'une ou l'autre loi du treillis comme multiplication. Le problème majeur posé par cet algorithme est la différence de traitement des deux lois, alors que dans un treillis elles ont exactement les mêmes propriétés.

Nous proposons un nouvel algorithme d'évaluation des circuits sur des treillis distributifs, qui est une généralisation de l'algorithme de Miller, Ramachandran et Kaltofen. Nous démontrons que la complexité de cet algorithme est

$$CREW_L(M(n), \log nd. \log n),$$

avec d le minimum des degrés calculés en considérant $\oplus$ puis $\otimes$ comme multiplication, si $\oplus$ et $\otimes$ sont les opérations du circuit. Nous introduisons ensuite une nouvelle mesure, h_a, le nombre d'alternances d'un circuit, qui est le nombre maximal d'alternances de $\oplus$ et de $\otimes$ sur un chemin du circuit et nous montrons que la complexité est également bornée par

$$CREW_L(M(n), [h_a + \log n] \log n).$$

Nous discutons la complexité de cet algorithme sur une $CRCW-PRAM$ si le treillis est totalement ordonné : le facteur $\log n$ de la complexité temporelle disparaît, éventuellement au prix d'un accroissement du nombre de processeurs. Nous illustrons cela sur l'exemple du tri par insertion, pour lequel la complexité temporelle est $\mathcal{O}(\log^2 n)$ sur une $CREW-PRAM$ et $\mathcal{O}(\log n)$ sur une $CRCW-PRAM$; la complexité des meilleurs algorithmes parallèles de tri est prédite automatiquement. Enfin, l'algorithme est adapté au cas du treillis de Boole $(\{Vrai,\ Faux\}, \vee, \wedge, Vrai,\ Faux, \neg)$.

4.1 Algorithmes existants

Un treillis distributif $(L, \oplus, \otimes, \epsilon, e)$ peut être considéré comme l'un des semi-anneaux unitaires $(L, \oplus, \otimes, \epsilon, e)$ ou $(L, \otimes, \oplus, e, \epsilon)$. Il est donc possible d'appliquer l'algorithme 13 en utilisant le treillis comme l'un de ces semi-anneaux. Pour savoir lequel de ces deux semi-anneaux choisir, il faut pré-calculer les degrés correspondant à chaque cas. Ce pré-calcul étant une tâche aussi complexe qu'évaluer le circuit, cette solution est à proscrire.

Il est également possible d'exécuter en parallèle deux versions de l'algorithme, une pour chaque semi-anneau. L'exécution qui se termine la première arrête l'autre ; par exemple, une exécution peut signaler sa fin en mettant à 1 un drapeau initialisé à 0 et consulté avant chaque application de la procédure *Phase* par l'autre calcul.

Après avoir discuté ces solutions, Miller et Teng [MT87] ont proposé un algorithme qui alterne les applications de $Phase_\oplus$ et $Phase_\otimes$, où $Phase_\oplus$ (resp. $Phase_\otimes$) est l'opération de *Phase* avec $\oplus$ (resp. $\otimes$) comme multiplication.

Dans tous les cas, ces algorithmes n'exploitent pas la symétrie des opérations $\oplus$ et $\otimes$. Leur complexité est

$$CREW_L(M(n), \log nd. \log n)$$

si $d = \min(d_\oplus, d_\otimes)$, où $d_\oplus$ (resp. $d_\otimes$) est le degré du circuit avec $\oplus$ (resp. $\otimes$) considéré comme multiplication.

4.2 Un nouvel algorithme

Les treillis distributifs $(L, \oplus, \otimes, \epsilon, e)$ sont des structures où les deux lois $\oplus$ et $\otimes$ possèdent les mêmes propriétés de distributivité, d'absorption et d'idempotence. Une idée naturelle est d'étendre chacune des procédures de l'algorithme 13 pour symétriser le traitement de ces deux opérations. Cela permettra par exemple de traiter les circuits de la figure 4.1 en temps polylogarithmique, bien que leur degré soit exponentiel.

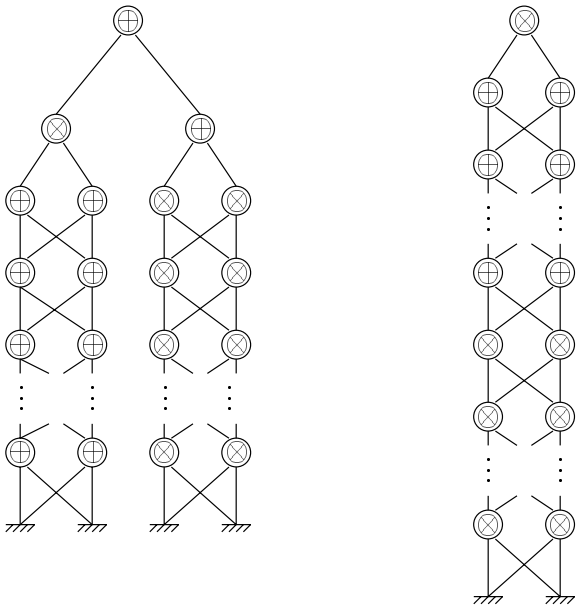


FIG. 4.1 - *Des circuits avec un degré exponentiel.*

4.2.1 Les fonctions d'arcs

Tout d'abord, il est clair que si l'on peut « shunter » à la fois les nœuds $\oplus$ et $\otimes$, les fonctions d'arcs vont être affines : $(a \oplus x) \otimes b$ ou $(a \otimes x) \oplus b$, ces deux notations étant équivalentes : $(a \otimes x) \oplus b = (a \oplus b) \otimes (x \oplus b) = (b \oplus x) \otimes c$ avec $c = a \oplus b$. Choisissons-en une au hasard, par exemple la deuxième, qui est plus conforme à la représentation habituelle d'une fonction affine : $f(x) = (a \otimes x) \oplus b$. Ces fonctions sont stables par composition, par $\oplus$

et par $\otimes$:

$$\begin{array}{lll}
\text{si} & f(x) & = (a \otimes x) \oplus b \\
\text{et} & g(x) & = (c \otimes x) \oplus d, \\
 & & \\
\text{alors} & (f \circ g)(x) & = [a \otimes ([c \otimes x] \oplus d)] \oplus b \\
 & & = (a \otimes [c \otimes x]) \oplus (a \otimes d) \oplus b \\
 & & = ([a \otimes c] \otimes x) \oplus ([a \otimes d] \oplus b), \\
 & & \\
 & (f \oplus g)(x) & = [(a \otimes x) \oplus b] \oplus [(c \otimes x) \oplus d] \\
 & & = ([a \oplus c] \otimes x) \oplus (b \oplus d), \\
 & & \\
 & (f \otimes g)(x) & = [(a \otimes x) \oplus b] \otimes [(c \otimes x) \oplus d] \\
 & & = ([a \otimes x] \otimes [c \otimes x]) \oplus ([a \otimes x) \otimes d) \\
 & & \oplus (b \otimes [c \otimes x]) \oplus (b \otimes d) \\
\text{(idempotence)} & & = ([(a \otimes c) \oplus (a \otimes d) \oplus (b \otimes c)] \otimes x) \oplus (b \otimes d).
\end{array}$$

4.2.2 L'opération de regroupement : *Group*

Tout comme des nœuds + successifs sont regroupés en un seul nœud + n-aire dans l'opération *Group* de l'algorithme 13, les nœuds $\oplus$ et $\otimes$ peuvent être regroupés. Cela impose tout d'abord que les nœuds $\otimes$ tout comme les nœuds $\oplus$ puissent être d'arité quelconque. Ensuite, les regroupements de nœuds vont encore s'effectuer à l'aide de produits de matrices ; pour cela il faut tenir compte des fonctions affines d'arcs et les produits de matrices sont modifiés en conséquence. Soit U la matrice d'adjacence du circuit, quatre matrices sont définies à partir de U :

– $U^{\oplus\oplus}$ est la matrice d'adjacence des nœuds $\oplus$:

$$U^{\oplus\oplus}_{i,j} = \begin{cases} U_{i,j} \text{ si } i \text{ et } j \text{ sont des nœuds } \oplus, \\ \epsilon \text{ sinon ;} \end{cases}$$

– $U^{\oplus\cdot}$ est la matrice des arcs connectant un nœud $\oplus$ à un nœud non $\oplus$:

$$U^{\oplus\cdot}_{i,j} = \begin{cases} U_{i,j} \text{ si } i \text{ est un nœud } \oplus \text{ et } j \text{ est un nœud } \otimes \text{ ou une feuille,} \\ \epsilon \text{ sinon ;} \end{cases}$$

– $U^{\otimes\otimes}$ est la matrice d'adjacence des nœuds $\otimes$:

$$U^{\otimes\otimes}_{i,j} = \begin{cases} U_{i,j} \text{ si } i \text{ et } j \text{ sont des nœuds } \otimes, \\ e \text{ sinon ;} \end{cases}$$

– $U^{\otimes\cdot}$ est la matrice des arcs connectant un nœud $\otimes$ à un nœud non $\otimes$:

$$U^{\otimes\cdot}_{i,j} = \begin{cases} U_{i,j} \text{ si } i \text{ est un nœud } \otimes \text{ et } j \text{ est un nœud } \oplus \text{ ou une feuille,} \\ e \text{ sinon.} \end{cases}$$

Ces quatre matrices forment une partition de la matrice d'adjacence U. Pour regrouper les nœuds $\oplus$ (resp. $\otimes$), on effectue un pas de calcul de fermeture transitive sur les sous-graphes engendrés par ces nœuds :

$$\left\{ \begin{array}{lcl} U^{\oplus\oplus} & \longleftarrow & U^{\oplus\oplus}.U^{\oplus\oplus}, \\ U^{\oplus\cdot} & \longleftarrow & U^{\oplus\oplus}.U^{\oplus\cdot} + U^{\oplus\cdot}, \end{array} \right.$$

avec comme produit de matrices

$$(A.B)_{i,j} = \bigoplus_{k=1}^{n} (A_{i,k} \circ B_{k,j})(x)$$

pour regrouper les nœuds $\oplus$ et

$$\left\{ \begin{array}{lcl} U^{\otimes\otimes} & \longleftarrow & U^{\otimes\otimes}.U^{\otimes\otimes}, \\ U^{\otimes\cdot} & \longleftarrow & U^{\otimes\otimes}.U^{\otimes\cdot} + U^{\otimes\cdot}, \end{array} \right.$$

avec comme produit de matrices

$$(A.B)_{i,j} = \bigotimes_{k=1}^{n} (A_{i,k} \circ B_{k,j})(x)$$

pour regrouper les nœuds $\otimes$. Cette procédure de regroupement, appelée *Group*, a par conséquent la même complexité qu'un produit de matrices dans la structure considérée, soit

$$CREW_L(M(n), \log n)$$

si $M(n)$ est le nombre minimum de processeurs nécessaires pour effectuer un produit de matrices modifié dans L, sur une $CREW-PRAM$, en temps $\mathcal{O}(\log n)$. Une borne classique est $M(n) = \mathcal{O}(n^3)$.

Dans certains treillis, il est possible d'effectuer ces produits de matrices en temps constant sur une $CRCW-PRAM$, au prix éventuellement d'un accroissement du nombre de processeurs. Si $M'(n)$ est ce nombre de processeurs, la complexité de *Group* est $CRCW_L(M'(n), 1)$. Plus précisément, si le treillis considéré est un ensemble totalement ordonné, *i.e.* $\forall (a,b) \in L^2$, $a \oplus b \in \{a,b\}$ et $a \otimes b \in \{a,b\}$, alors les produits de matrices de l'opération *Group* peuvent être effectués en temps constant avec $\mathcal{O}(n^4)$ processeurs : pour calculer

$$C = A \times B,\ C = (C_{i,j})_{1 \le i,j \le n} = (\bigodot_{k=1}^{n} A_{i,k}.B_{k,j})_{1 \le i,j \le n}$$

avec $\odot = \oplus$ ou $\otimes$ et $. = \oplus$, $\otimes$ ou $\circ$, on affecte n^2 processeurs au calcul de chaque coefficient $C_{i,j}$; tout d'abord tous les $C_{i,j,k} = A_{i;k}.B_{k,j}$ sont calculés en parallèle avec n processeurs, ensuite on cherche à déterminer « le plus grand » des $C_{i,j,k}$ si $\odot = \oplus$, « le plus petit » sinon. Pour cela, chacun des n^2 processeurs se charge d'un couple $(C_{i,j,k_1}, C_{i,j,k_2})$ et écrit 1 en case mémoire k_2 (resp. k_1) si $C_{i,j,k_1} \odot C_{i,j,k_2} = C_{i,j,k_1}$ (resp. $C_{i,j,k_1} \odot C_{i,j,k_2} = C_{i,j,k_2}$), pour signaler que C_{i,j,k_2} (resp. C_{i,j,k_1}) n'est pas le résultat cherché. Les cases mémoires sont

supposées être initialisées à 0. Ensuite les cases mémoire k contenant la valeur 0 écrivent $C_{i,j,k}$ dans la case $C_{i,j}$. Cela fonctionne quel que soit le modèle $CRCW-PRAM$ utilisé ; en particulier cet algorithme est valable pour une $COMMUNE\ CRCW-PRAM$: lors du premier pas, les processeurs qui écrivent en mémoire écrivent tous 1, ensuite les cases mémoire qui contiennent 0 correspondent au résultat $C_{i,j}$; comme ce résultat est unique, elles écrivent toutes la même valeur. La complexité de l'opération *Group* est donc sur une $CRCW-PRAM$

$$CRCW_L(M'(n), 1).$$

Dans l'algèbre de Boole $(\{Vrai, Faux\}, \vee, \wedge, Vrai, Faux)$, les produits de matrices peuvent même s'effectuer en temps constant sans augmentation du nombre de processeurs : par exemple si $\odot = \vee$, les processeurs qui ont calculé un produit $A_{i,k}.B_{k,j}$ n'ont qu'à écrire $Vrai$ dans $C_{i,j}$ si $A_{i,k}.B_{k,j} = Vrai$ et rien sinon, les cases étant initialisées à $Faux$.

4.2.3 L'opération d'évaluation : *Éval*

Cette procédure est la plus naturelle, elle n'est pas modifiée par rapport à l'algorithme 13 et elle peut encore se décomposer en une évaluation des nœuds $\oplus$ et une évaluation des nœuds $\otimes$, ces deux procédures étant effectuées en parallèle.

Algorithme 14 $\acute{E}val_{\oplus}$

si v, un nœud $\oplus$, a m fils tous feuilles (w_i pour $i \in [1, \ldots m]$) alors
  $val(v) \leftarrow \bigoplus_{i=1}^{n} U_{v,w_i}(val(w_i))$
  pour $i = 1$ à m faire
    $U_{v,w_i} \leftarrow \epsilon$
  v devient une feuille

fin

Algorithme 15 $\acute{E}val_{\otimes}$

si v, un nœud $\otimes$, a m fils tous feuilles (w_i pour $i \in [1, \ldots m]$) alors
  $val(v) \leftarrow \bigotimes_{i=1}^{m} U_{v,w_i}(val(w_i))$
  pour $i = 1$ à m faire
    $U_{v,w_i} \leftarrow e$
  v devient une feuille

fin

Algorithme 16 *Éval*

en parallèle faire
  $\acute{E}val_{\oplus}$
  $\acute{E}val_{\otimes}$

fin

Cette procédure effectue, au pire, un calcul de produit itéré sur chaque nœud, sa complexité est par conséquent

$$CREW_L(n^2, \log n).$$

Il s'agit d'une majoration grossière, mais suffisante puisqu'ici encore la complexité de l'opération *Group* domine.

Dans un treillis totalement ordonné, la technique de calcul d'un $\oplus$ (ou d'un $\otimes$) de n termes en temps constant présentée pour *Group* peut encore être utilisée. La complexité de la procédure *Éval* est dans ce cas

$$CRCW_L(n^3, 1).$$

4.2.4 L'opération d'évaluation partielle : *ÉvalPartiel*

Il reste désormais à adapter la procédure de *Shunt* de l'algorithme de Miller, Ramachandran et Kaltofen. Le problème est que les nœuds $\oplus$ comme les nœuds $\otimes$ ne sont pas binaires. Il est cependant possible de « shunter » un nœud n-aire, dès qu'un seul de ses fils n'est pas une feuille. Pour faciliter la tâche, l'adaptation de l'opération de *Shunt* est réalisée en deux temps. Tout d'abord, chaque nœud qui a des fils feuilles calcule son résultat partiel à partir de leur valeur et met le résultat sur un seul des arcs vers ses fils. Il paraît judicieux d'effectuer cette évaluation partielle pour tous les nœuds et non seulement pour les nœuds n'ayant qu'un seul fils non feuille, pour homogénéiser le traitement des nœuds opérateurs. Il est ensuite facile de déterminer si un nœud a un seul ou plusieurs fils non feuilles. Reste à déterminer sur quel arc stocker le résultat de l'évaluation partielle. Comme cette opération tend à supprimer les fils feuilles, une stratégie possible est de stocker le résultat sur l'un des arcs vers les fils non feuilles s'il en existe, ou sur tous les arcs pour ne pas avoir de problème de choix (grâce à l'idempotence des opérations $\oplus$ et $\otimes$) ; si tous les fils sont des feuilles, un arc vers un nœud artificiel réservé à cet usage, le nœud de numéro 0 ou -1 par exemple, est créé pour conserver le résultat partiel.

L'algorithme d'évaluation partielle, appelé *ÉvalPartiel*, réalise cela :

Algorithme 17 *ÉvalPartiel*

pour tous les nœuds internes v faire en parallèle
{ On calcule le résultat partiel. }
$ResPartiel \leftarrow \bigoplus_{w=1}^{n} U_{v,w}(val(w))$ si v est un nœud $\oplus$
$ResPartiel \leftarrow \bigotimes_{w=1}^{n} U_{v,w}(val(w))$ si v est un nœud $\otimes$

{ On détruit les arcs de v vers ses fils feuilles. }

$$U_{v,w} \leftarrow \begin{cases} \epsilon & \text{si } v \text{ est un nœud } \oplus \text{ et } w \text{ est une feuille,} \\ e & \text{si } v \text{ est un nœud } \otimes \text{ et } w \text{ est une feuille.} \end{cases}$$

{ On compose le résultat partiel avec les arcs de v vers ses fils non feuilles. }

$$U_{v,w} \leftarrow \begin{cases} U_{v,w} \oplus ResPartiel & \text{si } v \text{ est un nœud } \oplus, \\ & w \text{ n'est pas une feuille et} \\ & U_{v,w} \neq \epsilon, \\ U_{v,w} \otimes ResPartiel & \text{si } v \text{ est un nœud } \otimes, \\ & w \text{ n'est pas une feuille et} \\ & U_{v,w} \neq e. \end{cases}$$

fin

Cette procédure est schématisée sur la figure 4.2.

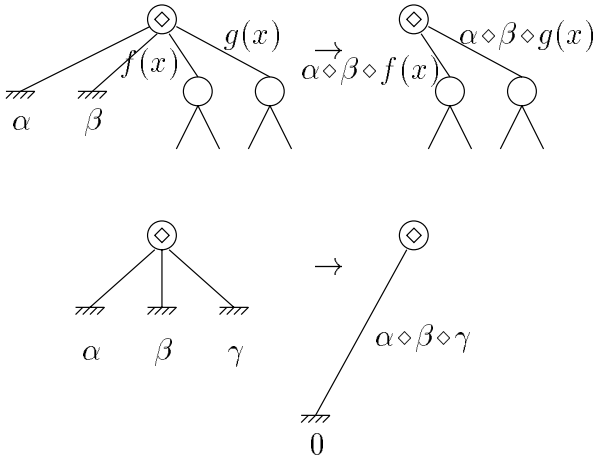


FIG. 4.2 - *Évaluation partielle.*

La mise à jour des arcs peut s'effectuer en temps constant avec $\mathcal{O}(n^2)$ processeurs, puisque le nombre d'arcs est inférieur à $\mathcal{O}(n^2)$, qui est le nombre d'arcs d'un circuit complet. Cette procédure a donc la même complexité que la procédure *Éval*, c'est-à-dire

$$CREW_L(n^2, \log n)$$

et

$$CRCW_L(n^3, 1)$$

si le treillis est totalement ordonné.

4.2.5 L'opération de suppression des nœuds unaires : *Suppress*

Enfin, il est possible de « shunter » les nœuds $\oplus$ et $\otimes$ si, après l'opération d'évaluation partielle, ces nœuds sont devenus unaires. Une technique de saut de pointeur (cf. §1.4.5) appliquée à l'ensemble du circuit permet de supprimer tous les nœuds unaires du circuit, sauf

éventuellement les nœuds de sortie. Il est particulièrement important de supprimer les nœuds unaires du circuit parce qu'ils introduisent de façon totalement artificielle une séquentialité dans le circuit : une chaîne de nœuds unaires $\oplus$ et $\otimes$ alternés étant de hauteur 1, sa suppression est impérative.

L'opération de suppression des nœuds unaires, *Suppress*, s'écrit :

Algorithme 18 *Suppress*

tant qu'il reste des nœuds unaires faire
pour tous les nœuds internes v faire en parallèle
pour tous les fils w de v faire en parallèle
si w est unaire et a pour fils x alors

$$U_{v,x} \leftarrow \begin{cases} U_{v,x} \oplus (U_{v,w} \circ U_{w,x}) & \text{si } v \text{ est un nœud } \oplus, \\ U_{v,x} \otimes (U_{v,w} \circ U_{w,x}) & \text{si } v \text{ est un nœud } \otimes. \end{cases}$$

$$U_{v,w} \leftarrow \begin{cases} \epsilon & \text{si } v \text{ est un nœud } \oplus, \\ e & \text{si } v \text{ est un nœud } \otimes. \end{cases}$$

fin

Cette procédure a pour complexité $CREW_L(n^2, \log n)$, puisque le nombre de passages dans l'instruction d'itération *tant qu'il reste des nœuds unaires faire* est majoré par $\log n$. Sur une $CRCW-PRAM$ il est possible de tester en temps constant s'il reste des nœuds unaires : chaque nœud teste son nombre de fils (1 ou plus), écrit 1 dans une case spéciale de la mémoire s'il est unaire et le processus sera répété tant qu'il y aura un 1 dans cette case (supposée être remise à 0 au début de chaque itération).

La procédure *Suppress* aura pour complexité *pour l'ensemble de l'évaluation du circuit* $CRCW_L(n^2, \log n)$.

4.2.6 L'algorithme d'évaluation

L'algorithme d'évaluation est constitué de l'application, autant de fois qu'il le faut, de la séquence *Group* – *Éval* – *ÉvalPartiel* – *Suppress*. Cette séquence portera encore le nom de *Phase*, par analogie avec l'algorithme 13. Cela pourrait suffire mais, pour les besoins du calcul de la complexité, un pré-traitement est effectué sur le circuit pendant lequel une fermeture transitive complète des nœuds $\oplus$ et $\otimes$, assortie de la suppression de tous les nœuds unaires, est réalisée à l'aide d'une opération *Group** qui consiste en $\lceil \log n \rceil$ applications de la procédure *Group*, suivie de *Suppress*.

L'algorithme s'écrit alors :

Algorithme 19 *Évaluation des circuits sur un treillis*

pré-traitement
    $Group^* = \lceil \log n \rceil$ *Group*
    Suppress
tant qu'il reste des nœuds non évalués répéter
Phase
    Group
    Éval
    ÉvalPartiel
    Suppress

fin

La complexité du pré-traitement est bornée par celle de $\lceil \log n \rceil$ applications de *Phase*. La procédure *Phase* a pour complexité

$$CREW_L(M(n), \log n)$$

et

$$CRCW_L(M'(n), 1)$$

si le treillis est totalement ordonné.

Ici encore, le problème est de déterminer le nombre d'applications de la procédure *Phase*. Avant le calcul de cette complexité, illustrons l'algorithme 19 par un exemple.

4.3 Exemple

Pour l'exemple de la figure 4.3, les feuilles sont repérées par des lettres grecques et les fonctions d'arcs (même constantes) seront notées par des lettres latines. Le pré-traitement n'est pas appliqué sur cet exemple.

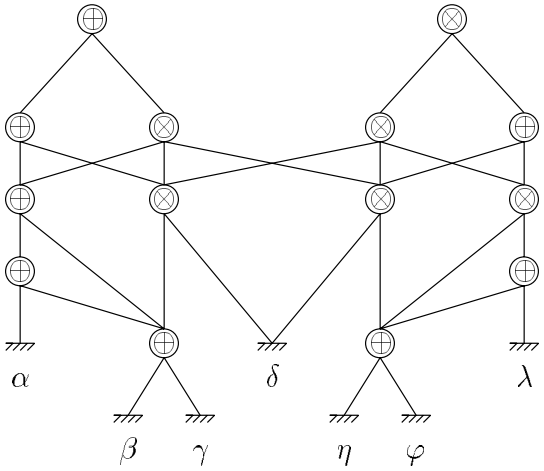


FIG. 4.3 - *Exemple.*

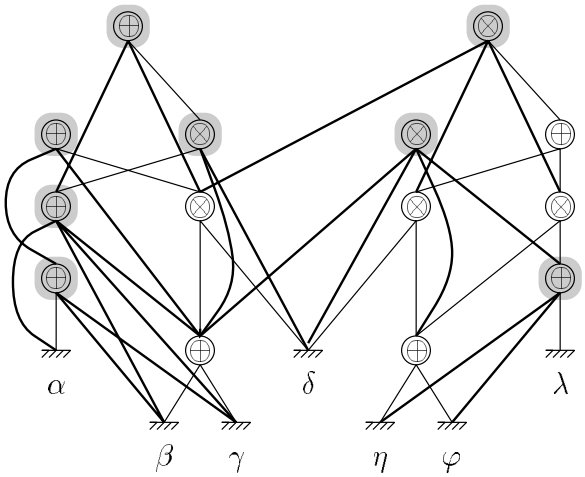
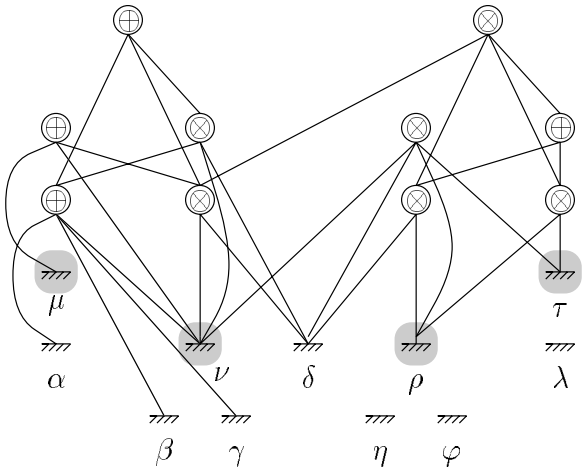


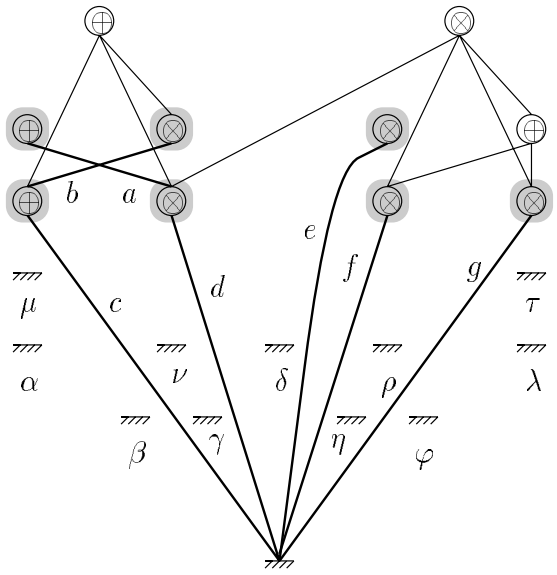
FIG. 4.4 - *Après l'opération Group.*



$\mu = \alpha \oplus \beta \oplus \gamma$ $\qquad$ $\rho = \eta \oplus \varphi$

$\nu = \beta \oplus \gamma$ $\qquad$ $\tau = \eta \oplus \varphi \oplus \lambda$

FIG. 4.5 - *Après l'opération Éval.*



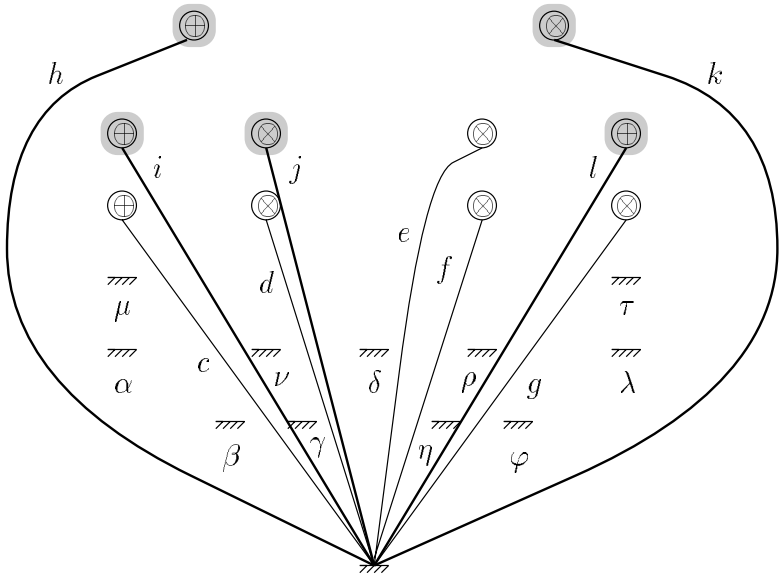
$a(x) = \alpha \oplus \beta \oplus \gamma \oplus x$ $\qquad$ $e(x) = (\beta \oplus \gamma) \otimes \delta \otimes (\eta \oplus \varphi)$

$b(x) = (\beta \oplus \gamma) \otimes \delta \otimes x$ $\qquad$ $f(x) = \delta \otimes (\eta \oplus \varphi)$

$c(x) = \alpha \oplus \beta \oplus \gamma \oplus \nu$ $\qquad$ $g(x) = \eta \oplus \varphi$

$d(x) = (\beta \oplus \gamma) \otimes \delta$

FIG. 4.6 - *Après l'opération Évalpartiel.*



$h(x) = \alpha \oplus \beta \oplus \gamma$ $\qquad$ $k(x) = (\beta \oplus \gamma) \otimes \delta \otimes (\eta \oplus \varphi)$

$i(x) = \alpha \oplus \beta \oplus \gamma$ $\qquad$ $l(x) = \eta \oplus \varphi$

$j(x) = (\beta \oplus \gamma) \otimes \delta$

FIG. 4.7 - *Après l'opération Suppress.*

Tout d'abord, les nœuds grisés ont regroupé leurs fils et leurs petits-fils grâce à l'opération *Group* (figure 4.4). Les nœuds dont tous les fils étaient des feuilles ont alors calculé leur valeur avec l'opération *Éval* (figure 4.5). Les nœuds qui avaient des fils feuilles ont ensuite calculé leur résultat partiel (opération *ÉvalPartiel*) et l'ont stocké sur un arc vers un fils non feuille s'ils en ont, sur un arc vers un nœud artificiel (au centre de la figure) sinon (figure 4.6). Après *Suppress* qui a déconnecté les nœuds unaires, tous les nœuds n'ont qu'un seul fils, qui est la feuille artificielle. Une seule application de *Éval* suffira pour que ce circuit soit totalement évalué. Deux applications de *Phase* suffisent donc pour évaluer ce circuit.

4.4 Complexité

Tout d'abord, la technique de preuve du §3.4 permet de montrer que le nombre d'applications de la procédure *Phase* est borné par $\log nd$, où d est le minimum des degrés $d_\oplus$ et $d_\otimes$ du circuit. Cependant cette borne est beaucoup trop large pour les circuits de la figure 4.1, qui sont évalués en au plus quatre applications de *Phase*. Nous introduisons une quantité h_a, le nombre d'alternances du circuit, qui permet d'affiner cette borne : le nombre d'itérations est également majoré par $h_a + \log n$, or h_a vaut 4 pour les circuits de degré exponentiel de la figure 4.1.

Définition 40 *Soit un circuit à opérations dans un treillis* $(L, \oplus, \otimes, \epsilon, e)$. *Le degré formel* d *de ce circuit est le minimum des degrés* $d_\oplus$ *et* $d_\otimes$, *avec*

$$d_\oplus(v) = \begin{cases} 1 & \textit{si } v \textit{ est une feuille,} \\ \sum_w d_\oplus(w \textit{ fils de } v) & \textit{si } v \textit{ est un nœud } \oplus, \\ \max_w(d_\oplus(w \textit{ fils de } v)) & \textit{si } v \textit{ est un nœud } \otimes, \end{cases}$$

$$d_\otimes(v) = \begin{cases} 1 & \textit{si } v \textit{ est une feuille,} \\ \sum_w d_\otimes(w \textit{ fils de } v) & \textit{si } v \textit{ est un nœud } \otimes, \\ \max_w(d_\otimes(w \textit{ fils de } v)) & \textit{si } v \textit{ est un nœud } \oplus\,. \end{cases}$$

4.4.1 Première majoration

Cette première majoration s'obtient en reprenant la preuve 3.4. Soit un circuit comportant n nœuds, qui est évalué par l'algorithme 19. Tout d'abord, montrons que le nombre d'applications de *Phase* est majoré par $\log nd = \log n + \log d$. Pour cela, deux hauteurs correspondant à la hauteur du §3.4, $h_\oplus$ et $h_\otimes$, sont définies pour chaque nœud du circuit et les hauteurs $h_\oplus$ et $h_\otimes$ d'un circuit sont les maxima des hauteurs correspondantes de ses nœuds. Ces hauteurs vérifient deux propriétés essentielles pour le calcul de la complexité :

$$h_\oplus \leq \frac{1}{2}n^2 d_\oplus \text{ et } h_\otimes \leq \frac{1}{2}n^2 d_\otimes$$

et elles sont divisées par 2 par chaque application de l'opération *Phase*.

Définition 41 *$h_\oplus$ est définie pour tout nœud v du circuit par :*

$$\begin{aligned} h_\oplus(v) &= 1 \text{ si } v \text{ est une feuille,} \\ h_\oplus(v) &= \max_w \begin{pmatrix} h_\oplus(w \text{ fils } \oplus \text{ de } v) + \frac{1}{2}, \\ h_\oplus(w \text{ fils } \otimes \text{ de } v), \\ h_\oplus(w \text{ fils feuille de } v) \end{pmatrix} \text{ si } v \text{ est un nœud } \oplus \end{aligned}$$

et

$$h_\oplus(v) = \textstyle\sum_w h_\oplus(w \text{ fils de } v) \text{ si } v \text{ est un nœud } \otimes .$$

Un fils dominant *w d'un nœud $\oplus$ v est un fils de v tel que $h_\oplus(v) = h_\oplus(w) + \frac{1}{2}$ si w est un nœud $\oplus$, $h_\oplus(v) = h_\oplus(w)$ sinon.*

Définition 42 *$h_\otimes$ est définie pour tout nœud v du circuit par :*

$$\begin{aligned} h_\otimes(v) &= 1 \text{ si } v \text{ est une feuille,} \\ h_\otimes(v) &= \max_w \begin{pmatrix} h_\otimes(w \text{ fils } \otimes \text{ de } v) + \frac{1}{2}, \\ h_\otimes(w \text{ fils } \oplus \text{ de } v), \\ h_\otimes(w \text{ fils feuille de } v) \end{pmatrix} \text{ si } v \text{ est un nœud } \otimes \end{aligned}$$

et

$$h_\otimes(v) = \textstyle\sum_w h_\otimes(w \text{ fils de } v) \text{ si } v \text{ est un nœud } \oplus .$$

Un fils dominant *w d'un nœud $\otimes$ v est un fils de v tel que $h_\otimes(v) = h_\otimes(w) + \frac{1}{2}$ si w est un nœud $\otimes$, $h_\otimes(v) = h_\otimes(w)$ sinon.*

Définition 43 *La hauteur $h_\oplus$ d'un circuit est le maximum des hauteurs $h_\oplus$ de ses nœuds. La hauteur $h_\otimes$ d'un circuit est le maximum des hauteurs $h_\otimes$ de ses nœuds.*

Remarque
Les circuits considérés par la suite sont supposés avoir subi l'étape de pré-traitement. Ceci permet d'assurer, dans la première partie du calcul de complexité, qu'il n'y a pas de nœuds unaires dans le circuit et, dans la seconde partie, que tous les nœuds $\oplus$ et $\otimes$ ont été regroupés.

Les procédures *Phase*, *Group*, *Éval*, *ÉvalPartiel* et *Suppress* peuvent être considérées comme des fonctions sur les circuits, qui laissent inchangés les sommets (mis à part le fait qu'une étiquette « nœud $\oplus$ » ou « nœud $\otimes$ » peut se transformer en « feuille »), mais qui modifient la structure des arcs. On parlera donc de l'image d'un nœud ou d'un circuit par *Phase* et, de manière un peu abusive, de leur antécédent par *Phase*. Dans ce dernier cas, il existe un circuit, sur lequel la procédure *Phase* a été appliquée, qui fait référence. On notera désormais v' l'image d'un nœud v et U' l'image d'un circuit U. Par convention, w' désignera de façon générique l'un des fils d'un nœud v' et w son antécédent par *Phase*.

Si v est un nœud $\otimes$ et x un de ses fils est un nœud $\oplus$, si de plus l'image x' de x n'est pas un fils de v' l'image de v, le seul cas de figure possible est que x soit devenu un nœud

unaire après *ÉvalPartiel* et qu'il ait été déconnecté de v par *Suppress*. L'arc v–x a donc été remplacé par un seul arc v–w où w est le seul fils de x après *ÉvalPartiel*. Cette remarque justifie le lemme suivant :

Lemme 3 *Si v' est un nœud $\otimes$, sa hauteur est*

$$h_\oplus(v') = \sum_{w'} h_\oplus(w' \textit{ fils de } v')$$

et la hauteur de son antécédent v est

$$h_\oplus(v) \geq \sum_{w} h_\oplus(w \textit{ antécédent de } w').$$

Échanger les signes $\oplus$ et $\otimes$ conduit à un résultat vrai également.

Le lemme qui suit est essentiel pour conclure à la première majoration de la complexité de cet algorithme.

Lemme 4 *Si U est un circuit et U' est son image par Phase, si v' est un nœud de U' qui n'est ni une feuille ni un nœud de sortie (un nœud sans père) et si v est son antécédent dans U, alors*

$$\begin{cases} h_\oplus(v) & \geq & 2h_\oplus(v'), \\ h_\otimes(v) & \geq & 2h_\otimes(v'). \end{cases}$$

DÉMONSTRATION
Comme les rôles et le traitement des opérations $\oplus$ et $\otimes$ sont symétriques, seul le cas de $h_\oplus$ sera traité. La démonstration se fait par récurrence sur la taille de $U'_{v'}$ le sous-circuit de U' qui calcule v', avec v' qui n'est ni une feuille ni un nœud de sortie. $U'_{v'}$ est le sous-graphe de U' engendré par l'ensemble des nœuds x' de U' pour lesquels un chemin allant de v' à x' existe. Le cas des feuilles et des nœuds de sortie est exclu parce qu'il n'est pas certain que leurs hauteurs soient divisées par 2 par *Phase*, du fait de leur position extrême dans le circuit.

INITIALISATION DE LA RÉCURRENCE
Le cas initial de la récurrence est celui d'un nœud v' n'ayant que des fils feuilles. Soit v l'antécédent de v', montrons que $h_\oplus(v) \geq 2h_\oplus(v')$:

- si v' est un nœud $\oplus$, v est également un nœud $\oplus$ et la hauteur $h_\oplus(v')$ de v' vaut 1. Montrons par l'absurde que $h_\oplus(v) \geq 2$. Si $h_\oplus(v)<2$, les seules possibilités sont $h_\oplus(v) = 1$ ou $h_\oplus(v) = \frac{3}{2}$:
  - si $h_\oplus(v) = 1$, alors v est un nœud $\oplus$ dont tous les fils sont des feuilles puisqu'il n'y a pas de nœuds unaires. Après *Éval*, v est devenu une feuille et v' est une feuille également, ce qui est contraire à l'hypothèse.
  - si $h_\oplus(v) = \frac{3}{2}$, alors v est un nœud $\oplus$ dont les fils dominants sont des nœuds $\oplus$ qui n'ont que des fils feuilles, ses autres fils étant des feuilles, toujours grâce à l'absence de nœuds unaires. Après *Group*, v n'a donc que des fils feuilles et après *Éval* v est une feuille. v' est donc également une feuille, ce qui contredit encore l'hypothèse sur v'.

Il vient donc d'être montré par l'absurde que si v' est un nœud $\oplus$ qui n'a que des fils feuilles, son antécédent v vérifie :

$$h_{\oplus}(v) \geq 2h_{\oplus}(v').$$

– si v' est un nœud $\otimes$ dont tous les fils sont des feuilles, alors

$$h_{\oplus}(v') = \sum_{w' \text{ fils de } v'} h_{\oplus}(w') = \sum_{w' \text{ fils de } v'} 1 = Card\{w' \text{ fils de } v'\}.$$

Il suffit de vérifier que chaque antécédent w de w' fils de v' a une hauteur $h_{\oplus}$ supérieure ou égale à 2, le lemme 3 permettra alors de conclure. Montrons-le par l'absurde. Supposons que $h_{\oplus}(w)<2$ avec w antécédent de w'. Comme dans le cas précédent, w est un nœud $\oplus$ avec $h_{\oplus}(w) = 1$ ou $h_{\oplus}(w) = \frac{3}{2}$ et w se retrouve être une feuille après *Éval*. Mais alors *ÉvalPartiel* déconnecte v de w et w' ne peut donc pas être fils de v'. $h_{\oplus}(w)$ ne pouvant pas être strictement inférieure à 2, on a donc, grâce au lemme 3,

$$h_{\oplus}(v) \geq \sum_{w} h_{\oplus}(w \text{ antécédent de } w') \geq 2\sum_{w'} h_{\oplus}(w') = 2h_{\oplus}(v').$$

Le cas initial de la récurrence étant établi, passons au cas général.

CAS GÉNÉRAL

L'hypothèse de récurrence est que pour tout circuit $U'_{w'}$ de taille inférieure à k, tel que w' n'est ni une feuille ni un nœud de sortie, la hauteur $h_{\oplus}(w)$ de w l'antécédent de w' est divisée par 2 par *Phase* : $h_{\oplus}(w) \geq 2h_{\oplus}(w')$.

Soit v un nœud de U tel que $U'_{v'}$ comporte $k+1$ nœuds et que v' l'image de v ne soit ni une feuille ni un nœud de sortie, montrons que la hauteur $h_{\oplus}$ de v est divisée par 2 par *Phase* : $h_{\oplus}(v) \geq 2h_{\oplus}(v')$:

– si v est un nœud $\oplus$, son image v' est également un nœud $\oplus$. Soit w' le fils dominant de v' et soit w l'antécédent de w. w' n'est pas un nœud de sortie puisqu'il a v' pour père.

  – si w' est une feuille, on se retrouve dans le cas initial de la récurrence.
  – si w' est un nœud $\otimes$, alors $h_{\oplus}(v') = h_{\oplus}(w')$ et $2h_{\oplus}(w') \leq h_{\oplus}(w)$ par hypothèse de récurrence. Comme w est un descendant de v, $h_{\oplus}(v) \geq h_{\oplus}(w)$ et donc

$$h_{\oplus}(v) \geq 2h_{\oplus}(v').$$

  – si w' est un nœud $\oplus$, $h_{\oplus}(v') = h_{\oplus}(w') + \frac{1}{2}$; il suffit de montrer qu'il existe un chemin de longueur au moins 2 connectant v à w. Si ce chemin n'existait pas, v et w seraient deux nœuds $\oplus$ adjacents et la procédure *Group* les aurait déconnectés. Il existe donc un nœud x sur un chemin entre v et w. Si x est un nœud $\oplus$, alors $h_{\oplus}(v) \geq h_{\oplus}(x) + \frac{1}{2} \geq h_{\oplus}(w) + 1$. Si x est un nœud $\otimes$, x ne peut pas être un nœud unaire (ils sont déconnectés de leurs pères par la procédure *Suppress*), donc

$h_\oplus(x) \geq h_\oplus(w)+1$ et alors $h_\oplus(v) \geq h_\oplus(x) \geq h_\oplus(w)+1$. On a donc, en appliquant l'hypothèse de récurrence à w,

$$h_\oplus(v') = h_\oplus(w') + \frac{1}{2} \leq \frac{1}{2}[h_\oplus(w)+1] \leq \frac{1}{2}h_\oplus(v).$$

La propriété est vraie pour les nœuds $\oplus$.

– si v est un nœud $\otimes$, son image v' est également un nœud $\otimes$ et $h_\oplus(v') = \sum_{w'} h_\oplus(w'$ fils de $v')$. Grâce au lemme 3, on sait que $h_\oplus(v) \geq \sum h_\oplus(w$ antécédent de $w')$. Il reste à vérifier que $h_\oplus(w) \geq 2h_\oplus(w')$ pour chaque fils w' de v'. w' n'est pas un nœud de sortie puisqu'il a v' pour père. Si w' est un nœud $\oplus$ ou $\otimes$, l'hypothèse de récurrence s'applique :

$$h_\oplus(w) \geq 2h_\oplus(w').$$

Si w' est une feuille, il a été montré lors de l'initialisation de la récurrence que son antécédent w est de hauteur supérieure à 2, sinon il est déconnecté de v' ; donc $h_\oplus(w) \geq 2h_\oplus(w')$. Dans tous les cas,

$$h_\oplus(v') = \sum_{w' \text{ fils de } v'} h_\oplus(w') \leq \frac{1}{2} \sum_{w \text{ antécédent de } w'} h_\oplus(w) \leq \frac{1}{2}h_\oplus(v).$$

Nous avons montré par récurrence que pour tout nœud du circuit qui n'est pas transformé en feuille ou en nœud de sortie par la procédure *Phase*, sa hauteur est divisée par 2.

Il en est de même pour la hauteur $h_\otimes$, grâce à la symétrie des opérations $\oplus$ et $\otimes$ et des traitements qui leur sont appliqués. □

Définition 44 *La hauteur h d'un circuit est le minimum de ses hauteurs $h_\oplus$ et $h_\otimes$.*

$$h = \min(h_\oplus, h_\otimes).$$

Corollaire 3 *Chaque application de Phase sur un circuit divise sa hauteur h par 2.*

Ce corollaire permet de déduire qu'après $\lceil \log h \rceil$ applications de la procédure *Phase*, tout circuit de hauteur h est transformé en un circuit constitué exclusivement de feuilles et de nœuds de sortie. Une application de l'opération *Éval* suffit alors pour évaluer complètement ce circuit.

Pour relier la hauteur d'un circuit à d'autres grandeurs plus usuelles, établissons le lemme suivant :

Lemme 5 *Soit U un circuit de degrés formels $d_\oplus$ et $d_\otimes$ et qui possède $a_\oplus$ arcs connectant deux nœuds $\oplus$ et $a_\otimes$ arcs connectant deux nœuds $\otimes$, les hauteurs $h_\oplus$ et $h_\otimes$ de U vérifient*

$$\begin{cases} h_\oplus & \leq & \frac{1}{2}a_\oplus d_\oplus + d_\oplus, \\ h_\otimes & \leq & \frac{1}{2}a_\otimes d_\otimes + d_\otimes. \end{cases}$$

DÉMONSTRATION
Démontrons-le uniquement pour $h_\oplus$, une preuve analogue permettant de conclure pour $h_\otimes$. On le démontre par récurrence sur la taille de tout sous-circuit U_v, où v est un nœud du circuit U. Le cas initial étant celui où v est une feuille, on a

$$\begin{array}{rrcl} & h_\oplus(v) & = & 1, \\ & a_\oplus(v) & = & 0, \\ & d_\oplus(v) & = & 1 \\ \text{et} & 1 & \leq & \frac{1}{2} * 0 * 1 + 1. \end{array}$$

Si cette relation est vérifiée pour tout sous-circuit U_w de taille inférieuure ou égale à k, montrons qu'elle est vérifiée pour tout sous-circuit de taille $k+1$. Soit donc U_v un sous-circuit de U comportant $k+1$ nœuds. Si v a pour fils $v_1, v_2, \ldots v_m$, de hauteur $h_{\oplus_1}, h_{\oplus_2}, \ldots h_{\oplus_m}$, de degré $d_{\oplus_1}, d_{\oplus_2}, \ldots d_{\oplus_m}$ et dont les sous-circuits comportent respectivement $a_{\oplus_1}, a_{\oplus_2}, \ldots a_{\oplus_m}$ arcs $\oplus$–$\oplus$, l'hypothèse de récurrence indique que

$$h_{\oplus_i} \leq \frac{1}{2} a_{\oplus_i} d_{\oplus_i} + d_{\oplus_i},\ 1 \leq i \leq m.$$

– si v est un nœud $\otimes$,

$$\begin{array}{rrcl} & h_\oplus(v) & = & \sum_{i=1}^m h_\oplus(v_i) \\ & & = & \sum_{i=1}^m h_{\oplus_i} \\ & & \leq & \frac{1}{2} \sum_{i=1}^m (a_{\oplus_i} d_{\oplus_i}) + \sum_{i=1}^m d_{\oplus_i}, \\ \text{or} & a_\oplus(v) & \geq & \max_{i=1}^m a_{\oplus_i} \text{ puisque } v \text{ est un nœud } \otimes, \\ & d_\oplus(v) & = & \sum_{i=1}^m d_{\oplus_i} \\ \text{et} & a_\oplus(v) d_\oplus(v) & \geq & (\max_{i=1}^m a_{\oplus_i})(\sum_{i=1}^m d_{\oplus_i}) \\ & & \geq & \frac{1}{2} \sum_{i=1}^m (a_{\oplus_i} d_{\oplus_i}), \\ \text{donc} & h_\oplus(v) & \leq & \frac{1}{2} a_\oplus(v) d_\oplus(v) + d_\oplus(v). \end{array}$$

– si v est un nœud $\oplus$, soit v_l son fils dominant,

  – si v_l est une feuille, alors

$$\begin{array}{rrcl} & h_\oplus(v) & = & 1, \\ & a_\oplus(v) & = & 0, \\ & d_\oplus(v) & = & 1 \\ \text{et} & h_\oplus(v) & \leq & \frac{1}{2} a_\oplus(v) d_\oplus(v) + d_\oplus(v). \end{array}$$

  – si v_l est un nœud $\otimes$

$$\begin{array}{rrcl} & a_{\oplus_l} & \leq & a_\oplus(v), \\ & d_{\oplus_l} & \leq & d_\oplus(v) \\ \text{et} & h_\oplus(v) = h_{\oplus_l} & \leq & \frac{1}{2} a_{\oplus_l} d_{\oplus_l} + d_{\oplus_l} \\ & & \leq & \frac{1}{2} a_\oplus(v) d_\oplus(v) + d_\oplus(v). \end{array}$$

– si v_l est un nœud $\oplus$,

$$\begin{array}{lll} & a_{\oplus_l} & \leq a_\oplus(v)+1, \\ & d_{\oplus_l} & \leq d_\oplus(v) \\ \text{et} \quad & h_\oplus(v)=h_{\oplus_l}+\frac{1}{2} & \leq \frac{1}{2}a_{\oplus_l}d_{\oplus_l}+d_{\oplus_l}+\frac{1}{2} \\ & & \leq \frac{1}{2}(a_{\oplus_l}+1)d_{\oplus_l}+d_{\oplus_l} \\ & & \leq \frac{1}{2}a_\oplus(v)d_\oplus(v)+d_\oplus(v). \end{array}$$

On a donc montré par récurrence que, pour tout sous-circuit U_v,

$$h_\oplus(U_v) \leq \frac{1}{2}a_\oplus(U_v)d_\oplus(U_v)+d_\oplus(U_v).$$

Soit v un nœud du circuit U tel que $h_\oplus(U)=h_\oplus(v)$, on a

$$h_\oplus = h_\oplus(U) = h_\oplus(v) \leq \frac{1}{2}a_\oplus(v)d_\oplus(v)+d_\oplus(v) \leq \frac{1}{2}a_\oplus d_\oplus + d_\oplus.$$

□

Corollaire 4 *Pour tout circuit U à n nœuds, de hauteur h et de degré d,*

$$h \leq \frac{1}{2}n^2d+d.$$

DÉMONSTRATION
Si $d=d_\oplus$ par exemple,

$$h=\min(h_\oplus,h_\otimes) \leq h_\oplus \leq \frac{1}{2}a_\oplus d_\oplus + d_\oplus \leq \frac{1}{2}a_\oplus d + d.$$

et $a_\oplus$ est majoré par le nombre d'arcs d'un graphe complet, c'est-à-dire n^2 (ou plus exactement $\frac{n^2}{2}$, mais les calculs étant tous en ordre de grandeur, une telle précision n'est pas indispensable), donc

$$h \leq \frac{1}{2}n^2d+d$$

□

D'après le lemme 4, $\mathcal{O}(\log h)$ applications de *Phase* suffisent à évaluer complètement un circuit de hauteur h. Or $\mathcal{O}(\log h)=\mathcal{O}(\log n+\log d)$ grâce au corollaire 4, d'où ce premier théorème :

Théorème 15 *La complexité de l'algorithme 19 est*

$$CREW_L(M(n),[\log n+\log d]\log n)$$

et

$$CRCW_L(M'(n),\log n+\log d)$$

si le treillis est totalement ordonné.

4.4.2 Seconde majoration

De façon intuitive, ce qui limite la vitesse d'évaluation d'un circuit est l'alternance de nœuds $\oplus$ et $\otimes$. Cette alternance peut être rompue par une évaluation partielle suivie d'une suppression uniquement dans le cas où l'un de ces nœuds a des fils feuilles. Nous introduisons donc une dernière quantité, le nombre d'alternances d'un circuit, qui permet de prendre ceci en compte. Cette quantité h_a est le plus grand nombre d'alternances de nœuds $\oplus$ et $\otimes$ sur tout chemin allant d'un nœud de sortie à une feuille. Plus formellement, h_a est défini comme suit :

Définition 45 *Pour tout nœud v d'un circuit, son nombre d'alternances $h_a(v)$ vaut*

$$
\begin{aligned}
h_a(v) &= 1 && \text{si } v \text{ est une feuille},\\
h_a(v) &= \max_w \left(\begin{array}{l} h_a(w \text{ fils } \otimes \text{ de } v) + 1,\\ h_a(w \text{ fils } \oplus \text{ de } v),\\ h_a(w \text{ fils feuille de } v) + 1) \end{array} \right) && \text{si } v \text{ est un nœud } \oplus,\\
h_a(v) &= \max_w \left(\begin{array}{l} h_a(w \text{ fils } \oplus \text{ de } v) + 1,\\ h_a(w \text{ fils } \otimes \text{ de } v),\\ h_a(w \text{ fils feuille de } v) + 1) \end{array} \right) && \text{si } v \text{ est un nœud } \otimes .
\end{aligned}
$$

Le nombre d'alternances h_a d'un circuit est le maximum des nombres d'alternances de ses nœuds.

Montrons que $h_a + \log n$ applications de *Phase*, ou un pré-traitement suivi de h_a applications de *Phase*, suffisent à évaluer un circuit.

Tout d'abord, aucune des différentes procédures *Group*, *Éval*, *ÉvalPartiel* ou *Suppress* ne fait augmenter h_a. Ensuite, l'importance du pré-traitement apparaît : après l'opération *Group** du pré-traitement, h_a est la longueur du plus long chemin du circuit. Il est alors clair que l'évaluation parallèle de ce circuit nécessitera un temps inférieur à h_a, puisque h_a applications de la procédure *Éval* suffisent à évaluer le circuit, ce qui revient à appliquer un algorithme glouton d'évaluation. *A fortiori*, h_a applications de la procédure *Phase* suffisent à évaluer le circuit.

Remarque
La complexité du pré-traitement est bornée par la complexité de $\log n$ applications de l'opération *Phase* ; il est même possible de remplacer ce pré-traitement par $\log n$ applications de *Phase* pour obtenir un algorithme plus homogène. Ceci revient à dire qu'avec un algorithme « homogène », sans pré-traitement, $h_a + \log n$ applications de *Phase* suffisent à évaluer le circuit.

Quel que soit le point de vue choisi, le théorème suivant est vérifié :

Théorème 16 *La complexité de l'algorithme 19 est*

$$CREW_L(M(n), [h_a + \log n] \log n)$$

et

$$CRCW_L(M'(n), h_a + \log n)$$

si le treillis est totalement ordonné.

4.4.3 Complexité

Les résultats des §4.4.1 et §4.4.2 permettent d'énoncer un théorème récapitulatif de la complexité de l'algorithme d'évaluation parallèle des circuits sur un treillis.

Théorème 17 *La complexité de l'algorithme 19 est*

$$CREW_L(M(n), [\min(\log d, h_a) + \log n] \log n)$$

avec $M(n)$ le nombre minimal de processeurs nécessaires pour multiplier en parallèle deux matrices $n \times n$ dans ce treillis en temps $\mathcal{O}(\log n)$ et

$$CRCW_L(M'(n), \min(\log d, h_a) + \log n)$$

si le treillis est totalement ordonné et si $M'(n)$ est le nombre de processeurs nécessaires pour effectuer un produit de matrices en parallèle dans ce treillis en temps constant.

Remarque
Ceci nous permet de prouver que les circuits de la figure 4.1 sont évalués en temps polylogarithmique, puisque h_a est une constante pour ces circuits.

Sur l'exemple détaillé au §4.3, le nombre de nœuds est 21, le degré vaut 9 et le nombre d'alternances 5. Le nombre d'applications de la procédure *Phase* est donc majoré par $\min(9, 5) + \log 21 = 5 + 5 = 10$. Deux applications ont suffi.

Remarque
Au cours de la démonstration du lemme 4, toutes les procédures qui composent l'algorithme 19 ont révélé leur utilité.

4.5 L'exemple du tri par insertion

Le tri par insertion est l'algorithme de tri le plus facile à concevoir, même s'il n'est pas le plus rapide en séquentiel. De plus il semble ne pas contenir de parallélisme, puisque les éléments sont rangés, « insérés », successivement dans un tableau trié. La structure utilisée est basée sur les opérations *min* et *Max* ; il s'agit d'un treillis distributif.

En entrée sont donnés n nombres $x_1, x_2, \ldots x_n$ supposés deux à deux distincts, en sortie est retourné un tableau $[y_1, y_2, \ldots y_n]$ contenant les éléments $(x_i)_{1 \leq i \leq n}$ triés par ordre croissant. L'algorithme est récursif ; un peu de parallélisme est introduit, en ce sens qu'à chaque étape le nouvel élément cherche en parallèle entre quels éléments s'insérer : si à la fin de l'étape

$i-1$, les éléments $x_1, x_2, \ldots x_{i-1}$ sont triés dans un tableau $[y_1^{(i-1)}, y_2^{(i-1)}, \ldots y_{i-1}^{(i-1)}]$, l'étape i consiste à insérer l'élément x_i dans ce tableau.

Algorithme 20 *Tri par insertion*

pour $i = 1$ à n faire
  si $i = 1$ alors
    $[y_1^{(1)}] \leftarrow [x_1]$
  si $i = 2$ alors
    $[y_1^{(2)}, y_2^{(2)}] \leftarrow [min(x_1, x_2), Max(x_1, x_2)]$
  si $i > 2$ alors
    $y_1^{(i)} \leftarrow min(x_i, y_1^{(i-1)})$
    $y_i^{(i)} \leftarrow Max(y_{i-1}^{(i-1)}, x_i)$
    pour $j = 2$ à $i-1$ faire en parallèle
      si $x_i < y_{j-1}^{(i-1)} < y_j^{(i-1)}$ alors
        $y_j^{(i)} \leftarrow y_{j-1}^{(i-1)}$
      sinon si $y_{j-1}^{(i-1)} < x_i < y_j^{(i-1)}$ alors
        $y_j^{(i)} \leftarrow x_i$
      sinon si $y_{j-1}^{(i-1)} < y_j^{(i-1)} < x_i$ alors
        $y_j^{(i)} \leftarrow y_j^{(i-1)}$

fin

On remarque que $y_j^{(i)}$ est le deuxième élément du tableau trié à trois éléments contenant x_i, $y_{j-1}^{(i-1)}$ et $y_j^{(i-1)}$. Il suffit donc de réaliser un circuit triant trois éléments (dont deux déjà triés) et qui rend ce deuxième élément, puis de le dupliquer pour chaque triplet $(y_{j-1}^{(i-1)}, y_j^{(i-1)}, x_i)$. Un circuit possible est celui de la figure 4.8.

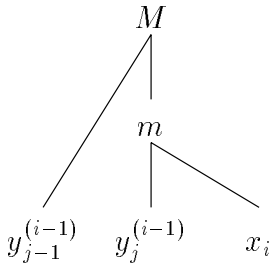


FIG. 4.8 - *Le circuit rendant le deuxième élément parmi* $(y_{j-1}^{(i-1)}, y_j^{(i-1)}, x_i)$.

Le circuit obtenu à partir de l'algorithme de tri par insertion, en utilisant ce circuit, a un nombre d'alternances au moins linéaire: la formule $y_2^{(i)} \leftarrow Max(y_1^{(i-1)}, min(y_2^{(i-1)}, x_i))$ entraîne $h_a(y_2^{(i)}) = 2 + h_a(y_2^{(i-1)})$, ce qui donne $h_a(y_2^{(i)}) = 2(i-1)$ et le nombre d'alternances

du circuit est supérieur à $h_a(y_2^{(i)})$.

Le degré d_{Max}, noté d_M, du circuit vérifie pour $2 \leq i \leq n$:

$$\left\{ \begin{array}{lcl} d_M(y_1^{(i)}) & = & d_M(y_1^{(i-1)}), \\ d_M(y_j^{(i)}) & = & d_M(y_j^{(i-1)}) + d_M(y_{j-1}^{(i-1)}),\ 2 \leq j \leq i-1, \\ d_M(y_i^{(i)}) & = & d_M(y_i^{(i-1)}) + 1. \end{array} \right.$$

On reconnaît les coefficients du triangle de Pascal et leur maximum est plus qu'exponentiel en n.

Le degré d_{min}, noté d_m, du circuit vérifie pour $2 \leq i \leq n$:

$$\left\{ \begin{array}{lcl} d_m(y_1^{(i)}) & = & d_m(y_1^{(i-1)}) + 1, \\ d_m(y_j^{(i)}) & = & \max(d_m(y_j^{(i-1)}) + 1, d_m(y_{j-1}^{(i-1)})),\ 2 \leq j \leq i-1, \\ d_m(y_i^{(i)}) & = & d_m(y_i^{(i-1)}). \end{array} \right.$$

Ce degré est donc linéaire.

Remarque
Il est possible de définir un algorithme où d_M est linéaire alors que d_m est exponentiel, en inversant les opérateurs Max et min dans le circuit de la figure 4.8.

On peut donc prédire une complexité

$$CREW_L(M(n), \log^2 n)$$

et même

$$CRCW_L(M'(n), \log n)$$

pour le problème du tri, à partir de l'algorithme du tri par insertion. Cette complexité rejoint les meilleures complexités connues pour ce problème [Col88, CT93, KR90], et ce de façon automatique, sans mettre en œuvre de techniques sophistiquées.

4.6 Le treillis de Boole $(\{Vrai, Faux\}, \wedge, \vee, \neg)$

Un treillis qui présente un intérêt particulier pour les informaticiens, qu'ils soient théoriciens en parallélisme, concepteurs de circuit ou arithméticiens, est l'algèbre de Boole ($\{Vrai, Faux\}$, $\wedge$, $\vee$, $Vrai$, $Faux$, $\neg$). Cependant, il existe une loi unaire, la loi $\neg$, qui n'est pas prise en compte par notre algorithme. Il est possible de transformer un circuit quelconque sur un treillis de Boole en un circuit sans portes $\neg$, qui a deux fois plus de portes, avec la technique du §**??**. Il est également possible d'adapter l'algorithme 19 au cas du treillis de Boole $\mathcal{B} = (\{Vrai, Faux\}, \wedge, \vee, Vrai, Faux, \neg)$.

En ce qui concerne le traitement des nœuds $\neg$, l'opération de *Group* n'a pas d'effet puisqu'elle est conçue pour augmenter le nombre de fils d'un opérateur et que $\neg$ n'a de signification que comme nœud unaire. L'opération d'évaluation peut être complétée pour prendre en compte cet opérateur, il suffit pour cela d'ajouter un appel, en parallèle des appels à $\acute{E}val_{\oplus}$ et $\acute{E}val_{\otimes}$, à une procédure $\acute{E}val_{\neg}$ dont la signification est immédiate : elle évalue tous les nœuds $\neg$ dont le fils est une feuille. La procédure d'évaluation partielle *ÉvalPartiel* peut soit ne pas toucher aux nœuds $\neg$, soit les connecter au nœud artificiel si leur fils est une feuille, sans que cela change quoi que ce soit. La procédure de suppression des nœuds unaires, *Suppress*, est particulièrement indiquée pour traiter ces nœuds et les déconnecter de leurs pères. Les fonctions d'arcs qui sont utilisées sont alors de la forme $(a \otimes x) \oplus (b \otimes \neg x) \oplus c$; dans le cas du treillis $\mathcal{B}$, les seules fonctions d'arcs possibles sont $Vrai$, $Faux$, x et $\neg x$.

Il faut alors en tenir compte dans le traitement des opérateurs $\wedge$ et $\vee$: en effet un nœud $\vee$ connecté à un autre nœud $\vee$ par un arc $\neg x$ « verra » ce dernier comme un nœud $\wedge$ du fait des lois de De Morgan :

$$\begin{cases} \neg(a \vee b) & = & (\neg a) \wedge (\neg b), \\ \neg(a \wedge b) & = & (\neg a) \vee (\neg b). \end{cases}$$

Pour respecter la sémantique des opérations, il ne faut pas regrouper ces deux nœuds. En revanche, un nœud $\vee$ connecté à un nœud $\wedge$ par un arc $\neg x$ aura la même signification qu'un nœud $\vee$ connecté à un autre nœud $\vee$ et il est possible de les regrouper.

La signification des matrices d'adjacence est modifiée, pour tenir compte de ces changements, U restant la matrice d'adjacence du circuit :

- $U^{\vee\vee}$ est la matrice des arcs différents de $\neg x$ connectant deux nœuds $\vee$ et des arcs portant la fonction $\neg x$ connectant un nœud $\vee$ à un nœud $\wedge$:
  $U^{\vee\vee}_{i,j} = U_{i,j}$ si i et j sont des nœuds $\vee$ et $U_{i,j}(x)$ est différent de $\neg x$, ou si i est un nœud $\vee$, j un nœud $\wedge$ et $U_{i,j}(x) = \neg x$;

- $U^{\vee\cdot}$ est la matrice des arcs $\neg x$ connectant deux nœuds $\vee$, des arcs portant une fonction différente de $\neg x$ connectant un nœud $\vee$ à un nœud $\wedge$ et les arcs connectant un nœud $\vee$ à un nœud $\neg$ ou à une feuille :
  $U^{\vee\cdot}_{i,j} = U_{i,j}$ si i et j sont des nœuds $\vee$ et $U_{i,j}(x) = \neg x$, si i est un nœud $\vee$, j un nœud $\wedge$ et $U_{i,j}(x) \neq \neg x$, ou si i est un nœud $\vee$ et j est un nœud $\neg$ ou une feuille ;

- $U^{\wedge\wedge}$ est la matrice des arcs différents de $\neg x$ connectant deux nœuds $\wedge$ et des arcs portant la fonction $\neg x$ connectant un nœud $\wedge$ à un nœud $\vee$:
  $U^{\wedge\wedge}_{i,j} = U_{i,j}$ si i et j sont des nœuds $\wedge$ et $U_{i,j}(x)$ est différent de $\neg x$, ou si i est un nœud $\wedge$, j un nœud $\vee$ et $U_{i,j}(x) = \neg x$;

- $U^{\wedge\cdot}$ est la matrice des arcs $\neg x$ connectant deux nœuds $\wedge$, des arcs portant une fonction différente de $\neg x$ connectant un nœud $\wedge$ à un nœud $\vee$ et les arcs connectant un nœud $\wedge$ à un nœud $\neg$ ou à une feuille :
  $U^{\vee\cdot}_{i,j} = U_{i,j}$ si i et j sont des nœuds $\wedge$ et $U_{i,j}(x) = \neg x$, si i est un nœud $\wedge$, j un nœud $\vee$ et $U_{i,j}(x)$ est différent de $\neg x$, ou si i est un nœud $\wedge$ et j est un nœud $\neg$ ou une feuille.

Ces quatre matrices ne forment plus une partition de U puisque les arcs issus d'un nœud $\neg$ ne sont pas répertoriés. Il ne nous semble pas utile de définir une nouvelle matrice pour ces arcs, puisqu'une seule application de *Suppress*, lors du pré-traitement s'il existe, ou à la première application de *Phase* sinon, suffit à déconnecter tous les nœuds $\neg$ de leurs pères et rendent ces nœuds inutiles, l'information qu'ils représentent étant alors stockée sur les arcs.

Ces nouvelles matrices sont conçues pour que l'opération de *Group* puisse s'écrire avec les mêmes produits matriciels. L'utilisation de ces nouvelles fonctions d'arcs n'entraînent pas de modification des autres procédures, ni de l'algorithme.

En ce qui concerne les grandeurs caractéristiques du circuit, elles nécessitent également un aménagement pour prendre en compte cet opérateur supplémentaire. Les nouveaux degrés $d_\oplus$ et $d_\otimes$ sont définis par :

$$d_\otimes(v) = \begin{cases} 1 & \text{si } v \text{ est une feuille,} \\ \sum_w d_\otimes(w \text{ fils de } v) & \text{si } v \text{ est un nœud } \otimes, \\ \max_w(d_\otimes(w \text{ fils de } v) & \text{si } v \text{ est un nœud } \oplus, \\ d_\oplus(w \text{ fils de } v) & \text{si } v \text{ est un nœud } \neg, \end{cases}$$

$$d_\oplus(v) = \begin{cases} 1 & \text{si } v \text{ est une feuille,} \\ \sum_w d_\oplus(w \text{ fils de } v) & \text{si } v \text{ est un nœud } \oplus, \\ \max_w(d_\oplus(w \text{ fils de } v) & \text{si } v \text{ est un nœud } \otimes, \\ d_\otimes(w \text{ fils de } v) & \text{si } v \text{ est un nœud } \neg. \end{cases}$$

Le degré d'un circuit est le maximum des degrés de ses nœuds.

De la même façon le nombre d'alternances d'un nœud v devient :

$$h_a(v) = \max \begin{pmatrix} h_a(w \otimes \text{ fils de } v) + 1, \\ h_a(w \oplus \text{ fils de } v), \\ h_a(w \text{ fils feuille de } v) + 1, \\ h_a(x \otimes \text{ petit-fils de } v \text{ et fils de } w \neg), \\ h_a(x \oplus \text{ petit-fils de } v \text{ et fils de } w \neg) + 1 \end{pmatrix}$$

si v est un nœud $\oplus$ et

$$h_a(v) = \max \begin{pmatrix} h_a(w \oplus \text{ fils de } v) + 1, \\ h_a(w \otimes \text{ fils de } v), \\ h_a(w \text{ fils feuille de } v) + 1, \\ h_a(x \oplus \text{ petit-fils de } v \text{ et fils de } w \neg) \\ h_a(x \otimes \text{ petit-fils de } v \text{ et fils de } w \neg) + 1 \end{pmatrix}$$

si v est un nœud $\otimes$. Le nombre d'alternances h_a d'un circuit est le maximum des nombres d'alternances h_a de ses nœuds.

La complexité de l'algorithme reste inchangée. Au §1.2.2, la transformation d'un circuit booléen quelconque en circuit monotone (sans portes $\neg$) équivalent a été détaillée, et il est

facile de vérifier que cette adaptation 19 correspond à l'algorithme original appliqué à un circuit monotone, et que les nouveaux degrés et nombre d'alternances sont les degrés et nombre d'alternances du circuit monotone.

Remarque
Dans le cas où l'algorithme 19 et son adaptation pour les treillis de Boole sont réellement utilisés comme interpréteurs, et non pas comme prédicteurs de complexité (cf. chapitre 5) ou comme compilateurs (cf. chapitre 6.1), il est possible d'utiliser la valeur effective des fonctions d'arcs : si une fonction d'arc est une fonction constante, le nœud père peut considérer son fils comme une feuille lors des procédures d'évaluation et d'évaluation partielle. Par conséquent, la procédure *Suppress* peut déconnecter les nœuds plus tôt et éventuellement « gagner du temps ».

4.7 Conclusion

Dans ce chapitre, nous avons présenté notre contribution à l'évaluation parallèle de circuits arithmétiques, en proposant un algorithme d'évaluation des circuits sur des treillis. Les deux caractéristiques majeures de cet algorithme sont tout d'abord le traitement symétrique des opérations $\oplus$ et $\otimes$ et le fait qu'une nouvelle borne de complexité, le nombre d'alternances du circuit, permette d'affiner les bornes déjà connues pour ce problème.

Cet algorithme permet de montrer par exemple que le problème du tri admet une solution parallèle, simplement en évaluant une version du tri par insertion, qui n'est pourtant pas connu pour être un modèle de tri parallèle.

Nous avons également adapté cet algorithme au cas du treillis de Boole ($\{Vrai, Faux\}$, $\vee$, $\wedge$, $Vrai$, $Faux$, $\neg$) afin de gérer les nœuds $\neg$ des circuits booléens, qui servent de modèle en parallélisme.

Deuxième partie

Applications

Chapitre 5

Expérimentations

Résumé

Des simulations, destinées à valider les algorithmes de la partie I, ont été effectuées sur deux problèmes tests : le problème de l'addition de deux entiers à n bits et le problème P-complet du calcul de l'ensemble lexicographiquement maximal de sommets indépendants d'un graphe. Pour ces deux problèmes, la complexité théorique est linéaire ; en pratique, le circuit d'addition de deux entiers est évalué en temps polylogarithmique et le second en temps linéaire. L'algorithme 19 est ensuite utilisé comme prédicteur de complexité pour différents problèmes issus du calcul booléen et du calcul formel. Enfin, d'autres champs d'application sont mentionnés.

5.1 Introduction

La première partie de cette thèse a été consacrée à l'étude d'algorithmes parallèles théoriques d'évaluation des expressions ou des circuits arithmétiques, ainsi qu'au calcul de leur complexité. Il est légitime de s'interroger sur la précision de ces complexités : s'agit-il de surestimations grossières systématiques ou bien y a-t-il concordance entre la borne calculée et la durée observée du calcul ? Le meilleur moyen de s'assurer de la validité des bornes théoriques sur la complexité des algorithmes d'évaluation est de les expérimenter sur différents problèmes. Dans ces calculs de complexité, la durée de la procédure *Phase* est logarithmique en n puisque les produits de matrices sont effectués, même s'ils sont totalement inutiles. Le nombre d'applications de cette procédure *Phase* est par contre inconnu. Pour mesurer ce nombre, nul besoin d'exécuter un algorithme parallèle, une simulation séquentielle suffit, ce qui simplifie considérablement le travail de programmation. Pour s'assurer que la simulation séquentielle respectera bien l'exécution parallèle de l'algorithme d'évaluation, on travaille sur deux copies du circuit, l'une contenant le circuit dans l'état où il se trouvait à la fin de l'itération précédente et une copie de travail, qui est modifiée. Ce fonctionnement respecte la convention $PRAM$ de lecture puis écriture de la mémoire et il évite également d'évaluer complètement (et séquentiellement) le circuit en une seule passe. Le programme de simulation incrémente également un compteur qui mémorise le nombre d'applications déjà effectuées de la procédure *Phase*.

Comme l'algorithme 19 est indépendant de la valeur des feuilles du circuit, il est clair que toutes les valeurs des entrées conduiront à des exécutions identiques. Une seule exécution du programme, sur un jeu de valeurs quelconques, suffit donc à déterminer le nombre cherché.

Pour vérifier les complexités déterminées dans la première partie de cette thèse, nous avons tout d'abord choisi un problème simple à programmer « en booléen » et tel que le résultat soit facile à vérifier, ce qui permet de détecter les erreurs de programmation. L'addition de deux entiers de n bits est un problème qui répond à ces critères. Les entiers sont supposés positifs et écrits en base deux. L'addition s'effectue des poids faibles vers les poids forts, en propageant la retenue, une démarche intrinsèquement séquentielle.

Ensuite, nous nous sommes demandé quelle était la puissance de notre algorithme d'évaluation. Nous avons donc décidé de l'essayer sur un problème réputé difficile, le calcul de l'ensemble lexicographiquement maximal de sommets d'un graphe ou *LMISP*. Ce problème est en effet un problème P-complet [Coo81] et Cook [Coo85] conjecture que ces problèmes n'appartiennent pas à NC. Comme les algorithmes d'évaluation utilisent la connaissance du circuit entièrement déroulé pour transformer ce circuit en un circuit moins profond, qui est leur graphe de tâches, ils peuvent être vus comme des transformations P-uniformes sur ces circuits. Si cet algorithme permettait d'évaluer le circuit booléen correspondant au *LMISP* en temps polylogarithmique[1], un élément de réponse serait que tout algorithme séquentiel polynomial peut être transformé – par une transformation polynomiale en temps – en un circuit booléen de profondeur polylogarithmique.

Les résultats obtenus confirment ces prévisions et réhabilitent même l'usage de l'algorithme de type contraction d'arbre 9 dans certains cas. Pour le problème de l'addition, le nombre d'applications de la procédure *Phase* est un logarithme ou un logarithme au carré du nombre de chiffres selon l'algorithme booléen choisi, il est linéaire pour le *LMISP*. La fiabilité des estimations de complexité confirmée, nous avons alors testé la puissance de notre algorithme sur divers problèmes issus du calcul booléen ou du calcul formel. Pour la plupart des cas, nous retrouvons la complexité déjà connue pour ces problèmes. Pour le problème du calcul d'un pgcd, les bornes données ne sont pas les meilleures connues, cependant elles indiquent que la parallélisation ne sera pas évidente.

Enfin, deux domaines où ces algorithmes d'évaluation parallèle pourraient s'appliquer avec succès, sont les recherches de chemins dans les graphes et la simulation de systèmes à événements discrets modélisés par des réseaux de Petri temporisés.

5.2 Présentation des simulations

5.2.1 Addition

L'addition de deux entiers de n bits

$$\begin{array}{cccc} & a_{n-1} & \dots & a_0 \\ + & b_{n-1} & \dots & b_0 \\ \hline r_n & r_{n-1} & \dots & r_0 \end{array}$$

1. On peut toujours espérer !

a pour résultat un entier d'au plus $n+1$ bits, $r_n r_{n-1} \ldots r_0$ (r comme résultat). À chaque couple (a_i, b_i) est associé un couple (r_i, c_i) où r_i est le chiffre correspondant au résultat et c_i est la retenue (ou *carry*) qui va entrer en compte dans le calcul de r_{i+1}. Les équations booléennes s'écrivent à l'aide des fonctions ET, OU et NON sous la forme non simplifiée :

$$\begin{aligned} r_i &= ((\neg a_i) \wedge (\neg b_i) \wedge c_{i-1}) \\ &\vee ((\neg a_i) \wedge b_i \wedge (\neg c_{i-1})) \\ &\vee (a_i \wedge (\neg b_i) \wedge (\neg c_{i-1})) \\ &\vee (a_i \wedge b_i \wedge c_{i-1}), \end{aligned}$$

$$\begin{aligned} c_0 &= Faux, \\ c_i &= ((\neg a_i) \wedge b_i \wedge c_{i-1}) \\ &\vee (a_i \wedge (\neg b_i) \wedge c_{i-1}) \\ &\vee (a_i \wedge b_i \wedge (\neg c_{i-1})) \\ &\vee (a_i \wedge b_i \wedge c_{i-1}). \end{aligned}$$

Le degré et le nombre d'alternances du circuit correspondant sont déterminés par ceux de c_n. Le degré $d_\vee$ de c_i vérifie $d_\vee(c_i) = 3 * d_\vee(c_{i-1}) + d_\wedge(c_{i-1})$, et le degré $d_\wedge$ de c_i est $d_\wedge(c_i) = \max(2 + d_\wedge(c_{i-1}), 2 + d_\vee(c_{i-1}))$, ce qui donne un degré exponentiel. De plus, le nombre d'alternances est linéaire. La complexité parallèle théorique de l'évaluation de ce circuit par l'algorithme 19 est par conséquent $CRCW(M(n), n)$.

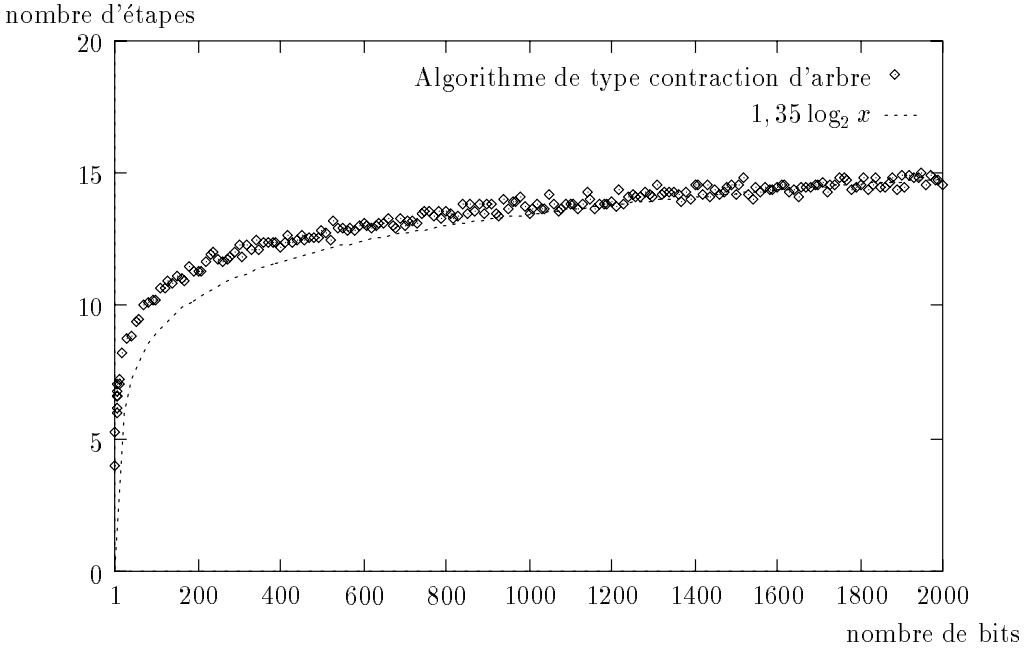


FIG. 5.1 - *Évaluation par l'algorithme de type contraction d'arbre.*

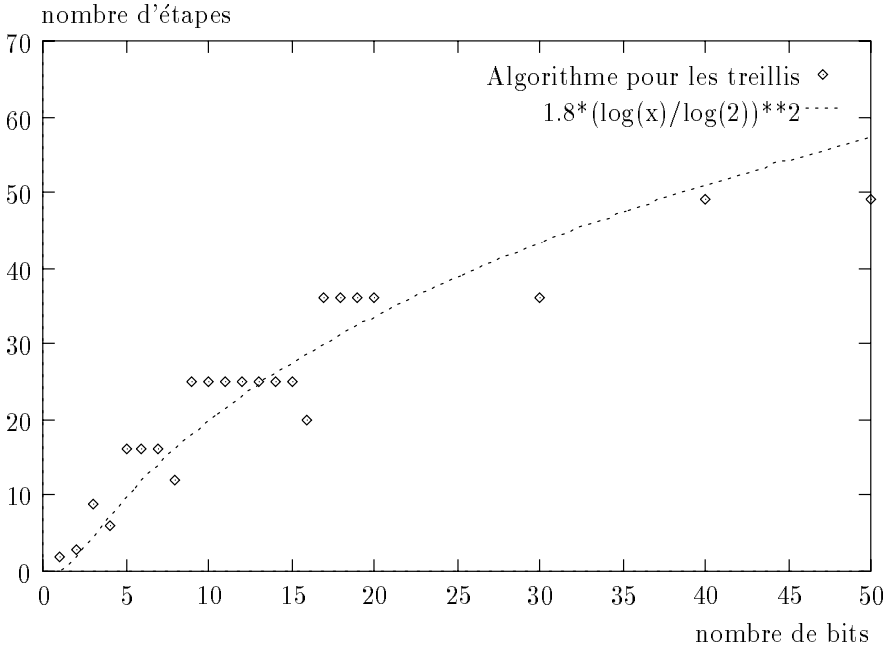


FIG. 5.2 - *Évaluation par l'algorithme pour les treillis.*

La figure 5.2 représente la complexité temporelle pour une $CRCW-PRAM$. Elle offre l'agréable surprise d'être un logarithme au carré (très exactement $(\lfloor \log n \rfloor + 2)^2 = (i+1)^2$ pour $n \in]2^{i-1}, 2^i[$ et $(\lfloor \log n \rfloor + 1)(\log n + 2) = i(i+1)$ pour $n = 2^i$). Ceci signifie que l'algorithme fonctionne au-delà de toute espérance, mais également que ni le degré ni le nombre d'alternances ne permettent de détecter toutes les possibilités de *Shunt*.

Remarque
On peut également utiliser les équations du chapitre 1 :

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

Dans ce cas, les degrés $d_\vee$ et $d_\wedge$ sont linéaires en n et donc la complexité théorique est

$$CRCW(M(n), \log n).$$

Les résultats expérimentaux confirment ces résultats : le nombre d'applications de *Phase* est exactement $\lceil \log n \rceil + 1$ (cf. courbe 5.3).

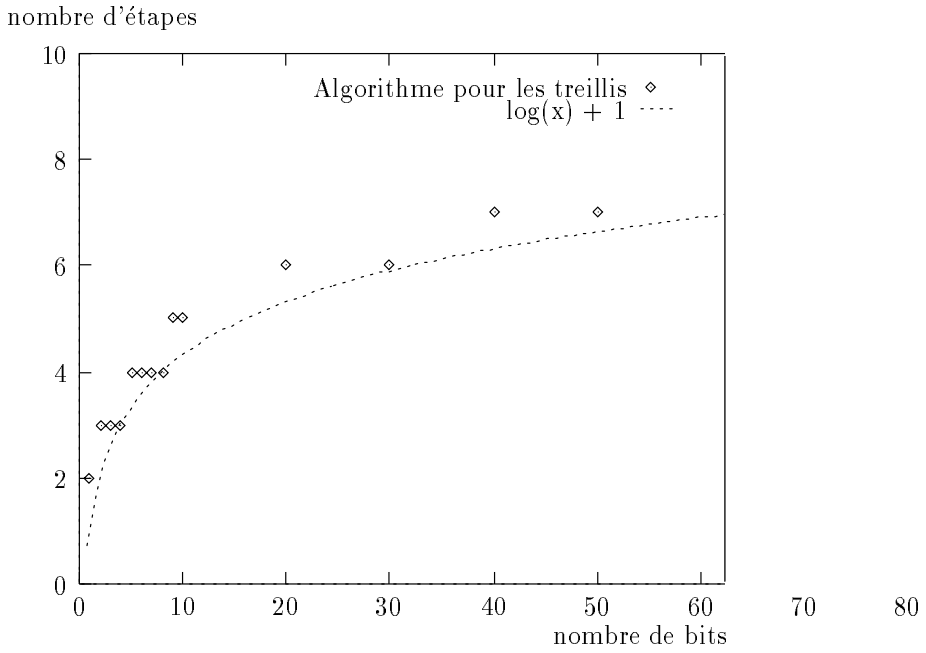


FIG. 5.3 - *Évaluation à partir d'un algorithme plus pertinent.*

L'algorithme de type contraction d'arbre appliqué à ce circuit donne également de très bons résultats : expérimentalement, le circuit est évalué en temps $1,35 \log n$ (figure 5.1). Il conduit à un temps optimal pour ce circuit. Il rejoint l'algorithme de Brent et Kung, présenté au chapitre 1, en ce sens qu'il « reconnaît » et réalise une opération de préfixe dès que le premier *Éval* et le premier *Shunt* sont effectués.

Sur ce problème simple et néanmoins très séquentiel, si on se contente de rechercher le parallélisme sur le graphe de dépendance des calculs, les algorithmes d'évaluation de la partie I permettent d'atteindre de très bons résultats, même quand l'écriture de l'algorithme ne favorise pas la détection du parallélisme. Il est à noter que le problème de l'addition « bien écrite » représente un « meilleur cas » pour ces algorithmes d'évaluation, puisque sa complexité coïncide avec le minimum des complexités possibles. En effet, par construction de l'algorithme 19, il est impossible – sur le modèle $CREW-PRAM$ – de passer sous la barre d'un logarithme élevé au carré. Comme de plus les constantes multiplicatives sont proches de 1, ces résultats de complexité sont très encourageants.

Remarque
Des statistiques sur le remplissage de la matrice d'adjacence utilisée par l'algorithme 19 mettent en évidence une structure très creuse, avec en moyenne moins de 5 éléments par ligne et au maximum une fonction qui ressemble à un logarithme.

5.2.2 *LMISP*

Ce problème est un problème P-complet (cf. chapitre 1), il est donc un mauvais candidat à la parallélisation. Le problème est de calculer, dans un graphe dont les sommets sont totalement ordonnés, un ensemble, lexicograhiquement maximal pour cet ordre et de cardinal maximal, de sommets du graphe qui ne sont pas connectés. Ce problème joue l'analogue du problème de la satisfaisabilité dans l'étude de l'inclusion $P \subset NP$, où NP est la classe définie en remplaçant « machine de Turing déterministe » dans la définition de P par « machine de Turing non déterministe ». Ce problème a été choisi parce qu'il s'agit d'un problème P-complet très précis, beaucoup plus que le CVP et le $MCVP$ par exemple, où il faut travailler sur un circuit quelconque.

Définition 46 *Soit $G = (V, E)$ un graphe non orienté, avec $V = \{v_1, v_2, \ldots v_n\}$ totalement ordonné par l'ordre lexicographique $v_1 > v_2 > \ldots > v_n$. L'ensemble, noté LMIS, est l'ensemble maximal pour l'ordre lexicographique des sommets indépendants de G, i.e. le sous-graphe engendré par cet ensemble n'a pas d'arête. Le problème du calcul de cet ensemble est noté LMISP, pour Lexicographic Maximal Independent Set Problem.*

Un algorithme séquentiel glouton permet de résoudre ce problème : si l'ensemble de sommets cherché est noté *LMIS*, un parcours selon l'ordre lexicographique décroissant de l'ensemble des sommets remplit le *LMIS*, en ajoutant un nœud si ce nœud n'est adjacent à aucun des nœuds dont l'appartenance au *LMIS* a déjà été déterminée.

Algorithme 21 *Algorithme glouton*

$LMIS \leftarrow \emptyset$
pour $i = 1$ à n faire
    si v_i n'est connecté à aucun sommet appartenant au *LMIS* alors
        $LMIS \leftarrow LMIS \bigcup \{v_i\}$

fin

Cet algorithme s'écrit sous une forme booléenne n'utilisant que des portes ET et OU, ainsi que la matrice d'adjacence $\mathcal{G}$ du graphe et cette matrice complémentée $\bar{\mathcal{G}} = (\neg \mathcal{G}_{i,j})$. Si S est la table de vérité du *LMIS* et $\bar{S}$ son complément, alors le circuit correspondant est défini par :

Algorithme 22 *Algorithme booléen pour le LMISP*

pour $i = 1$ à n faire
    $S_i \leftarrow Faux$
    $\bar{S}_i \leftarrow Vrai$

$S_1 \leftarrow Vrai$
$\bar{S}_1 \leftarrow Faux$

pour $i = 2$ à n faire
  pour $j = 1$ à $i - 1$ faire
    $S_i \leftarrow S_i \wedge (\bar{S}_j \vee \bar{\mathcal{G}}_{i,j})$
    $\bar{S}_i \leftarrow \bar{S}_i \vee (S_j \wedge \mathcal{G}_{i,j})$

fin

Cette écriture s'explique par la transformation d'un circuit en circuit monotone détaillée au chapitre 1.

Remarque
Le nombre de nœuds du circuit est quadratique en le nombre de sommets du graphe.

Ce circuit est assez enchevêtré, comme le montre la figure 5.4 pour un graphe à quatre sommets.

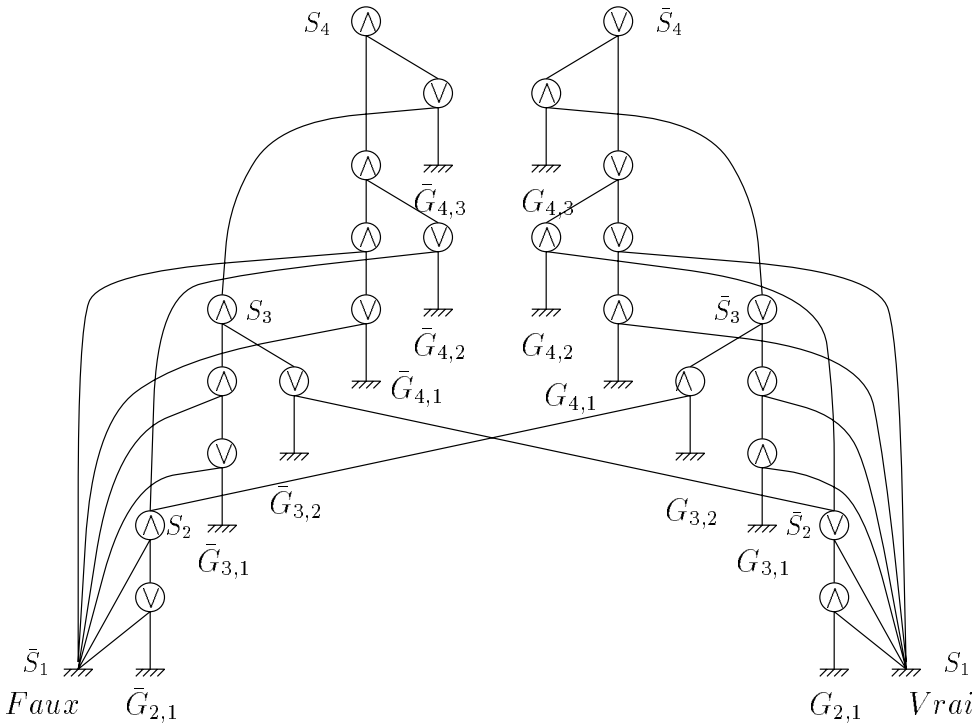


FIG. 5.4 - *Le circuit du LMISP pour un graphe à 4 sommets.*

Calculons le degré de ce circuit. Comme il est absolument symétrique (et pas uniquement à cause de la construction d'un circuit monotone à partir d'un circuit quelconque), on a $d_\vee = d_\wedge$. Calculons $d_\wedge$:

$$\begin{array}{lcl} d_\wedge S_1 & = & 1, \\ d_\wedge \bar{S}_1 & = & 1, \\ d_\wedge S_2 & = & 2, \\ d_\wedge \bar{S}_2 & = & 2, \end{array}$$

$$\begin{array}{lcl} S_i & = & \bigwedge_{j=1}^{i-1} \bar{S}_j \vee \bar{\mathcal{G}}_{i,j}, \\ d_\wedge S_i & = & \sum_{j=1}^{i-1} \max(d_\wedge \bar{S}_j, d_\wedge \bar{\mathcal{G}}_{i,j}) \\ & = & \sum_{j=1}^{i-1} d_\wedge \bar{S}_j, \end{array}$$

$$\begin{array}{lcl} \bar{S}_i & = & \bigvee_{j=1}^{i-1} S_j \wedge \mathcal{G}_{i,j}, \\ d_\wedge \bar{S}_i & = & \max_{1 \le j \le i-1}(d_\wedge S_j + d_\wedge \mathcal{G}_{i,j}) \\ & = & d_\wedge S_{i-1} + 1. \end{array}$$

On remplace $d_\wedge \bar{S}_j$ dans $d_\wedge S_i$:

$$d_\wedge S_i \ge (i-1) + \sum_{j=1}^{i-2} d_\wedge S_j.$$

Notons

$$\begin{array}{lcll} \Sigma_i & = & \sum_{j=1}^{i-2} d_\wedge S_j, & \\ \Sigma_3 & = & d_\wedge S_1 = 1, & \\ \Sigma_4 & = & d_\wedge S_1 + d_\wedge S_2 = 3, & \\ \Sigma_{i+1} & = & d_\wedge S_{i-1} + \Sigma_i & i \ge 3 \\ & \ge & (i-2) + \Sigma_{i-1} + \Sigma_i & \\ & \ge & 2\Sigma_{i-1}. & \end{array}$$

Séparons les cas i pair et i impair.

$$\left.\begin{array}{rcl} i = 2k+1 \Rightarrow \Sigma_{2k+1} & \ge & 2\Sigma_{2(k-1)+1} \\ \Rightarrow \Sigma_{2k+1} & \ge & 2^{k-1}\Sigma_3 \\ \Rightarrow \Sigma_{2k+1} & \ge & 2^{k-1} \end{array}\right\} i \ge 3\ k \ge 1,$$

$$\left.\begin{array}{rcl} i = 2k \Rightarrow \Sigma_{2k} & \ge & 2\Sigma_{2(k-1)} \\ \Rightarrow \Sigma_{2k} & \ge & 2^{k-2}\Sigma_4 \\ \Rightarrow \Sigma_{2k+1} & \ge & 2^{k-1} \end{array}\right\} i \ge 4,\ k \ge 2,$$

d'où

$$\Sigma_i \ge 2^{\lfloor \frac{i}{2} \rfloor - 1}$$

et

$$d_\wedge S_i = (i-1) + \Sigma_i \ge 2^{\lfloor \frac{i}{2} \rfloor - 1},$$

donc le degré de ce circuit est exponentiel.
Calculons maintenant le nombre d'alternances de ce circuit :

$$\begin{array}{rcl} h_a(S_1) & = & 1, \\ h_a(\bar{S}_1) & = & 1, \\ h_a(S_2) & = & 3, \\ h_a(\bar{S}_2) & = & 3, \end{array}$$

$$\begin{array}{rcl} S_i & = & \bigwedge_{j=1}^{i-1} \bar{S}_j \vee \bar{\mathcal{G}}_{i,j}, \\ h_a(S_i) & = & 2 + h_a \bar{S}_{i-1}, \end{array}$$

$$\begin{array}{rcl} \bar{S}_i & = & \bigvee_{j=1}^{i-1} S_j \wedge \mathcal{G}_{i,j}, \\ h_a(\bar{S}_i) & = & 2 + h_a S_{i-1}, \end{array}$$

$$\begin{array}{rcl} h_a(S_i) & = & 2(i-1)+1, \\ h_a(\bar{S}_i) & = & 2(i-1)+1. \end{array}$$

h_a est linéaire en n. La complexité est donc linéaire en n :

$$CRCW(M(n), n).$$

Ceci signifie que cette technique ne permet pas de conclure quant au lien entre P et (P-uniforme)NC : ce problème P-complet est évalué en parallèle en temps linéaire.

Quand on applique l'algorithme, on remarque qu'au début de l'évaluation, un seul *Shunt* a effectivement lieu, sur les coefficients de $\mathcal{G}$ et $\bar{\mathcal{G}}$. Si les portes du circuit sont n-aires, *Group* ne modifie pas le circuit et, *grosso modo*, les seules procédures actives sont $\acute{E}val_+$ et $\acute{E}val_*$ et *Shunt* « tout en bas » du circuit. On calcule un seul $S[i]$ et un seul $\bar{S}[i]$ à chaque application de *Phase*. L'étude fine de ce qui se passe lors de l'évaluation du circuit correspondant au $LMISP$ met cependant en évidence des motifs qui peuvent être simplifiés algébriquement. Par exemple, il y a des motifs du type

$$\begin{array}{l} (S_j \wedge \bar{S}_j \wedge \mathcal{G}_{,,.}) \vee \ldots, \\ (S_j \vee \bar{S}_j \vee \bar{\mathcal{G}}_{,,.}) \wedge \ldots, \end{array}$$

qui pourraient être simplifiés, ce qui permettrait de supprimer des termes. Très exactement on a

$$\begin{array}{rcl} S_i & = & [\bar{\mathcal{G}}_{i,i-1} \vee (S_{i-1} \wedge \mathcal{G}_{i-1,i-2}) \vee \ldots] \wedge [\bar{\mathcal{G}}_{i,i-2} \vee \bar{S}_{i-1}] \\ & = & ([\bar{\mathcal{G}}_{i,i-1} \vee \ldots] \wedge [\bar{\mathcal{G}}_{i,i-2} \vee \bar{S}_{i-1}]) \\ & & \vee [(S_{i-1} \wedge \mathcal{G}_{i-1,i-2}) \wedge \bar{\mathcal{G}}_{i,i-2}] \vee [(S_{i-1} \wedge \mathcal{G}_{i-1,i-2}) \wedge \bar{S}_{i-1}] \\ & & \vee \ldots \\ & = & ([\ldots] \wedge [\ldots]) \vee [\ldots] \vee [Faux \wedge \mathcal{G}_{i-1,i-2}] \\ & = & ([\ldots] \wedge [\ldots]) \vee [\ldots]. \end{array}$$

Il semble donc que l'utilisation de propriétés algébriques supplémentaires permettrait d'accélérer l'évaluation parallèle du circuit. Il est cependant difficile de prédire leur action exacte : les formules à manipuler deviennent rapidement complexes (de longueur au moins polynomiale,

puisqu'il s'agit de transformations polynomiales !). Quoi qu'il en soit, l'équivalence des circuits booléens et des circuits booléens monotones sur le plan de la complexité laisse présager que les gains obtenus ne permettront pas de conclure sur le lien entre P et (P-uniforme)NC.

Les figures 5.5 et 5.6 présentent les résultats des simulations avec les deux algorithmes testés. Sur la figure correspondant à l'algorithme 5.6 nous avons représenté le nombre d'applications de *Phase*, *i.e.* la complexité sur une $CRCW-PRAM$. L'algorithme 9 tout comme l'algorithme 19 évalue ce circuit en temps linéaire.

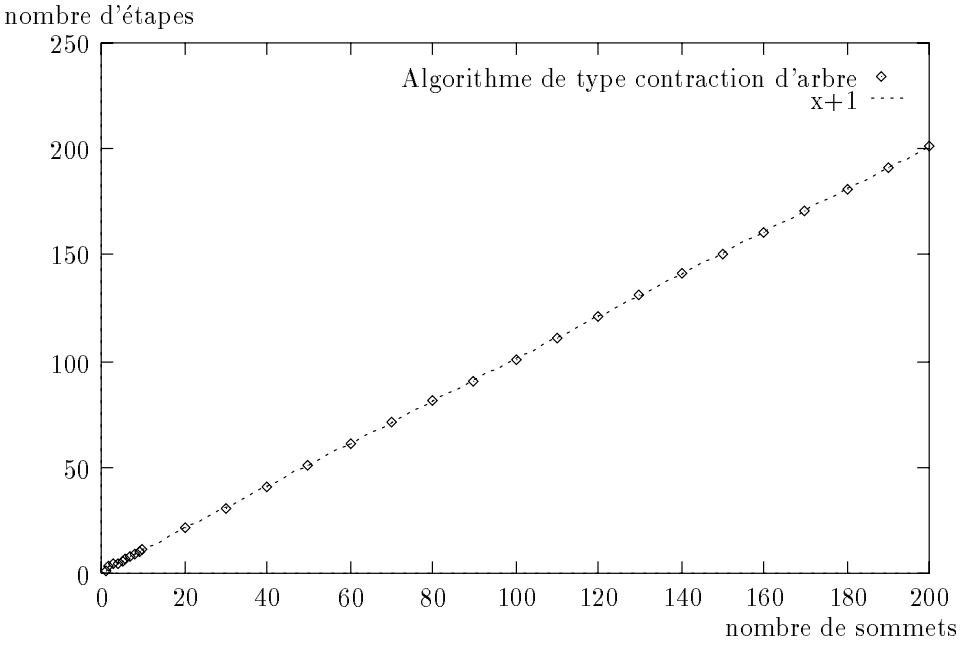


FIG. 5.5 - *Évaluation par l'algorithme de type contraction d'arbre.*

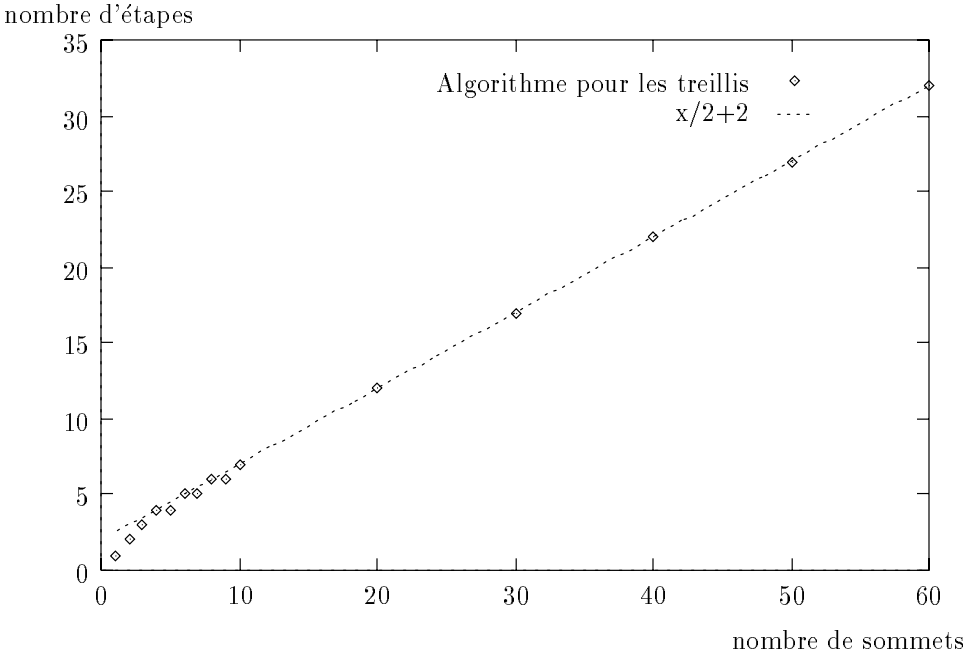


FIG. 5.6 - *Évaluation par l'algorithme pour les treillis.*

Ici, les complexités théoriques sont atteintes, et les constantes multiplicatives sont encore de l'ordre de 1. Ce problème nous a permis d'atteindre les maxima possibles de complexité. Il s'agit donc bien d'un pire cas pour nos algorithmes.

5.2.3 Commentaires

L'expérimentation des algorithmes d'évaluation sur deux problèmes très différents a permis de mettre en évidence plusieurs points.

Tout d'abord, les bornes de complexité calculées dans les chapitres précédents, qui sont *a priori* seulement des majorations, sont souvent assez fines, mais elles méritent d'être vérifiées expérimentalement. Quoi qu'il en soit, si les prévisions sont optimistes, il est certain qu'une solution parallèle existe, sinon la surprise ne peut qu'être agréable. L'expérimentation permet de raffiner ces bornes théoriques. De plus, les constantes multiplicatives qui apparaissent (et qui étaient cachées par les calculs en ordre de grandeur) sont proches de 1.
Ces expérimentations ont confirmé que le problème P-complet du *LMISP* est un pire cas pour notre algorithme.
Enfin, ces deux exemples extrêmes ont réhabilité l'algorithme de type contraction d'arbre du chapitre 3, qui est tout à fait utilisable dans certains cas : pour l'addition avec l'une ou l'autre écriture booléenne, où les opérations de *Shunt* ont réalisé la majeure partie des calculs, et pour le *LMISP*, pour lequel il est presque certain qu'une évaluation en temps sous-linéaire est impossible.

La distinction entre l'algorithme de Miller et Teng [MT87] et le nôtre n'a pas pu se faire sur ces deux problèmes. En effet, les calculs sont essentiellement réalisés par l'opération de *Shunt* sur des nœuds $\wedge$ dans le problème de l'addition (quelle que soit l'écriture choisie), et par les évaluations pour le *LMISP*. Les constantes multiplicatives sont les mêmes pour les deux algorithmes dans ces deux cas.

En ce qui concerne la réalisation des expériences, il s'avère que les matrices sont très creuses et que si on ne les stocke pas comme telles, alors la taille des problèmes traitables est très petite (12 bits maximum pour l'addition de deux entiers). Avec un stockage creux, seul le circuit est conservé en mémoire. Le temps des simulations est important aussi : chaque exécution de *Phase* est réalisée en séquentiel en temps $\mathcal{O}(n^3)$, ou au minimum $\mathcal{O}(n^2)$ en tenant compte du remplissage de la matrice, et ce temps est encore multiplié par le nombre de *Phase*.

5.3 Application au calcul booléen

5.3.1 Addition de n bits

Le problème est ici de calculer la somme de n bits :

$$\sum_{i=1}^{n} b_i.$$

Ce problème intervient notamment dans la multiplication de deux entiers par l'algorithme standard. Nous présentons deux algorithmes pour ce problème : le premier consiste à utiliser un produit standard de produit itéré en prenant garde au grossissement des résultats des calculs. Le second utilise une astuce pour ramener ce problème à celui de l'addition de deux (grands) entiers.

1. *Avec un produit itéré*, l'algorithme est le suivant :

   Algorithme 23 *Addition de n bits par un produit itéré*

   BitAdd$(b_0, \ldots b_{n-1}, n)$:=
   $r_0 =$ BitAdd$(b_0, \ldots b_{\frac{n}{2}-1}, \frac{n}{2})$
   $r_1 =$ BitAdd$(b_{\frac{n}{2}}, \ldots b_{n-1}, \frac{n}{2})$
   $res =$ Addition $(r_0, r_1, \log n)$
   {Cette procédure effectue l'addition de deux nombres de $\log n$ bits.}

   fin

   Si on suppose que l'on utilise l'algorithme de degré linéaire pour additionner deux nombres (sinon il n'y a aucun espoir de prédire une complexité parallèle intéressante), il apparaît que le degré, aussi bien $d_\vee$ que $d_\wedge$, du circuit est multiplié par $4^{\log(\text{taille des opérandes})}$

à chaque addition. Le degré total du circuit est donc $\mathcal{O}(4^{\log^2 n})$. En ce qui concerne le nombre d'alternances h_a du circuit, la contribution de chaque addition étant l'ajout de log(taille des opérandes), le nombre total d'alternances du circuit est $\mathcal{O}(\log^2 n)$. Cela permet de prédire une complexité $\mathcal{O}(\log^2 n)$, qui est effectivement la complexité parallèle de cet algorithme. Le prédicteur n'a pas permis d'affiner cette borne.

2. *La règle du « 2 pour 3»* permet de remplacer l'addition de trois nombres $a = a_{k-1}, \ldots a_0$, $b = b_{k-1}, \ldots b_0$ et $c = c_{k-1} \ldots c_0$ par l'addition de deux nombres $u = u_{k+1}, \ldots u_0$ et $v = v_k, \ldots v_0$ tels que $a+b+c = 2u+v$. Les calculs sont simples : $a_i + b_i + c_i = 2u_i + v_i$, u_i est donc le chiffre de la retenue générée par $a_i + b_i + c_i$ et v_i est le chiffre de poids faible de cette somme. On peut calculer u et v en temps constant en parallèle, avec k processeurs. L'application récursive de cette règle transforme en temps $\log_{\frac{3}{2}} n$, le problème de l'addition de n nombres à $n^{\mathcal{O}(1)}$ chiffres en une addition de deux nombres à $n^{\mathcal{O}(1)}$ chiffres, ce qui prend un temps $\log n$.
   Très clairement, sur cet algorithme le nombre d'alternances sera $\mathcal{O}(\log n)$ et le degré $\mathcal{O}(n)$, ce qui donnera une complexité parallèle logarithmique. Les calculs sont suffisamment bien équilibrés pour que les techniques d'anticipation ne puissent pas jouer.

5.3.2 Puissance de matrices booléennes

Il s'agit d'élever à la puissance n une matrice à coefficients booléens de taille $n \times n$. Sur une $CRCW-PRAM$, le produit de deux matrices booléennes s'effectue en temps constant : les produits sur les coefficients s'effectuent en parallèle en temps 1, ensuite la matrice résultat est initialisée à $Faux$ et les processeurs qui ont calculé une valeur $Vrai$ l'écrivent dans la case mémoire correspondant à l'élément calculé. Cet algorithme fonctionne quel que soit le modèle de $CRCW-PRAM$ choisi. Un algorithme parallèle possible est d'effectuer ces produits de matrices, qui sont associatifs, selon un schéma produit itéré, en temps $\log n$. La complexité parallèle de ce problème est donc majorée par $\log n$ pour une $CRCW-PRAM$. Calculons la complexité prédite par notre algorithme de contraction. L'algorithme séquentiel est le suivant ;

Algorithme 24 *Puissance $n^{ième}$ de matrices*

pour $i = 1$ à n faire
    $A^i \leftarrow A^{i-1} * A$

fin

Le degré $d_\wedge$ du circuit correspondant est 2 pour chaque produit de matrices. Comme n produits sont enchaînés, cela donne un degré $2n$, la complexité de l'évaluation de ce circuit par l'algorithme 19 est donc $CRCW(M(n^4), \log n)$. Dans ce cas également, la complexité parallèle du problème est prédite automatiquement.

5.4 Application au calcul formel

5.4.1 Puissance de matrices à coefficients entiers

Le problème est de calculer l'élévation à la puissance n d'une matrice $n \times n$ dont les coefficients sont des entiers à n bits. Dans le décompte des opérations interviennent les opérations booléennes réalisant les opérations arithmétiques en précision infinie.

Une solution parallèle en temps $\mathcal{O}(\log^2 n)$ existe : comme l'addition de n entiers de longueur n a la même complexité que celle de deux entiers de $n^{\mathcal{O}(1)}$ bits, grâce à la règle du « 2 pour 3 » du §5.3.1, on en déduit qu'un produit de matrices prend un temps $\mathcal{O}(\log n)$. Comme il s'agit d'un calcul de produit itéré faisant intervenir l'opération matricielle associative de produits de matrices, ce produit itéré se réalise en temps $\mathcal{O}(\log n)$ fois la complexité du produit de matrices. Cela signifie que $\mathcal{O}(\log^2 n)$ est le temps parallèle nécessaire pour élever une matrice à coefficients entiers à la puissance n.

Cherchons s'il est possible d'améliorer cet algorithme parallèle. Rompre ce découpage en « niveau haut » (opérations matricielles) et « niveau bas » (opérations sur les entiers), en autorisant le mécanisme d'anticipation à jouer à travers ces niveaux de conception logique, pourrait améliorer encore le temps parallèle, au moins pour une $CRCW-PRAM$.
Calculons le nombre d'alternances du circuit parallèle : pour chaque calcul $A^{2^{i+1}} \leftarrow A^{2^i} * A^{2^i}$, au niveau d'un calcul de coefficient, la contribution de la multiplication à h_a est de $\mathcal{O}(\log n)$ si les calculs sont organisés en arbre équilibré. Il est en de même pour les additions. Comme il y a $\log n$ calculs de produits de matrices dans un plus long chemin de l'arbre, on en déduit que $h_a = \mathcal{O}(\log^2 n)$.
Calculons maintenant le degré du circuit parallèle : pour une multiplication, le degré des opérandes (aussi bien $d_\wedge$ que $d_\vee$) est multiplié par $\mathcal{O}(4^{\log n})$; il en va de même pour les n additions qui suivent chaque multiplication (on peut le vérifier à partir du calcul de degré de l'addition). Le degré total de ce circuit est $\mathcal{O}(4^{\log^2 n})$.
Ces deux mesures conduisent à une complexité parallèle temporelle de $\mathcal{O}(\log^2 n)$. La complexité de ce problème n'est donc pas affinée (mais pas dégradée non plus) par les estimations automatiques.

5.4.2 Calcul du pgcd d'entiers

Ce problème n'admet pas à l'heure actuelle de solution parallèle rapide. Soient A et B deux entiers, de taille m et n respectivement, le problème est de calculer d le plus grand diviseur commun de A et B. L'algorithme systolique de Brent et Kung [BK83] est le suivant :

Algorithme 25 *Calcul de PGCD [Brent et Kung]*

$accu \leftarrow 0$ { cet accumulateur conserve le nombre de décalages effectués simultanément sur les opérandes. }
pour ($i = 1$ à $m + n$) et si $fini = Faux$ faire
1.diviser A et B par 2, jusqu'à ce que l'un des deux soit impair.
Cette opération est réalisée par α décalages et $accu \leftarrow accu + \alpha$.

2.diviser le nombre pair par 2 jusqu'à ce qu'il soit impair également, le *pgcd* d'un nombre pair et d'un nombre impair étant toujours impair. Cette opération se réalise également à l'aide de décalages.

3.
$$A \leftarrow \begin{cases} \frac{A+B}{4} & \text{si } A+B = 0 \; mod \; 4 \\ \frac{|A-B|}{4} & \text{sinon} \end{cases}$$
Cette opération requiert le calcul de $A+B$ et éventuellement de $|A-B|$, la division par 4 s'effectue par des décalages.

4.si $\frac{A-B}{2} < B$, échanger A et B.

5.si $A - B < 0$ alors
$d \leftarrow A$
$fini \leftarrow Vrai$

$d \leftarrow d * 2^{accu}$

fin

Le nombre d'alternances de ce circuit sera linéaire, ne serait-ce qu'à cause de l'écriture sous forme de boucle séquentielle de l'algorithme.
À chaque étape, le degré du circuit est multiplié par $\log n$ à chaque opération arithmétique, et par n par le comparateur. Le degré cumulé est finalement $\mathcal{O}(n^n)$, ce qui donne une complexité superlinéaire. Sur ce problème, notre estimateur de complexité s'avère moins performant qu'une expertise humaine: Kannan, Miller et Rudolph ont proposé un algorithme sous-linéaire pour ce problème [KMR84]. Il indique cependant que la parallélisation de ce problème n'est pas *a priori* évidente; c'est d'ailleurs un problème ouvert.

5.5 Autres domaines d'application

Il existe deux domaines où les calculs dans les semi-anneaux unitaires intègres du type $(\bar{\mathbb{R}}, \max, +, -\infty, 0)$, avec $\bar{\mathbb{R}} = \mathbb{R} \bigcup \{-\infty\}$ ou $(\mathbb{R}^+, \max, *, 0, 1)$ sont monnaie courante [CG79].

5.5.1 Cheminements dans les graphes

Le premier est celui de la théorie des graphes, où le problème est de parcourir un graphe à la recherche d'une solution vérifiant un critère donné. Les problèmes d'optimisation font intervenir le plus souvent les opérateurs min ou max et $+$, mais également pour tester la fiabilité d'un réseau, les opérateurs Δ et $\times$. Les problèmes d'énumération sont résolus par des parcours à la recherche de toutes les solutions résolvant un problème donné, l'une des opérations étant alors l'union ensembliste, l'autre permettant de vérifier si l'ensemble trouvé répond à la question. Les problèmes de connexité et de dénombrement enfin utilisent des ensembles de nombres ≥ 0 sur lesquels agissent min et max. Dans ce contexte, les structures algébriques de semi-anneaux et de treillis sont regroupées sous le vocable d'« algèbre de chemins » [GM79].

5.5.2 Réseaux de Petri temporisés

La simulation de systèmes à événements discrets, ou divers problèmes de production (étude et optimisation d'ateliers de production par exemple), sont modélisés par des réseaux de Petri temporisés [CMQV85]. Ils peuvent être représentés à l'aide de la structure algébrique $(\bar{\mathbb{R}}, \max, +, -\infty, 0)$, noté $(\bar{\mathbb{R}}, \oplus, ., \epsilon, 0)$, qui est un semi-anneau.

Un réseau de Petri est basé sur un graphe dont les sommets sont de deux sortes, les *places* et les *transitions*, tels que tout chemin est formé d'une alternance de places et de transitions. Le fonctionnement d'un réseau de Petri est le suivant : chaque place contient des jetons, appelés *marques*, et les jetons circulent dans le réseau selon une règle simple : quand toutes les places en amont d'une place considérée possèdent au moins un jeton chacune (on dit alors qu'elles sont *actives* ou *marquées*) et que ceux-ci ont séjourné un certain temps dans leur place, ils franchissent la transition les séparant de la place étudiée. Les places en amont perdent alors un jeton chacune, on dit qu'elles ont *consommé* un jeton. Ces réseaux sont temporisés, ce qui signifie que chaque jeton doit passer une certaine durée dans une place, durée dépendant uniquement de la place où ils se trouvent, et qu'à chaque transition est également associée une durée. Si chaque place est liée à au plus une transition en amont et une transition en aval (pour que l'on sache quelles sont les places qui activent une place donnée, et quelle sera la place activée par une place qui consomme son jeton) et si la règle de fonctionnement adoptée est le déclenchement au plus tôt des transitions, tout indéterminisme quant au fonctionnement de ces réseaux est supprimé.

Un tel réseau permet de décrire des événements discrets. Les événements étudiés ne sont pas, en général, l'évolution au cours du temps de l'ensemble des marques accessibles, mais plutôt les dates d'occurrence des transitions. Si $X_i^{(n)}$ est la date de la $n^{ième}$ occurrence de la transition i, $X_i^{(n)}$ dépend linéairement des dates $X_j^{(n-k)}$ d'occurrences précédentes de la même transition et des autres transitions ; il est possible d'écrire ces dépendances par une suite de transformations linéaires

$$X^{(n)} = X^{(n)}A_0 \oplus \ldots \oplus X^{(n-\alpha)}A_\alpha,$$

où $X^{(n)}$ est le vecteur formé par les $X_i^{(n)}$, α est une donnée du réseau et les A_j sont des matrices dépendant des conditions initiales. Le problème est de résoudre le système linéaire définissant $X^{(n)}$. En général, la symétrisation de la loi max, c'est-à-dire l'introduction d'une soustraction, détruit le lien entre l'opération max et l'existence d'un ordre total et n'aide pas à la résolution des problèmes. Il existe une théorie des pseudo-inverses A^* de matrices dans cette structure, $A^* = A^0 + A^1 + \ldots A^{n-1}$ où n est le nombre de transitions du réseau. La solution au problème précédent est alors $X^{(n)} = X^{(n-1)}.A_1.A_0^* \oplus \ldots \oplus X^{(n-\alpha)}.A_{n-\alpha}.A_0^*$. Ceci est un exemple de calcul possible dans ce formalisme. Les graphes étant souvent importants, il est intéressant de paralléliser les calculs. Les algorithmes d'évaluation, utilisés dans un compilateur comme celui qui sera présenté au chapitre suivant, trouvent là une utilité supplémentaire. De plus, même si les réseaux de Petri utilisés admettent plutôt des temporisations variables au cours du temps, la structure des matrices ne varie pas et la compilation reste possible.

5.6 Conclusion

Les algorithmes présentés en première partie de ce mémoire permettent de fournir facilement des estimations pas trop grossières de la complexité parallèle d'un problème. Si une estimation ainsi produite est polylogarithmique, elle signifie que le problème donné admet une bonne parallélisation et que les recherches en ce sens ont de bonnes chances d'aboutir (ne serait-ce qu'en utilisant les outils de parallélisation automatique présentés au chapitre 6.1). Si en revanche l'estimation est une quantité trop grande, tout espoir de parallélisation n'est pas perdu, mais la tâche risque d'être plus complexe.

Chapitre 6

Compilation et parallélisation

Résumé

Ce chapitre présente les modalités d'emploi des algorithmes d'évaluation pour une compilation/parallélisation des programmes sans boucle. La compilation de tels programmes fournit des programmes parallèles dont la structure est connue et sur laquelle un placement est possible. Dans cette optique, une extension du langage C a été étudiée et un prototype de compilateur a été réalisé. L'opération la plus coûteuse de ce compilateur, le produit de matrices, a été étudiée tant du point de vue théorique que du point de vue pratique. Une autre approche, qui s'insère dans les travaux classiques en parallélisation automatique, est l'aide à l'identification de réductions dans un programme parallèle.

Les algorithmes d'évaluation présentés dans la première partie de cette thèse peuvent être utilisés comme interpréteurs. Cependant, les résultats temporels obtenus sur des machines parallèles à mémoire distribuée ne sont pas encore établis, à cause des communications qui ne sont pas structurées. L'idée est d'utiliser les algorithmes d'évaluation pour compiler des circuits en d'autres circuits moins profonds, qui calculent encore les mêmes résultats. Nous avons défini une extension du langage C, dans laquelle il est possible d'annoter les parties de programmes à paralléliser par une indication sur la nature de la structure algébrique utilisée. Un prototype de compilateur a été réalisé. La grammaire définie pour ce langage sera présentée, ainsi que l'organisation du compilateur.

Le problème majeur se révèle alors être le produit de matrices : le temps de compilation devient très dépendant du temps de ce produit. Les algorithmes rapides de produit de matrices ont été étudiés ; un algorithme applicable dans les anneaux est celui de Strassen [Str69] ou celui de Strassen-Winograd [Win73] et ils sont programmables. D'autres algorithmes plus rapides ont été proposés, mais il s'agit d'algorithmes approchés (donc dans $\mathbb{R}$ ou $\mathbb{C}$) et théoriques, dont la raison d'être est de fournir une majoration de ω, avec ω l'exposant qui apparaît dans la complexité du produit de matrices : $\mathcal{O}(n^\omega)$. À l'heure actuelle, la meilleure majoration, déterminée par Coppersmith et Winograd, est de 2,376. Cependant, la démonstration de cette borne n'est pas constructive. De plus, ces produits s'appliquent à des matrices pleines, alors que l'étude statistique réalisée au chapitre 5 montre que les

matrices sont très creuses et pas structurées. Nous décrirons l'implémentation réalisée pour ces matrices.

Nous tracerons ensuite les grandes tendances en parallélisation automatique et plus particulièrement les travaux d'identification de réduction (opération de préfixe) dans des nids de boucles. Ces techniques sont fragiles, souvent dépendantes de l'écriture du programme, et mettent en œuvre des techniques sophistiquées pour résoudre ce problème. Notre utilisation des bornes de complexité des algorithmes d'évaluation permet d'estimer le degré et le nombre d'alternances du programme et cela relativement indépendamment de l'écriture du programme. Une interprétation immédiate de ces valeurs fournit une prédiction de l'existence ou non de réductions et même de réductions pour lesquelles la fonction associative mise en jeu n'est pas simplement l'addition.

6.1 Compiler : pourquoi, comment ?

6.1.1 Pourquoi compiler plutôt qu'interpréter ?

Tels qu'ils sont présentés dans la première partie de cette thèse, les algorithmes d'évaluation de circuits sont utilisés comme interpréteurs parallèles pour machines abstraites. Leur intérêt est théorique, mais leur utilisation comme interpréteurs sur des machines parallèles à mémoire distribuée nécessite une étude fine des communications pour parvenir à une implémentation efficace. En effet les communications ne sont absolument pas structurées. Supposons que le circuit à évaluer a n nœuds et que la machine parallèle cible a n processeurs. Dans le cas des expressions, chaque nœud n'a qu'un seul père susceptible d'utiliser sa valeur, il ne la communique qu'à un seul nœud et les communications peuvent être assimilées à des permutations. Sur la plupart des réseaux d'interconnexion classiques, une permutation est réalisable en un temps proportionnel au diamètre du réseau (cf. chapitre 2, §2.6.3). Dans le cas des circuits, on ne dispose plus d'une information aussi précise sur l'allure des communications et le seul schéma général que l'on puisse appliquer est un échange total (*all-to-all*), dont le temps d'exécution est minoré par le quotient du nombre de processeurs par le degré du réseau[1]. Le degré d'un réseau à n processeurs est limité pour des raisons architecturales et il est égal à une constante ou au plus au logarithme de n dans le cas d'un hypercube par exemple [CT93]. Le temps parallèle théorique de l'algorithme d'évaluation sera donc multiplié par un terme au moins égal à $\frac{n}{\log n}$, autant dire que le temps parallèle réel sera supérieur au temps séquentiel.

Ceci nous incite donc à utiliser les algorithmes d'évaluation parallèle pour compiler des programmes. En effet, à la compilation la structure du circuit parallèle est connue et, comme il n'y a pas de contrainte de temps, il est possible de calculer un placement du programme et même, éventuellement, de précalculer le routage des données. De plus, les matrices qui interviennent dans ces calculs sont très creuses et il est également possible de déterminer à

1. Le degré d'un réseau d'interconnexion est le nombre – supposé identique pour tous les processeurs du réseau – de voisins d'un processeur.

l'avance quels seront les produits $a_{i,k} * b_{k,j}$ à calculer et comment les effectuer rapidement. Les statistiques effectuées au chapitre 5 indiquent que si seuls les calculs utiles sont effectués alors le nombre d'opérations induites par les produits de matrices va diminuer de façon spectaculaire. Même pour le calcul de l'ensemble lexicographiquement maximal de sommets indépendants, qui est l'un des problèmes les plus durs à paralléliser, le nombre d'opérations d'un produit de matrices sera $\mathcal{O}(\sqrt{n}n^2)$ au lieu de $\mathcal{O}(n^3)$.

Remarque
Le niveau de parallélisation sera ici celui des opérations, ou grain fin, le grain étant défini comme le rapport du nombre de tâches à effectuer et du nombre de processeurs. Comme les machines actuelles sont prévues pour fonctionner avec des programmes à gros grain, il faut soit paralléliser des circuits effectuant des opérations coûteuses, comme celles qui interviennent en calcul formel (opérations sur des polynômes à coefficients rationnels, ou matrices à coefficients polynomiaux sur $\mathcal{Q}$, ...), soit utiliser des techniques de regroupement de tâches, ou *clustering*. Ces dernières tiennent compte des relations de précédence et des communications qui existent entre les tâches pour les regrouper selon différentes heuristiques [GY92].

6.1.2 Comment compiler avec des algorithmes d'évaluation ?

Maintenant que nous avons souligné l'importance des algorithmes d'évaluation de la partie I pour compiler des circuits arithmétiques plutôt que pour les interpréter, nous allons préciser de quelle manière cela peut être réalisé. Le principe ici consiste à tracer le graphe de tâches de l'algorithme d'évaluation appliqué au circuit à compiler. Ce graphe de tâches peut être produit facilement en remplaçant, dans l'algorithme d'évaluation, chaque calcul par une fonction de construction de tâches qui précise les envois et les réceptions de messages (valeur ou fonction d'arc) et les opérations à effectuer sur ces données. On obtient alors, selon les substitutions effectuées, la description d'un graphe de précédence et de communication qui sera pris en entrée d'un outil d'ordonnancement et de placement, ou un programme parallèle.

6.2 Le prototype de compilateur

Décrivons l'outil de compilation de façon plus précise et technique. On suppose dans la suite de cette section que le lecteur est un familier du langage C.

6.2.1 Un langage pour la parallélisation des programmes sans boucle

Nous avons tout d'abord défini un langage pour la parallélisation de programmes sans boucle, basé sur une version restreinte de C (pour simplifier l'analyse lexicale et syntaxique) et qui l'étend, pour comprendre des déclarations de structures algébriques et des instructions de parallélisation qui signifient qu'un bloc de calcul devra être parallélisé en accord avec la structure algébrique spécifiée au début de ce bloc.

Dans ce langage, quatre points sont à souligner :

- la définition de structures algébriques.
- le découpage du programme en parties séquentielles et en parties parallèles, le lien étant assuré par des variables qui « traversent » cette séparation, nommées variables `input` ou `output`.
- la parallélisation s'effectue au niveau des blocs `Par (...){ ...}`. Ces blocs sont signalés par le programmeur comme étant les parties à paralléliser dans son programme.
- le modèle de machine parallèle est à l'heure actuelle un modèle où les processus communiquent par envoi de message. Ce modèle pourra être un modèle fonctionnel (création de processus fils) comme en Athapascan, le langage parallèle développé par l'équipe APACHE (Algorithmique parallèle et partage de charge, LMC-LGI), sans changement dans le principe du compilateur. Il suffira de modifier la génération finale de code.

Remarque
Nous avons choisi C comme langage de base pour être en adéquation avec les machines parallèles dont nous disposions (le MegaNode de Telmat essentiellement). Il est clair qu'une implémentation dans un langage orienté objet (par exemple C++) aurait apporté plus de flexibilité, les structures algébriques pouvant être définies comme des classes génériques.

Un sous-langage de C

De C, notre langage reprend les caractéristiques les plus simples.

Déclarations Pour ce qui est des déclarations, les possibilités sont restreintes aux types de base : `void`, `char`, `short`, `int`, `long`, `float` et `double`. Les types qui se définissent en deux mots ne sont pas acceptés, comme par exemple `long int`, `signed int` ou `unsigned long`. À partir de ces types de base, les seuls constructeurs de types autorisés sont les pointeurs, les tableaux et les fonctions, `union` et `struct` ne sont pas autorisés.
Les classes de stockage autorisées pour ces objets sont les classes `extern`, `static`, `auto` et `typedef`. Le spécificateur `const`, présent dans la norme ANSI, n'est pas traité.
Les déclarations de fonctions doivent être sous forme ANSI, comme par exemple `int somme_iter (int *res, int a[n])`.
On ne peut déclarer qu'une variable à la fois, par exemple `int a, b;` n'est pas autorisé, il faut écrire

```
int a;
int b;
```

Les initialisations des variables lors de leur déclaration ne sont pas autorisées.

Instructions L'ensemble des instructions traitées reprenant presque toutes les instructions du C, nous allons procéder par exclusion : les instructions de sélection multiples (`switch`), les instructions de branchement (`goto`) et les définitions d'étiquettes (`label`) ne sont pas incluses dans le langage, de même que les opérateurs de pré- et post-décrémentation `--` et de pré- et post-incrémentation `++`, les opérateurs d'indirection `->` et . pour accéder aux champs des structures (ce qui est cohérent avec le fait que les structures ne sont pas autorisées) et les manipulations de chaînes de caractères.

Ce qui a motivé nos choix est la recherche de simplicité pour l'écriture de l'analyseur lexical et syntaxique, tout en laissant la possibilité d'exprimer la plupart des algorithmes. Les instructions omises représentant, pour une large part, des sucres syntaxiques, il reste effectivement possible de programmer avec ce sous-ensemble de C.

Une extension de C

Nous avons essentiellement ajouté un `typedefalgstruct` pour définir les structures algébriques (au sens mathématique du terme et non pas au sens du `struct` de C) et un délimiteur de bloc à paralléliser qui, dans sa syntaxe, inclut le nom de la structure algébrique employée.

Déclaration d'une structure algébrique La définition d'une structure s'écrit à l'aide du mot clé `typedefalgstruct`, suivi du nom de la structure et, entre accolades, les noms (précédés d'un mot clé pour indiquer de quoi il s'agit) du type des éléments, de la fonction d'affectation, facultativement de la fonction de test d'égalité (nous avons également rajouté le type `BOOLEAN` et les deux valeurs `TRUE` et `FALSE`, pour pouvoir définir ces fonctions d'égalité) et ensuite, une liste des noms des opérations d'addition, de soustraction, de multiplication, de division, des neutres pour l'addition et la multiplication, si tous ces objets sont définis pour la structure en question. Par exemple, pour définir le semi-anneau des entiers naturels, en précision infinie, on écrira :

```
typedefalgstruct  SemiRing {
        type TypeNaturel,
        affect AffectNaturel,
        equality EgalNaturel,
        plusneutral Zero,
        plus Plus,
        multneutral Un,
        mult Mult
} Naturel;
```

Les mots clés pour déclarer une structure algébrique sont `Monoide`, `Group`, `SemiRing`, `Ring`, `Field` et `Lattice`.
Les prototypes de fonctions doivent être compatibles avec leur sémantique d'opérations sur la structure, par exemple, la fonction d'addition `Plus` a comme prototype `TypeNaturel *Plus (TypeNaturel *a, TypeNaturel *b)` et les éléments neutres doivent appartenir à la structure, par exemple `Zero` est déclaré comme `TypeNaturel Zero;`.

Le délimiteur de bloc s'appelle `Par`, il s'utilise suivi, entre parenthèses, du nom de la structure algébrique utilisée, qui a été définie à l'aide du mot clé `typedefalgstruct`, puis, entre accolades, du bloc d'instructions : `Par (< nom de la structure algébrique >) {< Bloc >}`.

Par exemple, pour définir un bloc utilisant le semi-anneau `Naturel`, on écrira

```
Par(Naturel) { Bloc }
```

À l'intérieur d'un bloc, les variables sont obligatoirement définies avec l'une des trois classes de stockage suivantes :

`input` :	il s'agit d'une feuille, il faudra générer le code de réception de sa valeur au début du programme parallèle
`output` :	le nœud correspondant du circuit est un nœud de sortie, il faudra générer le code d'envoi de sa valeur à la fin du programme parallèle
`parallel` :	le nœud correspondant est un nœud interne, il sera détruit en fin de calcul parallèle (cette classe est la classe par défaut, analogue à la classe `auto`)

Pour que le programme puisse recevoir les valeurs et envoyer les résultats, il faut que les variables `input` et `output` soient déclarées à l'extérieur du bloc parallèle. Seules les variables de classe `parallel` sont déclarées à l'intérieur du bloc.

Ensuite viennent les instructions. Pour que les instructions correspondent à un circuit, les tests portant sur des expressions non évaluables à la compilation sont interdits, ainsi que les appels à des fonctions autres que celles définissant la structure algébrique et dont la liste est donnée lors de la déclaration de la structure.

Exemple

Par exemple, le produit de matrices sur des doubles, de taille 100×100 s'écrit :

```
/* Ce programme effectue un produit de matrices 100x100. */
/* Le produit de matrices sera parallelise.              */
double NeutreMult;          /* A initialiser avant      *
double NeutrePlus;          /* d'utiliser la structure  */

/* L'addition dans la structure                          */
double *Plus (double *a, double *b)
{
...
}

/* La multiplication dans la structure                   */
double *Mult (double *a, double *b)
{
...
}
```

```
/* L'affectation dans la structure                        */
double *Affect(double *a, double *b)
{
...
}

/* Definition d'un semi-anneau base sur les doubles       */
typedefalgstruct SemiRing {
        type double;
        affect Affect;
        plusneutral NeutrePlus;
        multneutral NeutreMult;
        plus Plus;
        multiply Mult
} doubleSR;
/* L'ordre n'a pas d'importance dans typedefalgstruct.    */

/* La procedure de produit de matrices                    */
/* Elle contient un bloc a paralleliser.                  */
void Produit_Mat_Par(double A[100][100], double B[100][100],
                     double C[100][100])
{
   Par(doubleSR){
     input A;
     input B ;
     output C ;
     int i ; int j ; int k ;

     for (i=0 ; i<100; i=i+1 )
       {
       for (j=0 ; j<100; j=j+1)
         {
         /*C[i][j] = 0 ;*/
         Affect(&(C[i][j]), &NeutrePlus);
         for (k=0 ; k<100; k=k+1 )
           /*C[i][j] = C[i][j] + A[i][k] * B[k][i] ;*/
           Affect(&(C[i][j]),
                  Plus(&(C[i][j]),
                       Mult(&(A[i][k]), &(B[k][j]))));
         }
       }
     }
}
```

Maintenant que le langage est défini, passons à l'architecture du compilateur.

6.2.2 L'organisation du compilateur

Détaillons chacune des phases du compilateur, dont le schéma général est représenté sur la figure 6.1 :

- la phase d'*analyse*. Elle contient les analyses lexicale, syntaxique et sémantique du programme source. L'analyseur lexical et syntaxique a été réalisé à l'aide de LEX a pour l'analyse lexicale et de BISON pour l'analyse syntaxique.

- la phase d'*expansion*. Elle s'effectue en trois temps : tout d'abord, le programme correspondant à la partie séquentielle est produit, sans le code pour les envois de données et les réceptions de résultats qui sera généré à la fin de la phase de contraction. Dans un deuxième temps, un autre programme est obtenu à partir des blocs `Par:` chaque opération $+$, $*$,... est remplacée par une fonction de construction d'un nœud du DAG correspondant au bloc parallèle. Par exemple, `input a` sera remplacé par « nom parallèle associé `= MakeLeaf ();` », et `Plus(a,b)` sera remplacé par « variable parallèle associée à ce nœud `= MakeNode(+,` la variable désignant le code de `a,` la variable désignant le code associé à `b);` ». Ces deux étapes logiques sont réalisées simultanément, en une seule passe sur le programme source. Finalement, le programme correspondant à la partie parallèle est compilé par un compilateur C classique et exécuté. Le résultat est le graphe d'exécution correspondant à la partie parallèle du programme source.

- la phase de *contraction*. Cette phase utilise de façon symbolique l'algorithme d'évaluation qui correspond à la structure spécifiée. Ce compilateur opère donc sur un graphe de calcul (construit par la phase d'expansion), à l'aide d'une fonction de contraction qui prend un DAG en argument. Comme le modèle de machine est un modèle par processus communicants, il y a génération de code pour les communications. Avec un modèle par lancement de processus fils, le code généré serait une création de *thread*.

  Cet algorithme symbolique fabrique les tâches du graphe d'exécution de l'algorithme d'évaluation, ainsi que le code pour les communications. (Pour les corps, la seule opération appliquée aux divisions est la procédure $\acute{E}val_*$.) À l'heure actuelle, cette partie n'étant réalisée que pour les arbres, le problème des produits de matrices ne s'est pas encore posé.
  Cette phase de contraction génère également le code pour les communications effectuées par la partie séquentielle vers la partie parallèle (correspondant aux variables `input` et `output`)). La phase de contraction peut être complétée pour générer également un graphe de tâches correspondant au programme parallèle produit, sous un format acceptable par un outil de *clustering* si le besoin s'en fait sentir, puis de placement/ordonnancement.

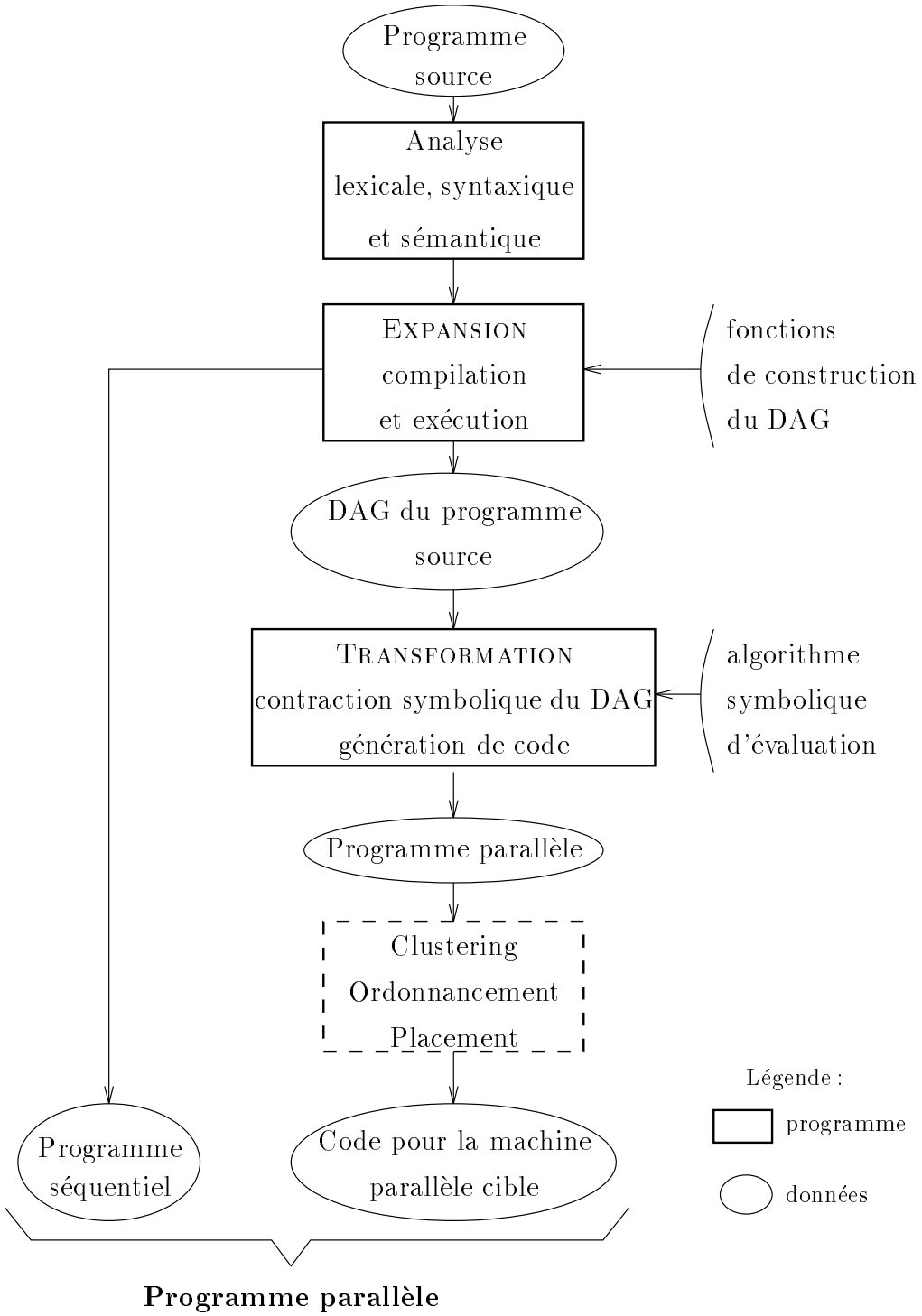


FIG. 6.1 - *Schéma du compilateur.*

Le langage cible de ce prototype est OUF (*Occam Users Frustration*) [BFP+91], une bibliothèque développée au LMC qui permet de s'affranchir de la machine cible, en offrant

la possibilité d'écrire les communications comme pour une machine à réseau complet et qui s'occupe de l'interface avec un routeur. Le modèle de programme parallèle est un modèle par tâches communicantes, les synchronisations entre tâches s'effectuant par envoi de message. Il existe une tâche reliée au monde extérieur, qui gère les réceptions de données et les envois de message.

Ce compilateur a été réalisé avec Thierry Gautier, en stage de DEA, sous une première forme, restreinte aux semi-anneaux [Gau92]. Son développement a été stoppé par l'attente de l'avènement d'un langage nouveau et plus expressif. Ce prototype est donc voué à une extension prochaine vers la production de programmes parallèles en langage Athapascan, puisque ce langage est désormais disponible sur réseau de stations comme sur la dernière recrue de l'équipe, un SP1 IBM. Il devra être augmenté, de façon à traiter les circuits arithmétiques généraux.

6.3 Les produits de matrices

Comme cela a été mentionné, les produits de matrices induits par l'opération *Group* ne sont pas gérés actuellement dans ce compilateur. Pourtant cette partie du programme est la plus coûteuse à la fois en temps et en espace, parce que les matrices d'adjacence sont rapidement de grande taille, même pour de petits problèmes. Pour pallier à ces problèmes, nous avons étudié les algorithmes rapides de produits de matrices et nous avons porté nos efforts sur la gestion de matrices creuses, en vue de leur intégration au compilateur.

6.3.1 Algorithmes théoriques de produits de matrices rapides

Une première idée, en présence de produits de matrices gourmands en temps, est de chercher une réponse parmi la panoplie des produits de matrices rapides. En effet, depuis 1969, il existe un produit de matrices simple,proposé par Strassen [Str69], dont la complexité est $\mathcal{O}(n^{2,808})$. En vingt-cinq ans, la recherche dans ce domaine a été active [Gas71, Lad76, BCRL79, Pan80, Sch81, CW82, Pan84a] – voir la présentation de [Pan84b] sur ce sujet – et en 1987 Coppersmith et Winograd [CW90] ont proposé une borne de $\mathcal{O}(n^{2,376})$ pour le produit de matrices.

L'algorithme de Strassen est un algorithme très simple dans son écriture. Il suppose seulement que les opérations s'effectuent dans un anneau plutôt qu'un semi-anneau. Nous présentons uniquement l'algorithme de Strassen-Winograd [Win73], qui réduit la constante multiplicative qui est cachée dans le $\mathcal{O}$ par rapport à l'algorithme original de Strassen. Si les matrices à multiplier sont deux matrices 2×2, A et B et que $C \leftarrow A.B$, l'algorithme s'écrit :

$$C = A.B = \begin{pmatrix} M_0 + M_1 & M_0 + M_4 + M_5 - M_6 \\ M_0 - M_2 + M_3 - M_6 & M_0 + M_3 + M_4 - M_6 \end{pmatrix}$$

avec

$$\begin{aligned} M_0 &= A_{1,1}.B_{1,1}, \\ M_1 &= A_{1,2}.B_{2,1}, \\ M_2 &= A_{2,2}.(B_{1,1} - B_{1,2} - B_{2,1} + B_{2,2}), \\ M_3 &= (A_{1,1} - A_{2,1}).(B_{2,2} - B_{1,2}), \\ M_4 &= (A_{2,1} + A_{2,2}).(B_{1,2} - B_{1,1}), \\ M_5 &= (A_{1,1} + A_{1,2} - A_{2,1} - A_{2,2}).B_{2,2}, \\ M_6 &= (A_{1,1} - A_{2,1} - A_{2,2}).(B_{1,1} - B_{1,2} + B_{2,2}). \end{aligned}$$

On peut l'appliquer pour des matrices de taille quelconque, en découpant les matrices en 4 blocs. Cet algorithme est récursivement appliqué aux blocs plus petits. La complexité de cet algorithme récursif est, si $t_m(n)$ est le nombre de multiplications effectuées pour un produit de matrices de taille $n \times n$ et $t_a(n)$ est le nombre d'additions :

$$\begin{aligned} t_m(n) &= 7t_m(\tfrac{n}{2}), \\ t_a(n) &= 7t_a(\tfrac{n}{2}) + 15\tfrac{n^2}{4}, \end{aligned}$$

ce qui donne finalement :

$$\begin{aligned} t_m(n) &= n^{\log_2 7}, \\ t_a(n) &= 5n^{\log_2 7} - 5n^2. \end{aligned}$$

Cet algorithme, ainsi que plusieurs variantes, ont été expérimentés sur le MegaNode et il apparaît que pour des tailles de matrices supérieures à 100, ce produit devient intéressant [RT92]. En séquentiel, la valeur à partir de laquelle il devient plus rapide que le produit classique est également de 50.

Remarque
La restriction portant sur la structure d'anneau peut être levée, en considérant qu'il est possible de symétriser un semi-anneau pour ajouter une loi $-$ par la méthode dite justement de symétrisation (sous certaines conditions algébriques, la régularité par exemple). Si la structure est un semi-anneau $(S, +, *, 0)$ régulier par exemple, la relation d'équivalence $\mathcal{R}$ est définie par $(a, b)\mathcal{R}(c, d)$ si et seulement si $a + d = c + b$. Chaque couple représente la différence $a - b$. Si on travaille modulo $\mathcal{R}$, l'addition s'étend sur $S/\mathcal{R}$:

$$(a, b) + (c, d) = (a + c, b + d)$$

et la multiplication également :

$$(a, b) * (c, d) = (a * c + b * d, a * d + b * c)$$

En appliquant cette symétrisation, la soustraction est implantée comme $(a, b) - (c, d) = (a + d, b + c)$ et l'algorithme de Strassen est utilisable.

Si maintenant on s'intéresse aux produits de matrices plus rapides, on s'aperçoit très vite qu'ils ne sont pas conçus pour fournir des algorithmes effectifs, mais pour approcher au

plus près la complexité intrinsèque du produit de matrices, le ω de la complexité en $\mathcal{O}(n^\omega)$. Trivialement, on a $2 \leq \omega \leq 3$ et même, grâce à l'algorithme de Strassen, $2 \leq \omega \leq 2,808$. Les méthodes employées reposent sur des calculs de rang approché du tenseur correspondant au produit de matrices plutôt que sur des réorganisations des calculs. Coppersmith et Winograd ont ainsi déterminé une valeur de 2,376 pour ω [CW90]. Une version simple de leur démonstration fournit pour ω la valeur 2,41. Cette démonstration sera esquissée dans le seul but de convaincre le lecteur de l'aspect mathématique théorique pur de ce ω. L'idée consiste à utiliser un algorithme qui multiplie des matrices 1×1 par des matrices $1 \times q$ et des matrices $q \times 1$ par des matrices 1×1 simultanément avec $q + 2$ multiplications, pour multiplier des matrices carrées. On effectue des produits de matrices carrées de taille quelconque en appliquant récursivement l'« algorithme » précédent à des matrices 3×3, avec $3N$ niveaux de récursion. On obtient 3^{3N} produits de blocs élémentaires. Il est alors possible, si on sait calculer ces 3^{3N} produits, de calculer $(\frac{1}{4})^N 3^{3N} = (\frac{27}{4})^N$ produits indépendants de blocs élémentaires, grâce au théorème de Salem et Spencer qui permet d'assurer l'existence d'un ensemble d'indices tel que, si on suppose que les blocs dont le numéro n'appartient pas à cet ensemble sont nuls, alors tout produit restant n'utilise jamais le même bloc qu'un autre produit. Si les blocs élémentaires sont de taille q^N et sont multipliés en temps $q^{N\omega}$, alors le τ-théorème de Schönhage conduit à l'inégalité

$$(q+2)^{3N} \geq (\frac{27}{4})^N q^{N\omega}.$$

Pour $q = 8$, on obtient la valeur minimale de ω, soit $2,404$.

Les autres algorithmes conduisant à des valeurs de ω intermédiaires entre 2,808 et 2,376 sont basés sur des calculs du même type et ne sont pas, par conséquent, plus programmables.

6.3.2 Implémentation des matrices et de leurs produits

Lors des simulations séquentielles, des statistiques sur le nombre de coefficients non nuls par ligne dans les matrices d'adjacence des circuits ont été réalisées. Pour les problèmes simples, comme l'addition ou la multiplication de deux entiers de n bits, ce nombre de coefficients est en moyenne de 2, au maximum de 5 et le produit de matrices correspondant, où seuls les coefficients non nuls sont pris en compte, a une complexité en $\mathcal{O}(n^2)$. Pour le problème plus complexe du $LMISP$, le nombre de coefficients est une racine carrée du nombre de nœuds du circuit, ce qui donne pour complexité $\mathcal{O}(\sqrt{n}n^2) = \mathcal{O}(n^{2,5})$.

Les matrices d'adjacence sont donc très creuses, c'est entendu, mais également très peu structurées. Comme les circuits sont des graphes orientés sans circuit[2], il existe une numérotation des nœuds qui rend cette matrice triangulaire. En particulier, quand le circuit est construit à partir d'un programme sans boucle, la numérotation des nœuds dans leur ordre d'apparition permet d'obtenir des matrices triangulaires inférieures. Cependant, cette structure triangulaire mise à part, il ne se dégage pas de structure plus fine de la matrice.

2. Le vocabulaire de la théorie des graphes et celui de la complexité parallèle interfèrent parfois fâcheusement.

Le stockage qui nous a paru adéquat a été un stockage par listes d'adjacence, par ligne et par colonne pour pouvoir multiplier facilement une matrice par elle-même, et doublement chaînées pour permettre d'insérer et de supprimer facilement un élément (sinon il faut parcourir à la fois la ligne et la colonne contenant cet élément pour mettre les pointeurs à jour). Une matrice est représentée figure 6.2 pour illustrer ce stockage.

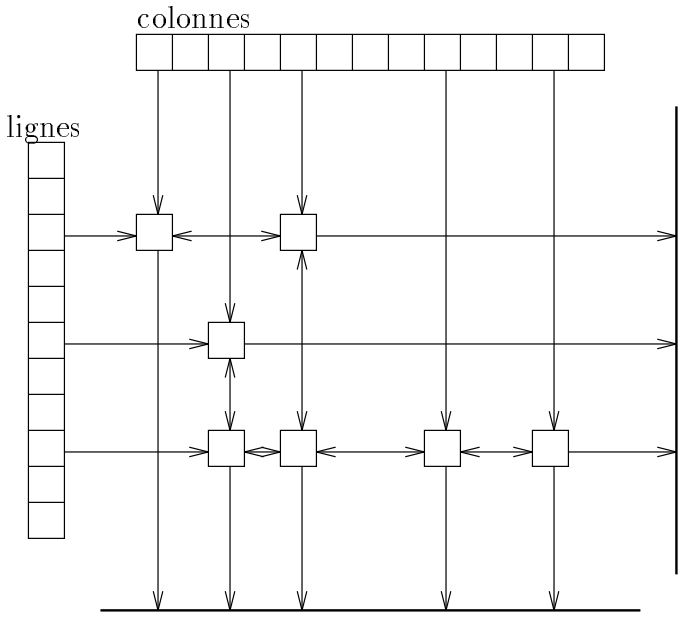


FIG. 6.2 - *Stockage des matrices par listes d'adjacence.*

Chaque coefficient de la matrice est composé de :

- un pointeur vers son prédécesseur et son successeur dans la même ligne ;
- un pointeur vers son prédécesseur et son successeur dans la même colonne ;
- les numéros de la ligne et de la colonne auxquelles il appartient ;
- le type d'arc qu'il représente ; ceci nous permet de stocker toutes les matrices d'adjacence U^{++}, U^{+*}, ... en une seule, sans risque de conflit puisque ces matrices sont, de par leur définition, disjointes ;
- le codage de la fonction d'arc portée par l'arc représenté par cet élément de matrice.

Cette gestion de matrices devient un peu complexe mais elle a permis de nous affranchir des limitations dues à la mémoire utilisée, qui apparaissaient très rapidement avec des matrices pleines.

Enfin, il est possible de prédéterminer, à la compilation, quels seront les produits $a_{i,k} * b_{k,j}$ à effectuer et de ne générer que le code qui leur est propre. La génération de code ne devrait

pas devenir beaucoup plus ardue, en revanche le nombre de tâches du programme parallèle s'en trouvera considérablement diminué.

6.3.3 Utilisation d'autres outils

Le compilateur fournit en sortie un programme dont les tâches sont de la granularité des opérations de base. Cette granularité peut ne pas correspondre à la granularité de la machine parallèle : par exemple, si les tâches sont des opérations booléennes et que la machine cible est le SP1. Si la granularité des tâches est beaucoup trop importante, comme des opérations sur des rationnels en précision infinie alors que les processeurs sont des processeurs élémentaires, du style Connection Machine 2, on ne peut rien faire, à part recompiler en remplaçant les opérations rationnelles par les opérations booléennes qui effectuent ces opérations rationnelles. Le cas de figure inverse est plus probable. Il sera donc utile d'utiliser soit un outil dédié au regroupement de tâches, ou *clustering*, soit un outil de placement qui inclut cette phase de *clustering*, pour obtenir des tâches de granularité désirée. Un outil de placement/ordonnancement permettra ensuite d'améliorer le programme produit.

L'utilisation de ces outils implique une génération du graphe de tâches correspondant au programme parallèle, dans le format d'entrée de l'outil, et ensuite de savoir exploiter les résultats pour réécrire les tâches en des tâches plus importantes.

6.3.4 Critiques

Une critique souvent formulée à l'encontre de ce type de compilateur est le fait qu'il déroule entièrement le graphe d'exécution du programme séquentiel. Cette méthodologie implique des limites sur la taille des programmes traités : il faut pouvoir stocker ces derniers complètement déroulés et, pour cela, il faut que le compilateur dispose d'une mémoire assez grande. Elle implique aussi que chaque changement de taille des données donne lieu à une nouvelle compilation.

À cela, plusieurs réponses sont possibles :

- tout d'abord, comme la possibilité de ne paralléliser que des parties de code est offerte, il est possible de découper le programme séquentiel en plusieurs blocs parallèles séparés par des frontières de synchronisation, les blocs parallèles étant effectués l'un après l'autre. Ceci permet de compiler de gros programmes sans saturer la mémoire du compilateur. Pour les gros programmes, pour lesquels l'on désire changer la taille des données, il est raisonnable d'espérer que le programme est écrit à l'aide de boucles imbriquées, en particulier s'il s'agit d'un programme de calcul numérique. Il est donc possible de découper une boucle exécutée n fois, où n est un multiple de 1000 :

```
pour i=1 a n faire
  f(i)
```

en

```
pour i1=1 a (n/1000) faire
  pour i2=1 a 1000 faire
    f(1000*(i1-1)+i2)
```

On pourra ne paralléliser que le bloc de taille 1000 et faire varier n sans tout recompiler.

– On peut enfin n'utiliser que les bornes de complexité obtenues pour les algorithmes d'évaluation parallèle, pour ajouter un peu d'expertise dans les compilateurs qui proposent de la parallélisation automatique. C'est ce qui sera détaillé au prochain paragraphe.

6.4 Utilisation en parallélisation automatique

En parallélisation automatique, plusieurs approches coexistent, qui reprennent les grands principes de l'algorithmique parallèle énoncés au chapitre 1.

Deux techniques peuvent être regroupées sous un même dénominateur : le travail sur le graphe de précédence des tâches. Les études en ordonnancement prennent en entrée un graphe de précédence et un nombre de processeurs, qui sont supposés totalement connectés ; elles ont pour but de fournir une date d'exécution pour chaque tâche, l'objectif étant de minimiser le temps total d'exécution [Chr92]. Le graphe de tâches est supposé entièrement déroulé et la durée de chacune d'entre elles connue. Les critiques adressées à notre compilateur sont donc encore de mise ici.
La deuxième grande famille de travaux sur les graphes de précédence est celle sur les nids de boucle [Lam74, Pol88, DR92, Fea92]. Un programme constitué de boucles imbriquées[3] et d'une série d'instructions placées au cœur de ce nid de boucles est donné en entrée ; ce programme est représenté par un graphe de dépendance condensé, où une dépendance du type

$$A[i_1 \dots i_n] \leftarrow B[j_1 \dots j_n]$$

se représente dans $\mathbb{Z}^m$ par un vecteur $(j_1 - i_1, \dots j_n - i_n)$ et ces vecteurs sont supposés indépendants des valeurs particulières de $j_1, \dots j_n, i_1, \dots i_n$. On évite ainsi de dérouler le graphe de précédence de tout le programme. Les travaux portent sur le choix d'une permutation de boucles valide (telle que le programme transformé calcule les mêmes résultats que le programme initial) qui ait un graphe de dépendance condensé contenant plus de parallélisme. On peut également interpréter géométriquement ce travail comme la recherche d'un hyperplan h, qui contienne un maximum de sommets indépendants et d'un vecteur π de translation de h, tel qu'à l'étape t, on exécute toutes les tâches d'indices $i_1, \dots i_n$ tels que $\pi.(i_1 \dots i_n) = t$ (π est appelé vecteur de temps). Le choix de l'hyperplan correspond à une réécriture aisée des bornes pour les indices de boucle (cf. figure 6.3).

3. Les bornes des boucles internes sont supposées être des fonctions affines des indices des boucles englobantes, ou des extrema (max pour la borne inférieure, min pour la borne supérieure) d'un ensemble fini de fonctions affines.

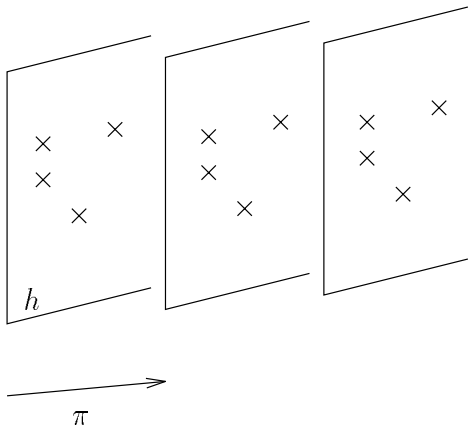


FIG. 6.3 - *Interprétation géométrique.*

Dans ces travaux sur les nids de boucles, l'introduction de redondance permet souvent d'améliorer grandement la qualité de la parallélisation, en dupliquant certaines variables temporaires qui servent d'accumulateur comme dans un produit scalaire par exemple, ou en dupliquant certains calculs ; souvent des dépendances sont ainsi brisées.

Enfin, des travaux de recherche de réductions sont menés par [JD89, PP91, Cal91, Red94, RF94]. Les réductions sont des calculs de préfixe et le problème est de les détecter et d'identifier la fonction associative mise en jeu. Ces réductions sont des schémas de calculs qui sont bien parallélisés, à tel point qu'ils font partie intégrante des nouveaux langages parallèles : MPI, HPF, où les `scatter<op>` sont désormais des fonctions intrinsèques (*intrinsic*) du langage, avec `<op>` qui peut être `max`, `min`, `sum`, `product`, `all`, `any`, `count`,...Il est cependant assez difficile de les détecter. Plusieurs méthodes ont été proposées. Jouvelot et Dehbonei [JD89] remplacent chaque donnée utilisée au cœur de la boucle par un symbole, puis essaient d'identifier le motif obtenu avec un motif connu correspondant à une fonction associative. Pinter et Pinter [PP91] travaillent sur le graphe de dépendance condensé respectant le nid de boucles et le déroulent jusqu'à ce que toutes les dépendances lient des instructions calculées à la même itération ou à l'itération précédente. De nouveau, une recherche du motif obtenu dans une base de données contenant les motifs correspondants à un certain nombre de fonctions associatives permet de conclure. Enfin, Redon et Feautrier [Red94, RF94] ont mis au point un prototype qui transforme un tel programme en un ensemble d'équations linéaires de récurrence portant sur les variables et les indices (pour indiquer les bornes des indices), ensuite le compilateur normalise, en un certain sens, le système d'équations de récurrence, puis il réécrit cette récurrence avec un formalisme inspiré du λ-calcul (le prototype est écrit en Lisp, ce qui éclaire ce choix) et enfin compare le résultat à une base de données d'exemples connus.

Toutes ces techniques de recherche de motifs déjà connus sont basées sur des transformations assez lourdes (6000 lignes de Lisp pour l'outil de X. Redon, sans compter l'utilisation de logiciels de description de polyèdres par leurs sommets à partir d'une caractérisation par des inégalités par exemple). Il serait alors possible d'aiguiller les outils de recherche de réduction

en effectuant une première passe, où chaque opération dans la boucle d'indice $i_1, \ldots i_n$ serait remplacée par les calculs du degré et du nombre d'alternances h_a correspondants. On obtiendrait deux relations de récurrence simples qui peuvent être résolues par les outils adéquats de X. Redon ou tout simplement par la fonction $rsolve$ de résolution d'équations de récurrence de Maple. Les ordres de grandeur du degré et du nombre d'alternances en fonction des bornes des boucles fourniraient une indication sur l'existence possible ou non de ces réductions et leur type. Si h_a est petit, il y aura certainement des réductions possibles. Si le degré aussi est petit, elles impliqueront des additions, sinon il faut rechercher des fonctions associatives à base de multiplications. Cette technique présente l'avantage d'éviter de dérouler le graphe de précédence et d'être assez insensible à l'écriture du corps de la boucle, ce qui constitue un des handicaps fréquents des méthodes plus classiques. De plus, elle indique l'existence de réductions même dans le cas où la fonction associative n'est pas réduite à +, par exemple dans le programme de produit scalaire,

`pour` $i = 1$ `à` n `faire`

$$f(i) \leftarrow f(i-1) + a(i) * b(i)$$

où a et b sont des vecteurs de valeurs, le degré est égal à 2 et h_a vaut 3, ce qui signifie que l'opération effectuée utilisera additivement la variable sur laquelle porte l'itération, f en l'occurrence. Si seul le degré est petit (par exemple linéaire), le schéma à rechercher sera vraisemblablement de la forme

`pour` $i = 1$ `à` n `faire`

$$f(i) \leftarrow f(i-1) * a(i) + b(i).$$

Utiliser ce précalcul simple permettrait de guider la recherche de récurrences ou éventuellement de la décourager.

6.5 Conclusion

Dans ce chapitre, nous avons présenté notre principe d'utilisation des algorithmes d'évaluation pour une compilation des programmes sans boucle. Un compilateur, basé sur ce principe, est présenté. Tout d'abord, le langage choisi est précisé : il s'agit d'un sous-ensemble du C, auquel ont été ajoutés une déclaration de structures algébriques et un délimiteur de bloc parallèle. Puis l'organisation du compilateur est détaillée : une compilation consiste en une phase usuelle d'analyse et une séparation des parties séquentielles et parallèles. Une première compilation, suivie de l'exécution du programme compilé, permet de construire le circuit associé au programme initial. Enfin, une évaluation symbolique de ce circuit génère le code final pour le programme parallèle.

Les produits de matrices limitant la capacité du compilateur, à cause de la place mémoire requise pour les matrices et du temps de calcul de ces produits, nous avons donc étudié les algorithmes rapides. Nous avons conclu – l'algorithme de Strassen mis à part – qu'ils ne sont pas applicables en pratique ; une implantation de ces matrices creuses à l'aide de listes d'adjacence a permis en revanche de repousser les limites du compilateur.

Enfin, on peut reprocher à ce compilateur son manque de souplesse dû à l'obligation de fixer à la compilation toutes les tailles de données. Cependant, les principes qui prévalent dans un tel compilateur fournissent des bornes sur la complexité parallèle de programmes séquentiels et ces connaissances peuvent être intégrées dans les paralléliseurs existants. En

particulier, il existe des schémas dont la parallélisation est bien maîtrisée, les réductions (ou préfixes). Identifier de tels schémas dans un programme est une tâche assez ardue. Or calculer le degré et le nombre d'alternances d'un circuit sous forme d'équations de récurrence et les estimer fournit des indications sur la présence – ou l'absence – de réductions dissimulées dans un programme, y compris des réductions non triviales.

Conclusion

Les études en complexité parallèle et notamment sur la classe NC se sont révélées et se révèlent encore d'une grande importance pratique, et ce pour plusieurs raisons : la première est que cette classe comprend les problèmes dont la parallélisation sur machine réelle admet une accélération significative ; ensuite, même si les temps polylogarithmiques ne peuvent pas être atteints sur des machines concrètes pour certains problèmes, pour des raisons de contention liées aux communications, leur appartenance à NC implique néanmoins l'existence d'un vaste spectre de décompositions possibles en sous-tâches indépendantes ; enfin, de nombreux algorithmes parallèles dont l'intérêt pratique n'est plus à démontrer, comme le tri et de nombreux problèmes de mathématiques discrètes, d'arithmétique ou d'algèbre linéaire, appartiennent à NC.

Dans ce contexte, les techniques de contraction des expressions et des circuits arithmétiques peuvent être vues comme des extracteurs du parallélisme intrinsèque contenu dans les programmes séquentiels, parallélisme qui dépasse celui qui peut être lu sur le graphe de précédence et qui tient à la sémantique des opérateurs utilisés. La connaissance des propriétés algébriques, comme l'associativité ou la distributivité, permet une réorganisation des calculs qui n'affecte pas les résultats. Plus la structure algébrique utilisée sera riche en propriétés, plus il sera possible d'en tirer parti pour améliorer les algorithmes d'évaluation. Généralisant les algorithmes conçus pour les semi-anneaux, nous proposons un algorithme qui améliore les majorations précédemment connues pour la contraction de circuits arithmétiques dans un treillis. En prolongement de ce travail, il paraît intéressant d'élargir la palette des structures algébriques utilisées afin de profiter des propriétés additionnelles qui les caractérisent et par exemple d'évaluer la classe des circuits booléens en prenant en compte la loi d'absorption et le principe du tiers exclu, la classe des circuits arithmétiques booléens, où une structure arithmétique est couplée à l'algèbre de Boole, ou bien la classe des RAM, qui comporte quelques instructions arithmétiques ainsi que des instructions booléennes.

Ces algorithmes consistent en une réorganisation profonde des calculs. Cette réorganisation est explicitement effectuée grâce à un prototype de compilateur, qui permet d'une part de générer le code associé et d'autre part d'obtenir des estimations de la complexité parallèle du problème traité. Une extension possible de ce prototype, où le compilateur effectue une parallélisation explicite à gros grain (opérations matricielles pour l'évaluation d'un polynôme en une matrice par exemple), est le couplage avec Athapascan, langage parallèle développé dans le cadre du projet IMAG APACHE, qui à partir d'un programme à parallélisme explicite, permet une exécution parallèle sur une machine donnée, le placement étant réalisé de manière implicite (statique et dynamique) grâce à des techniques de régulation de charge.

Par ailleurs, la voie de l'expertise en parallélisation des nids de boucles, pour guider la

recherche de réductions cachées dans ces nids, est prometteuse, parce qu'elle est peu coûteuse à mettre en œuvre et fournit des informations de qualité. En cela aussi, les recherches en algorithmique parallèle théorique rejoignent les préoccupations de la parallélisation effective.

Finalement, les techniques d'anticipation permettent d'obtenir une bonne approximation de la complexité parallèle accessible à partir d'un programme donné, qu'il soit séquentiel ou déjà parallèle ; par là-même, elles guident le programmeur dans le vaste domaine de l'algorithmique parallèle pour l'étude algorithmique préliminaire à la phase de programmation pratique sur une machine donnée.

Bibliographie

[ADKP89] K. Abrahamson, N. Dadoun, D.G. Kirkpatrick, and T. Przytycka. A simple parallel tree contraction algorithm. *Journal of Algorithms*, 10(2):287–302, June 1989.

[AF87] J.M. Arnaudiès and H. Fraysse. *Cours de mathématiques - 1 - Algèbre*. Dunod Université, 1987.

[AU72] A.V. Aho and J. D. Ullman. *The Theory of Parsing, Translation, and Compiling*, volume I : Parsing of *Series in Automatic Computation*, chapter 4 : General parsing methods, section 4.2 : Tabular parsing methods, pages 314–332. Prentice-Hall, 1972.

[Bau92] F. Baude. Tree contraction on distributed-memory parallel architectures. Communication personnelle, November 1992.

[BCGR92] S.R. Buss, S.A. Cook, A. Gupta, and V. Ramachandran. An optimal parallel algorithm for formula evaluation. *SIAM Journal on Computing*, 21(4):755–780, August 1992.

[BCH86] P. W. Beame, S. A. Cook, and H. J. Hoover. Log depth circuits for divisions and related problems. *SIAM Journal on Computing*, 15(4):994–1003, November 1986.

[BCRL79] D. Bini, M. Capovani, F. Romani, and G. Lotti. $\mathcal{O}(n^{2,7799})$ complexity for $n \times n$ approximate matrix multiplication. *Information Processing Letters*, 8(5):234–235, June 1979.

[BDG90] J.L. Balcázar, J. Díaz, and J. Gabarró. *Structural Complexity II*. Springer Verlag, 1990.

[Ber73] C. Berge. *Graphes et Hypergraphes*. Gauthier-Villars, 1973.

[BFP+91] P. Bouvry, H. Frydlender, J. Prévost, J.-L. Roch, A. Touzène, and G. Villard. Manuel du méganode. Technical Report 63, LMC-IMAG, 46, avenue Félix Viallet,
38031 Grenoble cedex, France, April 1991.

[BK82] R.P. Brent and H.T. Kung. A regular layout for parallel adders. *IEEE Transactions on Computers*, C31(3):260–264, 1982.

[BK83] R.P. Brent and H.T. Kung. Systolic VLSI arrays for linear-time gcd computations. In *Proc. VLSI '83*, pages 145–154, 1983.

[BOV85] I. Bar-On and U. Vishkin. Optimal parallel generation of a computation tree-form. *ACM Transactions on Programming Languages and Systems*, 7(2):348–357, April 1985.

[Bre73] R. P. Brent. The parallel evaluation of arithmetic expressions in logarithmic time. In J.F. Traub, editor, *Complexity of Sequential and Parallel Numerical Algorithms*, pages 83–102. Academic Press, New York, 1973.

[Bre74] R. P. Brent. The parallel evaluation of general arithmetic expressions. *J.ACM*, 21(2):201–206, April 1974.

[BS83] W. Baur and V. Strassen. The complexity of partial derivatives. *Theoretical Computer Science*, 22:317–330, 1983.

[Bus87] S.R. Buss. The boolean formula value problem is in ALOGTIME. In *Proc. 19th Annual ACM Symposium on Theory of Computing*, pages 123–131, 1987.

[Bus93] S. R. Buss. Algorithm for boolean formula evaluation and for tree contraction. In P. Clote and J. Krajícek, editors, *Arithmetic, proof theory, and computational complexity*, pages 96–115. Clarendon Press - Oxford, 1993.

[Cal91] D. Callahan. Recognizing and parallelizing bounded recurrences. In U. Banerjee et al., editor, *Proc. of the 4th Int. Workshop on Languages and Compilers for Parallel Computing*, volume 589 of *LNCS*, pages 266–282. Springer Verlag, 1991.

[Car66] M. Carvallo. *Monographie des treillis et algèbre de Boole*. Gauthier-Villars, 1966.

[CG79] R. Cuninghame-Green. *Minimax Algebra*. Springer-Verlag, 1979.

[Chi85] A.L. Chistov. Fast parallel calculation of the rank of matrices over a field of arbitrary characteristic. In LNCS, editor, *Fundamentals of Computation Theory*, number 199, pages 63–69, 1985.

[Chr92] P. Chrétienne. Ordonnancement et parallélisme. In M. Cosnard et al., editor, *Algorithmique parallèle*, pages 297–312. Masson, collection ERI, 992.

[CMQV85] G. Cohen, P. Moller, J.P. Quadrat, and M. Viot. Une théorie linéaire des systèmes à événements discrets. Technical report, INRIA Rocquencourt, 1985.

[Col88] R. Cole. Parallel merge sort. *SIAM Journal on Computing*, 17(4):770–785, 1988.

[Coo81] S.A. Cook. Towards a complexity theory of synchronous parallel computations. *Enseignement Mathématique*, 27:99–124, 1981.

[Coo85] S. A. Cook. A taxonomy of problems with fast parallel algorithms. *Information and Control*, 64:2–22, 1985.

[CP90] R. Cypher and C. G. Plaxton. Deterministic sorting in nearly logarithmic time on the hypercube and related computers. In *22nd Annual ACM Symposium on Theory of Computing*, pages 193–203, 1990.

[Csa76] L. Csanky. Fast parallel matrix inversion algorithm. *SIAM Journal on Computing*, 5(4):618–623, December 1976.

[CT93] M. Cosnard and D. Trystram. *Algorithmes et architectures parallèles*. InterEditions, 1993.

[CV88] R. Cole and U. Vishkin. Optimal parallel algorithms for expression tree evaluation and list ranking. In *Proceedings Aegean Workshop on Computing*, number 319, pages 91–110. Lecture Notes in Computer Science, 1988.

[CW82] D. Coppersmith and S. Winograd. On the asymptotic complexity of matrix multiplication. *SIAM Journal of Computing*, 11(3):472–492, August 1982.

[CW90] D. Coppersmith and S. Winograd. Matrix multiplication via arithmetic progression (full paper). *Journal of Symbolic Computation*, 9(3):251–280, March 1990.

[DR92] A. Darte and Y. Robert. Séquencement des nids de boucle. In M. Cosnard et al., editor, *Algorithmique parallèle*, pages 344–368. Masson, collection ERI, 1992.

[Fea92] P. Feautrier. Techniques de parallélisation. In M. Cosnard et al., editor, *Algorithmique parallèle*, pages 244–257. Masson, collection ERI, 1992.

[Gas71] N. Gastinel. Sur le calcul des produits de matrices. *Numerische Mathematik*, Band 17(Heft 13):222–229, 1971.

[Gat86] J. von zur Gathen. Parallel arithmetic computations : a survey. In Springer Lecture Notes in Computer Science, editor, *Proc. 12^{th} Int. Symp. Math. Foundations of Computer Science, Bratislava*, number 233, 1986.

[Gat88] J. von zur Gathen. Algebraic complexity theory. *Ann. Rev. Comput. Sci.*, 3:317–347, 1988.

[Gau92] T. Gautier. Contraction d'expressions algébriques : un extracteur automatique de parallélisme. Master's thesis, DEA de Mathématiques Appliquées de l'Université Joseph Fourier, Rapport de fin d'études ENSIMAG, 1992.

[GM79] M. Gondran and M. Minoux. *Graphes et algorithmes*, volume 37 of *Collection de la Direction des Etudes et Recherches d'Electricité de France*, chapter Chapitre 9 : Matroïdes, pages 307–341. Eyrolles, 1979.

[Gol77] L. M. Goldschlager. The monotone and planar circuit value problems are log space complete for P. *SIGACT News*, 9(2):25–29, 1977.

[GR86] A. Gibbons and W. Rytter. An optimal parallel algorithm for dynamic tree evaluation and its applications. In *Symposium on Foundations of Software Technology and Theoretical Computer Science*, pages 453–469, 1986.

[GR88a] A. Gibbons and W. Rytter. *Efficient parallel algorithms*. Cambridge University Press, 1988.

[GR88b] A. Gibbons and W. Rytter. Optimal parallel algorithms for dynamic expression evaluation and context-free recognition. *Lecture Notes in Computer Science, VLSI Algorithms and Architecture*, (319):32–45, 1988.

[GS89a] J. von zur Gathen and G. Seroussi. Boolean versus arithmetic circuits. In *(preliminary version in the Proc. 6^{th} Int. Conf. Computer Science, Santiago, Chile, 1986)*, pages 171–184, 1989.

[GS89b] A. M. Gibbons and Y.N. Srikant. A class of problems efficiently solvable on mesh-connected computers including dynamic expression evaluation. *Information Processing Letters*, 32(6):305–311, October 1989.

[GY92] A. Gerasoulis and T. Yang. A comparison of clustering heuristics for scheduling dags on multiprocessors. *Journal of Parallel and Distributed Computing*, 1992.

[HKP84] H.G. Hoover, M.M. Klawe, and N.J. Pippenger. Bounding fan-out in logical networks. *Journal of the ACM*, 31(1):13–18, January 1984.

[JD89] P. Jouvelot and B. Dehbonei. A unified semantic approach for the vectorization and parallelization of generalized reductions. In *Proc. of the 3rd Int. Conf. on Supercomputing*, pages 186–194. ACM Press, 1989.

[Kal88] E. Kaltofen. Greatest common divisors of polynomials given by straight-line programs. *J.ACM*, 35(1):231–264, January 1988.

[Kal89] E. Kaltofen. Parallel algebraic algorithm design, 1989. Lecture Notes for a Tutorial, ISSAC'89.

[KD88] S. R. Kosaraju and A. L. Delcher. Optimal parallel evaluation of tree-structured computations by raking (extended abstract). *Lecture Notes in Computer Science, VLSI Algorithms and Architecture*, (319):103–110, 1988.

[KMR84] R. Kannan, G.L. Miller, and L. Rudolph. Sublinear parallel algorithms for computing tthe greatest common divisor of two integers. In *Proc. 25^{th} Annual IEEE Symposium on Foundations of Computer Science*, pages 7–11, 1984.

[KR90] R. M. Karp and V. Ramachandran. *Handbook of Theoretical Computer Science, J. van Leeuwen, editor*, chapter Parallel Algorithms for Shared-Memory Machines, pages 869–941. Elsevier Science Publishers, B.V., 1990.

[KT90] E. Kaltofen and B. Trager. Computing with polynomials given by black boxes for their evaluation: Greatest common divisors, factorization, separation of numerators and denominators. *Journal of Symbolic Computations*, 9(3):301–320, 1990.

[Lad75] R.E. Ladner. The circuit value problem is log space complete for p. *SIGACT News*, 7(1):18–20, January 1975.

[Lad76] J. D. Laderman. A noncommutative algorithm for multiplying 3×3 matrices using 23 multiplications. *Bulletin of the American Mathematical Society*, 82(1):126–128, January 1976.

[Lam74] L. Lamport. The parallel execution of DO loops. *Communications of the ACM*, 17(2):83–93, February 1974.

[LF80] R.E. Ladner and M.J. Fisher. Parallel prefix computation. *J. ACM*, 27:831–838, 1980.

[MR85] G. L. Miller and J. H. Reif. Parallel tree contraction and its application. In *IEEE, 26^{th} IEEE Symposium on Foundations of Computer Science*, pages 478–489, 1985.

[MRK86] G. L. Miller, V. Ramachandran, and E. Kaltofen. Efficient parallel evaluation of straight-line code and arithmetic circuits. In Lecture Notes in Computer Science, editor, *Proceedings Aegean Workshop on Computing*, number 227, July 1986.

[MT87] G. L. Miller and S.-H. Teng. Dynamic parallel complexity of computational circuits. *J.ACM*, pages 254–263, 1987.

[Pan80] V. Pan. New fast algorithms for matrix operations. *SIAM Journal of Computing*, 9(2):321–342, May 1980.

[Pan84a] V. Pan. How can we speed up matrix multiplication? *SIAM Review*, 26(3):393–415, July 1984.

[Pan84b] V. Pan. *How to multiply matrices faster.* Number 179. Lecture Notes in Computer Science, 1984.

[Pap94] C. H. Papadimitriou. *Computational Complexity.* Addison-Wesley, 1994.

[Pol88] C.D. Polychronopoulos. Compiler optimizations for enhancing parallelism and their impact on architecture design. *IEEE Transactions on Computers*, 37(8):991–1004, 1988.

[PP91] S. Pinter and R.Y. Pinter. Program optimizations and parallelizations using idioms. In *POPL'91*, 1991.

[PRRV91] B. Plateau, A. Rasse, J.L. Roch, and J.P. Verjus. *Parallélisme.* Polycopiés ENSIMAG, 1991.

[Red94] X. Redon. Détection et exploitation des récurrences dans les programmes scientifiques. In P. Fraigniaud L. Bougé, M. Cosnard, editor, *RenPar'6*, pages 81–85, June 1994.

[Rei86] J.H. Reif. Logarithmic depth functions for algebraic functions. *SIAM Journal on Computing*, 15:231–242, 1986.

[RF94] X. Redon and P. Feautrier. Detection of simple recurrences using a graph reduction algorithms. Communication personnelle, January 1994.

[RJ90] K. W. Ryu and J. JàJà. Efficient algorithms for list ranking and for solving graph problems on the hypercube. *IEEE Transactions on Parallel and Distributed Systems*, 1(1):83–90, January 1990.

[Roc94] J.-L. Roch. *Algorithmes parallèles - analyse et conception*, chapter Complexité parallèle et algorithmique PRAM, pages 105–128. Hermès, 1994.

[RT92] J.-L. Roch and D. Trystram. Parallel Winograd matrix multiplication. In W. Joosen and E. Milgrom, editors, *Parallel Computing : from Theory to Sound Practice*, pages 578–581. IOS Press, September 1992. Grenoble.

[Ruz81] W.L. Ruzzo. On uniform circuit complexity. *Journal of Computer and System Sciences*, 22(3):365–383, June 1981.

[Sch81] A. Schönhage. Partial and total matrix multiplication. *SIAM Journal of Computing*, 10(3):434–455, August 1981.

[Str69] V. Strassen. Gaussian elimination is not optimal. *Numerische Mathematik*, Band 13(Heft 4):354–356, 1969.

[Str73] V. Strassen. Vermeidung von Divisionen. *Journal für Mathematik*, Band 264:184–202, 1973.

[Str90] V. Strassen. *Handbook of Theoretical Computer Science, J. van Leeuwen, editor*, chapter Algebraic Complexity Theory, pages 633–672. Elsevier Science Publishers, B.V., 1990.

[TV85] R.E. Tarjan and U. Vishkin. An efficient parallel biconnectivity algorithm. *SIAM Journal on Computing*, 14(4):862–874, November 1985.

[Val90] L.G. Valiant. General purpose parallel architectures. In J. van Leuwen, editor, *Handbook of Theoretical Computer Science (vol. A : Algorithms and Complexity)*, chapter 18, pages 943–971. Elsevier Science Publishers, B.V., 1990.

[Win73] S. Winograd. Some remarks on fast multiplication of polynomials. In J.F. Traub, editor, *Complexity of sequential and parallel numerical algorithms*, pages 181–916. Academic Press, 1973.

Résumé

Les algorithmes d'évaluation parallèle des expressions et des circuits arithmétiques peuvent être vus comme des extracteurs du parallélisme intrinsèque contenu dans les programmes séquentiels, parallélisme qui dépasse celui qui peut être lu sur le graphe de précédence et qui tient à la sémantique des opérateurs utilisés. La connaissance des propriétés algébriques, comme l'associativité ou la distributivité, permet une réorganisation des calculs qui n'affecte pas les résultats. Plus la structure algébrique utilisée sera riche en propriétés, plus il sera possible d'en tirer parti pour améliorer les algorithmes d'évaluation. Généralisant les algorithmes conçus pour les semi-anneaux, nous proposons un algorithme qui améliore les majorations précédemment connues pour la contraction de circuits arithmétiques dans un treillis.

Des simulations de cet algorithme ont permis de mettre en évidence ses qualités de « prédicteur automatique de complexité ». Réorganiser explicitement les calculs à l'aide de ces algorithmes, c'est-à-dire réaliser un compilateur complet, permet de comparer la réalité des algorithmes parallèles sur machines à mémoire distribuée et la puissance des algorithmes théoriques. Un prototype a été réalisé, basé sur une simplification/extension du langage C. Enfin, l'intérêt de ces techniques dans le domaine de la parallélisation des nids de boucles, pour guider la recherche de réductions cachées dans ces nids, semble prometteuse, parce qu'elle est peu coûteuse à mettre en œuvre et fournit des informations de qualité. En cela, les recherches en algorithmique parallèle théorique rejoignent les préoccupations de la parallélisation effective.

Abstract

Algorithms for the parallel evaluation of expressions and arithmetic circuits may be considered as extractors of the intrinsic parallelism contained in sequential programs; far beyond the parallelism that can be read from the dependence graph, this parallelism comes from the meaning of the operators that are employed. The knowledge of their algebraic properties, such as associativity or distributivity, allows the reorganization of the computations without affecting the results. The more the algebraic structure used in the program possesses such properties, the more they can be taken into account to speed up the parallel evaluation of the program. We generalize the algorithms designed for programs over semi-rings in order to propose an algorithm, the complexity of which improves previously known upper bounds for the evaluation of arithmetic circuits over lattices.

Simulations of this algorithm highlight its power as an "automatic complexity predictor". Furthermore, the explicit reorganization of the computations by means of these evaluation algorithms, by means of a complete compilation, helps to compare real algorithms for distributed memory machines with theoretical parallel ones. A prototype of the compiler has been developed, based on a simplification/extension of the C language. Then, the use of these techniques in the area of automatic parallelization of nested loops is discussed: they can help the detection of hidden reductions in these nested loops in an easy and efficient way, by providing relevant information on the probability of the existence of reductions. This last point proves that the design of theoretical parallel algorithms is related to the search for effective parallelization.