Contributions à la recherche dans des ensembles ordonnés : du séquentiel au parallèle

Afonso Galvao Ferreira

THESE

Présentée par Afonso GALVAO FERREIRA

Pour l'obtention du titre de **DOCTEUR**

DE L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

(Arrêté ministériel du 23 novembre 1988)

Spécialité : **INFORMATIQUE**

CONTRIBUTIONS A LA RECHERCHE DANS DES ENSEMBLES ORDONNES : DU SEQUENTIEL AU PARALLELE

présentée le 17 janvier 1990

Jury : **M. NIVAT** (président)
S. AKL (rapporteur)
R. CORI (rapporteur)
M. COSNARD
F. DEHNE
B. PLATEAU

préparée au sein des laboratoire LIP - IMAG et TIM3 - IMAG

INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

46 avenue Felix Viallet
38031 GRENOBLE cedex

Tél. : 76.57.45.00

Année universitaire 1989

Président de l'Institut :
Monsieur Georges LESPINARD

Professeurs des Universités

BARIBAUD Michel	ENSERG	JAUSSAUD Pierre	ENSIEG
BARRAUD Alain	ENSIEG	JOST Rémy	ENSPG
BAUDELET Bernard	ENSPG	JOUBERT Jean-Claude	ENSPG
BEAUFILS Jean-Pierre	INPG	JOURDAIN Geneviève	ENSIEG
BLIMAN Samuel	ENSERG	LACOUME Jean-Louis	ENSIEG
BOIS Philippe	ENSHMG	LADET Pierre	ENSIEG
BONNETAIN Lucien	ENSEEG	LESIEUR Marcel	ENSHMG
BONNET Guy	ENSPG	LESPINARD Georges	ENSHMG
BRISSONNEAU Pierre	ENSIEG	LONGEQUEUE Jean-Pierre	ENSPG
BRUNET Yves	IUFA	LORET Benjamin	ENSHMG
CAILLERIE Denis	ENSHMG	LOUCHET François	ENSEEG
CAVAIGNAC Jean-François	ENSPG	LUCAZEAU Guy	ENSEEG
CHARTIER Germain	ENSPG	MASSE Philippe	ENSIEG
CHENEVIER Pierre	ENSERG	MASSELOT Christian	ENSIEG
CHERADAME Hervé	UFR PGP	MAZARE Guy	ENSIMAG
CHERUY Arlette	ENSIEG	MOHR Roger	ENSIMAG
CHOVET Alain	ENSERG	MOREAU René	ENSHMG
COHEN Joseph	ENSERG	MORET Roger	ENSIEG
COLINET Catherine	ENSEEG	MOSSIERE Jacques	ENSIMAG
CORNUT Bruno	ENSIEG	OBLED Charles	ENSHMG
COULOMB Jean-Louis	ENSIEG	OZIL Patrick	ENSEEG
COUMES André	ENSERG	PA ULEAU Yves	ENSEEG
CROWLEY James	ENSIMAG	PERRET Robert	ENSIEG
DARVE Félix	ENSHMG	PIAU Jean-Michel	ENSHMG
DELLA-DORA Jean	ENSIMAG	PIC Etienne	ENSERG
DEPEY Maurice	ENSERG	PLATEAU Brigitte	ENSIMAG
DEPORTES Jacques	ENSPG	POUPOT Christian	ENSERG
DEROO Daniel	ENSEEG	RAMEAU Jean-Jacques	ENSEEG
DESRE Pierre	ENSEEG	REINISCH Raymond	ENSPG
DOLMAZON Jean-Marc	ENSERG	RENAUD Maurice	UFR PGP
DURAND Francis	ENSEEG	ROBERT André	UFR PGP
DURAND Jean-Louis	ENSPG	ROBERT François	ENSIMAG
FAUTRELLE Yves	ENSHMG	SABONNADIERE Jean-Claude	ENSIEG
FOGGIA Albert	ENSIEG	SAUCIER Gabrièle	ENSIMAG
FONLUPTJean	ENSIMAG	SCHLENKER Claire	ENSPG
FOULARD Claude	ENSIEG	SCHLENKER Michel	ENSPG
GANDINI Alessandro	UFR PGP	SERMET Pierre	ENSERG
GAUBERT Claude	ENSPG	SILVY Jacques	UFR PGP
GENTIL Pierre	ENSERG	SIRIEYS Pierre	ENSHMG
GENTIL Sylviane	ENSIEG	SOHM Jean-Claude	ENSEEG
GREVEN Hélène	IUFA	SOLER Jean-Louis	ENSIMAG
GUEGUEN Claude	ENSIEG	SOUQUET Jean-Louis	ENSEEG
GUERIN Bernard	ENSERG	TROMPETTE Philippe	ENSHMG
GUYOT Pierre	ENSEEG	VINCENT Henri	ENSPG
IVANES Marcel	ENSIEG	ZADWORNY François	ENSERG

Personnes ayant obtenu le diplôme d'HABILITATION A DIRIGER DES RECHERCHES

BECKER Monique
BINDER Zdenek
CHASSERY Jean-Marc
CHOLLET Jean-Pierre
COEY John
COLINET Catherine
COMMAULT Christian
CORNUEJOLS Gérard
COULOMB Jean- Louis
COURNIL M.
DALARD Francis
DANES Florin
DEROO Daniel
DIARD Jean-Paul
DION Jean-Michel
DUGARD Luc
DURAND Madeleine
DURAND Robert
GALERIE Alain
GAUTHIER Jean-Paul
GENTIL Sylviane
GHIBAUDO Gérard
HAMAR Sylvaine
HAMAR Roger
LACHENAL D.
LADET Pierre
LATOMBE Claudine
LE HUY H.
LE GORREC Bernard
MADAR Roland
MEUNIER G.
MULLER Jean
NGUYEN TRONG Bernadette
NIEZ J.J.
PASTUREL Alain
PLA Fernand
ROGNON J.P.
ROUGER Jean
TCHUENTE Maurice
VINCENT Henri
YAVARI A.R.

Chercheurs du C.N.R.S

DIRECTEURS DE RECHERCHE CLASSE 0

LANDEAU	Ioan
NAYROLLES	Bernard

Directeurs de recherche 1ère Classe

ANSARA Ibrahim
CARRE René
FRUCHART Robert
HOPFINGER Emile
JORRAND Philippe
KRAKOWIAK Sacha
LEPROVOST Christian
VACHAUD Georges
VERJUS Jean-Pierre

Directeurs de recherche 2ème Classe

ALEMANY Antoine
ALLIBERT Colette
ALLIBERT Michel
ARMAND Michel
AUDIER Marc
BERNARD Claude
BINDER Gilbert
BONNET Roland
BORNARD Guy
CAILLET Marcel
CALMET Jacques
CHATILLON Chritiant
CLERMONT Jean-Robert
COURTOIS Bernard
DAVID René
DION Jean-Michel
DRIOLE Jean
DURAND Robert
ESCUDIER Pierre
EUSTATHOPOULOS Nicolas
GARNIER Marcel
GUELIN Pierre
JOUD Jean-Charles
KAMARINOS Georges
KLEITZ Michel
KOFMAN Walter
LEJEUNE Gérard
MADAR Roland
MERMET Jean
MICHEL Jean-Marie
MEUNIER Jacques
PEUZIN Jean-Claude
PIAU Monique
RENOUARD Dominique
SENATEUR Jean-Pierre
SIFAKIS Joseph
SIMON Jean-Paul
SUERY Michel
TEODOSIU Christian
VAUCLIN Michel
VENNEREAU Pierre
WACK Bernard
YONNET Jean-Paul

Personnalités agrées à titre permanent à diriger des travaux de recherche
(décision du conseil scientifique)

E.N.S.E.E.G
HAMMOU Abdelkader
MARTIN-GARIN Régina
SARRAZIN Pierre
SIMON Jean-Paul

E.N.S.E.R.G
BOREL Joseph

E.N.S.I.E.G
DESCHIZEAUX Pierre
GLANGEAUD François
PERARD Jacques
REINISCH Raymond

E.N.S.H.M.G
ROWE Alain

E.N.S.I.M.A.G
COURTIN Jacques

C.E.N.G
CADET Jean
COEURE Philippe
DELHAYE Jean-Marc
DUPUY Michel
JOUVE Hubert
NICOLAU Yvan
NIFENECKER Hervé
PERROUD Paul
PEUZIN Jean-Claude
TAIEB Maurice
VINCENDON Marc

Laboratoires extérieurs :

C.N.E.T
DEVINE Rodericq
GERBER Roland
MERCKEL Gérard
PAULEAU Yves

Situation particulière

PROFESSEURS D'UNIVERSITE

DETACHEMENT

ENSIMAG	LATOMBE	J..Claude	Détachement	21/10/1989
ENSHMG	PIERRARD	J.Marie	Détachement	30/04/1989
ENSIMAG	VEILLON	Gérard	Détachement	30/09/1990
ENSIMAG	VERJUS	J.Pierre	Détachement	30/09/1989
ENSPG	BLOCH	Daniel	Recteur à c/	21/12/1988

SURNOMBRE

INPG	CHIAVERINA	Jean	30/09/1989
ENSHMG	BOUVARD	Maurice	30/09/1991
ENSEEG	PARIAUD	J.Charles	30/09/1991

A ma maman,
sans qui cette tâche
aurait été presque
insurmontable.
C'est bête mais c'est vrai.

Ticking away the moments that make up a dull day
You fritter and waste the hours in an off hand way
Kicking around on a piece of ground in your home town
Waiting for someone or something to show you the way

Tired of lying in the sunshine staying home to watch the rain
You are young and life is long and there is time to kill today
And then one day you find ten years have got behind you
No one told you when to run, you missed the starting gun

And you run and you run to catch up with the sun, but it's sinking
And racing around to come up behind you again
The sun is the same in the relative way, but you're older
Shorter of breath and one day closer to death

Every year is getting shorter, never seem to find the time
Plans that either come to naught or half a page of scribbled lines
Hanging on in quiet desperation is the English way
The time is gone the song is over, thought I'd something more to say.

(Pink Floyd)

Mas é preciso ter raça, é preciso ter força, é preciso ter gana, sempre.

(Milton Nascimento)

Oui, je crois que c'est par là...

(Un passant)

Je voudrais exprimer ici toute ma reconnaissance aux membres du jury :

Maurice Nivat, pour l'honneur qu'il me fait de présider ce jury,

Selim Akl, pour s'intéresser à cette thèse et d'avoir accepté d'en être rapporteur, même depuis les bords gelés de son lac à Kingston, Canada,

Robert Cori, pour avoir accepté d'être rapporteur de ce travail malgré les contraintes temporelles,

Michel Cosnard, pour m'avoir accepté en tant que thésard, tout en sachant me motiver sans m'étouffer, avec la confiance, le bonheur et la capacité intelectuelle qui lui sont caractéristiques. De par son intérêt et sa dédication, cette thèse, mon bébé, l'aurait volontiers appelé tonton. Toute mon admiration et respect lui sont dédiés,

Frank Dehne, pour m'avoir invité à Carleton University, où j'ai pu trouver la joie de travailler avec lui,

Brigitte Plateau, pour avoir accepté de faire partie de ce jury, en lui apportant à la fois de la compétence et du charme.

Pour quelqu'un qui abandonne tout dans son pays d'origine en vue de l'obtention d'un doctorat, y arriver n'est pas du tout évident. Loin de la famille, des amis et de la chaleur caractéristique du peuple et de la terre natals, j'aurais envie de remercier ici tous ceux qui ont contribué à rendre possible cette aventure et la réalisation émotionnelle, intéllectuelle, sentimentale et matérielle de cette thèse. Malheureusement ni l'espace ni la mémoire ne le permettraient. Ainsi, ceux qui ne trouveront pas leur nom dans ces remerciements "à la brésilienne", sachez que je vous aime autant.

Je tiens donc à exprimer toute ma gratitude à *Bernard*, dont l'appui plus que fondamental m'a empeché de couler quand je sentais déjà l'eau qui me baignait les oreilles. A *Felipe*, *Bia*, *Brito* e principalmente à pequena *Helena*, que seguraram uma barra pesadissima com a força que lhes é caracteristica, meu muito obrigado. De même à ma *mamy*, mon *papa* et *Nivai* qui ont su assurer quand il l'a fallu.

Mes remerciements vont à *Jean D.*, *Janez* et *Andrew* qui m'ont donné le plaisir de partager leurs bonnes idées au travail. Aussi à *JDD* qui a bien voulu m'accueillir au sein de l'équipe Algorithmique Parallèle et Calcul Formel du Laboratoire TIM3 et nous offrir des gâteaux de temps en temps, à *Yves* qui a bien voulu diriger mes recherches et à IBM-Rome qui l'en a empêché... (joking !!!). Egalement à *Jean-Michel* pour son attention ; et spécialement à *Jean-Laurent* (coman sasse crit sa, Jeanlô ??) pour sa disponibilité et gentillesse, à *Michel Gastaldo*, mon terne sbire, sans qui je serais certainement en tôle et à la toujours efficace *Valérie* qui a supporté (supporte, plutôt) avec flegme mes allers et retours.

Merci *Gilles*, *Pascale*, *Françoise*, *Jean-Louis* et tous les copains, copines et personnel des 25 labos auxquels j'ai appartenu, pour les coups de main toujours bienvenus.

Ma gratitude aussi à *Denis* pour la musique et Antoine, à *Antoine* pour le saxo et Isabelle, à *Isabelle* pour les fêtes, le tennis, les gâteaux, mes dents et *Pascale*. A celle-ci pour la force morale, qui a failli m'abandonner près de l'arrivée.

A *Jean-Yves*, qui n'a pas faibli, même après la 200-ième balle sur le filet ; à *Béné* - avec tendresse - et *Isa*, pour le rock et la joie ; merci de votre patience et compagnie. Mes remerciements aux troupes brésiliennes, *Ligia* et *Claudio*, *Lilian* et *Vladimir*, et avant eux, *Renata*, *Antonio* et *Rosana*, et *Eytan* qui ont, tous, aidé à diminuer la distance qui nous sépare du Brésil. Un merci spécial à la gentillesse infinie des *Witkowski* et des *François*.

Je remercie le Conseil du Dept. d'Informatique de l'Institut des Mathématiques et Statistiques de l'Université de Sao Paulo, Brésil, et la CAPES/COFECUB (n° 503/86-9) pour leur support financier et logistique. Je remercie également le personnel du Service de la Scolarité de l'INPG, de la Médiathèque et de la Reprographie pour la qualité de leur travail.

SOMMAIRE

INTRODUCTION

La recherche d'un élément dans un ensemble donné est un des problèmes fondamentaux de la vie quotidienne. Qui n'a jamais cherché un mot dans un dictionnaire, ou une recette dans un livre de cuisine ou même la meilleure façon de remplir son sac pour cette petite semaine au soleil ??

Le problème de la recherche se pose aussi en informatique non-numérique, étant à la base de beaucoup d'applications comme, par exemple, les bases de données, la compilation, l'optimisation combinatoire, etc ... Dans cette thèse nous abordons le problème de la recherche dans des ensembles ordonnés et quelques problèmes qui lui sont liés.

Etant données des entiers $A = \{a_1, \ldots, a_n\}$ et z, la version de base du problème de la recherche est celui de décider s'il existe k tel que $z = a_k \in A$. Pour résoudre ce problème, il suffit (et il est nécessaire) de parcourir toute la liste A. Par contre, si la liste A est triée, on peut utiliser la recherche dichotomique pour le résoudre en temps $O(\log n)$[1] et on ne peut pas faire mieux ([Knu72]).

Le jeu de mots n'étant pas voulu, dans les pages qui suivent sont décrites quelques trois années de *recherches* au sein des laboratoires TIM3 et LIP de l'IMAG et de PARADISE de la School of Computer Science de la

[1] Dorénavant, tous les logarithmes sont par rapport à la base 2 et arrondis supérieurement, sauf quand explicité contrairement.

Carleton University à Ottawa, Canada. Bien que tous nos résultats ne seront pas rapportés ici, nous pensons que ce travail résume bien la totalité de nos travaux menés sous la direction de Michel COSNARD, en présentant des algorithmes pour la résolution de quelques problèmes de recherche, sur des modèles de calcul séquentiel et parallèles, aussi bien que des bornes théoriques pour ces mêmes problèmes et modèles.

Le **Chapitre 1** de cette thèse porte sur des outils dont on aura besoin pour développer les résultats trouvés. Ces outils, connus sous le nom de *théorie de la complexité des algorithmes*, font déjà partie de la culture fondamentale de l'algorithmique, pouvant être trouvés dans de nombreux livres sur le sujet (e.g., [Aho74], [Akl85], [Kru85], [Akl89], ...).

Le lecteur qui connait les définitions et notations utilisées par une telle théorie pourra passer directement aux chapitres suivants, où nous allons démontrer des bornes inférieures, proposer des algorithmes séquentiels et étudier des solutions parallèles pour des problèmes de recherche, soit dans le domaine des ensembles partiellement ordonnés, soit dans l'étude des problèmes de décision ou d'optimisation combinatoire.

Les résultats que nous avons trouvés, et que nous décrivons par la suite, ne représentent pas la fin des discussions sur les sujets auxquels ils se rapportent. Au contraire, même si divers problèmes qui restaient ouverts ont été résolus, le nombre de nouvelles voies de recherches engendrées par ces mêmes réponses est assez important. Dans la **Conclusion** nous décrivons quelques problèmes ouverts, en espérant qu'ils trouveront des chercheurs intéressés et capables de les résoudre.

Matrices triées et X + Y

Les matrices définies par la somme cartésienne de deux vecteurs triés reçoivent le nom de matrices du type X + Y (triés). Ces matrices sont des sous-ensembles de l'ensemble des matrices triées, i.e., des matrices qui possèdent une relation d'ordre total sur chaque ligne et chaque colonne.

Pour les matrices triées (supposées n x n) les problèmes de la recherche ($\Theta(n)$), de la sélection du k-ième plus petit élément ($\Theta(\sqrt{k})$) et du tri de tous les éléments ($\Theta(n^2 log n)$) ont une complexité bien définie par l'existence d'algorithmes optimaux pour les résoudre. Ces mêmes algorithmes peuvent être utilisés pour les ensembles du type X + Y (triés) en substituant les références aux éléments (i,j) de la matrice d'entrée par des éléments (x_i+y_j), ce qui définit des bornes supérieures pour ces problèmes en X + Y (triés). Cependant, les bornes inférieures prouvées pour les matrices triées ne s'appliquent pas directement à X + Y (triés), puisque des nouvelles relations d'ordre y sont définies.

Ainsi, nous établissons la complexité du problème de la recherche ($\Theta(n)$) et du problème de la sélection du k-ième plus petit élément ($\Theta(\sqrt{k})$) dans X + Y (triés), à travers la démonstration de bornes inférieures pour ces problèmes. Ensuite nous faisons un tour d'horizon sur les résultats existants pour X + Y (triés) et pour des multi-ensembles, définis comme étant de la forme $\sum X_i$, la somme de vecteurs triés. Ayant pour but l'accélération de la recherche dans de tels ensembles, nous étudions le problème de la sélection du successeur d'un élément quelconque, donné, dans X + Y (triés), pour lequel nous démontrons que la complexité est $\Theta(n)$, dans le pire cas. En outre, pour des multi-ensembles, nous introduisons des algorithmes et

démontrons des résultats de complexité telle que, par exemple, l'appartenance à la classe NP-Complet du problème de la recherche dans $\sum X_i$.

Nous terminons ce **Chapitre 2** par une étude de la parallélisation de la recherche dans X + Y (triés) sur divers modèles de calcul parallèle. Nous proposons quatre algorithmes basés sur différentes stratégies, la démonstration de leurs optimalités étant dépendante du nombre de processeurs et des modèles utilisés.

Une application : le problème du Sac-à-Dos

Le problème de décision du Sac-à-Dos est celui où on veut répondre à la question sur l'existence d'un n-uplet binaire qui est solution de l'équation $\sum w_i c_i = M$, pour M et $W = (w_1, \ldots, w_n)$ des entiers positifs donnés.

Etant NP-Complet, avec un espace solution de taille 2^n, ce problème a connu d'importantes réductions en temps et en espace nécessaires pour le résoudre. Si, d'une part, le meilleur algorithme pour le Sac-à-Dos a une complexité en $O(n2^{n/2})$, d'une autre, la complexité spatiale correspondant à une telle complexité temporelle a été améliorée et portée d'abord à $O(2^{n/2})$ et puis à $O(2^{n/4})$.

Le compromis temps/espace en mode séquentiel est donc $O(n2^{3n/4})$. En mode parallèle, le meilleur algorithme utilise $O(2^{n/6})$ processeurs pour un temps $O(n2^{n/2})$. Le compromis plus couramment utilisé dans l'algorithmique parallèle est temps/processeurs, mais, pour un tel problème, ce type de compromis ne tient pas compte de la complexité de l'espace mémoire, qui est très importante. Par conséquent, un compromis

temps/matériel existe, où le matériel est défini comme le nombre de processeurs plus le nombre de cellules mémoire utilisés par un algorithme. Pour cet algorithme parallèle le compromis temps/matériel égale $O(n2^{2n/3})$, puisque l'algorithme requiert $O(2^{n/6})$ cellules mémoire.

A partir des idées développées dans le Chapitre 2, nous avons conçu plusieurs algorithmes parallèles pour le problème du Sac-à-Dos, sur différents modèles de parallélisme, qui présentent des améliorations importantes au niveau des compromis temps/processeurs et temps/matériel.

Pour les machines à mémoire partagée, nous proposons d'abord des algorithmes qui utilisent strictement moins de $O(n2^{n/2})$ matériel pour une complexité temporelle elle aussi strictement inférieure à $O(n2^{n/2})$. Ensuite nous introduisons un algorithme qui présente une accélération optimale, induisant des compromis temps/processeur et temps/matériel en $O(n2^{n/2})$.

Pour finir ce **Chapitre 3** nous montrons que les meilleures solutions existantes pour le problème du Sac-à-Dos peuvent être réduites au problème du tri. Par conséquent nous en déduisons des solutions à accélération optimale pour les principales architectures des machines à mémoire distribuée.

Optimisation Combinatoire

Les problèmes d'optimisation combinatoire consistent en la recherche d'un élément de valeur optimale dans un ensemble fini d'objets combinatoires. Malheureusement, pour la plupart des applications, cet ensemble est non énumérable à cause de sa cardinalité exponentielle et le

problème respectif appartient à la classe des problèmes NP-Durs, d'où l'utilisation de méthodes heuristiques pour les résoudre.

La méthode du Branch and Bound, dont les dérivations (e.g., *depth-first*, tabou, minmax, etc...) comptent parmi les techniques les plus utilisées dans la résolution de problèmes d'optimisation combinatoire, est une recherche du type arborescent puisque l'espace solution est parcouru comme s'il s'agissait d'un arbre de recherche.

La parallélisation d'une telle méthode pose des difficultés diverses causées par une presque stricte dépendance entre deux itérations consécutives. La connaissance par la totalité des processeurs des informations obtenues par l'un d'entre eux peut être fondamentale pour le bon déroulement de l'algorithme. Outre une telle nécessité de connaissance globale sur l'état actuel de la recherche, un autre problème très commun est celui de l'équilibrage des tâches, pour éviter que quelques processeurs travaillent d'avantage que d'autres.

Dans la première partie de ce **Chapitre 4** nous présentons un algorithme de Branch and Bound sur un hypercube à granularité fine dont la principale caractéristique est l'accès des processeurs à quelques informations sur l'état global du système, ce qui conduit à un équilibrage optimal des tâches, obtenu grâce à un mécanisme d'allocation dynamique pour les processeurs. Visant la résolution de problèmes d'optimisation combinatoire sur des architectures à grain fin, cet algorithme est en phase finale d'implantation sur la Connection Machine connectée à PARADISE (Sch. of Comp. Sci., Carleton U.).

Dans la seconde partie nous abordons la technique du Recuit Simulé, aussi très répandue dans l'optimisation combinatoire. Nous nous intéressons au comportement de cet algorithme en ce qui concerne sa probabilité d'aboutissement, i.e., la probabilité qu'une solution soit trouvée en un nombre donné de pas.

Sous des conditions idéales, l'algorithme du Recuit Simulé converge asymptotiquement vers une solution optimale avec une probabilité 1. Pourtant, nous prouvons que cette convergence est moins rapide que celle de la technique - beaucoup plus simple - de Recherche Aléatoire Locale. Nous démontrons aussi que, pour des variables bien définies, ne dépendant que de l'application en vue, la probabilité d'aboutissement de l'algorithme du Recuit Simulé en un nombre fini d'itérations (i.e., dans tous les cas pratiques) est bornée supérieurement.

Enfin nous étendons ces résultats pour le mode parallèle puisque, comme pour le Branch and Bound, beaucoup d'études portent sur l'utilisation de machines parallèles pour la diminution du temps de calcul, assez important, de l'algorithme du Recuit Simulé.

CHAPITRE I. THEORIES ET MODELES

La recherche appartient à une classe de problèmes qui peuvent être résolus à l'aide uniquement de comparaisons (et éventuellement d'échanges) effectuées entre deux éléments. Outre les problèmes basés sur des comparaisons, la classe des *problèmes non-numériques* connaît une définition assez floue et peut contenir, selon les différentes notions, le calcul symbolique, l'optimisation combinatoire et l'intelligence artificielle. Le sujet de cette thèse porte donc sur la résolution algorithmique de quelques problèmes de cette classe, liés à la recherche dans des ensembles ordonnés.

1.1 COMPLEXITÉ DES ALGORITHMES

La définition de l'*opération de base* d'un algorithme diffère selon le modèle de calcul choisi. Elle peut être une comparaison, une addition ou un échange de deux éléments. Pourvu qu'une telle opération puisse être implantée en un nombre constant d'unités de temps sur un ordinateur typique, on pourra mesurer la qualité (*complexité*) d'un algorithme par la quantité d'opérations de base (*temps d'exécution*) qu'il utilise pour résoudre une instance du problème dans le *pire cas*. Pour que la complexité d'un algorithme soit indépendante d'une implantation sur des machines existantes, cette mesure est faite asymptotiquement, en oubliant les facteurs constants qui peuvent intervenir suivant l'ordinateur choisi. Le temps d'exécution d'un algorithme, aussi bien que l'espace mémoire dont il a besoin, est défini en fonction de la taille de l'instance du problème à traiter.

Pour cela on définit les notations utilisées pour la description du comportement d'un algorithme sur une instance de taille *n* d'un problème donné.

Soient f(n) et g(n) deux fonctions dont l'argument est un entier et dont le résultat est un réel positif, alors :

(i) **g(n) = Ω(f(n))**, s'il existe des constantes positives c et n_0 telles que $g(n) \geq c.f(n)$ pour tout $n \geq n_0$. On dit que *g est d'ordre au moins f.*

(ii) **g(n) = O(f(n))**, s'il existe des constantes positives c et n_0 telles que $g(n) \leq c.f(n)$ pour tout $n \geq n_0$. On dit que *g est d'ordre au plus f.*

(iii) **g(n) = o(f(n))**, si le rapport g(n)/f(n) tend vers 0 quand n tend vers l'infini.

(iv) **g(n) = Θ(f(n))**, si à la fois g(n) = Ω(f(n)) et g(n) = O(f(n)).

L'existence d'une *borne inférieure* Ω(f(n)) pour un problème Q signifie qu'aucun algorithme ne peut le résoudre en moins de g(n) = Ω(f(n)) *pas* (opérations de base). Dans ce cas, on dit que Q est en Ω(f(n)). De même, une *borne supérieure* O(f(n)) pour Q signifie que parmi tous les algorithmes connus pour résoudre Q il existe un algorithme dont le nombre de pas (g(n)) est minimum **et** g est d'ordre au plus f. Alors on dit que Q est d'ordre f. Finalement, on dit que la complexité de Q est Θ(f(n)), si Ω(f(n)) = g(n) = O(f(n)). Dans ce cas, n'importe quel algorithme qui résout Q en O(f(n)) est dit *optimal.*

Pour illustrer l'utilisation de ces notations, étudions le problème de sélectionner le k-ième plus petit élément d'un ensemble D quelconque de taille n. Pour sélectionner l'élément souhaité il faut qu'on examine, au moins une fois, chacun des éléments de D. On aura donc un nombre de pas

au moins proportionnel à la taille de D. Ainsi, $\Omega(n)$ en est une borne inférieure. Par conséquent, la complexité d'un algorithme pour sélectionner le k-ième plus petit élément de D sera au moins *linéaire* en n. Un algorithme optimal (i.e., en O(n)) a été proposé par Blum et al. ([Blu73]), ce qui montre que la complexité du problème de sélection est $\Theta(n)$.

1.2 MODELES DE CALCUL

En ce qui concerne l'algorithmique séquentielle, le modèle de calcul le plus répandu et qui reflète l'organisation de la grande majorité des ordinateurs existants est celui des machines à accès aléatoire - *Random Access Machines* (RAM) ([Aho74]). Une telle machine est constituée d'un programme fini, d'une collection finie de registres et d'une mémoire (voir figure 1.2.1). En un pas on peut exécuter une opération arithmétique ou logique unique sur les contenus des registres spécifiés, ou bien stocker le contenu d'un registre dans une position de la mémoire désignée par une adresse contenue dans un autre registre.

Quand on passe au mode parallèle, on est enclin à définir une machine à accès aléatoire en parallèle - *Parallel Random Access Machine* (PRAM) -, comme étant simplement un système multi-processeur lié à une mémoire centrale. Cependant, plusieurs difficultés se présentent au niveau de la définition d'un tel modèle ; soit pour la résolution des conflits d'accès simultanés à la mémoire, soit pour la définition exacte de *processeur*. De ce fait on connaît différents modèles de machines parallèles, ce qui implique des mesures différentes de complexité d'algorithmes parallèles. D'autre part, s'il est vrai qu'en séquentiel les progrès réalisés en architecture des ordinateurs n'influençaient que la constante liée au temps de cycle des machines, il n'en est pas de même pour le parallélisme, où on ne peut plus

parler de la complexité d'un algorithme sans préciser à la fois le modèle et l'architecture choisis.

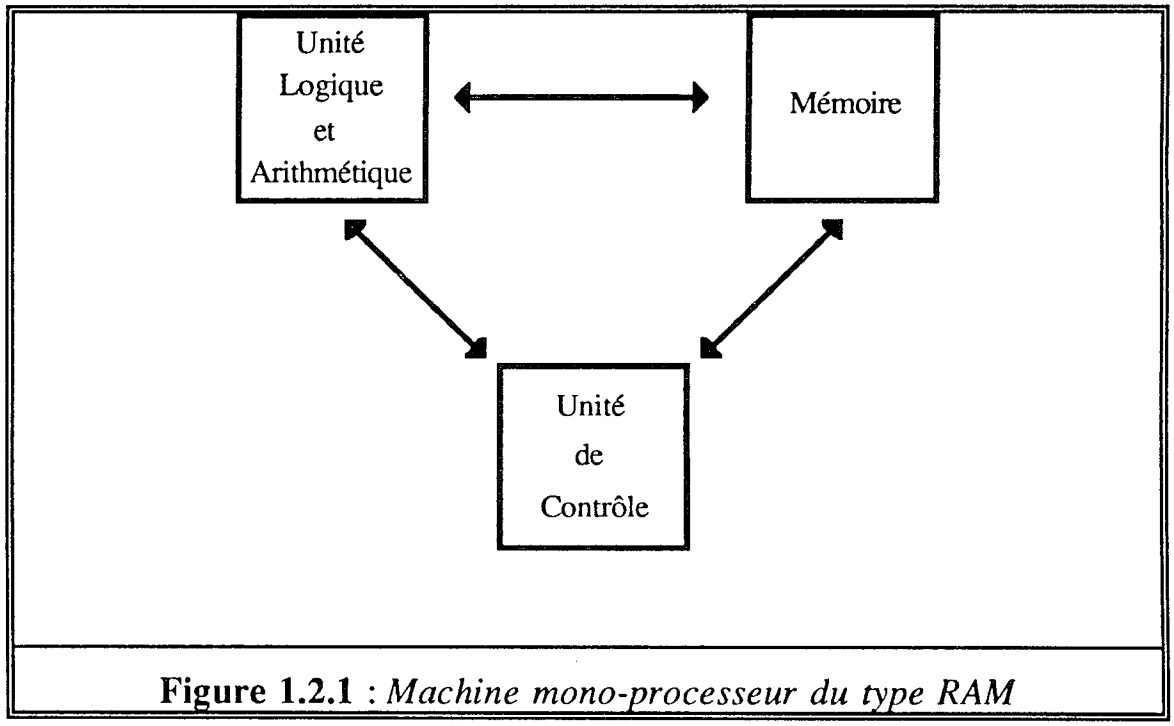

Figure 1.2.1 : *Machine mono-processeur du type RAM*

1.3 PRAM

En 1966 Flynn ([Fly66]) a proposé une classification des modèles des machines parallèles alors existantes, dont nous retiendrons ici celui des machines à flot d'instructions unique et données multiples (*Single Instruction Multiple Data* - SIMD). Ce modèle décrit des machines composées de processeurs (PE's) qui travaillent en mode synchrone, simultanément, sur des données multiples, contrôlés par un unique flot d'instructions issu d'une unité centrale de contrôle (voir figure 1.3.1). La communication inter-processeurs et leurs échanges de données sont assurés soit par un réseau d'interconnexion prédéfini soit par une mémoire partagée.

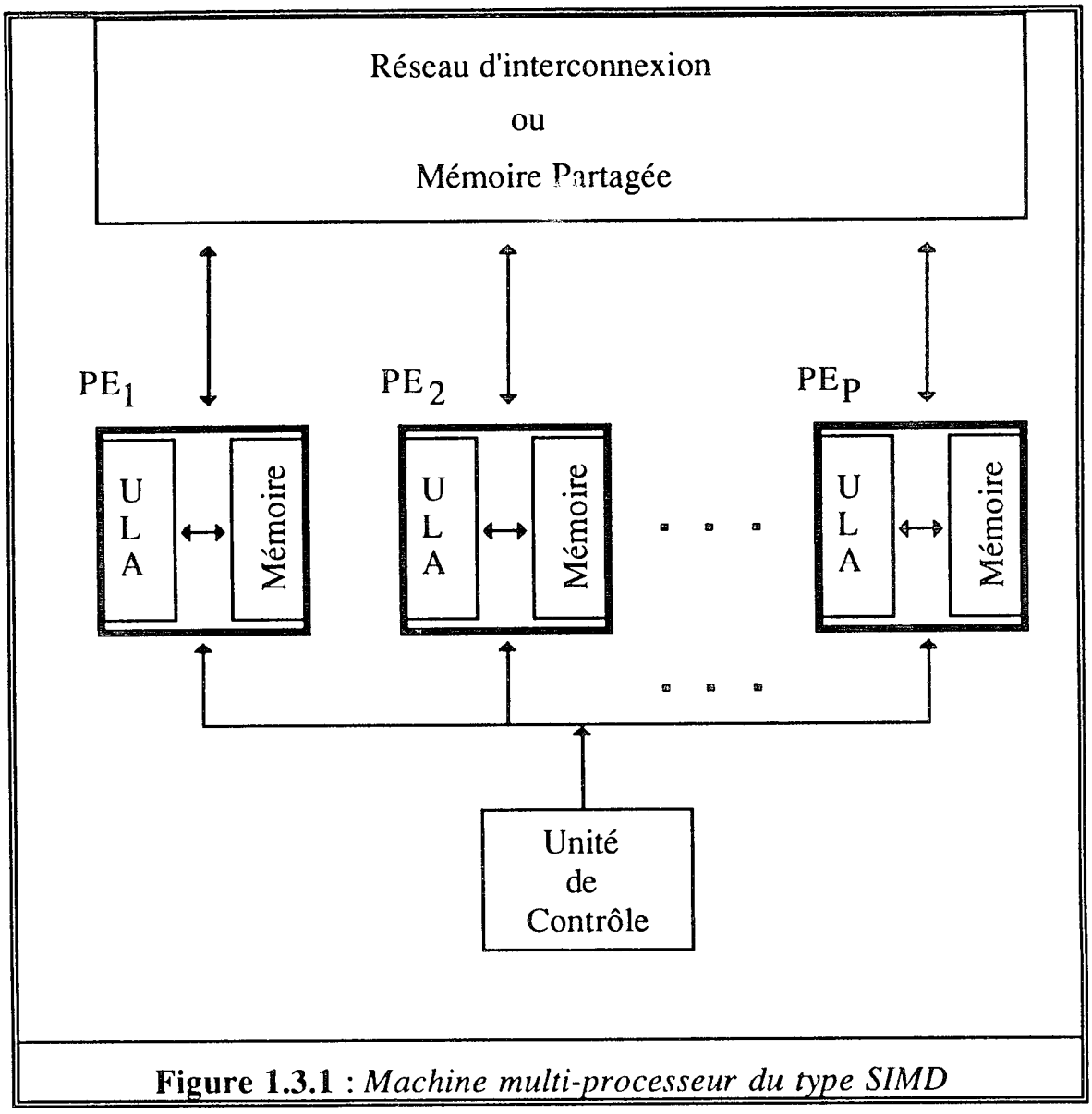

Figure 1.3.1 : *Machine multi-processeur du type SIMD*

Pour les *machines à mémoire partagée* (ou PRAM) on suppose que chaque PE contient une mémoire locale dont la capacité réduite est compensée par l'accès à une grande mémoire, commune à tous les processeurs. En un seul cycle un PE peut accéder à une cellule de la mémoire centrale ou exécuter sur sa propre mémoire locale une opération de base. Les conflits entre processeurs apparaissent quand plus d'un PE veut lire ou écrire sur la même position de la mémoire partagée. Les différentes règles employées pour restreindre l'accès multiple à une cellule de la mémoire dans un cycle définissent les différents modèles de machines à

mémoire partagée. Premièrement introduites par Snir ([Sni82]), les règles les plus courantes, par ordre croissant de contraintes, sont :

(i) **CRCW** (*Concurrent Read Concurrent Write*) ; des lectures et écritures multiples et simultanées dans les mêmes positions de la mémoire sont permises. Ici encore, lorsque des écritures multiples se produisent, plusieurs règles existent pour définir quelle valeur doit être finalement écrite sur la cellule requise.

(ii) **CREW** (*Concurrent Read Exclusive Write*) ; les cellules de la mémoire sont disponibles pour des lectures simultanées. Cependant, deux PE's n'ont pas accès simultanément, en écriture, à la même case mémoire.

(iii) **EREW** (*Exclusive Read Exclusive Write*) ; c'est le plus restrictif des modèles de PRAM et aussi le plus proche de l'organisation des machines existantes. N'y sont permises ni des lectures ni des écritures simultanées sur une même cellule de la mémoire. L'accès à une case mémoire par un PE bloque l'accès des autres PE's à cette position.

Il parait raisonnable que des règles plus restrictives définissent des modèles moins performants. En effet, il a été prouvé que, pour quelques problèmes spécifiques, une CRCW permet l'obtention de solutions dans un temps moindre que la borne inférieure pour ce même problème sur une CREW. De façon analogue une CREW a été prouvée plus puissante qu'une EREW.

La démonstration de la distinction entre les CREW et les EREW est basée sur le problème de la recherche d'un élément dans une liste triée de taille n. Snir ([Sni82]) a montré que sur une CREW avec P processeurs, ce problème est de complexité $\Theta((log(n+1))/(log(P+1)))$, tandis que sur une EREW la borne inférieure est en $\Omega(log n)$ - indépendante du nombre de

processeurs utilisés. Par conséquent il existe ici un facteur (logP) entre ces deux modèles.

Pour prouver que les CRCW sont plus puissantes que les CREW on considère le problème de du calcul du OU booléen sur n bits. Une machine CRCW le résout en temps constant avec n processeurs ([Kru85]). Par contre, ce problème sur CREW est en $\Omega(log n)$, borne démontrée par Cook et Dwork ([Coo82]). Par conséquent il existe un facteur (logP) entre les CRCW et CREW.

A ce point il est intéressant de noter que la simulation des lectures simultanées est obtenue en utilisant une simple opération de diffusion de la donnée requise pour tous les processeurs ; si les processeurs s'organisent, par exemple, en arbre binaire, alors cet algorithme s'exécute en O(logP) pas. En ce qui concerne les règles d'écriture simultanée sur une CRCW, le même argument montre que leur simulation par un modèle à écritures exclusives ne prend que O(logP) pas aussi. Ainsi, on en déduit la figure suivante :

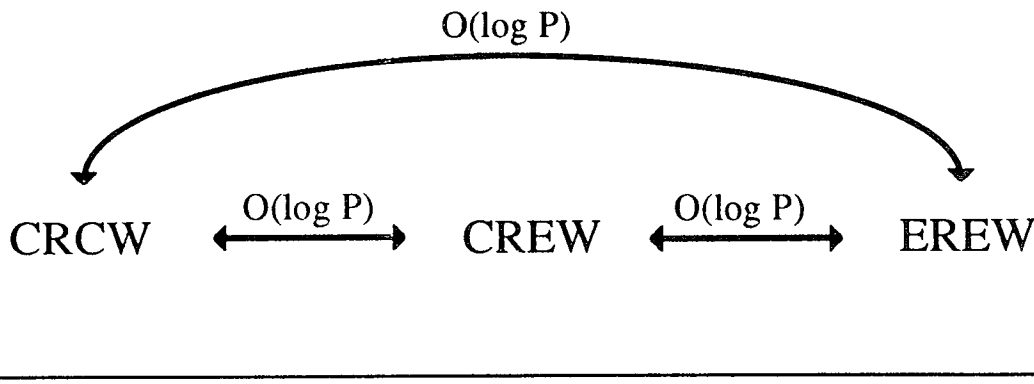

Figure 1.3.2 : *Relations entre les règles des PRAM*

1.4 ARCHITECTURES À MÉMOIRE DISTRIBUÉE

Au lieu d'une mémoire partagée, la communication et l'échange de données inter-processeurs peuvent être assurés par un réseau d'interconnexions. Plusieurs topologies ont été étudiées pour l'implantation d'un tel réseau et avec des notations et analyses empruntées à la théorie des graphes, des architectures en *réseau linéaire*, *grille*, *mélange parfait* et *hypercube* - pour n'en citer que quelques unes -, ont été proposées comme topologies pour l'implantation d'un tel réseau, donnant origine à ce qu'on appelle *Distributed Memory Machines* (DMM).

Par simplicité on représente les *machines à mémoire distribuée* (ou DMM) comme un graphe de processeurs interconnectés, où les PE's sont les noeuds et les liens de communication sont les arêtes. Chaque processeur dispose alors d'une mémoire privée, la communication entre deux PE's étant effectuée via un dispositif spécifique : un canal de communication. Deux processeurs sont *voisins* s'ils sont directement liés par un tel canal. La *distance* entre PE' et PE" est le minimum d tel que la séquence PE' = PE_0, PE_1, ..., PE_d = PE" est une suite de voisins. Le *diamètre* d'une architecture parallèle est la distance maximum parmi tous les couples de processeurs. Enfin, le *degré* d'un réseau est le nombre maximum de liens par processeur.

Un modèle communément utilisé suppose que le temps de communication entre deux processeurs voisins est proportionnel à la quantité d'informations échangées et, par conséquent, le temps de communication entre deux processeurs quelconques est proportionnel à leur distance. Il est important de remarquer que le modèle PRAM équivaut à un couplage des processeurs suivant la structure d'un graphe complet, puisque de ce fait tous les processeurs ont accès direct à la mémoire de tous les autres.

1.4.1 RÉSEAU LINÉAIRE ET ANNEAU

La plus simple des architectures à mémoire distribuée est celle du *réseau linéaire*, où P processeurs sont connectés de telle façon que chaque PE_i est directement lié à PE_{i-1} et PE_{i+1} (s'ils existent). Son degré est constant et égal à 2 et le diamètre est (P-1). Si l'on y ajoute une connexion entre PE_P et PE_1, on génère un *anneau* de processeurs, de même degré, mais où le diamètre est (P/2) au lieu de (P-1).

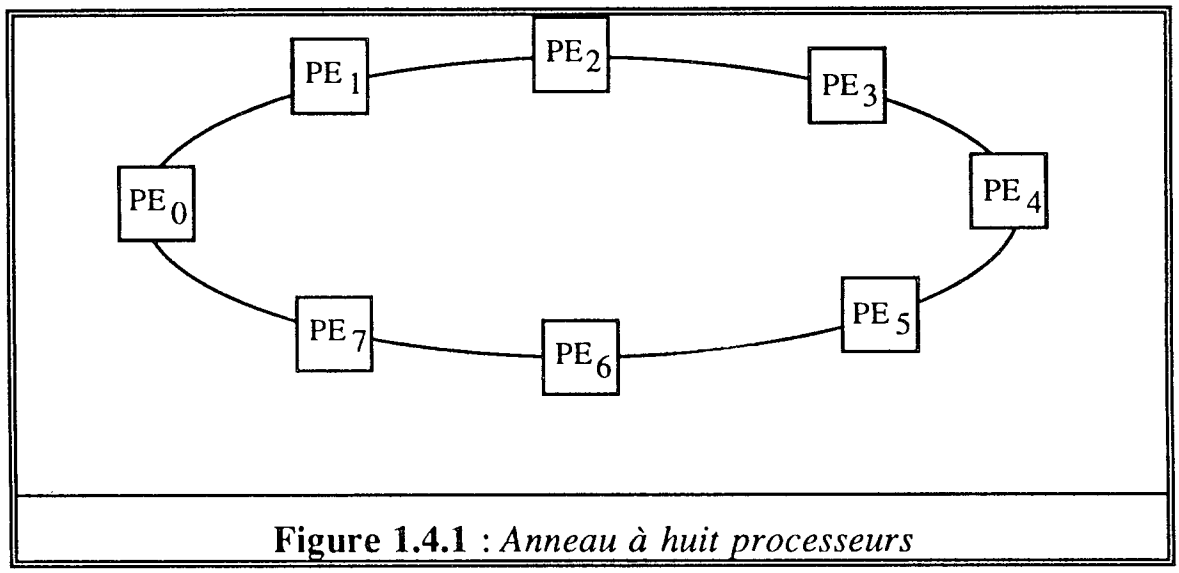

Figure 1.4.1 : *Anneau à huit processeurs*

1.4.2 GRILLE

Les distances inter-processeurs et le diamètre sont quelque peu réduits sur la *grille* à deux dimensions par rapport au réseau linéaire. Cette topologie est structurée sous forme d'une matrice ($\sqrt{P}$ x $\sqrt{P}$) de processeurs (où P est un carré) de telle façon qu'un processeur (i,j) est relié aux processeurs (i+1,j), (i,j+1), (i-1,j) et (i,j-1), s'ils existent. Le diamètre dans ce cas vaut ($2\sqrt{P}$- 1), i.e., la distance entre les processeurs qui occupent les coins opposés de la grille, tandis que le degré est égal à 4.

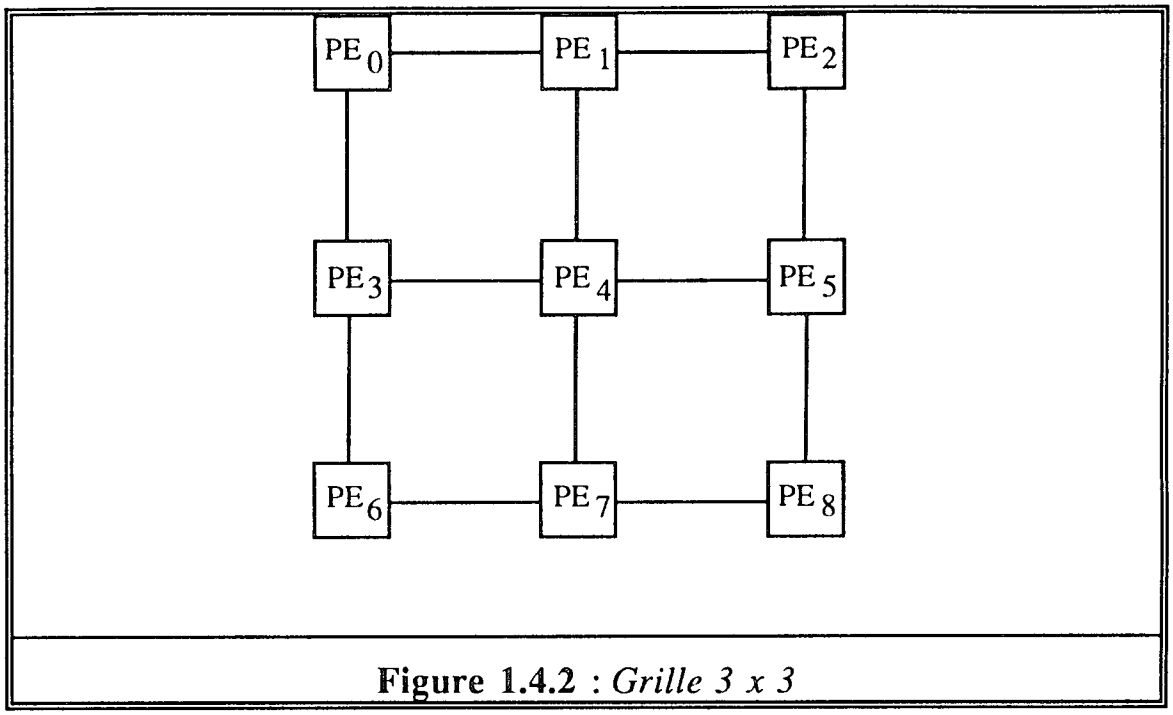

Figure 1.4.2 : *Grille 3 x 3*

1.4.3 MÉLANGE PARFAIT

Dans la classe des architectures à degré constant, celle du *mélange parfait* (perfect shuffle) est la base de la *machine à mélange et décalage*, dont le diamètre est inférieur ou égal à $2log P$. La fonction mélange (parfait) est définie, pour $P = 2^q$, par :

$$\sigma(i) = \begin{cases} 2i, & \text{si } i < P/2 ; \\ 2i - P + 1, & \text{si } i \geq P/2 \end{cases}$$

et la fonction inverse est telle que :

$$\sigma^{-1}(i) = \begin{cases} i/2, & \text{si } i \text{ est pair} ; \\ \dfrac{i - 1 + P}{2}, & \text{si } i \text{ est impair} \end{cases}$$

Les quatre voisins de chaque PE_i dans une machine à mélange et décalage sont les suivants :

(1) Gauche : $PE_{(i-1) \bmod P}$

(2) Droit : $PE_{(i+1) \bmod P}$

(3) Mélange: $PE_{\sigma(i)}$

(4) Inverse : $PE_{\sigma^{-1}(i)}$.

Sur la figure 1.4.3 un exemple où n = 8 :

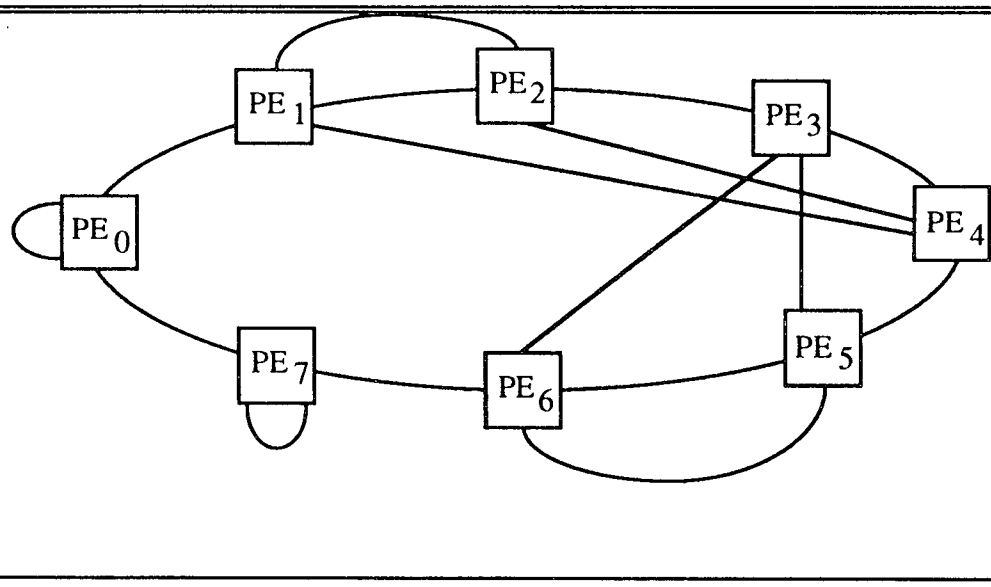

Figure 1.4.3 : *Machine à mélange et décalage avec huit processeurs*

1.4.4 HYPERCUBE

La dernière architecture que nous mentionnons ici est celle de l'*hypercube*. D'utilisation courante pour la construction de machines commerciales et de recherches, la topologie en hypercube est composée de P = 2^d processeurs.

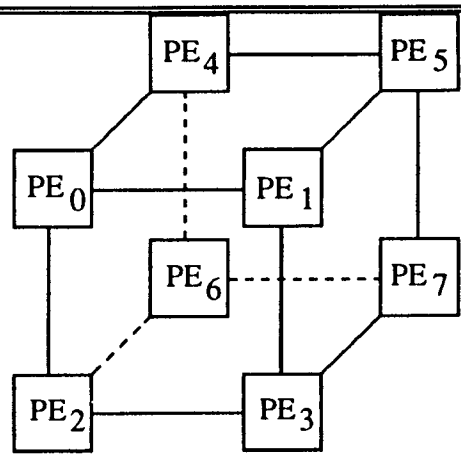

Figure 1.4.4 : *Hypercube de dimension 3*

Soit $i_{d-1}i_{d-2}...i_1i_0$ la représentation binaire de i. Alors les voisins de PE_i sont tous les PE_j tels que la représentation binaire de j diffère de celle de i en exactement une position. De cette façon, les processeurs sont les sommets d'un hypercube de dimension d, chacun relié aux d sommets voisins. La plus grande distance dans un hypercube est celle entre deux processeurs dont les représentations binaires diffèrent dans toutes les d positions. De ce fait le diamètre d'un hypercube est d. Le principale désavantage d'une telle architecture vient de son degré, qui n'est plus constant mais logarithmique en le nombre de processeurs.

1.5 COMPLEXITÉ DES ALGORITHMES PARALLELES

Après avoir décrit les notions de modèle et architecture, le prochain pas vers l'évaluation d'algorithmes parallèles consiste à définir les outils pour en mesurer la qualité. Le temps d'exécution parallèle est défini comme le nombre de pas du processeur le plus lent quand l'algorithme traite l'instance du problème qui représente son pire cas. Les communications inter-processeurs, bien que beaucoup plus coûteuses que les opérations de

base définies précédément, sont aussi comptées comme ayant un coût constant, i.e., envoyer ou recevoir un élément à travers un canal de communication prend O(1) pas.

Formellement, on définit le facteur d'*accélération* (Acc) apporté par un algorithme parallèle comme étant le rapport entre les temps d'exécution séquentiel et parallèle pour un problème donné :

$$\text{Acc} = \frac{\text{Complexité du meilleur algorithme séquentiel connu}}{\text{Temps d'exécution de l'algorithme parallèle}}$$

De par le fait que la définition de l'accélération utilise le meilleur algorithme séquentiel connu (i.e., le plus rapide), l'accélération d'un algorithme parallèle doit être inférieure ou égale au nombre de processeurs utilisés. Sinon, la simulation de ces processeurs en séquentiel donnerait un nouvel algorithme séquentiel plus rapide que celui d'où on était parti.

Si l'algorithme parallèle produit une accélération d'ordre P, alors *son accélération est dite optimale.* Un *algorithme parallèle est dit optimal* si de deux choses l'une : ou il produit une accélération optimale et le meilleur algorithme séquentiel connu est lui même optimal, ou son temps d'exécution parallèle égale la borne inférieure parallèle pour le problème traité.

Une autre mesure largement utilisée dans l'algorithmique parallèle est celle de l'*efficacité* (e) d'un algorithme, qui montre le taux de travail des processeurs. L'efficacité, inférieure ou égale à 1, est définie par :

$$e = \frac{\text{Accélération}}{P}$$

Dans le cas où (e = 1) les processeurs travaillent tout le temps, sans avoir des périodes d'inactivité. *L'efficacité est par conséquent dite optimale.*

1.6 GRANULARITÉ DU PARALLÉLISME

Enfin, un dernier mot doit être adressé sur le nombre de processeurs utilisés pour la construction effective de machines et leur puissance de calcul et de stockage. Si l'on opte pour l'utilisation de processeurs très puissants, alors le coût de connexion peut être prohibitif - s'il s'agit d'une architecture complexe. Or, il s'avère qu'il existe un compromis entre le nombre de processeurs d'un système parallèle et leur capacité : ou bien on en couple des petits par dizaines de milliers, comme sur la Connection Machine ([Hil87]), (système à *grain fin de parallélisme*) ou bien on utilise un nombre réduit de processeurs très performants, comme sur le CRAY ([Sto87]), (système à *gros grain de parallélisme*). Il est en tout cas certain que ces chiffres (et non pas l'idée sous-jacente) sont passibles de changement dû au développement de la technologie.

Grosso modo on considère qu'un système à grain fin utilise $P = O(n)$ processeurs pour la résolution d'un problème de taille n et donc chaque PE s'occupe d'un élément. Pour les gros grain $P = o(n)$ et peut être défini comme $P = n^{1-\varepsilon}$, $0<\varepsilon<1$, par exemple.

Pour décider quel type d'application est plus approprié pour quel système, Stone ([Sto87]) donne un critère basé sur le rapport du nombre d'opérations arithmétiques sur la quantité de communications requises. Soit donc *Op* le nombre d'opérations arithmétiques nécessaires pour accomplir une exécution d'un algorithme ; soit aussi *C* le nombre de communications effectuées pendant cette même exécution. Si le rapport **Op/C** est faible, alors il est préférable d'utiliser un système qui peut communiquer beaucoup et rapidement, peut-être à l'aide de dispositifs matériels spécifiques. Par conséquent le grain fin est le plus adéquat. Par contre, si une grande masse

de calcul est faite entre chaque communication, le mieux est d'avoir un système où les processeurs calculent efficacement, même s'ils communiquent moins bien ou sont en moindre nombre. Dans ce cas on utilise un système à gros grain de parallélisme ([Sto87]).

1.6.1 PRAM OU DMM

Sans doute le modèle à mémoire partagée est plus puissant que le modèle à mémoire distribuée, puisqu'il réprésente en fait la topologie du graphe complet. Une question assez intéressante concerne la portabilité d'algorithmes écrits pour les machines à mémoire partagée sur des machines à mémoire distribuée de topologies différentes. Effectivement, le problème qui se pose est celui du routage des informations à travers le réseau d'interconnexion. Sur, par exemple, une CREW-PRAM quand tous les PE's veulent lire une même position de la mémoire, cela se fait automatiquement en un temps de cycle. Cependant, cette même instruction sur une DMM requiert une procédure de diffusion pour router l'information.

Deux versions du problème de la simulation automatique des PRAM sur des machines à mémoire distribuée ont été étudiées par Nassimi et Sahni ([Nas81]) : *lecture à accès aléatoire* et *écriture à accès aléatoire*. Ces deux modes d'accès sont les plus courants dans les algorithmes pour les machines à mémoire partagée, et signifient qu'à un moment donné n'importe quel processeur peut lire sur n'importe quelle position de la mémoire (de même pour l'écriture).

Ici nous ne montrerons que les procédures de base proposées pour la simulation des CREW et EREW sur des hypercubes (et sur des machines à mélange et décalage puisque celles-ci peuvent simuler des hypercubes), dont

nous aurons besoin au chapitre 4. Soit O(t(n)) la complexité d'un algorithme sur une PRAM avec n processeurs, alors, en utilisant les procédures ci-dessous, cet algorithme induit un algorithme en temps $O(t(n)log^2n)$ sur un hypercube de dimension n.

On suppose n processeurs disponibles, numérotés de 0 à n-1. Pour cela la formalisation des problèmes est telle que :

(i) **Lecture à accès aléatoire** : Chaque PE_i contient un indice S(i), $0 \leq i < n$, indiquant que PE_i doit recevoir une donnée de $PE_{S(i)}$. Il est admis que cette donnée se trouve dans l'enregistrement G(S(i)), où **enregistrement(i)** indique la variable **enregistrement** de PE_i. Si PE_i ne veut pas lire pendant ce cycle, alors $S(i) = \infty$.

(ii) **Ecriture à accès aléatoire** : Chaque PE_i contient un indice D(i), indiquant qu'il veut envoyer son enregistrement G(i) au processeur $PE_{D(i)}$. Si PE_i ne veut rien écrire, $D(i) = \infty$.

1.6.2 PROCÉDURES DE ROUTAGE

L'implantation de ces deux procédures de diffusion sont faites en utilisant quelques procédures bien définies, décrites dans ([Nas81]) :

(1) **Sort** : Trie des enregistrements G(i) en ordre non croissant selon une clé prédéfinie.

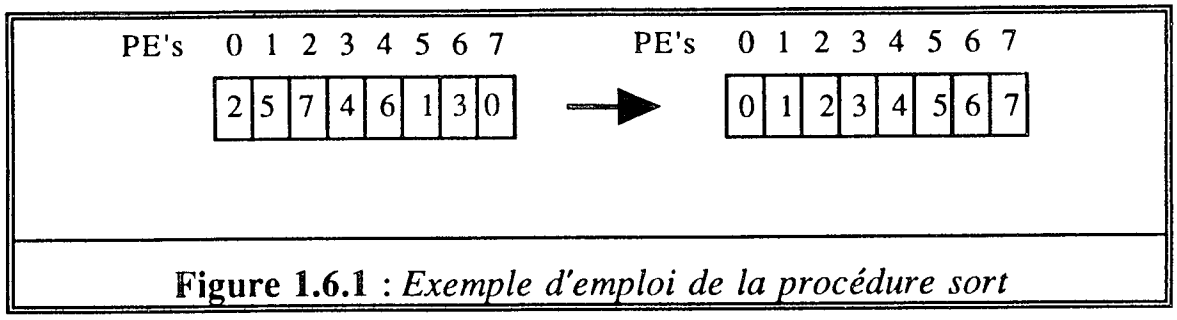

Figure 1.6.1 : *Exemple d'emploi de la procédure sort*

(2) **Rank** : Le rang d'un enregistrement G(i) sélectionné est le nombre d'enregistrements G(j) sélectionnés tel que j < i.

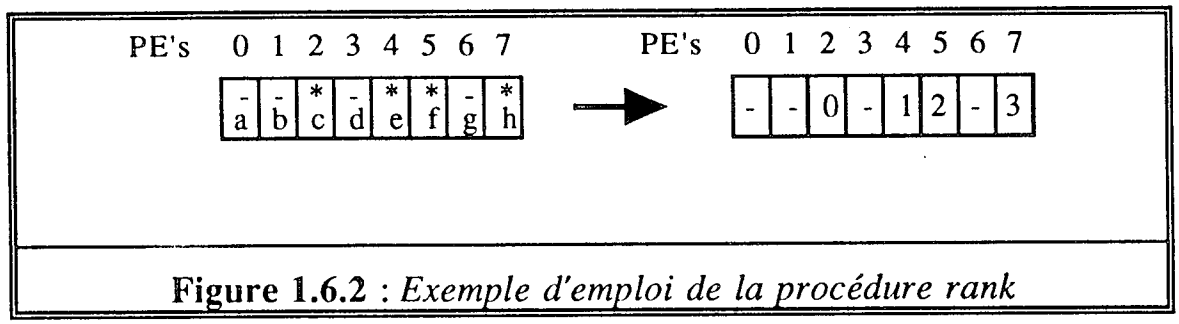

Figure 1.6.2 : *Exemple d'emploi de la procédure rank*

(3) **Concentrate** : Elle suppose que des enregistrements $G(i_r)$ ont un rang r, r=0,1,...,j<n. Un appel de la procédure Concentrate positionne l'enregistrement $G(i_r)$ sur le processeur PE_r.

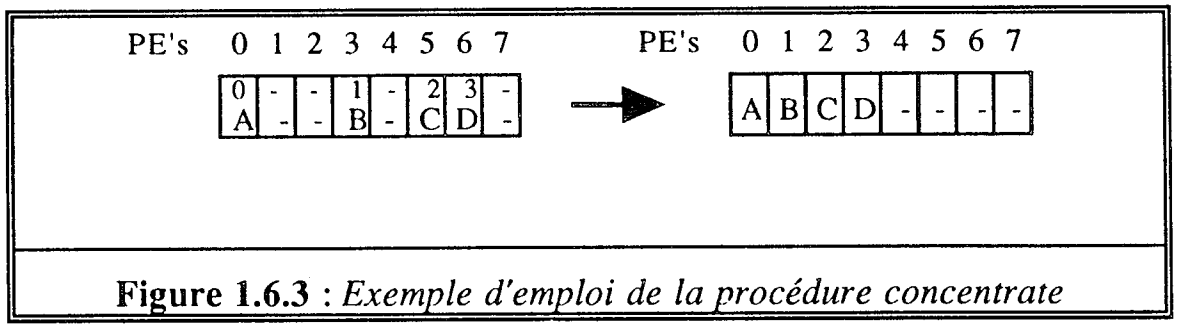

Figure 1.6.3 : *Exemple d'emploi de la procédure concentrate*

(4) **Distribute** : Cette procédure effectue l'opération inverse de Concentrate. Elle suppose que des enregistrements G(i), i=0,1,...j < n, ont des destinations D(i), $0 \leq i \leq j$, **telles que** $0 \leq D(i) < D(i+1) < n$, pour $0 \leq i < j$. Un appel de Distribute route les G(i) aux processeurs $PE_{D(i)}$.

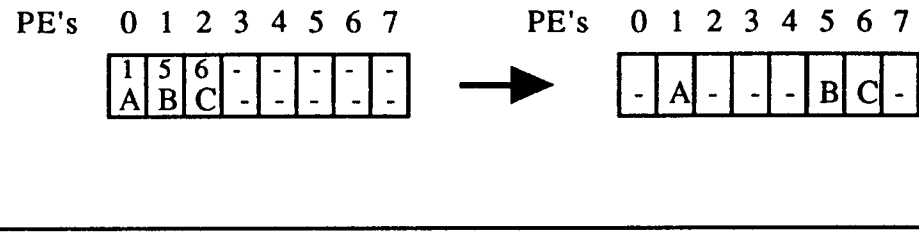

Figure 1.6.4 : *Exemple d'emploi de la procédure distribute*

(5) **Generalize** : Suppose que des enregistrements G(i), i=0,1,...j < n, ont des destinations D(i), **telles que** $0 \leq D(i) < D(i+1) < n$, pour $0 \leq i < j$. Un appel de Generalize route une copie de G(i) à chaque PE_k, $D(i-1) + 1 \leq k \leq D(i)$, où D(-1) est supposée 0 et $D(r) = \infty$ pour $r > j$.

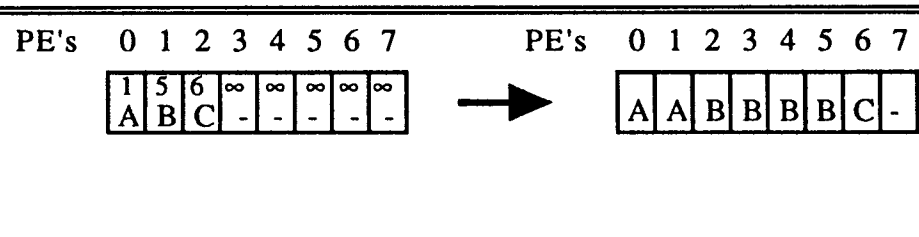

Figure 1.6.5 : *Exemple d'emploi de la procédure generalize*

Les problèmes de diffusion de données sur des DMM sont donc résolus par une suite convenable d'appels des procédures décrites ci-dessus. La simulation de l'écriture à accès aléatoire d'une CREW (ou EREW), par exemple, ne nécessite qu'un appel de Sort suivi d'un appel de Distribute, en ayant les destinations comme clé. A l'exception de Sort, toutes les autres procédures sont de complexité optimale et coûtent O(log n) pas sur l'hypercube. En ce qui concerne le tri en $O(\log^2 n)$ sur l'hypercube (celui proposé par Batcher ([Knu72])), il n'a pas été prouvé optimal, bien qu'il soit depuis plus d'une vingtaine d'années le meilleur algorithme de tri connu sur de telles architectures.

Ce qu'on doit retenir est le coût total de la simulation automatique d'un pas d'un algorithme sur une CREW avec n processeurs : $O(\log^2 n)$ sur un hypercube avec n processeurs.

En outre, nous aurons aussi besoin d'une procédure de sommes partielles, exécutée en temps $O(\log n)$ sur n processeurs, soit sur une CREW-PRAM, soit sur un hypercube :

(6) **SomPar** : Elle suppose que des sommes partielles sont à effectuer sur une variable S, affectant le total à la variable SP(i). Un appel de la procédure SomPar calcule, pour toute variable S(i) sélectionnée, la somme SP(i) := S(0) + S(1) + ... + S(i).

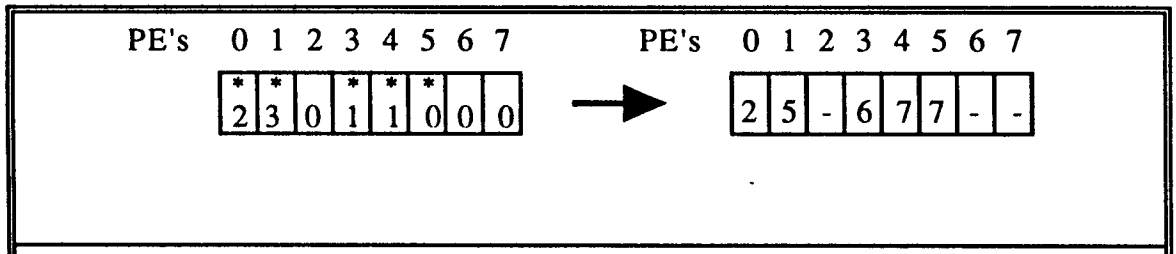

Figure 1.6.6 : *Exemple d'emploi de la procédure de sommes partielles*

CHAPITRE II. MATRICES TRIEES ET X + Y

Une matrice nxn est dite triée si chaque ligne et chaque colonne est ordonnée. Si X et Y sont deux vecteurs ordonnés, alors la somme cartésienne A = X + Y, où $a_{ij} = x_i + y_j$, est un cas particulier de telles matrices (*cf.* figure 2.1, ci-dessous). Bien qu'ils aient n^2 éléments, l'avantage de tels ensembles vient du fait qu'ils ne requièrent que 2n cellules de mémoire pour être représentés. Dans ce chapitre on propose un tour d'horizon des complexités de trois problèmes corrélés sur des matrices triées et sur X + Y (les deux vecteurs étant triés) : sélectionner le k-ième plus petit élément, rechercher l'occurrence d'un élément donné et trier tous les éléments.

$$A = \begin{bmatrix} 2 & 4 & 5 \\ 6 & 7 & 8 \\ 7 & 8 & 9 \end{bmatrix}$$

$$A = X+Y = \begin{bmatrix} 2 & 4 & 7 \\ 3 & 5 & 8 \\ 4 & 6 & 9 \end{bmatrix}$$

$X = \{0, 1, 2\}$

$Y = \{2, 4, 7\}$

Figure 2.1 : *Exemple d'une matrice triée et d'une matrice X + Y*

Le tri, la recherche et la sélection dans de telles matrices ont reçu une attention considérable dans la littérature, dûe à leurs applications en statistique, conception de circuits VLSI, recherche opérationnelle et combinatoire, entre autres ([Joh78a], [Joh78b], [Mir84], [Mir85], ...). Ces problèmes ont été étudiés avec différentes hypothèses sur les ensembles partiellement ordonnés concernés. Dans le cas où X et Y sont triés,

l'ensemble des matrices de la forme X + Y est un sous-ensemble de l'ensemble des matrices triées. Les algorithmes conçus pour ce dernier s'appliquent donc immédiatement au premier. Par conséquent, les bornes supérieures proposées pour la résolution de problèmes dans des matrices triées sont valables aussi pour X + Y (triés). De même, les bornes inférieures prouvées pour X + Y (triés) induisent les mêmes bornes inférieures pour les matrices triées.

Dans la suite on s'intéresse au cas où X et Y sont déjà triés ; sauf mention contraire, on suppose que X et Y sont de taille n et ordonnés en ordre croissant.

La complexité des problèmes cités ci-dessus est bien définie pour les matrices triées, ce qui veut dire que des bornes supérieures existent pour ces mêmes problèmes dans X + Y. Par contre, des bornes inférieures étaient inexistantes. Les résultats originaux qui suivent ont été trouvés en compagnie, toujours agréable, de Michel Cosnard et Jean Duprat ([Cos89a], [Cos89b], [Cos89c], [Cos89d]).

2.1 SÉLECTION DU K-IEME

Le problème de la sélection dans une matrice triée A est celui de déterminer, pour un rang k donné, un élément qui est le k-ième plus petit dans l'ordre total défini sur A. Frederickson et Johnson ([Fre84]) ont démontré que la complexité de ce problème de sélection dépend du rang k, en contraste avec le problème classique de sélection dans des ensembles quelconques, dont la complexité est linéaire sur la taille de l'ensemble et où le rang k n'influence que la constante associée.

On montre ci-dessous l'algorithme proposé dans ([Fre84]) dans sa version de base, dont la complexité est en O(n), la dimension de la matrice A donnée, l'argument qui fait descendre la complexité de l'algorithme à $O(\sqrt{k})$ étant ensuite abordé. Un tel algorithme est immédiatement valable pour la sélection dans X + Y. Par contre, la démonstration de la borne inférieure - elle aussi proportionnelle à $\sqrt{k}$ -, construite dans ([Fre84]), ne s'applique qu'à des matrices triées.

Pour prouver la borne inférieure pour le problème de la sélection dans X + Y nous construisons une matrice X + Y spéciale, tel que ce problème soit réduit au problème de trouver la médiane d'un ensemble non ordonné quelconque. Une telle construction se montrera très utile par la suite, puisqu'elle sera utilisée tout au long de ce chapitre pour l'obtention de bornes inférieures pour des problèmes correlés.

2.1.1 BORNE SUPÉRIEURE

Le problème de la sélection du k-ième plus petit élément où n excède k peut être transformé en temps constant en une autre instance équivalente où n = k, par la simple affectation de k à n. On suppose, donc, $n \leq k$ et n une puissance de 2 pour faciliter les démonstrations.

L'algorithme Sélection ([Fre84]) présenté à la figure 2.1.1 partitionne la matrice initiale A en des sous-matrices (*cellules*) et exécute des sélections sur ces cellules. Les sélections sont faites par paires, de façon à déduire une borne inférieure et une borne supérieure pour le k-ième plus petit élément. Il est donc possible de déterminer des cellules qu'il est inutile de considérer. A chaque itération les cellules restantes, qui forment l'ensemble *CELLS*, sont à nouveau partitionnées pour permettre l'obtention de bornes de plus

en plus fines. Ces itérations aboutissent à la formation d'un ensemble CELLS dont les cellules sont composées d'un seul élément, et l'algorithme classique de sélection ([Blu73]) est utilisé pour sélectionner un k'-ième plus petit élément de CELLS, qui est en fait le k-ième plus petit élément de A.

Comme les cellules sont des matrices triées, le plus petit et le plus grand éléments d'une cellule sont disponibles en temps constant. La division décrite au **pas 2.1** de l'algorithme Sélection crée quatre nouvelles cellules à partir de chaque cellule existante, soit par la division de chaque dimension en deux ou, si la cellule est un tableau à une dimension, par la division de l'unique dimension en quatre.

La structure des matrices triées induit une partition de l'ensemble des cellules existantes en des sous-ensembles qu'on appelle *chaines*. Si deux cellules C' et C'' sont dans la même chaine alors ou max(C') ≤ min (C''), ou max(C'') ≤ min (C'). Si B_i est le nombre maximum de chaines différentes possibles après la division des cellules à la i-ième itération, alors on montre que $B_i = \min \{n, 2^{i+1} - 1\}$.

Au **pas 2.2**, l'élément a_s sélectionné majore le nombre d'éléments candidats. Par conséquent, il suffit de garder q - 1 cellules dont le plus petit élément est plus petit que a_s. De façon similaire, au **pas 2.3** l'élément sélectionné a_u est une borne inférieure pour tous sauf moins de k' éléments. Par conséquent, les cellules dont le plus grand élément est plus petit que celui sélectionné peuvent être supprimées. Enfin, au **pas 4**, CELLS compte O(n) éléments et le k'-ième plus petit est sélectionné en utilisant l'algorithme linéaire de sélection ([Blu73]). Voir exemple sur les figures 2.1.2.a, 2.1.2.b et 2.1.2.c.

Algorithme *Sélection*(CELLS, k)

1 k' := k ;

2 pour i = 1 jusqu'à (log_4 (n^2)) faire

2.0 $B_i = min$ {n, 2^{i+1} - 1} ;

2.1 Diviser chaque cellule de CELLS ;

2.2 Soit q := $\lceil k'/(n^2/4^i) \rceil + B_i$.

si (q ≤ |CELLS|) alors sélectionner un q-ième plus petit élément a_s dans l'ensemble {*min* (C) | C ∈ CELLS}. Supprimer |CELLS|-q+1 cellules de CELLS, tout en gardant toutes les cellules C où *min*(C) < a_s et aucune cellule C où *min*(C) > a_s ;

fin-si ;

2.3 Soit r := $\lfloor k'/(n^2/4^i) \rfloor - B_i$.

si (r ≥ 1) alors sélectionner un r-ième plus petit élément a_u dans l'ensemble {*max* (C) | C ∈ CELLS}. Supprimer r cellules de CELLS, tout en gardant toutes les cellules C où *max*(C) > a_u et aucune cellule C où *max*(C) < a_u et faire k' := k' - r($n^2/4^i$) ;

fin-si ;

3 fin-pour ;

4 Sélectionner le k'-ième plus petit élément de CELLS.

Figure 2.1.1 : *Algorithme Sélection*

La correction de l'algorithme repose sur le fait que le k'-ième plus petit élément de l'union de toutes les cellules restantes est bien le k-ième plus petit élément de la matrice initiale. Ceci n'est pas difficile à voir puisque sa mise à jour, à la fin du **pas 2.3**, tient compte de l'ancienne valeur de k' et du nombre d'éléments plus petits que le k'-ième qui ont été supprimés.

La complexité de l'algorithme provient d'une étude sur le nombre de cellules présentes à chaque itération. Frederickson et Johnson ([Fre84]) ont prouvé que moins de $2B_i$ cellules sont toujours présentes à la fin de la i-ième itération. Ce qui veut dire que les sélections à l'itération suivante porteront sur moins de $4(2B_i)$ cellules. L'utilisation d'un algorithme linéaire pour les sélections amène la complexité de chaque itération à être proportionnelle au nombre de cellules existantes : la complexité du **pas 2** est en O(n). A la fin de ce pas, moins de 2n cellules resteront et le **pas 3** prendra un temps O(n) aussi. De cette façon la complexité de l'algorithme Sélection(CELLS, k) est linéaire en n.

Exemple d'un appel de Sélection (figures 2.1.1.a-c), où k = 7, n = 4 et A est la matrice réprésentée à gauche dans la figure 2.1.1.a :

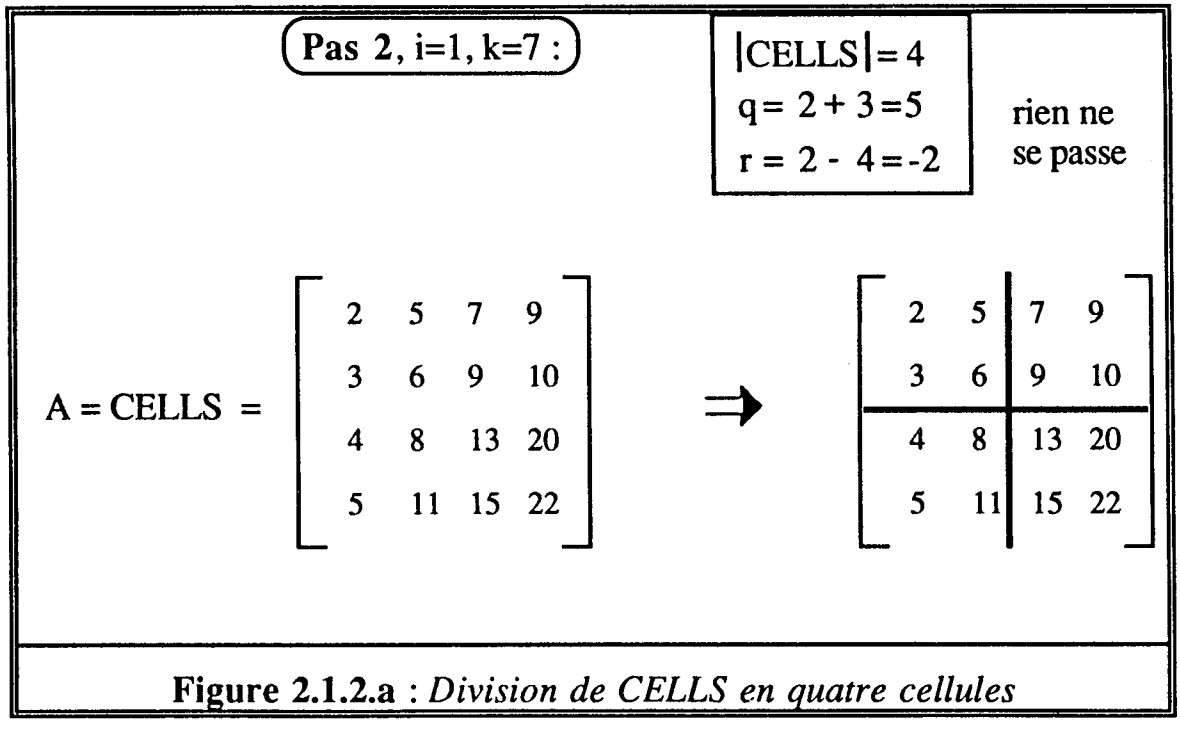

Figure 2.1.2.a : *Division de CELLS en quatre cellules*

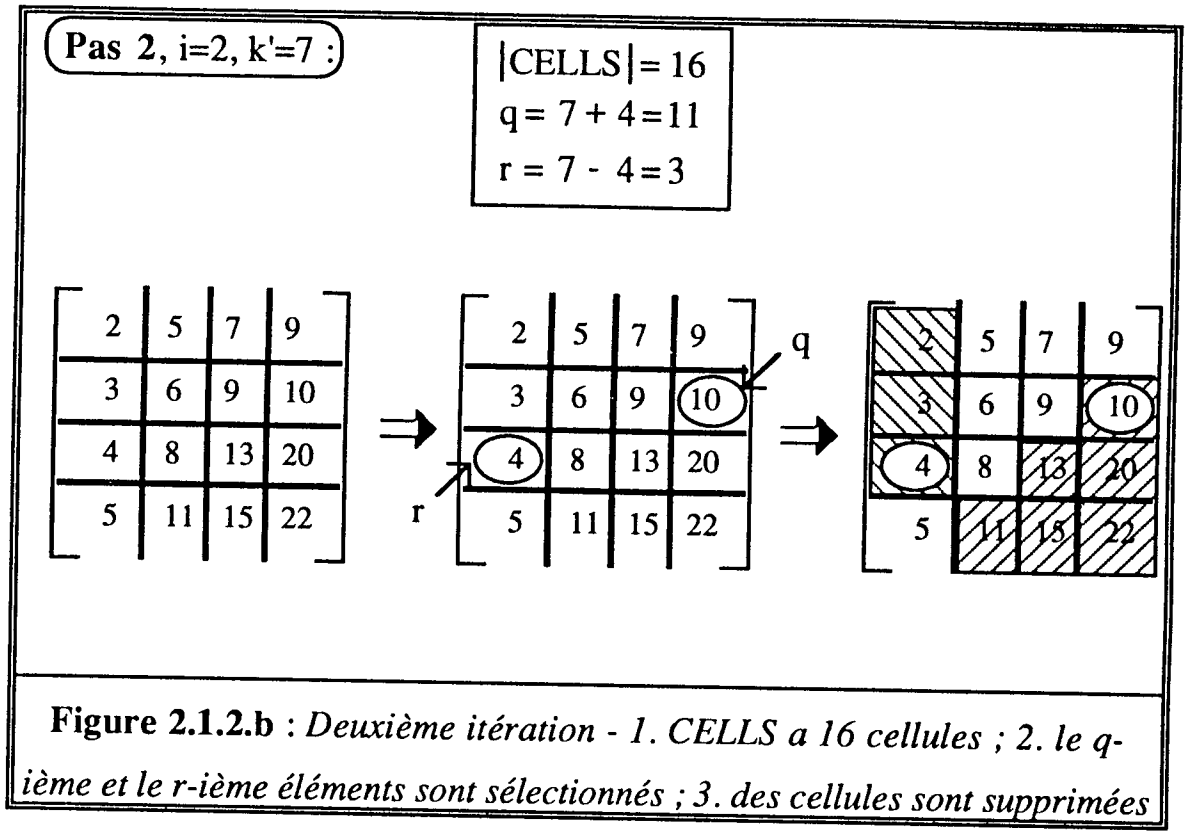

Figure 2.1.2.b : *Deuxième itération - 1. CELLS a 16 cellules ; 2. le q-ième et le r-ième éléments sont sélectionnés ; 3. des cellules sont supprimées*

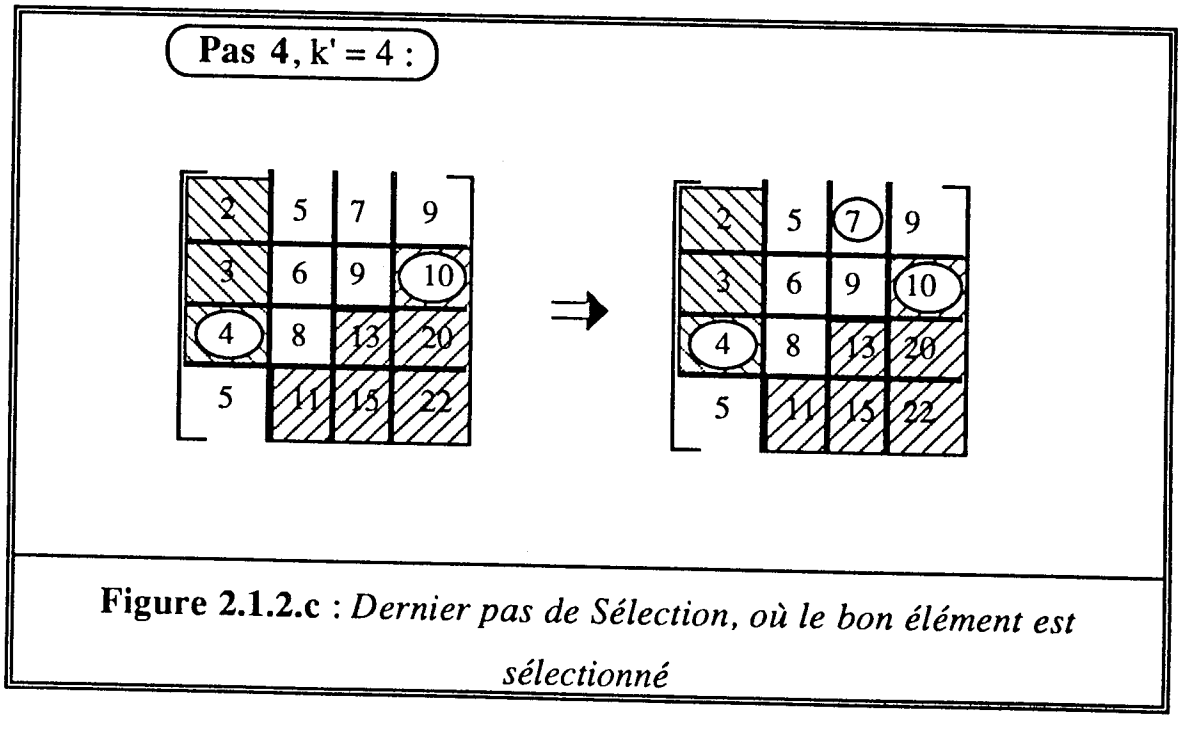

Figure 2.1.2.c : *Dernier pas de Sélection, où le bon élément est sélectionné*

Dans l'optique de matrices triées cet algorithme n'est pas optimal si $k = o(n^2)$, puisque la borne inférieure pour le problème de la sélection dans les matrices triées est en $\Omega(\sqrt{k})$ ([Fre84]). Pour accélérer cet algorithme et le faire tourner en temps optimal, il suffit d'extraire de la matrice donnée un ensemble de sous-matrices contenant tous les éléments inférieures au k-ième plus petit et au moins k éléments qui ne sont pas supérieurs. Même si ces sous-matrices ont différentes formes, elles ont toutes la même taille et l'algorithme Sélection peut être directement appliqué. Ainsi on prouve le résultat suivant ([Fre84]) :

Théorème 2.1.1 La complexité du problème de la sélection dans des matrices triées est $\Theta(\sqrt{k})$.

2.1.2 BORNE INFÉRIEURE

La sélection dans X + Y, où les vecteurs sont supposés indistinctement triés ou non triés, est un problème très bien étudié ([Fre82], [Fre84], [Joh78a], [Joh78b], [Mir84], [Mir85]), le théorème 2.1.1 ci-dessus ayant comme conséquence immédiate :

Corollaire 2.1.2 Le k-ième élément de X + Y peut être sélectionné en $O(\sqrt{k})$.

On rappelle que toute mention à X + Y fait référence au cas où les deux vecteurs sont triés, sauf remarque contraire.

Si ce corollaire donne une méthode de sélection dans X + Y (l'algorithme Sélection de la figure 2.1.1), aucune borne inférieure n'avait été obtenue pour le cas où X et Y sont triés. Par la réduction du problème de sélection dans X + Y à celui de trouver la médiane d'un ensemble

quelconque, nous démontrons que le k-ième plus petit élément de X + Y ne peut être sélectionné en moins de $O(\sqrt{k})$ pas.

Soit D un vecteur de taille $q \leq n$. La médiane de D est définie comme son (q/2)-ième plus petit élément. Nous construisons une matrice X + Y, telle que la sélection du k-ième plus petit élément d'une telle matrice en temps T(k) induit un algorithme pour trouver la médiane de D en temps $O(T(q^2))$. Comme D est non trié, calculer sa médiane prend au moins q pas et la borne inférieure en découle immédiatement.

Soit alors

$$D = \{d_i\},\ 0 \leq d_i < M\ ;\ i = 1,2,...,q \leq n\ ;\ \text{pour un M donné.}\quad (1)$$

On définit X et Y comme suit (noter qu'ils sont triés en ordre croissant) :

$$x = (i-1).M + d_i\ ;\ i = 1,...,n \qquad (2)$$

$$y = (j-q).M\ ;\ j = 1,...,n \qquad (3)$$

Lemme 2.1.3 Soient X et Y définis ci-dessus et $A = X + Y = \{x_i + y_j \mid i,j = 1,2,...,n\}$. Soit $q \leq n$ et U l'ensemble des éléments de la q-ième anti-diagonale de A : $U = \{u_i = x_i + y_j \mid i+j = q+1\}$.

Alors $u_i = d_i$ (i.e., U = D).

Preuve :

$$u_i = x_i + y_{q+1-i} = (i-1).M + d_i + (q+1-i-q).M =$$

$$= (i-1+1-i).M + d_i = d_i.\quad \blacklozenge$$

Lemme 2.1.4 Soit U défini comme au lemme précédent. Soit $GU = \{x_i + y_j \mid i+j < q+1\}$ l'ensemble des éléments de A qui sont à gauche de U et $DU = \{x_i + y_j \mid i+j > q+1\}$ l'ensemble des éléments de A qui sont à droite de U. Soit u un élément quelconque de U.

Alors :

4.1. $\forall a \in GU, a < u.$

4.2. $\forall b \in DU, b > u.$

Preuve :

(4.1) $a \in GU \Rightarrow a = x_i + y_j \mid i+j < q+1$

$\Rightarrow a = (i-1).M + d_i + (j-q).M =$

$= (i+j-(q+1)).M + d_i$

$\Rightarrow a \leq d_i - M$

$\Rightarrow a < 0$ par (1)

$\Rightarrow a < u$ par le lemme 2.1.3

(4.2) Analogue à 4.1. ◆

En d'autres termes, les lemmes 2.1.3 et 2.1.4 montrent que U = D sépare A en deux ensembles spéciaux : la partie triangulaire en haut et à gauche (GU) où les éléments sont strictement inférieurs à ceux de U ; et la partie triangulaire en bas et à droite (DU) où les éléments sont strictement supérieurs à ceux de U (voir figure 2.1.3). De plus, l'ordre dans l'ensemble D est respecté dans l'ensemble U.

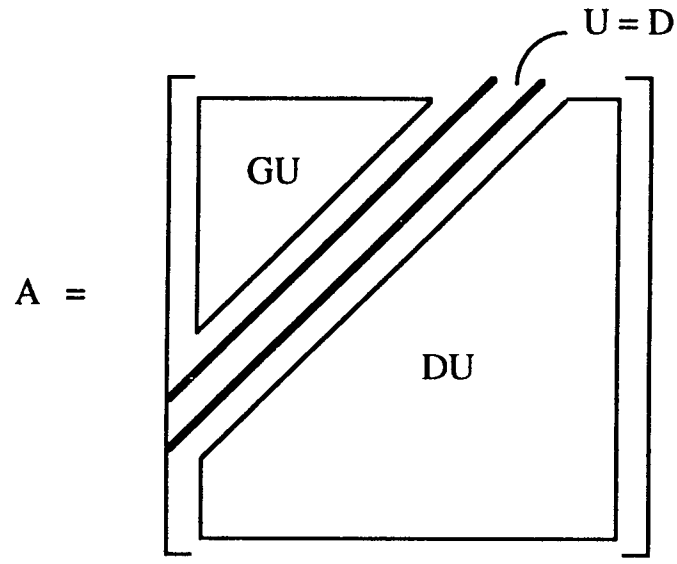

Figure 2.1.3 : *La matrice A et les ensembles U, D, GU et DU*

Proposition 2.1.5 S'il existe un algorithme pour sélectionner le k-ième plus petit élément de X + Y en temps T(k), alors il existe un algorithme pour trouver la médiane d'un ensemble quelconque de taille $q \leq n$ en temps $O(T(q^2))$.

Preuve :

Soit **Alg**(k,X,Y) un algorithme qui sélectionne le k-ième plus petit élément de X + Y en temps T(k). Soient M, D, X et Y comme définis de (1) à (3).

On introduit des modifications dans **Alg** pour créer l'algorithme **Med**(D), de la façon suivante :

a) Chaque référence à X[i] dans **Alg** est changée en (i-1).M + D[i] dans **Med**.

b) Chaque référence à Y[j] dans **Alg** est changée en (j-q).M dans **Med.**

c) Chaque référence à k dans **Alg** est changée en $(q/2 + (q^2-q)/2) = q^2/2)$ dans **Med.**

Comme GU a $(q^2-q)/2)$ éléments, le lemme 2.1.4 garantit que le $(q^2/2)$-ième plus petit élément de X + Y appartient à l'ensemble U. De plus il est la médiane de U et donc (lemme 2.1.3) la médiane de D.

Les modifications a, b et c introduites dans **Alg** n'incrémentent le temps T(k) que d'un facteur constant. Ainsi la complexité de **Med**(D) est la complexité de **Alg**$(q^2/2)$, i.e., $O(T(q^2))$. ♦

Théorème 2.1.6 Le problème de la sélection dans X + Y est en $\Omega(\sqrt{k})$ temps.

Preuve :

Le résultat est une conséquence directe de la proposition 2.1.5 et de la borne inférieure $\Omega(q)$ pour trouver la médiane d'un ensemble quelconque de taille q. ♦

Corollaire 2.1.7 La complexité du problème de la sélection en X + Y est $\Theta(\sqrt{k})$ temps.

Preuve :

Elle est immédiate à partir du théorème 2.1.6 ci-dessus et du corollaire 2.1.2, sur la complexité de l'algorithme Sélection(CELLS, k) pour X + Y. ♦

2.2 RECHERCHE

Des applications du problème de la recherche dans des matrices triées peuvent être trouvées dans plusieurs domaines de l'informatique, e.g., la résolution efficace de problèmes NP-Complets, aussi bien en séquentiel qu'en parallèle (*cf.* chapitre 3), ou le problème de décider si X et Y ont des éléments communs puisque, en fait, rechercher z dans X + Y et rechercher des éléments communs entre X et z-Y sont des problèmes équivalents.

Soit z un élément donné. Par la suite on désigne par *la recherche de z dans X + Y* le problème de trouver s'il existe p et q tels que $z = x_p + y_q$. Bien que ce problème, sous cette formulation, n'ait reçu l'attention que récemment ([Cos89b]), des algorithmes pour le résoudre existent depuis assez longtemps ([Hor74], [Knu72]). L'algorithme que l'on présente ici est très simple et élégant et s'exécute en temps linéaire. Nous prouvons que cet algorithme est en fait optimal. La construction de la borne inférieure est basée sur les idées développées pour le problème de la sélection dans X + Y.

2.2.1 BORNE SUPÉRIEURE

L'algorithme ci-dessous a été proposé par différents auteurs dans des contextes divers (e.g., [Hor74], [Knu72]) :

Algorithme *find*(z,X,Y)

```
1   i:=n ; j:=1 ;
2   Si (x_i + y_j = z) alors arrêter : la recherche a abouti ;
3   Si (x_i + y_j > z)
4         alors i:=i-1 sinon j:=j+1 ;
5   Si ((i < 1) ou (j > n)) alors arrêter : la recherche n'a pas abouti;
6   Aller en 2
```

Figure 2.2.1 : *Algorithme find*

Lemme 2.2.1 L'algorithme find(z,X,Y) recherche z dans X + Y en temps O(n).

Preuve :

La **ligne 4** peut être vue comme la suppression d'un élément de X ou de Y. La complexité de l'algorithme est en O(|X| + |Y|) = O(n), puisqu'on supprime au moins un élément à chaque itération.

Pour prouver sa correction on montre que si l'on supprime

$$x_i \in X$$

alors il n'existe aucun

$$y_j \in Y$$

tel que

$$x_i + y_j = z.$$

Y est en ordre croissant et les suppressions se produisent séquentiellement, d'où

$$x_i + y \geq x_i + y_j$$

quel que soit **y** parmi les éléments qui restent dans Y.

Ainsi, si $x_i + y_j > z$ à la **ligne 3** alors $x_i + y > z$ quel que soit **y** parmi les éléments qui restent dans Y. Ce qui implique que x_i peut être supprimé de X.

Le procédé pour justifier la suppression d'un élément $y_j \in Y$ est analogue. ♦

2.2.2 BORNE INFÉRIEURE

Pour démontrer la borne inférieure pour le problème de la recherche dans X + Y, on utilise la construction de la section 2.1.2. Le raisonnement

cette fois-ci est basé sur la réduction du problème de la recherche dans X + Y à celui de la recherche dans un ensemble quelconque de taille n.

Soient z un élément et D un vecteur de taille n donnés. On construit une matrice X + Y, telle que la recherche de z dans X + Y en temps T(n) induit un algorithme pour rechercher z dans D en temps O(T(n)). Comme D est non trié, on en déduit la borne inférieure en $\Omega(n)$.

Soit alors $D = \{d_i\}, 0 \leq d_i < M ; i = 1,2,...,n$; pour un M donné. (1)

On définit X et Y comme suit (noter qu'ils sont triés en ordre croissant) :

$x = (i-1).M + d_i ; i = 1,...,n$ (2)

$y = (j-n).M ; j = 1,...,n$ (3)

Lemme 2.2.2 Soient X et Y définis comme ci-dessus et $A = X + Y = \{x_i + y_j \mid i,j = 1,2,...,n\}$. Soit U l'ensemble des éléments de l'anti-diagonale principale de A : $U = \{u_i = x_i + y_j \mid i+j = n+1\}$.

Alors $u_i = d_i$ (i.e., U = D).

Les démonstrations de ce lemme et de celui qui suit sont basiquement les mêmes que celles décrites pour les lemmes 2.1.3 et 2.1.4.

Lemme 2.2.3 Soit U défini comme au lemme 2.2.2. Soit $GU = \{x_i + y_j \mid i+j < n+1\}$ l'ensemble des éléments de A qui sont à gauche de U et $DU = \{x_i + y_j \mid i+j > n+1\}$ l'ensemble des éléments de A qui sont à droite de U. Soit u un élément quelconque de U.

Alors :

3.1. $\forall a \in GU, a < u$.

3.2. $\forall b \in DU, b > u$.

Proposition 2.2.4 S'il existe un algorithme pour rechercher z dans X + Y en temps T(n), alors il existe un algorithme pour rechercher z dans un ensemble quelconque de taille n en temps O(T(n)).

Preuve :

Soit **Alg**(z,X,Y) un algorithme qui recherche z dans X + Y en temps T(n). Soient M, D, X et Y comme définis de (1) à (3).

On introduit des modifications dans **Alg** pour créer l'algorithme **Rec**(D), de la façon suivante :

a) Chaque référence à X[i] dans **Alg** est changée en (i-1).M + D[i] dans **Rec**.

b) Chaque référence à Y[j] dans **Alg** est changée en (j-n).M dans **Rec**.

On a alors deux réponses possibles produites par **Rec**(z,X,Y) :

1. $z \notin X + Y$.

Par le lemme 2.2.2, $X + Y \supset D$. Donc $z \notin D$.

2. $z \in X + Y$.

Alors il existe u et v tels que $z = x_u + y_v$. Les lemmes 2.2.2 et 2.2.3 impliquent que $z \in D \Leftrightarrow u + v = n + 1$. Ainsi il suffit de tester (u + v) pour décider si $z \in D$ ou non.

Les modifications a et b introduites dans **Rec** n'incrémentent le temps T(n) que d'un facteur constant. Par conséquent la complexité de **Rec**(D) est O(T(n)) temps. ♦

Théorème 2.2.5 Le problème de la recherche dans X + Y est en temps $\Omega(n)$.

Preuve :

Le résultat est une conséquence directe de la proposition 2.2.4 et de la borne inférieure $\Omega(n)$ pour rechercher dans un ensemble quelconque de taille n. ♦

Corollaire 2.2.6 La complexité du problème de la recherche dans X + Y est $\Theta(n)$ temps.

Preuve :

Elle est immédiate à partir du théorème 2.2.5 et du lemme 2.2.1 sur la complexité de l'algorithme find(z,X,Y). ♦

2.2.3 EXTENSION À DES MULTI-ENSEMBLES

Dans cette section, on s'intéresse à la recherche dans des multi-ensembles de la forme $X_1 + X_2 + \ldots + X_m$, où $m > 2$ et tous les vecteurs X_i ont n éléments triés. Par la suite, on notera un tel multi-ensemble $\sum X_i$. Parmi les résultats que l'on montre ici, certains généralisent ceux proposés

par Schroeppel et Shamir ([Sch81]), qui ont proposé des méthodes équivalentes pour résoudre de façon efficace une classe de problèmes NP-Complets. D'ailleurs, comme on verra plus loin dans le texte, certains problèmes NP-Complets et la recherche dans des multi-ensembles sont des problèmes étroitement liés.

Dans une première approche, l'algorithme find(z,X,Y) peut être immédiatement étendu pour la recherche dans des multi-ensembles.

Théorème 2.2.7 La recherche dans $A = \sum X_i$ peut être exécutée en temps $O(n^{m-1})$.

Preuve :

Soit z l'élément à rechercher. Pour chaque élément b_j de $\sum X_i$, $3 \le i \le m$, on applique find(z-b_j,X_1,X_2), ce qui conduit à un algorithme en temps $O(n^{m-1})$ pour la recherche dans $\sum X_i$. ♦

Corollaire 2.2.8 (m=3) Soient X, Y, W des vecteurs triés de taille n. Alors la recherche dans A = X + Y + W peut être réalisée en $O(n^2)$.

Preuve :

Algorithme *findthree*(z,X,Y,W)

```
1   k := 1 ;
2   Tant que (k ≤ n) et (find(z-w_k,X,Y) n'a pas abouti) faire k:=k+1 ;
3   si (k ≤ n) alors la recherche a abouti.
```

Figure 2.2.2 : *Algorithme findthree*

L'algorithme findthree fait appel à l'algorithme **find** n fois. Par conséquent sa complexité est $O(n^2)$ temps. ♦

Il est clair que l'algorithme de recherche dans $\sum X_i$ présenté dans le théorème 2.2.7 ne requiert que l'espace nécessaire pour garder les vecteurs X_i. De façon surprenante, si l'on autorise l'utilisation d'un espace supplémentaire, la recherche dans $\sum X_i$ peut être accélérée pour atteindre la complexité de $O(mn^{\lceil m/2 \rceil} log n)$ temps :

Algorithme *findmulti*$(z,X_1,\ldots,X_m)$

1 Construire un tableau $T_1 = X_1 + X_2 + \ldots + X_{\lfloor m/2 \rfloor}$
2 Construire un tableau $T_2 = X_{\lfloor m/2 \rfloor+1} + X_{\lfloor m/2 \rfloor+2} + \ldots + X_{2\lfloor m/2 \rfloor}$
3 Trier T_1 et T_2 en ordre croissant
4 Si {m est pair} alors find(z,T_1,T_2)
5 sinon findthree(z,T_1,T_2,X_m).

Figure 2.2.3 : *Algorithme pour rechercher dans* $\sum X_i$

On voit que :

Théorème 2.2.9 Si l'on autorise l'utilisation d'un espace supplémentaire de $O(n^{\lfloor m/2 \rfloor})$, alors la recherche dans $\sum X_i$ peut être effectuée en $O(mn^{\lceil m/2 \rceil} log n)$ temps.

Preuve :

L'espace supplémentaire est nécessaire pour garder les tables T_1 et T_2. La complexité temporelle dépend de la parité de m :

a) si m est pair, le tri coûte $O(mn^{m/2} log n)$ et l'appel de find coûte $O(mn^{m/2})$, ce qui donne un coût total en $O(mn^{m/2} log n)$.

b) si m est impair, le tri coûte $O(mn^{\lfloor m/2 \rfloor} log n)$ et l'appel de findthree coûte $O(mn^{\lceil m/2 \rceil})$, proportionnant un coût total en $O(mn^{\lceil m/2 \rceil})$.

◆

Encore plus inattendu est le fait que ce problème peut être résolu toujours en temps $O(mn^{\lceil m/2 \rceil} \log n)$, mais avec seulement $O(n^{\lceil m/4 \rceil})$ espaces supplémentaires, puisque l'algorithme find(z,T_1,T_2) examine les éléments des deux tables de façon séquentielle et ordonnée. Ainsi, si l'on connaît une technique permettant d'engendrer les éléments de T_1 et de T_2 dans l'ordre à partir des vecteurs X_i, alors on n'a pas besoin de construire effectivement les deux tables.

Pour des raisons de simplicité, on étudie le cas où m=4, sa généralisation étant directe. Il s'agit donc de rechercher dans quatre tables, à savoir (X + Y) + (R + S). Pour pouvoir appliquer l'algorithme find(z,X+Y,R+S) il faut qu'on soit capable d'engendrer rapidement (en temps $O(\log n)$)les éléments $(X + Y)_i$ et $(R + S)_j$. Le lemme ci-dessous, demontré indépendamment par Vyscoc ([Vys88]), fournit un tel algorithme.

Lemme 2.2.10 Soient X et Y deux vecteurs de taille n, X trié. Si l'on utilise O(n) espaces auxiliaires, alors la génération ordonnée de tous les éléments de X + Y peut être exécutée en temps O(n) pour le plus petit (grand) élément et puis en temps $O(\log n)$ par élément.

Preuve :

Comme X est trié, disons en ordre croissant, le plus petit élément de X + Y est $min\{x_i + y_j \mid 1 \le j \le n\}$.

Soit $x_1 + y_q$ un tel élément. Alors le deuxième plus petit élément de X + Y est $min\{\{x_1 + y_j \mid 1 \le j \le n, j \ne q\} \cup \{x_2 + y_q\}\}$.

Par un argument inductif, il est facile de voir que si l'on sélectionne le k-ième plus petit élément de X + Y - disons $x_u + y_v$ - à partir

d'un ensemble Q, alors le (k+1)-ième plus petit élément de X + Y appartient à l'ensemble $(Q \setminus \{x_u + y_v\}) \cup \{x_{u+1} + y_v\}$.

Par conséquent, pour prouver le lemme il suffit de trouver une structure de données qui permet des insertions et des suppressions arbritraires en temps logarithmique ; et qui donne accès, en temps constant, à la paire dont la somme est la plus petite.

Ce type de structure de données est connu comme liste de priorité et est implantée de façon efficace par un tas ([Aho74]), ce qui induit l'algorithme suivant :

Algorithme *sort*(X,Y)

1 Initialiser le tas H avec $\{x_1 + y_j \mid 1 \le j \le n\}$;
2 Pour k = 1 jusqu'à n^2 faire
3 $a_k := x_u + y_v = \min\{H\}$; % a_k est le k-ième plus petit élément de X + Y %
4 Supprimer $x_u + y_v$ de H ;
5 Insérer $x_{u+1} + y_v$ dans H ; % si x_{u+1} n'est pas défini, rien ne se passe %
6 fin-pour

Figure 2.2.4 : *Algorithme sort*

L'algorithme sort(X,Y) requiert O(n) cases mémoire en plus de celles nécessaires pour garder X et Y. Cet espace est utilisé pour la manipulation du tas H. L'initialisation du tas à la **ligne 1** coûte O(n) en temps et le plus petit élément de X + Y est immédiatement disponible. Pour les (n^2-1) éléments restants, une suppression suivie d'une insertion sont nécessaires. Chacune coûte O(*log*n) en temps et le lemme en découle. ◆

Il faut noter que le vecteur Y n'est pas forcément trié pour la démonstration de ce lemme. Cependant, même dans le cas où Y est trié, nous prouvons une borne inférieure en $\Omega(n)$ pour le calcul du successeur d'un élément donné, quelconque, de X + Y (*cf.* section 2.3). Cela implique qu'il existe une instance du problème pour laquelle l'algorithme sort(X,Y) nécessite un tas qui compare O(n) éléments, i.e., il est inutile de trier Y en ayant pour but l'accélération asymptotique de cet algorithme.

A l'aide des idées qui viennent d'être présentées on introduit un algorithme qui recherche un élément z dans (X + Y) + (R + S), X et R triés.

Algorithm *findfour*(z,X,Y,R,S)

```
1   i:=n^2 ; j:=1 ;
2   Initialiser le tas H1 avec {x_n + y_p | 1≤p≤n} ;
3   Initialiser le tas H2 avec {r_1 + s_q | 1≤q≤n} ;
4   Tant que ((i ≥ 1) et (j ≤ n^2)) faire
5         (a1,b1) := paire de somme maximum dans H1 ; % racine de H1 %
6         (a2,b2) := paire de somme minimum dans H2 ; % racine de H2 %
7         Si ((a1 + b1) + (a2 + b2) = z) alors arrêter : la recherche a abouti
8         fin-si ;
9         Si ((a1 + b1) + (a2 + b2) > z)
10              alors  supprimer (a1,b1) de H1 ;
11                     si {le prédécesseur a'1 de a1 dans X est défini}
12                           alors  insérer (a'1,b1) dans H1 ; i:=i-1 ;
13                     fin-si ;
14              sinon  supprimer (a2,b2) de H2 ;
15                     si {le successeur a'2 de a2 dans R est défini}
16                           alors  insérer (a'2,b2) dans H2 ; j:=j+1 ;
17                     fin-si ;
18        fin-si ;
19  fin tant que
20  Arrêter : la recherche n'a pas abouti
```

Figure 2.2.5 : *Algorithme findfour*

Le lemme 2.2.10 et cet algorithme conduisent à :

Théorème 2.2.11 Si l'on utilise O(n) espaces auxiliaires, alors la recherche dans $A = X_1 + X_2 + X_3 + X_4$, X_i triés, peut être effectuée en temps $O(n^2 log n)$. ♦

La généralisation de ce théorème au cas où $m > 4$ est la même que celle utilisée par le théorème 2.2.9. D'où :

Théorème 2.2.12 Si l'on peut utiliser $O(n^{\lceil m/4 \rceil})$ en espace auxiliaire, alors la recherche dans $A = \sum X_i$, $1 \le i \le m$, peut être effectuée en temps $O(mn^{\lceil m/2 \rceil} log n)$. ♦

Une question qui naturellement s'impose est de savoir s'il existe un algorithme encore plus performant pour la recherche dans un multi-ensemble du type $A = \sum X_i$. Malheureusement, nous montrons par le théorème 2.2.13 ci-dessous qu'il est très difficile de répondre à cette question, du moins en ce qui concerne la complexité exponentielle des algorithmes proposés jusqu'à présent.

Théorème 2.2.13 La recherche dans $A = \sum X_i$, quels que soient n et m, est un problème NP-Complet.

Preuve :

Soit z l'élément recherché. Il est facile de voir que le problème de la recherche dans $A = \sum X_i$ appartient à la classe NP ([Gar79]), puisque pour chaque solution possible $(u_1, \ldots, u_m)$, $1 \le u_i \le n$, vérifier si $z = X_1[u_1] + \ldots + X_m[u_m]$ peut être exécuté en temps polynomial.

D'autre part, le problème de décision du sac-à-dos est NP-Complet ([Kar72]). Ce problème consiste à déterminer l'existence d'un m-

uplet binaire $(t_1, t_2, \ldots, t_m)$ qui satisfasse $\sum w_i t_i = b$, étant donnés des entiers positifs ou nuls $w_1, w_2, \ldots, w_m$ et b. Un algorithme qui recherche b dans $A = \sum X_i$, où $X_i = \{0, w_i\}$, peut être utilisé pour résoudre ce problème. Par conséquent, l'existence d'un algorithme en temps polynomial pour le problème de la recherche dans $A = \sum X_i$ impliquerait l'existence d'un algorithme en temps polynomial pour un problème NP-Complet. ◈

2.2.4 PROBLEMES OUVERTS

Lineal et Saks ([Lin85]) ont étudié la complexité de la recherche dans des matrices triées à q dimensions. Soit A une telle matrice, alors l'ordre partiel défini sur A est tel que $A[i_1, i_2, \ldots, i_q] \leq A[j_1, j_2, \ldots, j_q]$ si et seulement si $i_k \leq j_k$, pour tout k, $1 \leq i_k, j_k \leq n$. Ils ont démontré par ailleurs que la complexité d'un tel problème est $\Theta(n^{q-1})$. De telles matrices sont aussi connues sous le nom de *cubes triés*, le cube étant celui en $\{1, 2, \ldots, n\}^q$. Quand la dimension du cube est égale à 2 on retrouve le cas des matrices triées. Un cas particulier de la recherche dans des cubes triés est celui qu'on vient de débattre : la complexité du problème de la recherche dans X + Y et d'autres multi-ensembles partiellement ordonnés.

Pour le cas des matrices triées, la complexité de la recherche est égale à celle qui a été prouvée ici pour X + Y, à travers la construction d'un hyperplan composé de n éléments deux à deux incomparables dans l'antidiagonale principale de X + Y, dans lequel nous avons montré comment immerger un ensemble quelconque non ordonné de taille n.

Par contre quand on s'intéresse au même problème en dimension 3, la question reste ouverte. Si l'on recherche dans un cube trié de dimension 3, un tel hyperplan existe, aussi bien qu'une façon d'y immerger un ensemble

non ordonné de taille n^2. Cependant, il n'en est pas de même quand on recherche dans X + Y + W. Un hyperplan dans X + Y + W peut être défini, par exemple, comme étant l'ensemble $U = \{x_i + y_j + z_k \mid i+j+k = n+2\}$ qui a $O(n^2)$ éléments. Malheureusement nous n'avons pas réussi à y immerger un ensemble *quelconque* de taille $O(n^2)$: en utilisant la construction des sections précédentes, on définit un ensemble D = U à partir des trois vecteurs de la façon suivante :

$x_i = (i-1).M + a_i$

$y_j = (j-1).M + b_j$

$w_i = (k-n).M$.

Par conséquent, l'hyperplan en question sera, lui aussi, de la forme D = A + B. On a vu que la recherche sur un tel ensemble s'exécute en O(n), alors la borne inférieure que nous arrivons à prouver n'est qu'en $\Omega(n)$.

Ainsi, la question ouverte à laquelle on a fait référence est liée à la borne inférieure pour le problème de la recherche dans X + Y + W : cette borne temporelle est-elle en $\Omega(n^2)$? On remarque d'abord qu'en effet l'hyperplan est généré seulement par 3n éléments (la cardinalité de X, Y et W), et puis qu'on a montré comment la recherche dans le cas où m = 4 s'effectue en temps $O(n^2 log n)$, bien que rechercher dans un cube trié de dimension 4 a comme borne inférieure $\Omega(n^3)$, puisque $O(n^3)$ est la taille d'un tel hyperplan - mais qui, dans le cas de X + Y + R + S, n'est généré que par 4n éléments.

2.3 SÉLECTION DU SUIVANT

Soit $z = x_u + y_v$ un élément de X + Y. Le problème de la sélection du suivant dans X + Y est défini comme celui de trouver *suivant*(z), le successeur de z dans l'ordre croissant défini sur X + Y.

La solution de ce problème intervient dans la recherche dans des multi-ensembles, puisqu'on a vu dans la section précédente que l'algorithme findfour(z,X,Y,R,S) est basé sur la capacité de générer les éléments de X + Y (et de R + S) les uns après les autres, dans l'ordre. S'il était possible de trouver rapidement (en temps proportionnel à *log*n) l'élément suivant d'un élément donné, on serait capable de concevoir un algorithme pour la recherche dans six tables, dont le coût serait supérieur seulement par un facteur *log*n à celui pour quatre tables. On gagnerait ainsi sur l'espace supplémentaire requis.

Cependant, nous démontrons que les mêmes bornes s'appliquent aussi à ce problème. La démonstration, comme attendu, repose sur les mêmes principes. Nous introduisons un algorithme linéaire en temps ainsi qu'une borne inférieure linéaire en temps pour le pire cas de ce problème, ce qui établit sa complexité.

2.3.1 BORNE SUPÉRIEURE

On suppose que l'élément $z = x_u + y_v$ est connu ; on veut calculer le successeur de z définit par l'ordre total dans X + Y.

Comme X et Y sont triés en ordre croissant,

$$k > u, q > v \Rightarrow x_k + y_q \geq z \text{ et} \qquad (*)$$

$j < u, m < v \Rightarrow x_j + y_m \leq z.$ (**)

Donc il suffit de rechercher suivant(z) parmi les éléments des sous-matrices :

$B_{i,j} = \{x_i + y_j \mid i > u \text{ et } j \leq v\}$ et (***)

$C_{i,j} = \{x_i + y_j \mid i \leq u \text{ et } j > v\}$ (****)

Algorithme *selectsuiv*(z,X,Y)

```
1    fonction MB(i,j,z)
2        début
3            k := i ; r := j ;
4            tant que (r ≥ 1) et (x_i + y_r ≥ z) faire r := r - 1 ;
5            si r = 0 alors MB := x_i + y_r+1 ; retourner MB ;
6            tant que (k ≤ n) et (x_k + y_r < z) faire k := k + 1 ;
7            si k = n + 1 alors MB := x_i + y_r+1 ; retourner MB ;
8            MB := min {x_i + y_r+1, MB(k,r,z)} ;
9        fin-fonction ;

10   fonction MC(i,j,z)
11       début
12           k := i ; r := j ;
13           tant que (k ≥ 1) et (x_k + y_j ≥ z) faire k := k - 1 ;
14           si k = 0 alors MC := x_k+1 + y_j ; retourner MC;
15           tant que (r ≤ n) et (x_k + y_r < z) faire r := r + 1
16           si r = n + 1 alors MC := x_k+1 + y_j ; retourner MC;
17           MC := min {x_k+1 + y_j, MC(k,r,z)} ;
18       fin-fonction;

19   main ()
20       début
21           suiv(z) := min{MB(u+1,v,z), MC(u,v+1,z)} ;
22       fin.
```

Figure 2.3.1 : *Algorithme selectsuiv*

Lemme 2.3.1 Si $x_i + y_j \geq z$ alors

1.1 la fonction MB(i,j,z) retourne le plus petit élément de la sous-matrice $B_{i,j} = \{x_k + y_r \mid i+1 \leq k \leq n \text{ et } 1 \leq r \leq j\}$ qui est supérieur à z.

1.2 la fonction MC(i,j,z) retourne le plus petit élément de la sous-matrice $C_{i,j} = \{x_k + y_r \mid 1 \leq k \leq i \text{ et } j+1 \leq r \leq n\}$ qui est supérieur à z.

Preuve :

(1.1) On prouve que si MB(k,r,z) retourne le plus petit élément de $B_{k,r}$ qui est supérieur à z pour tout $k > i$ et $r < j$, alors MB(i,j,z) retourne le plus petit élément de $B_{i,j}$ qui est supérieur à z.

On le démontre par récurrence sur le nombre d'appels récursifs de MB, à la **ligne 8**. Si aucun appel à MB(k,r,z) ne se produit dans MB(i,j,z) alors :

a. soit la fonction s'arrête à la **ligne 5** parce que r = 0. Dans ce cas, l'élément minimum $(x_i + y_1)$ de la ligne i de $B_{i,j}$, qui est supérieur à z, aura été calculé à la **ligne 4**. Par conséquent, les propriétés (*) et (**) s'appliquent avec u = i et v = 1, d'où (**) implique que $x_i + y_1$ est l'élément recherché, lequel est retourné à la **ligne 5**.

b. soit la fonction s'arrête à la **ligne 7** parce que k = n + 1. Dans ce cas le premier élément de la colonne r, qui est supérieur à z, aura été calculé à la **ligne 6**. Comme k = n + 1, en fait un tel élément n'existe pas. (*) implique alors que l'élément recherché est $x_i + y_{r+1}$, calculé à la **ligne 4**, lequel est retourné à la **ligne 7**.

Si l'on suppose que l'hypothèse de récurrence est vraie, alors il y aura un appel récursif à la fonction MB à la **ligne 8**. A ce point, les propriétés (*) et (**) s'appliquent pour $x_i + y_{r+1}$ et $x_k + y_r$ calculés, respectivement, aux **lignes 4** et **6**. Par conséquent, l'élément recherché est ou $x_i + y_{r+1}$ ou le plus petit élément de $B_{k,r}$ qui est supérieur à z et, par récurrence, il est calculé à la **ligne 8**.

(1.2) Analogue à 1.1. ♦

Lemme 2.3.2 Soit tmB(i,j) le temps requis pour un appel de la fonction MB(i,j,z), et tmC(i,j) le temps requis pour un appel de la fonction MC(i,j,z). Si $x_i + y_j \geq z$ alors :

2.1 $tmB(i,j,z) \leq n - i + j$.

2.2 $tmC(i,j,z) \leq n - j + i$.

Preuve :

(2.1) On démontre que si $tmB(k,r) \leq n - k + r$ pour tous $k > i$ et $r < j$ alors $tmB(i,j) \leq n - i + j$. On le prouve à nouveau par récurrence sur le nombre d'appels récursifs de la fonction MB à la **ligne 8**.

Si aucun appel récursif ne se produit dans MB(i,j,z) alors :

a. soit la fonction s'arrête à la **ligne 5** parce que r = 0. Dans ce cas tmB(i,j,z) = j.

b. soit la fonction s'arrête à la **ligne 7** parce que k = n + 1. Donc l'instruction de la **ligne 4** a été exécutée au plus (j - 1) fois tandis que l'instruction de la **ligne 6** a été exécutée (n - i) fois. Alors $tmB(i,j) \leq n - i + j$.

On suppose maintenant que l'hypothèse de récurrence est vraie. Alors il y aura un appel récursif à MB(k,r,z) à la **ligne 8**. A ce point l'instruction de la **ligne 4** a été exécutée (j - r) fois tandis que l'instruction de la **ligne 6** a été exécutée au plus k - i fois. D'où $tmB(i,j) \leq j - r + k - i + tmB(k,r) \leq j - r + k - i + n - k + r = n - i + j$.

(2.2) Analogue à 2.1. ♦

Les deux lemmes précédents conduisent au résultat suivant :

Théorème 2.3.3 Etant donnés X, Y et $z = x_u + y_v$, l'algorithme selectsuiv(z,X,Y) calcule l'élément suivant(z) dans X + Y en temps O(n).

Preuve :

La correction de l'algorithme est garantie par le lemme 2.3.1, tandis que sa complexité temporelle se déduit du lemme 2.3.2. ♦

2.3.2 BORNE INFÉRIEURE

Nous montrons maintenant que l'algorithme selectsuiv(z,X,Y) est optimal. En construisant une matrice A = X + Y qui ressemble à celles des sections précédentes, on déduit que dans le pire cas, suivant(z) appartient à l'ensemble U des éléments de l'antidiagonale principale de A. Comme, par construction, l'ordre de ces éléments dépend de l'ordre des éléments de Y, calculer suivant(z) = $min\{U\}$ coûte $\Omega(n)$ temps.

Soit $n \leq ß_i < 2n$, $i = 1,2,\ldots,n$. (1)

On définit X et Y comme suit :

$y_1 = ß_1$ (2)

$$y_i = y_1 + \ldots + y_{i-1} + \beta_i \; ; \; i = 2,\ldots,n \qquad (3)$$

$$x_1 = 0 \qquad (4)$$

$$x_i = y_{n-i+1} + \ldots + y_{n-1} \; ; \; i = 2,\ldots,n \qquad (5)$$

Lemme 2.3.4 Soit $A = X + Y = \{x_i + y_j \mid i,j = 1,2,\ldots,n\}$.

Alors :

4.1 Pour $1 \leq i \leq n-2$,

$a_{i,n-i} < a_{i+1,1}$.

4.2 Pour $2 \leq j \leq n-1$,

$a_{n,j} < a_{n-j+1,j+1}$.

Preuve :

(4.1) $a_{i,n-i} = x_i + y_{n-i} = y_{n-i+1} + \ldots + y_{n-1} + y_{n-i} =$

$= x_{i+1} < x_{i+1} + y_1 = a_{i+1,1}$.

(4.2) $a_{n,j} = x_n + y_j = y_1 + \ldots + y_{n-1} + y_j =$

$= y_1 + \ldots + y_j + y_j + \ldots + y_{n-1} <$

$< y_{j+1} + x_{n-j+1} = a_{n-j+1,j+1}$. ♦

Lemme 2.3.5 Soit U l'ensemble des éléments de l'antidiagonale de A : $U = \{u_j = x_i + y_j \mid i+j = n+1\}$.

Alors :

(5.1) $u_j = y_1 + y_2 + \ldots + y_{n-1} + \beta_j$.

(5.2) $a_{n-1,1} < u_j < a_{n,2}$, pour $1 \leq j \leq n$.

Preuve :

(5.1) $u_j = x_i + y_j = y_{n-i+1} + \ldots + y_{n-1} + y_1 + \ldots + y_{j-1} + \beta_j$.

Comme $i + j = n+1$, alors

$$u_j = y_j + \ldots + y_{n-1} + y_1 + \ldots + y_{j-1} + \beta_j = y_1 + \ldots + y_{n-1} + \beta_j.$$

(5.2) $a_{n-1,1} = y_1 + \ldots + y_{n-1} < u_j$, pour $1 \leq j \leq n$. (par 5.1 et (1))

$a_{n,2} = y_1 + \ldots + y_{n-1} + y_2 = y_1 + \ldots + y_{n-1} + \beta_1 + \beta_2 > u_j$, pour $1 \leq j \leq n$.

(par 5.1 et (1)) ♦

Théorème 2.3.6 La sélection du suivant dans X + Y nécessite $\Omega(n)$ temps dans le pire cas.

Preuve :

Soient X et Y des vecteurs vérifiant les propriétés (1) à (5). Soit $z = x_{n-1} + y_1$. Par les lemmes précédents, suivant(z) appartient à l'ensemble U, d'où suivant(z) = $min\{U\} = min\{\beta_i\} + \Sigma y_j$ (j=1,...,n-1) (lemme 2.3.4). Comme les éléments β_i ne sont pas connus à l'avance, le calcul de $min\{\beta_i\}$ a une borne inférieure en temps de $\Omega(n)$. ♦

Corollaire 2.3.7 La complexité du pire cas du problème de sélection du suivant dans X + Y est $\Theta(n)$ temps.

Preuve :

Elle est immédiate à partir des théorèmes 2.3.3 et 2.3.6. ♦

2.4 UN MOT SUR LE TRI

Le problème du tri des éléments de X + Y, lorsque X et Y sont triés, a été étudié par Harper, Payne, Savage et Straus dans un article assez complet ([Har75]). Le but est de rechercher une façon d'économiser du temps de calcul en utilisant le fait que les vecteurs sont triés.

Soit $W = (w_1, w_2,\ldots, w_r)$ un n-uplet de nombres réels, et soit π la permutation des indices telle que $w_{\pi 1} < w_{\pi 2} < \ldots < w_{\pi r}$, alors W est dit *de type* π. Si les valeurs des w ne sont pas distinctes, W n'a pas de type. Si Ξ est un ensemble de r-uplets, on définit $\tau(\Xi)$ comme le nombre de types distincts des r-uplets de Ξ. Par conséquent si l'on veut trier les éléments de Ξ, la comparaison à faire à un moment donné est basée sur les résultats des comparaisons antérieures. Si deux éléments de Ξ ont des types différents, alors la séquence de comparaisons effectuées ne peut être la même pour ces deux éléments. Par contre, il n'existe que deux résultats possibles pour chaque comparaison, donc on a au maximum 2^c différentes séquences de c comparaisons. D'où $2^c \geq \tau(\Xi)$ et $c \geq log\tau(\Xi)$.

Comme conséquence on a le théorème ci-dessous, issu de la théorie de l'information et montré dans ([Har75]) :

Théorème 2.4.1 $\Omega(log\tau(\Xi))$ est une borne inférieure en temps pour le problème du tri des éléments de Ξ. ♦

Par exemple, si Φ est l'ensemble de tous les n-uplets réels, alors $\tau(\Phi) = n!$, et ce théorème implique que $\Omega(n\log n)$ est une borne inférieure en temps pour le problème du tri. Dans ce cas classique il existe des algorithmes qui atteignent cette borne et donc la borne déduite du théorème est atteinte ([Knu72]).

D'autre part Fredman ([Fr75]) a prouvé que cette borne est bonne quand $\tau(\Xi)$ est grand, en utilisant un résultat similaire pour le problème de la *détermination successive des composantes* d'un r-uplet d'entiers positifs $U=(u_1, u_2,\ldots, u_r)$. Soit S un ensemble de r-uplets d'entiers positifs. Etant donné le vecteur U de S, un tel problème est celui de déterminer les

composantes de U dans l'ordre $u_1, u_2, \ldots$. La détermination de u_k s'effectue en posant des questions de la forme "est-ce que $u_k \leq j$?", pour j donné. La restriction fondamentale est que, avant de questionner sur u_k, on doit avoir déjà déterminé $u_1, \ldots, u_{k-1}$.

Fredman a démontré le théorème suivant ([Fr75]) :

Théorème 2.4.2 La détermination successive des composantes d'un vecteur de S peut être résolue avec $O(log|S| + 2r)$ questions. ◆

Si l'on remarque que la détermination successive des composantes d'un vecteur $(b_1, \ldots, b_r)$, où b_k est défini comme étant la cardinalité de l'ensemble $\{w_j \mid j \leq k \text{ et } w_j \leq w_k\}$, revient au même que trier W par insertion (b_k est le rang de w_k), alors on a immédiatement ([Fr75]) :

Théorème 2.4.3 $O(log\tau(\Xi) + 2r)$ est une borne supérieure en temps pour le problème du tri des éléments de Ξ. ◆

2.4.1 MATRICES TRIÉES

Si l'on applique le merge-sort pour trier les éléments d'une matrice triée, on a un algorithme en temps $O(n^2 log n)$. La question qui se pose est de savoir si cette borne peut être améliorée. Pour répondre à cette question, Harper et al. ([Har75]) ont prouvé que l'ordre existant sur chaque ligne et chaque colonne d'une matrice triée ne peut être utilisé pour accélérer le tri de la totalité des éléments :

Théorème 2.4.4 Le problème du tri pour les matrices triées est en $\Omega(n^2 log n)$.

Preuve :

L'affectation d'un entier distinct 1, ..., n^2 à chaque position d'une matrice nXn, telle que les entrées de chaque ligne et de chaque colonne soient en ordre croissant, induit une matrice nXn triée. Le nombre de telles affectations différentes qui induisent des matrices triées est donné par

$$\frac{n^2! \cdot \left(\prod_{i=1}^{n-1} i!\right)^2}{\left(\prod_{i=1}^{2n-1} i!\right)}$$

Ceci est exactement la quantité de différents ordres possibles dans l'ensemble des matrices triées. Par conséquent on a

$$\log(n^2!) + 2\sum_{i=1}^{n-1} \log(i!) - \sum_{i=1}^{2n-1} \log(i!) \cong n^2 \log n$$

comme borne temporelle inférieure pour le problème du tri des éléments d'une matrice triée. ♦

Une conséquence triviale du théorème 2.4.4 est la suivante :

Corollaire 2.4.5 La complexité du problème du tri pour les matrices triées est $\Theta(n^2 log n)$. ♦

2.4.2 X + Y

Si l'ordre sur les lignes et les colonnes d'une matrice triée n'est pas utile pour accélérer le tri de ses éléments, il n'en est pas de même pour

X + Y. En effet Harper et al. ([Har75]) prouvent que ($log(\tau(X+Y))$ = $O(n log n)$), i.e., que le nombre d'ordres distincts possibles dans une structure du type X + Y est borné par $O(n^n)$ - noter que cette borne est très inférieure à celle d'un ensemble normal à n^2 éléments ($O(n^2!)$). En utilisant ce résultat, Fredman ([Fr75]) a montré que :

Corollaire 2.4.6 Le problème du tri de X + Y peut être résolu avec $O(n^2)$ comparaisons.

Preuve :

Immédiate à partir du théorème 2.4.3, en sachant que ($log(\tau(X+Y))$ = $O(n log n)$). ♦

Malheureusement, ces démonstrations sont non constructives. On n'a toujours pas proposé d'algorithme qui atteigne cette borne. Jusqu'à présent le meilleur algorithme pour trier les éléments de X + Y est l'algorithme sort(X,Y) introduit dans la section 2.2.3 (Figure 2.2.4). Sa complexité étant en $O(n^2 log n)$.

2.5 PARALLÉLISATION DE LA RECHERCHE

Cette section est dédiée à l'étude d'algorithmes parallèles pour la recherche dans X + Y. En supposant les éléments de X et de Y déjà triés, nous proposons plusieurs façons de résoudre ce problème en parallèle. Nous introduisons plusieurs algorithmes, chacun d'entre eux mettant l'accent sur des aspects particuliers de la conception d'algorithmes parallèles, sur les modèles CREW PRAM et EREW PRAM avec $p \leq n$ processeurs en mode synchrone.

Quand ce sera nécessaire, on supposera que n est multiple de p. On rappelle que la complexité séquentielle de ce problème est $\Theta(n)$.

2.5.1 PAR PARTITIONNEMENT DE L'ESPACE

La parallélisation directe de l'algorithme find(z,X,Y) de la section 2.2 consiste en le partitionnement de l'espace de recherche parmi les processeurs. Sans perte de généralité, on suppose $p = q^2$ et n multiple de q. On numérote les processeurs avec une paire d'indices (i,j), $1 \leq i,j \leq q$. On partage X et Y, respectivement, en q sous-vecteurs BX et BY de taille n/q et on affecte BX_i et BY_j à $PE_{(i,j)}$.

Algorithme *Part*(z,X,Y)

pas **1** pour (i,j) = (1,1) jusqu'à (q,q) faire en parallèle
find(z,BX_i,BY_j)
pas **2** fin-pour

Figure 2.5.1 : *Algorithme par partitionnement de l'espace*

La taille des sous-vecteurs est n/q, d'où le temps total d'exécution de cet algorithme est $O(n/q) = O(n/\sqrt{p})$ quel que soit le modèle de calcul parallèle choisi, avec une accéleration de $O(\sqrt{p})$. Nous obtenons ainsi une borne supérieure pour les parallélisations proposées par la suite.

2.5.2 PAR FUSION

La solution que nous proposons ensuite est basée sur la fusion de deux listes triées. En effet, on veut répondre à la question de l'existence de u et de v tels que $x_u + y_v = z$, ce qui est équivalent à $x_u = z - y_v$. Par conséquent,

rechercher z dans X + Y est équivalent à détecter des éléments communs à X et z - Y.

Algorithme *Fusion*(z,X,Y)

pas **1** pour i = 1 jusqu'à n faire en parallèle
Y[i] := z - Y[i] ;

pas **2** C := Fusion-en-Parallèle(X,Y)

pas **3** pour i = 1 jusqu'à n faire en parallèle
si ((C[i] - C[i-1] = 0) et (C[i] et C[i-1]) n'ont pas le même vecteur comme origine))
alors arrêter : la recherche a abouti;

pas **4** arrêter : la recherche n'a pas abouti.

Figure 2.5.2 : *Algorithme par fusion*

Analyse :

Le **pas 1** est en O(n/p) quel que soit le modèle. Pour le **pas 2** on peut utiliser la fusion en parallèle proposée par Akl et Santoro ([Akl87]) pour le modèle EREW ou celle de Kruskal ([Kru83]) pour le CREW. La première solution coûte O(n/p + *log*p*log*n) tandis que la seconde coûte O(n/p + *log*(n/p)). Enfin, le troisième pas s'effectue en temps O(n/p) dans n'importe quel modèle.

Théorème 2.5.1 La complexité de l'algorithme Fusion est O(n/p + *log*p*log*n) sur une EREW. Il est optimal pour $p \leq n/log^2 n$. ♦

Théorème 2.5.2 La complexité de l'algorithme Fusion est O(n/p + *log*(n/p)) sur une CREW. Il est optimal pour p ≤ n/*log*n. ♦

2.5.3 BASÉE SUR LA RECHERCHE DANS Y

Une autre façon de résoudre ce problème en parallèle consiste à rechercher chaque élément de X dans (z-Y). En séquentiel la complexité d'une telle approche est en O(|X|*log*|Y|) si l'on effectue une recherche dichotomique dans (z-Y). L'algorithme parallèle pour la recherche dans X + Y, basé sur la recherche dans Y, est très simple. On suppose que Y est rangé dans la mémoire partagée tandis que X est partitionné en p blocs consécutifs (BX_1,..., BX_p), de taille n/p, un bloc par processeur. Donc il suffit que chaque processeur recherche son propre bloc d'éléments dans Y.

Algorithm *Rec_Y*(z,X,Y)

pas **1** pour i = 1 jusqu'à n/p faire en parallèle
rechercher bx_i dans Y.

Figure 2.5.5 : *Algorithme de la recherche dans Y*

Analyse :

Il est clair que la complexité de cet algorithme dépend de la stratégie de recherche choisie. La recherche dichotomique sur une CREW signifie l'exécution de n/p recherches de coût en O(*log*n) chacune, donnant une complexité totale en O((n/p)*log*n), qui n'est pas optimal puisque l'accélération est égale à O(p/*log*n).

Pour le modèle EREW on peut pipeliner les recherches dichotomiques. Au premier cycle PE_1 compare son élément courant à la médiane de Y. Au cycle suivant il va le comparer ailleurs et cette position (la médiane de Y) sera libre pour que PE_2 exécute sa comparaison, et ainsi de suite. Le premier processeur aura fini après avoir exécuté (logn) pas (la taille de Y), mais il ne pourra recommencer la recherche avant que PE_p n'ait exécuté sa première comparaison, i.e., avant (p+1) pas. Comme chaque processeur doit rechercher n/p éléments, le nombre de cycles nécessaires au dernier processeur sera de l'ordre de ((n/p).p + log n) = O(n), ce qui conduit à une accélération de O(logn) par rapport à la recherche dichotomique séquentielle de X dans Y, mais qui n'est pas de tout intéressant parce que O(n) est le coût de l'algorithme find, en séquentiel.

2.5.4 PAR BLOCS

Pour ce nouvel algorithme on suppose que X et Y sont rangés dans une mémoire partagée du type CREW - X en ordre décroissant et Y en ordre croissant - chacun partitionné en p blocs (BX_i et BY_j) de taille n. L'idée est de construire des vecteurs (DBX et DBY) avec les derniers éléments des blocs. Remarquer que DBX sera en ordre décroissant, tandis que DBY sera en ordre croissant. A l'aide d'une recherche, dans DBY, des éléments z-DBX on définit quelles sont les sous-matrices (BX_i + BY_j) qui sont censées contenir z (voir figure 2.5.6).

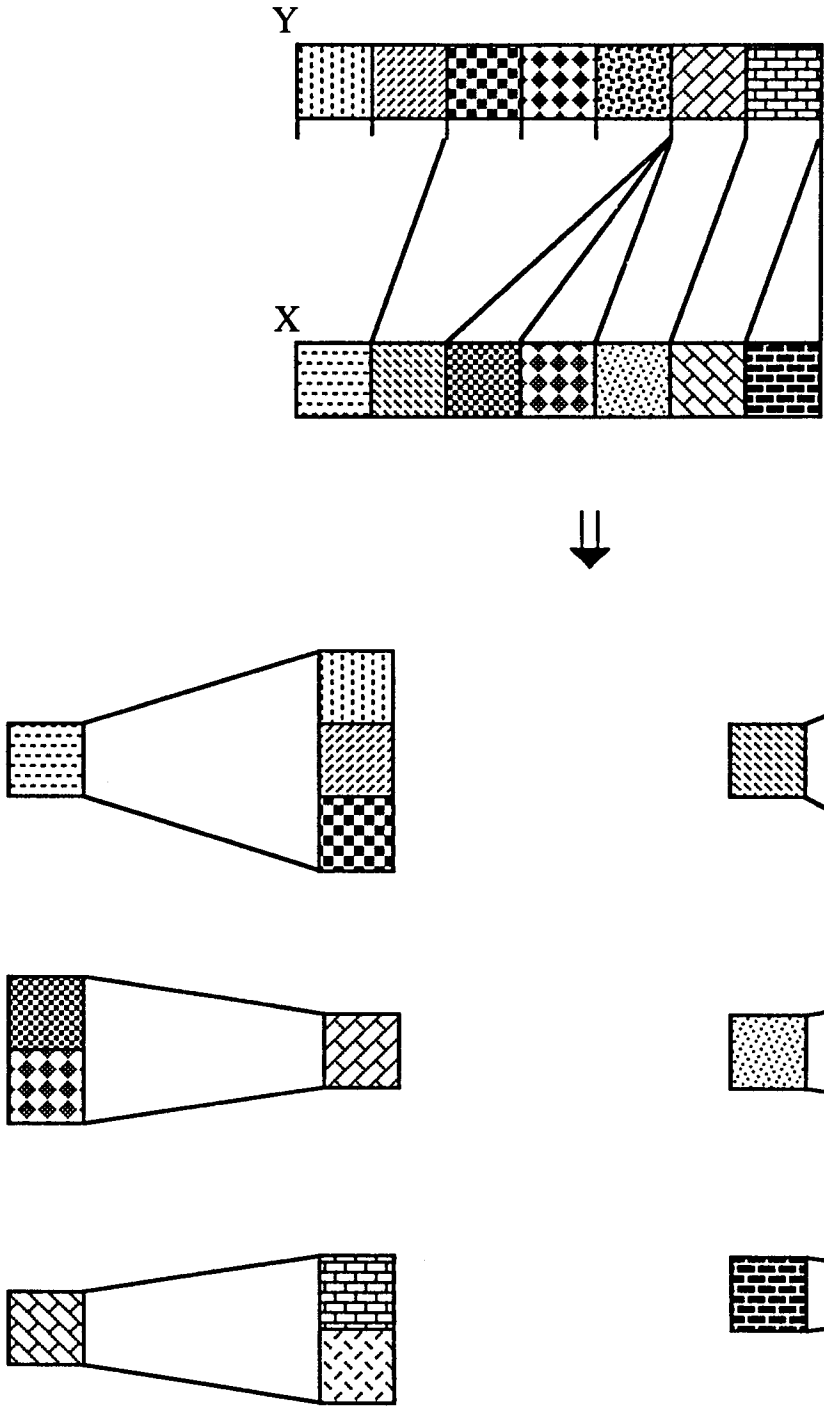

Figure 2.5.6 : *Détermination des sous-matrices qui peuvent contenir z*

Les deux vecteurs étant triés, le nombre maximum de sous-matrices trouvées est borné :

Lemme 2.5.3 Soit *nsm* le nombre de sous-matrices qui peuvent contenir z dans la structure de blocs définie ci-dessus, alors $nsm < 2p$. ♦

Preuve :

Soient DBX et DBY indexés de 1 à p. Soit r_i la position résultante de la recherche de l'élément dbx_i dans DBY :

$$DBY[r_i] > z\text{-}dbx_i > DBY[r_i + 1].$$

Alors, par rapport à BX_1, z peut se trouver dans les sous-matrices générées par la somme cartésienne de BX_1 et $(r_1 + 1)$ blocs de Y, à savoir $BY_1, \ldots, BY_{r_1}$ et BY_{r_1+1}. Pour BX_2, z peut se trouver dans les sous-matrices générées par la somme cartésienne de BX_2 et $BY_{r_1+1}, \ldots, BY_{r_2}$ et BY_{r_2+1}. En général, pour chaque PE_i, les blocs à rechercher sont ceux définis par la somme cartésienne de BX_i et $BY_{r_{i-1}+1}$ à BY_{r_i+1}.

Il est facile de voir qu'en général, pour chaque bloc BX_i, $1<i\leq p$, le nombre de sous-matrices à rechercher est $(r_i - r_{i-1} + 1)$. Ainsi

$$nsm = r_1 + 1 + \sum_{2}^{p} (r_i - r_{i-1} + 1) = p + r_p \leq 2p.$$

Cependant, pour que $nsm = 2p$ il faudrait que $r_p = p$, mais dans ce cas le nombre de sous-matrices à rechercher par le bloc BX_p est $r_p - r_{p-1}$ (et non pas $r_p - r_{p-1} + 1$), puisque $DBY[r_p]$ est le dernier élément de DBY. Par conséquent $nsm < 2p$. ♦

Une conséquence du lemme précédent est que la tâche de recherche dans les sous-matrices $(BX_i + BY_j)$ peut bien être partagée entre les

processeurs, où chacun s'occupe de deux sous-matrices au maximum. La difficulté, pourtant, est liée à la distribution de l'information concernant l'identité des blocs qu'un processeur doit rechercher : faire parvenir cette connaissance aux processeurs coûte, malheureusement, O(log p), puisque son implantation dépend de fonctions de diffusion basées sur la détermination de sommes partielles (*cf.* chapitre 1 et ([Kru85])).

Algorithme *Blocs*(z,X,Y)

pas **1** faire en parallèle
partitionner X en p blocs (BX_i) de n/p éléments chacun.
partitionner Y en p blocs (BY_j) de n/p éléments chacun.

pas **2** pour i = 1 jusqu'à p faire en parallèle
générer une liste DBX avec les éléments X[i.n/p].
générer une liste DBY avec les éléments Y[i.n/p].

pas **3** pour i = 1 jusqu'à p faire en parallèle
chaque PE_i trouve $1 \leq r_i \leq p$ tel que
$dby_{r_i} < z - dbx_i < dby_{r_{i+1}}$.

pas **4** {distribution de blocs entre les processeurs}

a pour i = 1 jusqu'à p faire en parallèle

b premierBY(i) := $r_{i-1}+1$

c nombreDeBlocs(i) := si ($i \neq p$) alors $r_i - r_{i-1}+1$

d sinon $r_i - r_{i-1}$

e si PE_i contient des blocs à rechercher alors cond(i) := vrai

f Concentrate(cond(i),G(i)) dans G' = (nDB,k,prY),
où G(i) = (nombreDeBlocs(i),i,premierY(i))

g SomPar(nDB(i),nombreDeBlocsTot(i))

h pour i = 1 jusqu'à p faire en parallèle

i sommeSansMoi(i) := nombreDeBlocsTot(i) - nDB + 1

j Generalize(destination(i),G(i))
dans G' = (sSM,k,prY),
où destination(i) = nombreDeBlocsTot(i),
et G(i) = (sommeSansMoi(i),i,PrY(i))

```
k       pour i = 1 jusqu'à p faire en parallèle
l               s := i - sSM + prY

pas 5   pour i = 1 jusqu'à p faire en parallèle
                chaque PE exécute find(z,BX_k,BY_s),
                pour k et s définis au pas 4.
```

Figure 2.5.6 : *Algorithme par blocs*

Analyse :

Les **pas 1** et **2** sont parallélisés de façon optimale. Au **pas 3**, chaque processeur peut exécuter une recherche dichotomique dans DBY pour essayer de trouver (z-dbx_i). Cependant, ce pas est équivalent à la fusion des deux listes et l'algorithme de Kruskal ([Kru83]) le résout en temps O(*loglog*p).

Pour distribuer les blocs restants entre les processeurs au **pas 4**, chaque PE_i calcule aux lignes **a** à **d** quels sont les blocs de DBY qu'il doit rechercher avec BX_i. S'il a au moins une tâche à exécuter, alors il est sélectionné à la ligne **e**.

L'appel de la procédure *Concentrate* (*cf.* section 1.6.2) à la ligne **f**, concentre le nombre de blocs à rechercher, l'indice i du bloc de DBX associé et l'indice du premier bloc de DBY à former paire avec BX_i, sur des variables nDB, k et prY, respectivement. Ensuite, à l'aide des *sommes partielles* (*cf.* section 1.6.2) de la ligne **g**, le nombre de processeurs à allouer pour la recherche des blocs est déterminé sur la variable nombreDeBlocsTot.

Si l'on suppose que les blocs sont en ce moment sur le processeur i et que chaque PE_i simule deux processeurs avec adresses 2i-1 et 2i, alors ces blocs seront distribués entre les processeurs sSM=nombreDeBlocsTot(i-1)+1 à nombreDeBlocsTot(i) (lignes **h** et **i**). Ensuite, le **pas 4** s'achève par l'affectation définitive, à la ligne **j**, des sous-matrices (BX_k + $BY_{i\text{-}sSM+prY}$) aux PE_i (simulés), qui ainsi pourront exécuter le **pas 5**.

A cause des opérations Concentrate, SomPar et Generalize, le coût du **pas 4** est O(*log*p) (*cf.* section 1.6.2), tandis que celui du **pas 5** est, selon le lemme 2.5.3, la complexité d'au plus deux exécutions de l'algorithme find, i.e., O(n/p).

Théorème 2.5.4 La complexité de l'algorithme Blocs est O(n/p + *log*p) sur une CREW. Il est optimal pour $p \leq n/\log n$. ◆

CHAPITRE III. UNE APPLICATION : LE PROBLEME DU SAC-A-DOS

Dans ce chapitre, on étudie le problème du Sac-à-Dos dans sa version liée à la cryptologie. Il s'agit ici du problème de décision où, donnés des entiers positifs

$$W = (w_1, w_2, \ldots, w_n) \text{ et } M,$$

on veut savoir s'il existe un n-uplet binaire

$$C^* = (c_1, c_2, \ldots, c_n)$$

solution de l'équation

$$\sum w_i c_i = M.$$

Ce problème appartient à la classe des problèmes NP-Complets ([Kar72]) et, sauf si NP = P, sa complexité est exponentielle en n. L'espace original de recherche a 2^n valeurs possibles, ce qui veut dire que, dans le pire cas, une recherche exhaustive coûte $O(2^n)$ temps pour trouver une solution.

Bien qu'il existe des cas où la taille d'une instance du problème du Sac-à-Dos est mesurée par le nombre de bits nécessaires pour le décrire ([Ste88]), nous gardons pour ce travail la valeur n définie comme la cardinalité du vecteur d'entrées W ([Hor74]) ou comme la taille de la sortie C^* ([Sch81]).

La résolution du problème du Sac-à-Dos peut être utilisée pour l'étude de plusieurs problèmes en théorie des nombres et, à cause de sa complexité exponentielle, des cryptosystèmes à clé publique sont basés sur lui. Pour

cette raison, beaucoup d'efforts ont été fournis pour le développement de techniques qui peuvent mener à des algorithmes portables, avec une complexité temporelle raisonnable ([Sch81]).

La résolution du Sac-à-Dos basée sur la technique de Branch and Bound (*cf.* chapitre suivant) fournit de bons résultats pour des instances particulières du problème, mais sa complexité asymptotique (dans le pire cas) reste toujours en $O(2^n)$. Horowitz et Sahni ([Hor74]) ont proposé l'*algorithme des deux-listes* qui a permis de réduire la complexité de ce problème à $O(n.2^{n/2})$ temps, à l'aide de $O(2^{n/2})$ cellules de mémoire supplémentaires. Il est clair que l'espace mémoire requis par cette solution est hors de portée pour des instances où *n* est grand. Pourtant il s'agissait de la première solution sous-exponentielle à un problème NP-Complet. Puis, utilisant l'algorithme des deux-listes, Schroeppel et Shamir ([Sch81]) ont réussi à réduire le besoin en espace mémoire avec *l'algorithme des deux-listes quatre-tables*. Leur algorithme ne nécessite que $O(2^{n/4})$ espaces pour résoudre le Sac-à-Dos en temps encore proportionnel à $O(n.2^{n/2})$.

Si l'on utilise le parallélisme pour essayer d'accélérer la résolution de problèmes NP-Complets, on sait qu'il y aura un facteur exponentiel quelque part dans le modèle proposé ; soit le temps, soit l'espace ou encore le nombre de processeurs requis (sauf si P = NP). Bien que la discussion sur la validité de solutions parallèles qui ont besoin d'un nombre exponentiel de processeurs soit encore loin d'être finie, il n'en reste pas moins certain que l'étude d'algorithmes parallèles pour ce type de problèmes aide à éclaircir des points qui ne sont pas toujours clairs en séquentiel.

Karnin ([Ka84]) a introduit la notion d'un compromis *Temps-Mémoire-Processeur* pour des problèmes NP-Complets sur une PRAM. Il

définit le *Matériel* (H) demandé par un algorithme parallèle comme étant l'Espace (S) plus le nombre de Processeurs (P) utilisés par un tel algorithme. Plus spécifiquement pour le problème du Sac-à-Dos, il propose une parallélisation de l'algorithme des deux-listes quatre-tables telle que le temps reste le même ($O(n.2^{n/2})$), mais où le besoin en Matériel (espace plus processeurs) est égal à $O(2^{n/6})$. Le compromis temps/matériel de son algorithme est donc $T.H = O(n.2^{2n/3})$.

Il est évident que ces progrès faits sur la complexité spatiale ont été importants. En ce qui concerne le stockage des données, des problèmes de taille déjà assez importante peuvent être attaqués. A l'aide de l'algorithme des deux-listes quatre-tables, des problèmes où n vaut par exemple 100 sont à la portée des machines existantes (en supposant de l'ordre d'un gigamot de mémoire vive disponible), puisque l'espace nécessaire sera proportionel à 2^{25} cellules mémoire . Par contre $T = O(2^{50})$ réprésente environ 42 ans sur une machine où le temps de cycle est de l'ordre de 1µs, i.e., le problème reste tout de même intraitable.

Conservant une approche théorique, nous avons étudié des parallélisations de ce problème sur différents modèles de machines parallèles et avec différentes hypothèses sur le Matériel disponible. Dans un premier temps nous avons proposé ([Fer88]) les algorithmes des *trois-tables equilibrées* et *trois-tables non-équilibrées*, basés sur la recherche dans X + Y (*cf.* le chapitre 2) qui atteignent les mêmes compromis T.H que ceux des algorithmes de ([Sch81]) et ([Ka84]), en utilisant une PRAM. Bien que ces algorithmes ne produisent pas d'accélération optimale pour le problème du Sac-à-Dos, ils ont l'avantage de demander strictement moins de $O(2^{n/2})$ espaces mémoires, pour un temps strictement inférieur à $O(2^{n/2})$.

Puis, toujours sur une PRAM, nous avons conçu l'algorithme *d'une-liste en parallèle*, avec une accélération optimale pour le Sac-à-Dos ([Fer89b]). Un tel algorithme donne lieu à une efficacité égale à 1 pour n'importe quel nombre de processeurs dans l'intervalle $[1,O(2^{n/2})]$. De plus son compromis $T.H = O(n.2^{n/2})$, quand on a $O(2^{n/2})$ processeurs disponibles, est le meilleur proposé jusqu'à présent.

Enfin, nous avons étudié la parallélisation de l'algorithme d'une-liste sur des architectures à mémoire distribuée ([Fer89a]), motivés par le fait que, si d'une part on a du mal à imaginer des machines à mémoire partagée comportant quelques centaines de processeurs, d'une autre on sait que des réseaux de Transputers ou la Connection Machine peuvent coupler des dizaines de milliers de processeurs, communiquant par échange de messages. Ainsi, nous avons proposé des bornes temporelles pour le Sac-à-Dos sur plusieurs architectures à mémoire distribuée.

3.1 DEUX ALGORITHMES SÉQUENTIELS

Dans cette section on introduit les algorithmes séquentiels qui ont réduit, d'abord, la complexité temporelle et ensuite la complexité spatiale du problème du Sac-à-Dos.

Le lecteur attentif remarquera que l'algorithme find(z,X,Y) et l'algorithme findfour(z,X,Y,R,S) du chapitre précédent sont en fait des réécritures, respectivement, de l'algorithme des deux-listes et de l'algorithme des deux-listes quatre-tables décrits par la suite. On verra aussi que les résultats présentés au **Chapitre 2** s'appliquent parfaitement à la résolution de ce problème.

3.1.1 L'ALGORITHME DES DEUX-LISTES

Etant donnés $W = (w_1, w_2, \ldots, w_n)$ et M, l'algorithme des deux-listes commence par diviser le vecteur W en *deux* nouveaux vecteurs : $W_1 = (w_1, \ldots, w_{n/2})$ et $W_2 = (w_{n/2+1}, \ldots, w_n)$. Ensuite tous les sous-ensembles de W_1 sont générés et leurs sommes triées en ordre croissant dans une liste A. Dans une liste B, on garde les sommes correspondants aux sous-ensembles de W_2, triées en ordre décroissant. Nous poserons par la suite $N = O(2^{n/2})$.

L'algorithme proposé par Horowitz et Sahni ([Hor74]) est le suivant :

Algorithme *deux-listes*

1 Générer les listes A et B ;
2 Trier A en ordre croissant ;
3 Trier B en ordre décroissant ;
4 i:= 1 ; j := 1 ;
5 Si (A[i] + B[j] = M) alors arrêter => il existe une solution ;
6 Si (A[i] + B[j] < M) alors i := i + 1 sinon j := j + 1 ;
7 Si (i > N) ou (j > N) alors arrêter => il n'existe pas de solution ;
8 Aller en **5**.

Figure 3.1.1 : *L'algorithme des deux-listes*

Analyse :

Chaque somme de A (respectivement, B) est obtenue à partir d'une combinaison d'éléments de W_1 (respectivement, W_2). Par conséquent la solution finale C* est la concaténation d'une combinaison qui appartient à la liste A avec une combinaison qui appartient à la liste B. La complexité de cet algorithme est $O(N.\log N)$ temps, puisque O(N) est la taille des deux listes.

D'autre part, deux listes de taille O(N) doivent être stockées dans la mémoire, ce qui montre que O(N) est la complexité en espace.

Théorème 3.1.1 ([Hor74]) Le problème du Sac-à-Dos peut être résolu en O(N.*log*N) temps et O(N) espaces. ♦

3.1.2 L'ALGORITHME DES QUATRE-TABLES

Schroeppel et Shamir ([Sch81]) ont remarqué que l'algorithme des deux-listes accède aux éléments des listes A et B d'une façon séquentielle et dans l'ordre, ce qui implique qu'on n'a pas besoin de garder toutes les combinaisons possibles simultanément en mémoire. Il est uniquement nécessaire d'engendrer ces combinaisons rapidement de façon ordonnée.

Pour cela, étant donnés $W = (w_1, w_2, \ldots, w_n)$ et M, le vecteur W est divisé en *quatre* nouveaux vecteurs : W_1 à W_4. Ensuite tous les sous-ensembles de W_i sont engendrés et leurs sommes triées dans une table T_i (i=1, ..., 4). Enfin, pour construire les deux listes A et B à partir de ces quatre tables, on utilise deux tas Q_1 et Q_2 :

Algorithme *deux-listes quatre-tables*

1 Diviser W en W_1, W_2, W_3 et W_4 ;
2 Engendrer les tables T_1, T_2, T_3 et T_4 ;
3 Trier T_2 dans un ordre croissant ;
4 Trier T_4 dans un ordre décroissant ;
5 Insérer dans le tas Q_1 toutes les paires (r, *premier*(T_2)) pour $r \in T_1$;
6 Insérer dans le tas Q_2 toutes les paires (s, *premier*(T_4)) pour $s \in T_3$.
7 Tant que Q_1 et Q_2 ne sont pas vides faire
8 (t_1,t_2) := paire avec la plus petite somme dans Q_1;
9 (t_3,t_4) := paire avec la plus grande somme dans Q_2;
10 $S := ((t_1 + t_2) + (t_3 + t_4))$;

```
11        Si S = M alors répondre "il existe une solution" et arrêter;
12        Si S < M alors
13              Supprimer (t1,t2) de Q1;
14              Si le successeur t'2 de t2 dans T2 est défini
15                    Alors insérer (t1,t'2) dans Q1;
16        Sinon
17              Supprimer (t3,t4) de Q2;
18              Si le successeur t'4 de t4 en T4 est défini
19                    Alors insérer (t3,t'4) dans Q2 ;
20        Fin-si
21  Fin-tant-que
22  Répondre "sans solution" et arrêter.
```

Figure 3.1.2 : *Algorithme des deux-listes quatre-tables*

Analyse :

L'espace demandé par cet algorithme est celui de la taille de tables T_i, i.e., $O(\sqrt{N})$. En ce qui concerne sa complexité temporelle, elle reste inchangée par rapport à celle de l'algorithme des deux-listes : $O(N.logN)$.

Théorème 3.1.2 ([Sch81]) Le problème du Sac-à-Dos peut être résolu en $O(N.logN)$ temps et $O(\sqrt{N})$ espaces. ♦

3.2 PARALLÉLISATION AVEC SIX TABLES

Si l'on se ramène aux résultats du chapitre précédent et, en particulier, au théorème 2.3.6, on voit qu'on ne peut espérer améliorer l'algorithme des deux-listes quatre-tables au moyen d'une technique quelconque de sélection. La borne prouvée pour la sélection du successeur d'un élément donné montre que, en utilisant deux listes, l'espace minimum nécessaire est celui proposé dans l'algorithme des deux-listes quatre-tables.

Karnin ([Ka84]) a conçu un algorithme parallèle pour des systèmes multiprocesseurs, qui calcule le successeur d'un élément donné dans X + Y en temps proportionnel au logarithme de la taille des listes, en utilisant des processeurs très simples couplés en arbre binaire. De cette façon, et à travers un découpage non équilibré du vecteur W, il a été capable de résoudre le problème du Sac-à-Dos avec $O(N^{1/3})$ cellules mémoire, $O(N^{1/3})$ processeurs et en temps $O(N.logN)$.

Remarquons que, dans ce cas, l'utilisation du parallélisme ne permet pas de réduire le temps d'exécution de l'algorithme, mais son occupation mémoire. Dans ce qu'on appelle *l'algorithme des deux-listes six-tables*, les éléments de chacune des deux listes sont engendrés à partir de trois tables. Le vecteur W est tout d'abord divisé en six sous-vecteurs, W_1 à W_6, de même taille. Ensuite les sommes des sous-ensembles de ces six sous-vecteurs sont triées en ordre croissant dans six tables T_1 à T_6, respectivement. On décrit ici la génération des éléments de la liste A, la procédure pour la liste B étant similaire.

3.2.1 GÉNÉRATION DES TABLES ET DES LISTES

L'idée de base est celle de l'algorithme des deux-listes quatre-tables, i.e., on a un tas Q_1 d'où les éléments de A seront construits d'une façon séquentielle et ordonnée. La différence réside dans le fait que maintenant la table T_2 est aussi définie comme la somme cartésienne de deux vecteurs ($T_2 = F + G$). On initialise donc le tas Q_1 avec $(x,premier(F + G))$, pour tout $x \in T_1$ et quand une paire (t_1,t_2) est supprimée dans la **ligne 13** de l'algorithme des deux-listes quatre-tables, il faut calculer le successeur t'_2 de t_2 dans $F + G$.

En d'autres termes on veut calculer

$$min\{t'_i \mid t'_i > t_2\}, t \in F + G. \quad (1)$$

Ce qui revient à calculer

$$min\{f_i + g_j \mid f_i + g_j > t_2\}. \quad (2)$$

Qui, à son tour est équivalent à

$$min\{f_i - (t_2 - g_j) \mid f_i - (t_2 - g_j) > 0\}. \quad (3)$$

Lorsque cet élément est trouvé, il est additioné à t_2, qui est connu, pour obtenir le prochain élément de la table T_2. On suppose que les tables F et G sont stockées de façon répartie dans les processeurs. Le contenu d'un processeur i est noté CP_i. Le calcul de l'expression (3) ci-dessus est fait selon les pas suivants :

1) Les processeurs qui ont un élément g_i calculent $(t_2 - g_i)$.

2) Tous les processeurs exécutent la fusion des listes $(f_1, \ldots, f_{N^{1/3}})$ et $((t_2 - g_1), \ldots, (t_2 - g_{N^{1/3}}))$ dans une nouvelle liste, en ordre croissant, conservée sous la forme d'une liste linéaire de $(2.N^{1/3})$ éléments. Dans le cas où des valeurs de $(t_2 - G)$ et F sont égales, celle de $(t_2 - G)$ doit précéder celle de F. Enfin, les processeurs se souviennent de la liste d'origine correspondant à leur nouveau contenu.

3) Si CP_i vient de la liste F **et** CP_{i-1} vient de la liste $(t_2 - G)$ **et** $CP_i - CP_{i-1} > 0$ alors $CP_i := CP_i - CP_{i-1}$ sinon $CP_i := \infty$.

4) Calculer $min\{CP_i\}$ en utilisant les processeurs couplés en topologie d'arbre binaire.

Analyse :

L'ordre croissant de la liste fusionnée garantit que l'élément recherché se trouve bien entre ceux calculés au **pas 3**, puisque, si CP_i contient une valeur de F, alors une soustraction de CP_j, pour $j > i$, donne un résultat

négatif, ce qui n'a pas d'intérêt. D'autre part, pour j = 0, 1, ..., i - 2, l'évaluation de $CP_i - CP_j$ ne sert à rien parce que CP_{i-1} est plus grand que tous ses prédécesseurs dans la liste, d'où $CP_i - CP_{i-1} \leq CP_i - CP_j$, j = 0, 1, ..., i - 2.

Le temps consommé est celui de la traversé des listes dans l'algorithme des deux-listes multiplié par le temps dépensé pour trouver le successeur d'un élément donné, suivant les **pas de 1 à 4**, ci-dessus. Les **pas 1 et 3** s'exécutent en temps constant. La fusion au **pas 2** peut être accomplie en temps proportionnel au logarithme de la taille des listes à fusionner ([Knu72]). On peut procéder de même pour la traversée de l'arbre des processeurs au **pas 4**.

L'algorithme des deux-listes six-tables requiert ainsi $O(N^{1/3})$ cellules mémoire, pour la manipulation des tas, placés dans une mémoire centrale du type PRAM (même si le modèle n'a pas été explicitement défini dans l'article original, on peut le supposer EREW), plus $O(N^{1/3})$ processeurs, chacun avec un espace de rangement constant.

Théorème 3.2.1 ([Ka84]) Sur une EREW avec $O(N^{1/3})$ processeurs le problème du Sac-à-Dos peut être résolu en temps $O(N.logN)$ et espace $O(N^{1/3})$. ♦

On rappelle que Karnin ([Ka84]) a introduit la notion de Matériel (H) comme une mesure pour des algorithmes parallèles. Le matériel requis par un algorithme englobe le nombre de cellules mémoire (S) et le nombre de processeurs (P) qu'il utilise. Sous la notation O, on a H = max{S, P} et, par conséquent :

Corollaire 3.2.2 Le compromis temps/matériel de l'algorithme des deux-listes est $T.H = O(N^2 log N)$. ♦

Corollaire 3.2.3 Le compromis temps/matériel de l'algorithme des deux-listes quatre-tables est $T.H = O(N^{3/2} log N)$. ♦

Corollaire 3.2.4 Le compromis temps/matériel de l'algorithme des deux-listes six-tables est $T.H = O(N^{4/3} log N)$. ♦

3.3 LES TROIS TABLES

Depuis l'apparition de l'algorithme des deux-listes, l'espace requis pour résoudre le problème du Sac-à-Dos a été reduit. Toutefois, sa complexité en temps reste inchangée tant en séquentiel qu'en parallèle, même dans le cas où un nombre exponentiel de processeurs a été employé. Nous utilisons dans la suite les résultats obtenus dans le domaine des matrices triées et X + Y pour proposer de nouveaux algorithmes, séquentiels et parallèles, qui sont meilleurs que ceux décrits jusqu'ici.

La structure de l'algorithme des deux-listes est essentiellement séquentielle, puisque l'itération (i+1) dépend entièrement de l'itération i. Si l'on veut proposer des algorithmes parallèles efficaces pour le problème du Sac-à-Dos, il faut en changer. Le premier algorithme décrit par la suite s'avère être moins intéressant que l'algorithme des deux-listes, en séquentiel. Par contre sa parallélisation est optimale, conduisant à des compromis temps/matériel aussi bons que ceux des sections précédentes, avec l'avantage d'accélérer la résolution du problème en utilisant moins de O(N) cellules mémoire ou processeurs.

3.3.1 LES TROIS TABLES ÉQUILIBRÉES

Les techniques de base utilisées dans cet algorithme, qu'on appelle l'*algorithme des trois-tables* pour des raisons qui deviendront claires, sont celles de la recherche dans des multi-ensembles, étudiées au chapitre 2. Le vecteur W est divisé maintenant en trois sous-vecteurs et leurs sommes générées et triées dans trois tables X, Y et Z. A ce point, l'application immédiate de l'algorithme findthree(M,X,Y,Z), de la section 2.2.3, donne une solution en temps $O(N^{4/3})$ pour ce problème.

L'algorithme des trois-tables peut être parallélisé de deux façons différentes. L'algorithme des trois-tables équilibrées divise le vecteur W en trois sous-vecteurs de même taille n/3, tandis que l'algorithme des trois-tables non-équilibrées le divise en deux sous-vecteurs de taille 3n/8, plus un sous-vecteur contenant les n/4 éléments restants.

Algorithme *Trois-Tables Equilibrées*

1 Diviser W en $W_1 = (w_1, \ldots, w_{n/3})$, $W_2 = (w_{(n/3)+1}, \ldots, w_{2n/3})$ et $W_3 = (w_{(2n/3)+1}, \ldots, w_n)$;
2 Générer une liste X à partir de W_1 ;
3 Générer une liste Y à partir de W_2 ;
4 Générer une liste Z à partir de W_3 ;
5 En parallèle faire
6 Trier X ;
7 En parallèle faire
8 Trier Y ;
9 Pour i = 1 jusqu'à |Z| faire en parallèle
10 Rechercher z_i de Z dans X + Y

Figure 3.3.1 : *L'algorithme des trois-tables équilibrées*

Analyse :

Chaque liste contient $2^{n/3}$ éléments ; par conséquent il faut $O(N^{2/3})$ positions dans la mémoire centrale pour garder les listes X et Y. Prennant l'EREW comme modèle, on utilise $P = O(N^{2/3})$ processeurs, où chacun s'occupe de rechercher un élément de Z. Les P processeurs sont numérotés de 0 à P - 1.

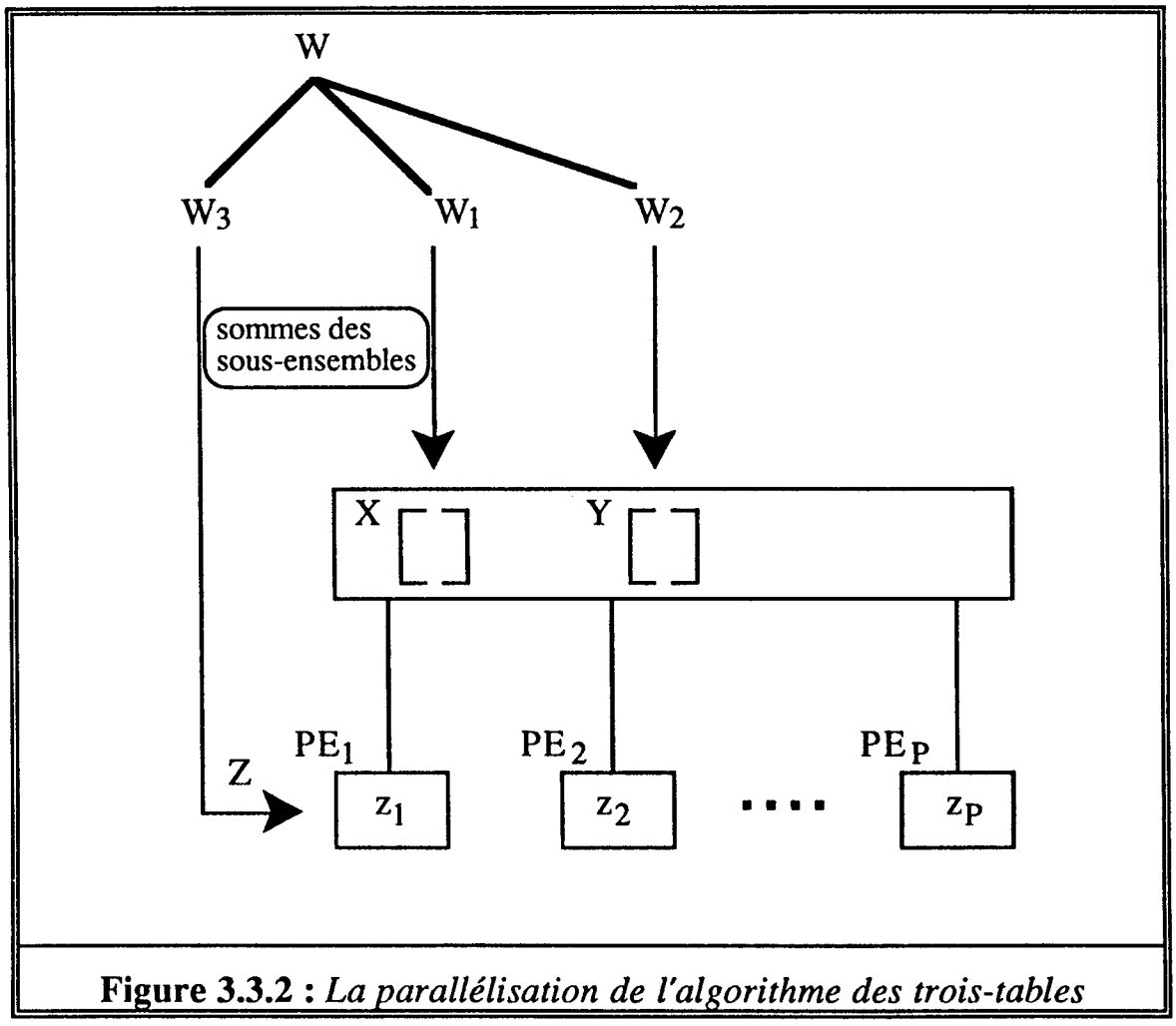

Figure 3.3.2 : *La parallélisation de l'algorithme des trois-tables*

La génération

Pour engendrer les éléments de la liste X, chaque processeur prend son propre numéro en représentation binaire et l'interprète comme une combinaison des éléments de W_1, un 1 sur le i-ème bit représentant la présence du i-ème élément. Ainsi chaque processeur génère la somme des éléments sélectionnés par son numéro et l'écrit sur la PRAM, dans une position de la mémoire qui dépend, elle aussi, du numéro du processeur : de cette façon on n'a pas besoin d'écritures concurrentes. Le procédé pour la génération de la liste Y est analogue.

La somme des éléments d'un sous-vecteur s'exécute en temps $O(log N)$ et telle est la complexité de cette étape.

Le tri des tables

On a le même nombre de processeurs que d'éléments à trier. Le tri des tables s'effectue en temps $O(log N)$ si l'on utilise l'algorithme optimal de tri proposé par Cole ([Col86]).

La recherche de la solution

De façon analogue à ce qui a déjà été fait pour l'étape de génération, chaque processeur engendre un élément z_i de Z selon son propre numéro et applique l'algorithme find(z_i, X,Y) pour rechercher z_i dans X + Y. Comme on travaille sur une EREW, les processeurs ne peuvent commencer l'exécution de l'algorithme **find** tous au même moment. Ce problème est facilement résolu à l'aide d'un pipeline des recherches (*cf.* section 2.5.3). Ainsi le premier processeur aura fini après $O(N^{2/3})$ pas (la complexité de l'algorithme find), tandis que le dernier processeur aura fini après $O(2N^{2/3})$ pas.

Corollaire 3.3.1 Sur une EREW avec $O(N^{2/3})$ processeurs, la complexité de l'algorithme des trois-tables équilibrées est $O(N^{2/3})$ temps et $O(N^{2/3})$ espaces. ♦

Théorème 3.3.2 Le compromis temps/matériel de l'algorithme des trois-tables équilibrées est $T.H = O(N^{4/3})$. ♦

Cet algorithme atteint un meilleur compromis T.H que l'algorithme des deux-listes six-tables (par un facteur (logN)). Si d'une part il utilise d'avantage d'espace mémoire et de processeurs, il faut remarquer qu'il est quand même $O(N^{1/3}log$N) fois plus rapide.

3.3.2 LE DESÉQUILIBRAGE DES TABLES

Il est clair qu'on n'est pas obligé de diviser W en trois sous-vecteurs de même taille. Selon les moyens disponibles en ce qui concerne le nombre de processeurs, taille mémoire et temps limite, on peut trouver des compromis entre ces trois variables suivant la façon de découper W. La solution du problème quand W est découpé en trois sous-vecteurs de tailles différentes va impliquer l'utilisation d'un nombre de processeurs moindre que le nombre d'éléments à manipuler sur la PRAM. Par conséquent, le tri en parallèle des tables stockées dans la mémoire centrale, exécuté sur les **lignes 5 à 8** de la version équilibrée de l'algorithme des trois-tables, doit être repris, puisque l'algorithme de tri proposé par Cole ([Col88]) ne s'applique plus.

Or, pour montrer une version non-équilibrée de l'algorithme des trois-listes qui égale le compromis temps/matériel de l'algorithme des deux-listes quatre-tables, il nous suffit de diviser le vecteur W en trois sous-

vecteurs, dont deux avec 3n/8 éléments et dont le troisième contient les 2n/8 éléments restants, et d'engendrer les sommes des sous-ensembles des deux premiers dans deux tables X et Y qui seront stockées dans la mémoire partagée. Chacune a $O(N^{3/4})$ éléments ce qui sera la complexité en espace de cet algorithme. Les $O(\sqrt{N})$ éléments de la dernière table (Z) sont répartis sur les P processeurs disponibles, pour que les processeurs les recherchent dans X + Y.

En fait, on s'intéresse à aller plus vite que O(N) en temps, avec le moins de Matériel possible. Une borne supérieure pour le nombre de processeurs utilisés pourrait être l'espace mémoire $O(N^{3/4})$, mais il est évident qu'il n'est pas nécessaire de travailler avec plus de $O(\sqrt{N})$ processeurs (la taille de Z).

Algorithme *Trois-Tables Non-Equilibrées*

1 Diviser W en $W_1 = (w_1, \ldots, w_{3n/8})$, $W_2 = (w_{(3n/8)+1}, \ldots, w_{3n/4})$ et $W_3 = (w_{(3n/4)}, \ldots, w_n)$;
2 Générer une liste X à partir de W_1 ;
3 Générer une liste Y à partir de W_2 ;
4 Générer une liste Z à partir de W_3 ;
5 En parallèle faire
6 Trier X ;
7 En parallèle faire
8 Trier Y ;
9 Pour i = 1 jusqu'à |Z| faire en parallèle
10 Rechercher z_i de Z dans X + Y ;

Figure 3.3.3 : *L'algorithme des trois-tables non-équilibrées*

Analyse :

Tout se passe comme dans la version précédente, sauf que maintenant on n'a que $P \leq O(\sqrt{N})$ processeurs disponibles. Par conséquent, la génération des listes s'exécute dans un temps $O(N^{3/4})/P$ suivant la même stratégie utilisée pour l'autre version.

Pour trier en parallèle une liste de taille k avec un nombre sous-linéaire de processeurs, Akl a introduit une version parallèle optimale du Quicksort ([Akl84a]), en temps $O(k^{\beta} log k)$ sur une EREW-PRAM avec $k^{1-\beta}$ processeurs, $0<\beta<1$. Ainsi le tri des listes X et Y, sur les **lignes 5 à 8**, peut être exécuté en $O(N^{3/4} log N)/P$ temps.

La recherche de la solution coûte $O(N^{3/4})$ par élément recherché. En pipelinant les recherches, sa complexité sera

$$(P + O(N^{3/4}) * O(\sqrt{N})/P) = O(N^{5/4})/P.$$

Lemme 3.3.4 Sur une EREW avec $P \leq O(\sqrt{N})$ processeurs, la complexité de l'algorithme des trois-tables non-équilibrées est $O(N^{5/4})/P$ temps et $O(N^{3/4})$ espaces. ♦

Théorème 3.3.5 Le problème du Sac-à-Dos peut être résolu sur une EREW avec $O(\sqrt{N})$ processeurs en temps $O(N^{3/4})$, avec $O(N^{3/4})$ cellules mémoire.

Preuve :

Immédiate à partir du lemme précédent pour $P = O(\sqrt{N})$. ♦

Corollaire 3.3.6 Le compromis temps/matériel de l'algorithme des trois-tables non-équilibrées avec $O(\sqrt{N})$ processeurs est $T.H = O(N^{3/2})$.

♦

A cause de sa complexité temporelle en $O(N^{5/4})/P$, il est intéressant de noter qu'il faut qu'on ait plus de $O(N^{1/4})$ processeurs pour que cet algorithme soit avantageux, i.e., ait une complexité temporelle inférieure à O(N). Le meilleur compromis T.H obtenu par cet algorithme, et qui égale celui du deux-listes quatre-tables, est atteint quand $P = O(\sqrt{N})$.

Les deux algorithmes présentés dans cette section égalent les meilleures bornes connues pour le compromis temps/matériel, en ayant l'avantage de requérir moins de O(N) en temps.

3.4 L'ALGORITHME D'UNE-LISTE

Dans le chapitre précédent on a montré plusieurs parallélisations possibles du problème de la recherche dans X + Y. Dans cette section et celle qui suit nous montrons comment des algorithmes décrits dans la section 2.5 peuvent être utilisés pour la parallélisation du problème du Sac-à-Dos. La parallélisation par le partitionnement de l'espace a été décrite dans ([Cos88]) mais ne sera pas abordée ici. On s'intéresse plutôt à la parallélisation basée sur la recherche dans Y et à la parallélisation par fusion.

On appelle *algorithme d'une-liste* celui qui découle de la parallélisation basée sur la recherche dans Y. On suppose que l'on veuille utiliser une PRAM du type CREW, où O(N) cellules mémoire sont disponibles. On suppose aussi que les $P \leq O(N)$ processeurs ont une capacité de mémoire constante. Sous ces hypothèses l'algorithme d'une-liste produit une

accélération optimale, de l'ordre de P, pour tout P dans l'intervalle [1, O(N)]. Dans le cas où il est implanté avec un seul processeur, les complexités temporelle et spatiale de l'algorithme d'une-liste égalent celles de l'algorithme des deux-listes. D'autre part, il résoud le problème du Sac-à-Dos à n variables en un temps proportionel à n, à l'aide de O(N) processeurs en parallèle. Ceci conduit à un compromis T.H = O(N*log*N), le meilleur jusqu'à présent.

Par une meilleure clarté de l'exposition, nous présenterons d'abord la version séquentielle de l'algorithme. Ensuite nous montrerons le cas où P = N, pour enfin présenter la description de l'algorithme quand $1 < P < O(N)$.

Algorithme *Une-liste*

1 Diviser W en $W_1 = (w_1, \ldots, w_{n/2})$ et $W_2 = (w_{(n/2)+1}, \ldots, w_n)$;
2 Générer une liste A à partir de W_1 ;
3 Trier A ;
4 Faire
5 Générer c, une nouvelle combinaison de W_2 et calculer sa somme c_s ;
6 Rechercher dans A un a_s tel que $a_s + c_s = M$;
7 Si la recherche a abouti, alors arrêter : la solution est la concaténation de a et c.
8 Jusqu'à ce que toutes les combinaisons de W_2 aient été testées ;
9 Arrêter : il n'existe pas de solution

Figure 3.4.1 : *L'algorithme d'une liste en séquentiel*

Analyse :

A nouveau, on peut relever les trois principaux pas de l'algorithme d'une-liste :

1. La génération de A.
2. Le tri de A.
3. La recherche dichotomique dans A.

Les deux premiers pas peuvent être réunis et résolus d'un seul coup en O(N) ([Cos88]). D'abord on trie le vecteur W_1 en O(n*log*n). Puis on commence avec la liste (naturellement) triée [0, w_1]. Ensuite, par l'addition de w_2 à chaque élément de la première liste, on construit une autre liste déjà triée [w_2, $w_1 + w_2$]. La fusion de ces deux listes ordonnées est faite en temps linéaire conduisant à la liste [0, w_1, w_2, $w_1 + w_2$]. Enfin on additione w_3 à la liste résultant, on les fusionne en une seule et on repète ce procédé jusqu'à ce que toutes les sommes des sous-ensembles de W_1 aient été générées.

La durée des **pas 1 et 2** est, par conséquent, égale à $\sum O(2^i) = O(N)$, $0 \leq i \leq n/2-1$.

Chaque recherche dichotomique dans la liste A a une durée égale à O(*log*N). Dans le pire cas, on doit tester les sommes de tous les sous-ensembles de W_2, ce qui implique que le troisième pas est en O(N*log*N), amenant la complexité totale de l'algorithme à T = O(N*log*N), avec le stockage d'une liste de N éléments, i.e., S = O(N).

3.4.1 PARALLÉLISATION AVEC (P = N) PROCESSEURS

Pour simplifier les notations, nous introduisons d'abord la version parallèle de l'algorithme d'une liste où (P = N). La parallélisation que nous proposons est évidente. Pourtant elle dépend strictement d'algorithmes de coût optimal, linéaires et sous-linéaires, pour le problème du tri et de la sélection ([Akl84a], [Akl84b]). On saute l'étude des deux premiers pas puisqu'elle se ramène à celle de la section précédente (3.3) où l'algorithme des trois-tables a été décrit. Pour la recherche, **au pas 3**, chaque processeur génère une somme à partir de W_2, selon son numéro, et exécute indépendamment la recherche dichotomique dans A. Cette étape est accomplie, aussi bien que les deux premières, en temps O(*log*N). Ainsi :

Théorème 3.4.1 La complexité de l'algorithme d'une-liste sur une CREW avec (P = N) processeurs est O(*log*N) temps et O(N) espaces. ♦

Corollaire 3.4.2 L'accélération de l'algorithme d'une-liste avec (P = N) processeurs est optimale.

Preuve :

Comme O(N*log*N) est le coût de l'algorithme séquentiel actuellement connu de plus faible complexité temporelle pour le problème du Sac-à-Dos (voir section 3.1), l'accélération de l'algorithme d'une-liste est égale au nombre de processeurs utilisés. ♦

En ce qui concerne le compromis temps/matériel, cet algorithme est le plus performant jusqu'à présent, puisque :

Corollaire 3.4.3 Sur une CREW avec N processeurs, le compromis temps/matériel de l'algorithme d'une-liste est T.H = O(N*log*N). ♦

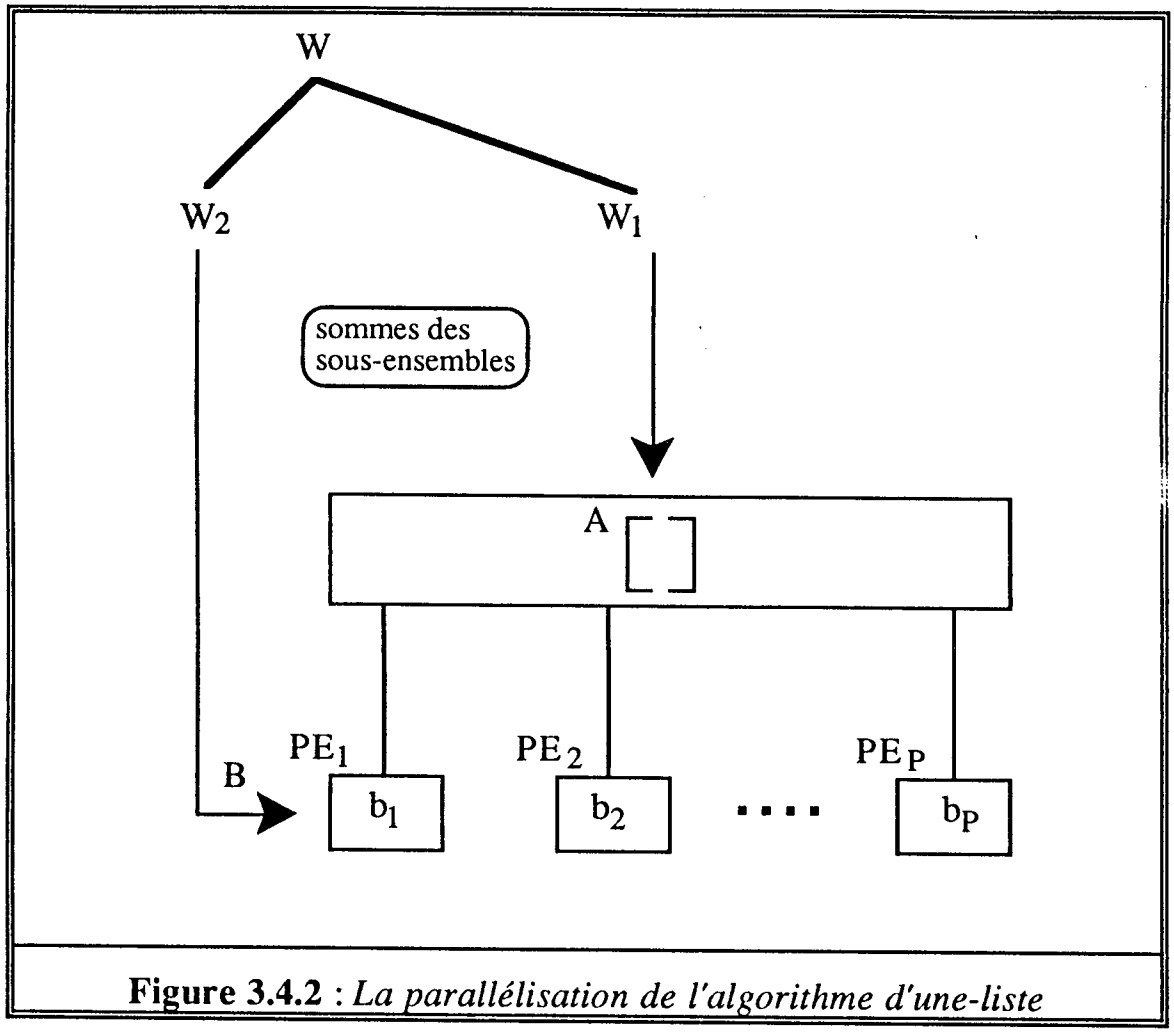

Figure 3.4.2 : *La parallélisation de l'algorithme d'une-liste*

3.4.2 CAS OU P < O(N)

Pour étendre ce résultat au cas où $P = N^{1-\varepsilon}$, $0 < \varepsilon < 1$, i.e., le cas où le nombre d'éléments de la liste A est plus grand que le nombre de processeurs, l'algorithme parallèle reste basé sur l'algorithme d'une-liste divisé en les mêmes trois étapes. En fait, cette parallélisation est similaire au cas de l'algorithme des trois tables non-équilibrées.

Chaque processeur engendre $N/N^{1-\varepsilon}$ éléments. Par conséquent on a N^{ε} éléments générés au coût de O(*log*N) chacun, ce qui donne un coût total de $O(N^{\varepsilon}\textit{log}N)$ pour le **pas 1**, de génération des éléments de la liste A. Au **pas**

2 on utilise à nouveau l'algorithme optimal de tri proposé par Akl ([Akl85]) et le tri aussi est accompli en $O(N^{\varepsilon} log N)$. La troisième étape de l'algorithme est celle où les processeurs génèrent les sommes des sous-ensembles de W_2, l'une après l'autre, et recherchent une solution dans la liste A à l'aide de la recherche dichotomique. Il existe $N^{1-\varepsilon}$ processeurs pour se partager cette tâche, chacun exécutant $N/N^{1-\varepsilon}$ fois le calcul d'une somme d'une nouvelle combinaison de W_2, aussi bien que la recherche d'une solution dans A. Comme chaque recherche dans A coûte $O(log N)$, la complexité de ce pas est aussi en $O(N/N^{1-\varepsilon}).O(log N) = O(N^{\varepsilon} log N)$.

L'algorithme parallèle est le suivant :

Algorithme *Une-liste en parallèle*

Pas 1 : {Génération de A}
J := numéro_de_processeur * N^{ε} ;
Tant que J < (numéro_de_processeur + 1) * N^{ε} faire
Engendrer a, la combinaison de W_1 induite par
la représentation binaire de J ;
Ecrire a_S, sa somme respective, dans la position J de la liste A ;
J := J + 1.

Pas 2 : {Tri de A}
Exécuter le tri parallèle de ([Akl85]) pour trier la liste A en $O(N^{\varepsilon} log N)$.

Pas 3 : {Recherche dichotomique dans A}
J := numéro_de_processeur * N^{ε} ;
Faire
Engendrer c, la combinaison de W_2 induite par la
réprésentation binaire de J;
Calculer sa somme respective c_S ;
Exécuter la recherche dichotomique dans la liste A pour trouver un
$a_S \in A$
tel que $c_S + a_S = M$;

Si la recherche aboutit, alors arrêter :
la solution est la concaténation de a et c ;
J := J + 1
Jusqu'à J = (numéro_de_processeur + 1) * N^{ε} ;
Arrêter : il n'existe pas de solution

Figure 3.4.3 : *L'algorithme d'une-liste en parallèle*

Théorème 3.4.4 La complexité de l'algorithme d'une-liste sur une CREW avec ($P = N^{1-\varepsilon}$) processeurs, $0<\varepsilon<1$, est $O(N^{\varepsilon} log N)$ temps et O(N) espaces. ♦

Corollaire 3.4.5 Sur une CREW avec $N^{1-\varepsilon}$ processeurs, l'accélération de l'algorithme d'une-liste en parallèle est optimale.

Preuve :
Voir corollaire 3.4.2. ♦

Corollaire 3.4.6 Sur une CREW avec $N^{1-\varepsilon}$ processeurs, le compromis temps/matériel de l'algorithme d'une-liste est $T.H = O(N^{1+\varepsilon} log N)$. ♦

3.4.3 REMARQUES

Karnin ([Ka84]) a obtenu plusieurs compromis du type temps/processeurs/mémoire pour le problème du Sac-à-Dos. Ils sont montrés sur la figure ci-dessous, où H représente les besoins en Matériel (nombre de processeurs, P, plus nombre de cellules mémoire, S). Les facteurs linéaires en n ont été négligés :

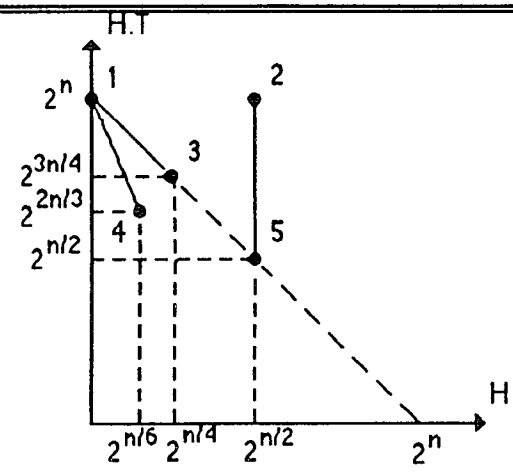

Figure 3.4.4 : *Schéma Matériel X Compromis temps/matériel*

Le point 1 correspond à la recherche exhaustive de l'espace solution, où H = O(1) et T = $O(2^n)$. Le point 2 représente l'algorithme des deux-listes en séquentiel. Le segment solide entre les points 1 et 3 est obtenu en utilisant l'algorithme des deux-listes quatre-tables. Le segment entre le point 4 et le point 1 est obtenu avec l'algorithme proposé par Karnin. Enfin, l'algorithme d'une-liste et sa version en parallèle, que nous venons de présenter, sont représentés par le segment qui lie le point 2 au point 5.

Pour préciser l'interdépendance entre le Temps et le Matériel pour ce problème, nous montrons un schéma Temps *versus* Matériel avec les algorithmes pour le Sac-à-Dos qui ont été l'objet de référence jusqu'ici dans le texte :

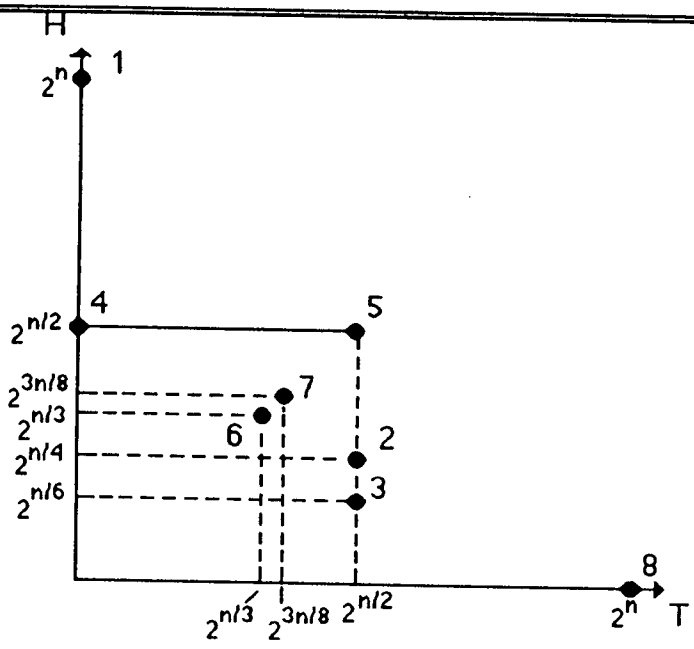

Figure 3.4.5 : *Schéma Temps X Matériel*

Le point 8 représente la recherche exhaustive, tandis que le point 1 représente sa parallélisation triviale en utilisant 2^n processeurs. Le point 2 représente l'algorithme des deux-listes quatre-tables et le 3 est obtenu avec l'algorithme des deux-listes six-tables. Les points 6 et 7 représentent respectivement les compromis atteints par l'algorithme des trois-tables, équilibrés et non-équilibrées. Les points 4 et 5 et le segment qui les lie ensemble représentent l'algorithme d'une-liste. On note que le point 5 correspond aussi à l'algorithme des deux-listes.

Finalement, on aborde le compromis temps/processeurs pour les algorithmes parallèles qui ont été conçus pour le problème du Sac-à-Dos, à l'aide de la figure 3.4.6 :

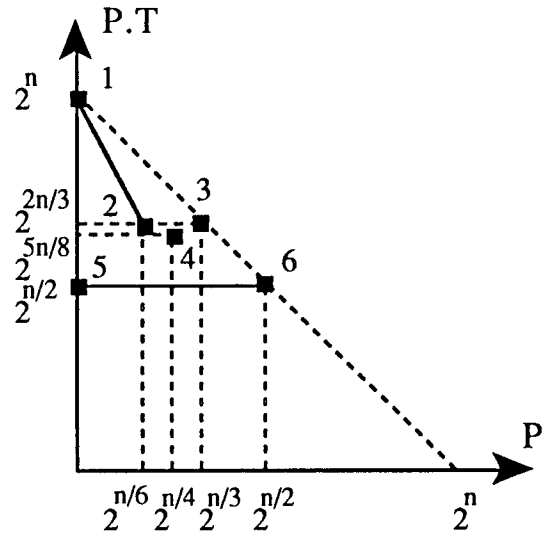

Figure 3.4.6 : *Schéma Processeurs X Compromis temps/processeurs*

L'algorithme des deux-listes six-tables est représenté par le segment qui lie les points 1 et 2. L'algorithme des trois-tables équilibrées est représenté par le point 3, tandis que sa version non-équilibrée est représentée par le point 4. L'algorithme d'une-liste en parallèle est représenté par le segment liant le point 5 au point 6.

3.5 SOLUTIONS DISTRIBUÉES

La réduction de l'espace requis pour la résolution du problème du Sac-à-Dos, apportée par l'algorithme des deux-listes quatre-tables, a permis le traitement de quelques instances auparavant intraitables. La capacité actuelle des mémoires permet de stocker les informations d'instances où n peut prendre des valeurs jusqu'à 100, bien que $T = O(2^{50})$ corresponde à environ 42 années de temps de calcul sur un ordinateur dont le temps de cycle est de 1µs.

Schroeppel et Shamir ([Sch81]) ont prétendu que des instances où n = 100 pourraient être résolues en deux semaines, à l'aide de quelques 1000 processeurs exécutant l'algorithme des deux-listes quatre-tables en parallèle. Cette remarque est basée sur la parallélisation optimale de leur algorithme : 2^{50} opérations sur une machine à temps de cycle de 2^{-20} secondes et une accélération optimale de 2^{10}.

Toutefois, on ne connaît pas de parallélisation optimale d'un tel algorithme sur des machines à mémoire distribuée. Il est vrai que l'on peut toujours utiliser la parallélisation par partitionnement de l'espace, proposée pour l'algorithme find à la section 2.5.1. Pourtant, bien qu'elle puisse être implantée sur n'importe quelle topologie, son accélération n'est proportionnelle qu'à la racine carrée du nombre de processeurs utilisés. Si l'on refait le calcul de Schroeppel et Shamir, on tombe sur une durée de 16 mois pour résoudre une instance à 100 variables, ce qui n'est pas du tout raisonnable.

D'autre part, il est vraiment difficile d'imaginer un ordinateur qui soit capable de coordonner quelques milliers de processeurs qui accèdent concurremment à une PRAM. Ainsi, une voie qui apparaît prometteuse est d'étudier la parallélisation d'un tel problème sur des machines à mémoire distribuée, où les processeurs peuvent être couplés selon différentes topologies et où ils communiquent par envoi de messages. Ainsi, nous avons introduit ([Fer89a]) des solutions implantables sur de telles machines, que nous décrivons par la suite.

3.5.1 RÉDUCTION AU PROBLEME DU TRI

L'approche qui se montre la plus pertinente est celle de la parallélisation de l'algorithme des deux-listes basée sur la fusion des deux listes, comme nous l'avons décrit en 2.5.2 pour l'algorithme **find.**

L'algorithme, dans une première version, est présenté ci-dessous.

Algorithme *Naïf*

1 Diviser W en deux sous-vecteurs W_1 et W_2 ;
2 Générer toutes les sommes des sous-ensembles de
W_1 et W_2 en deux listes A et B ;
3 Modifier la liste B, en faisant B := M - B ;
4 Trier les listes A et B en éliminant les doubles ;
5 Fusionner les deux listes dans une liste C ;
6 Vérifier s'il existe **k** tel que C[k] - C[k-1] = 0 ; le cas échéant
une solution a été trouvée, sinon il n'existe pas de solution.

Figure 3.5.1 : *Algorithme naïf*

La complexité temporelle de cet algorithme est O(N*log*N), avec un espace mémoire en O(N). De plus, à la suite d'une petite modification introduite au niveau du tri des listes A et B, l'algorithme naïf est transformé en un algorithme de tri, tout simplement :

Algorithme *Sac_tri(n, W, M)*

1 Diviser W dans deux sous-vecteurs W_1 et W_2 ;
2 Générer toutes les sommes des sous-ensembles de W_1 et W_2
en deux listes A et B ;
3 Modifier la liste B, en faisant B := M - B ;
4 Mettre les listes A et B dans une liste C ;

5 Trier la liste C ;
6 Si, pendant le tri, on trouve i et j tels que
C[i] = C[j] **et**
C[i] et C[j] ne sont pas originaires de la même liste (A ou B)
7 Alors une solution a été trouvée
8 Sinon il n'existe pas de solution.

Figure 3.5.2 : *Algorithme de tri pour résoudre le Sac-à-Dos*

Une conséquence de cet algorithme est la possibilité de résoudre le problème du Sac-à-Dos à l'aide de n'importe quelle procédure efficace pour le tri. Comme le tri est un des problèmes les plus étudiés en Informatique, on peut en déduire des bornes supérieures pour nombre d'architectures à multiprocesseurs.

Théorème 3.5.1 Quelle que soit l'architecture utilisée, la borne temporelle supérieure pour le problème du Sac-à-Dos est inférieure ou égale à celle du problème du tri. ♦

Plusieurs algorithmes parallèles pour le tri ont été proposés durant les trois dernières décennies. La discussion de ces algorithmes dépasse les limites des sujets qu'on veut aborder, d'autant plus que Knuth ([Knu72]) et Akl ([Akl85]) l'ont déjà fait de façon remarquable. Toutes les bornes qu'on propose par la suite pour le Sac-à-Dos sont des conséquences de résultats prouvés pour le tri.

3.5.2 LES PARALLÉLISATIONS SELON LES ARCHITECTURES

On rappelle qu'on s'intéresse à la résolution du problème du Sac-à-Dos à n variables, avec P processeurs couplés en architectures à mémoire

distribuée. De telles architectures sont simples, modulaires et permettent qu'un grand nombre de processeurs soient couplés.

Dans le cas où P < N, chaque processeur possède une mémoire locale (LS) pour stocker une sous-séquence de N/P entiers, au lieu d'un seul élément. Le pas de comparaison-échange dans l'algorithme de tri à N processeurs est substitué par un pas de fusion-séparation, où deux séquences d'entiers sont fusionnées et l'un des deux processeurs concernés garde la partie où se trouvent les plus petits éléments de la liste résultante, tandis que l'autre en garde la partie des plus grands.

Nous formulons une seule hypothèse de terminaison pour les algorithmes distribués que nous présentons dans la suite : une fois terminé l'algorithme de tri, une simple opération de rapatriement des données (du type *all-to-one* ([Jo89])) route une solution (on n'en nécessite qu'une) au processeur chargé des communications avec l'hôte. Cette opération prend O(d) pas, où d est le diamètre de l'architecture utilisée. Comme $\Omega(d)$ est une borne inférieure naturelle pour le tri ([Akl85]), l'*overhead* introduit par notre hypothèse de terminaison disparaît sous la notation O.

Réseaux linéaires

Il s'agit ici de la plus simple des architectures basées sur l'échange de messages entre processeurs. Il n'existe pas de problème majeur pour coupler un grand nombre de processeurs ensemble. L'inconvénient de sa simplicité est la faible accélération fournie pour la plupart des applications, ce qui inclut le tri.

Des résultats montrés dans ([Akl85]) on peut déduire :

Lemme 3.5.2 L'accélération obtenue sur un réseau linéaire à P processeurs pour le problème du tri de r éléments est $min\{O(log r), O(P)\}$.

♦

D'où nous prouvons :

Théorème 3.5.3 La parallélisation du problème du Sac-à-Dos sur un réseau linéaire de P processeurs produit une accélération optimale lorsque $P \leq n/2$. ♦

Bien que ce théorème porte sur l'optimalité de l'algorithme Sac_tri que nous venons d'introduire, il montre qu'en fait, sur un réseau linéaire il n'est pas intéressant de résoudre le Sac-à-Dos en utilisant cet algorithme, puisque:

Corollaire 3.5.4 Soient, sur un réseau linéaire de P processeurs, Acc_{qt} l'accélération obtenue par la parallélisation de l'algorithme des deux-listes quatre-tables et Acc_{st} l'accélération obtenue par la meilleure parallélisation de l'algorithme Sac_tri.

Alors $P > log^2 N$ implique $Acc_{qt} > Acc_{st}$.

Preuve :

La parallélisation triviale de l'algorithme des deux-listes quatre-tables - basée sur la parallélisation de l'algorithme find par le partitionnement de l'espace de recherche (*cf.* 2.5.1) -, donne une accélération de l'ordre de la racine carrée de P. Par conséquent, pour $P > log^2 N$ on a $Acc_{qt} > O(log N)$, et dans ce cas $Acc_{st} = O(log N)$, par le lemme 3.5.2. ♦

Grilles

Les grilles à deux dimensions de ($\sqrt{P}$ x $\sqrt{P}$) processeurs gardent encore beaucoup de la simplicité des réseaux linéaires, puisque, même si elles ont une dimension en plus, leurs connexions sont régulières et leur degré constant. Des études menées dans ([Akl85]) on conclut que :

Lemme 3.5.5 L'accélération obtenue sur une grille bidimensionnelle avec P processeurs pour le problème du tri de r éléments est $min\{O(\sqrt{P}log r), O(P)\}$. ♦

D'où nous prouvons :

Théorème 3.5.6 La parallélisation du problème du Sac-à-Dos sur une grille bi-dimensionnelle à P processeurs produit une accélération optimale lorsque $P \leq log^2 N$. ♦

Ce qui conduit à :

Corollaire 3.5.7 Soient, sur une grille bidimensionnelle de P processeurs, Acc_{qt} la meilleure accélération connue d'une parallélisation de l'algorithme des deux-listes quatre-tables et Acc_{st} l'accélération obtenue par la meilleure parallélisation de l'algorithme Sac_tri. Alors $Acc_{qt} < Acc_{st}$.

Preuve :

La seule parallélisation connue de l'algorithme des deux-listes quatre-tables est celle basée sur la parallélisation de l'algorithme find par le partitionnement de l'espace de recherche (*cf.* 2.5.1). Par conséquent, $Acc_{qt} = \sqrt{P}$, tandis que, par le lemme 3.5.2, on obtient :

$$Acc_{st} = min\{\sqrt{P} log N, P\} > \sqrt{P}.$$ ♦

Ce résultat est très intéressant parce qu'il montre, entre autre, que pour des machines à mémoire distribuée au moins aussi puissantes que la grille de processeurs, l'algorithme Sac_tri en parallèle est le meilleur existant actuellement pour le problème du Sac-à-Dos. En guise d'exemple, sur cette architecture l'algorithme Sac_tri résoud de façon optimale des instances du problème du Sac-à-Dos à 100 variables dès que $P \leq 2500$, borne tout à fait raisonnable sur le nombre de processeurs couplés en grille.

Hypercubes

L'algorithme bitonique de Batcher ([Knu72]) permet de trier N entiers en temps en $O(log^2 N)$ lorsqu'il est implanté sur un hypercube de taille N. Quoique ce soit assez difficile de concevoir un hypercube de dimension 50 pour la résolution d'instances à 100 variables, il existe déjà des machines qui peuvent être configurées jusqu'en 12-cube pour trier 64K entiers de 32-bits en environ 30 millisecondes ([Hil87]).

Dans le cas où le nombre de processeurs est plus petit que le nombre d'éléments à trier, des algorithmes optimaux ont été conçus pour l'implantation du tri sur un hypercube ([Aga88]), conduisant à :

Lemme 3.5.8 Soit $0<\varepsilon<1$, alors la parallélisation du problème du tri de r éléments est optimale sur un hypercube de $(P = r^{1-\varepsilon})$ processeurs. ♦

Par conséquent :

Théorème 3.5.9 La parallélisation du problème du Sac-à-Dos sur un hypercube produit une accélération optimale lorsque $P = N^{1-\varepsilon}$, pour $0<\varepsilon<1$. ♦

Il faut remarquer pourtant que pour ce type d'architecture, le réseau d'interconnexion n'est pas aussi simple et modulaire que sur les exemples précédents. Le nombre de liens de communication par processeur n'est plus constant. L'échange de messages est fait à travers *log*P canaux, qui, pour des grandes valeurs de P, n'ont pas la même longueur physique. Ainsi le temps pour la communication de données sera différent pour des cubes de dimensions différentes. Il faut donc s'attendre à des résultats expérimentaux distincts des résultats théoriques à cause des contraintes sur les communications ([Fer89d], [Fer89e]).

Mélanges parfaits

Le schéma bitonique peut être aussi bien implanté sur un hypercube que sur une machine à mélange et décalage (*cf.* chapitre 1). r entiers sont triés en temps en $O(log^2 r)$ à l'aide d'une machine à mélange et décalage avec r modules de stockage et r/2 comparateurs. Dans le cas où on a un moindre nombre de processeurs plus puissants, les résultats obtenus pour des hypercubes sont aussi valables ici. D'après les algorithmes montrés dans ([Aga88]), on a :

Lemme 3.5.10 La parallélisation du problème du tri de r éléments est optimale sur une machine à mélange et décalage de $(P = r^{1-\varepsilon})$ processeurs, où $0<\varepsilon<1$. ♦

Par conséquent :

Théorème 3.5.11 La parallélisation du problème du Sac-à-Dos sur une machine à mélange et décalage produit une accélération optimale lorsque $P = N^{1-\varepsilon}$, pour $0<\varepsilon<1$. ♦

L'avantage des machines à mélange et décalage sur les hypercubes vient du fait qu'elles ont un degré constant et égal à 4. Par contre les connexions n'ont pas de longueur fixe, puisque les fonctions 'mélange' et 'inverse' dépendent de la taille du réseau.

3.5.3 REMARQUES

La question qui naturellement s'impose maintenant est de savoir si ce nouvel algorithme peut résoudre des instances du problème du Sac-à-Dos jusqu'à 100 variables. Malheureusement la réponse est encore négative. Si d'une part on a vu comment espérer des délais d'exécution réalistes, sur des machines concevables, d'autre part les problèmes liés à la mémoire locale disponible sur chaque processeur sont cruciaux. Le schéma de départ pour l'algorithme Sac_tri a été celui du deux-listes, qui nécessite $S = O(N)$ cellules mémoire pour le stockage d'instances à n variables. Sa parallélisation produit une complexité de mémoire locale en

$$LS = \frac{O(N)}{P} \text{ par processeur}$$

et une complexité de mémoire totale en

$$S = O(N).$$

Si n = 100 alors

$$S = O(2^{50}) \text{ et } LS = \frac{O(2^{50})}{P}.$$

Si l'on substitue P par 2^{10} comme proposé par Schroeppel et Shamir ([Sch81]), chaque processeur doit avoir

$$LS = O(2^{40}) \text{ en espace mémoire}$$

ce qui n'est pas encore permis par la technologie actuelle.

En conclusion on peut dire que, malgré le fait qu'on ait développé des parallélisations optimales du problème du Sac-à-Dos sur des machines à mémoire distribuée, les instances de ce problème pour lesquelles n vaut 100 sont encore hors de portée pour le binôme technologie-algorithmes existant. Une solution envisageable est la parallélisation optimale de l'algorithme des deux-listes quatre-tables sur de telles machines.

CHAPITRE IV. OPTIMISATION COMBINATOIRE

Des problèmes d'ordre très divers trouvent des solutions efficaces à l'aide des techniques d'optimisation combinatoire. La plupart de ces problèmes consiste en la recherche d'un élément de meilleur valeur - par exemple, de coût minimal - dans un ensemble fini mais de très grande cardinalité et donc non énumérable ; de tels ensembles sont composés d'objets combinatoires (ou états), aussi appelés *solutions plausibles* ([Rou87]).

Les exemples plus courants de problèmes d'optimisation combinatoire appartiennent à la classe des problèmes NP-Durs ([Gar79]), leur complexité étant exponentielle en la taille du problème (disons, n). Par conséquent, des méthodes de recherche heuristiques ont été introduites pour les résoudre. Parmi ces méthodes on trouve les méthodes de recherche arborescente du type **Branch and Bound** comme la recherche A^*, la recherche *Alpha-Beta*, les recherches en *profondeur*, *largeur* ou *meilleur d'abord*, utilisées dans divers domaines comme la conception de circuits VLSI, la démonstration automatique de théorèmes, la programmation linéaire, les jeux d'échecs, et nombre d'autres applications en Intelligence Artificielle et Recherche Opérationnelle ([Akl89], [Nil80], [Rou87], [Win84]). Dans la suite on notera Branch and Bound par B&B.

D'autre part, l'utilisation de la technique du Recuit Simulé est aussi très répandue pour la résolution de problèmes d'optimisation combinatoire, principalement pour la conception de circuits VLSI et la CAO ([Rom84]). La technique de Recuit Simulé (désormais noté RS) est une approche

probabiliste basée sur une analogie avec le refroidissement statistique ([Kir83]). En métallurgie, le processus de recuit est composé d'une phase d'échauffement suivie d'une phase de refroidissement très lent et doux. Après ce processus, le métal se trouve dans une structure de basse énergie, la plus régulière possible. Cet état de structure régulière, considéré comme ayant une énergie optimale, serait alors l'état solution, avec un coût minimal. C'est pourquoi on simule ce processus de recuit pour la résolution de problèmes d'optimisation combinatoire.

Dans ce chapitre, nous nous intéressons aux parallélisations de ces deux techniques, en suivant des approches distinctes pour chacune d'entre elles. Nous proposons un algorithme de B&B pour des hypercubes à granularité fine (*cf.* Chapitre 1) qui garantit à la fois la distribution équilibrée des tâches parmi les processeurs et l'accès des processeurs à des informations globales concernant, entre autres, la forme courante de l'arbre de recherche. Nous essayons aussi d'indiquer les principales difficultés rencontrées pour appliquer une telle solution aux machines à grosse granularité. Accomplies pendant un séjour à Carleton University (Ottawa, CA), ces recherches ont été menées en collaboration avec F. Dehne et A. Rau-Chaplin ([Deh89a], [Deh89b]), avec beaucoup de plaisir.

Ensuite nous étudions le comportement asymptotique de l'algorithme de RS. Nous démontrons que ses versions parallèles (e.g., [Bon84], [Laa87]) sont asymptotiquement moins performantes qu'une simple parallélisation de la méthode de Recherche Aléatoire Locale. Son comportement fini est aussi abordé, à travers la démonstration d'une borne sur la probabilité de réussite d'un tel algorithme dans un nombre fixé d'itérations. Ces résultats sont une conséquence d'un résultat antérieur obtenu par J. Zerovnik ([Zer]). Nous avons pu les développer et mettre au point ([Bra89a], [Bra89b], [Fer89c])

lors de son séjour au laboratoire LIP-IMAG, avec la participation essentielle de B. Braschi.

4.1 DU BRANCH AND BOUND DISTRIBUÉ

Comme la plupart des problèmes traités par la méthode de B&B appartiennent à la classe des problèmes NP-Durs, de nombreuses recherches ont porté sur l'implantation de B&B en parallèle, permettant l'évaluation d'un nombre plus important de nœuds de l'*arbre de recherche*, qui représente l'*espace solution* ([Hua89], [New88], [Mar82], [Rou87], [Usu87]). Jusqu'à présent les études se concentraient sur les architectures multiprocesseurs à grosse granularité (des systèmes comportant un nombre relativement petit - moins de 1000 -, de processeurs assez puissants, chacun avec une bonne quantité de mémoire). De plus, ces derniers temps une attention particulièrement importante a été portée sur des algorithmes conçus pour des hypercubes à gros grain, tels que celui de FPS, de NCUBE ou d'Intel iPSC ([Li86], [Qui87], [And87], [Abd88], [Fel88], [Par88], [Scw88]).

Les méthodes de B&B en parallèle pour des multiprocesseur à gros grain ont pour but de diviser l'arbre de recherche du B&B séquentiel en des sous-arbres de recherche en parallèle pour les différents processeurs. Les solutions proposées se distinguent essentiellement par la façon dont elles résolvent les deux problèmes les plus importants posés par la division de l'arbre de recherche :

Equilibrage des tâches : quand un sous-problème (un sous-arbre de recherche) est affecté à un seul processeur, sa taille n'est pas connue à l'avance. C'est pourquoi les tailles de problèmes affectés à des processeurs distincts peuvent varier significativement et une distribution non équilibrée

de la charge de travail peut conduire à une dégradation des performances ([Lai84], [Li86]).

Information globale : pour les méthodes B&B séquentielles, l'élagage de l'arbre de recherche dépend souvent d'informations globales telle que, par exemple, la meilleure solution trouvée jusqu'alors. Sur une architecture multiprocesseur, la distribution d'information globale provoque une augmentation de la complexité de l'algorithme, dûe aux communications nécessaires. Par contre, si l'information n'est pas complètement distribuée, la méthode en parallèle peut parcourir davantage de nœuds qu'en séquentiel, en effectuant des évaluations inutiles, ce qui provoque, à nouveau, une dégradation des performances ([Rou87]).

Par la suite, nous étudions le B&B en parallèle sur des hypercubes à grain fin (des architectures avec un grand nombre - plus de 10000 - de processeurs élémentaires), la Connection Machine étant un très bon exemple d'un tel système. La présentation des algorithmes dans cette section change par rapport aux précédents puisque notre but en ce qui concerne cette parallélisation du B&B est son implantation sur une Connection Machine ([Hil87]), pour pouvoir étudier son comportement sur des multiprocesseurs à grain fin, qui ont l'avantage immédiat de compter davantage de parallélisme, bien que basé sur des processeurs élémentaires.

4.1.1 ARBRE ET CHEMIN DE RETOUR

Chaque arête (i,j) de l'arbre de recherche est une transition qui transforme l'état i en l'état j. Quel que soit le nœud p de l'arbre de recherche, si p représente un état final du problème, alors la solution trouvée est le *chemin de retour de p* (le chemin qui mène de p à la racine de l'arbre).

Les parallélisations citées précédemment pour des systèmes à gros grain supposent que chaque processeur garde dans sa mémoire les chemins de retour de tous les nœuds qu'il examine. Au moins jusqu'à la racine du sous-arbre auquel ces nœuds appartiennent.

Dans l'environment d'un multiprocesseur à grain fin , il est impossible pour un processeur de stocker dans sa mémoire restreinte le chemin de retour d'un nœud, même s'il ne s'agit que du chemin relatif au nœud qu'il est en train d'examiner. Par conséquent, les algorithmes qu'on trouve dans la littérature ne s'appliquent pas à notre cas. De plus, le nombre beaucoup plus important de processeurs (et donc de processus concourants de recherche) fait que l'accès continu à des informations globales par chaque processeur est impératif, sous peine d'avoir une quantité intolérable d'évaluations inutiles effectuées.

De ce fait nous proposons une solution qui utilise un schéma où tous les processeurs gardent *collectivement* la totalité des chemins de retour, de telle façon que chaque processeur n'a besoin que de garder une quantité constante d'information. Au lieu de garder les chemins de retour relatifs à chaque nœud examiné par l'algorithme, nous montrons comment garder plutôt l'*arbre courant de retour*, défini comme l'union des chemins de retour de tous les nœuds de l'arbre de recherche qui sont en train d'être examinés. Cela fournit aussi une utilisation optimale de l'espace mémoire, puisque les chemins dans l'arbre de recherche qui sont partagés par plusieurs chemins de retour ne sont stockés qu'une seule fois.

A chaque itération de l'algorithme, tous les nœuds de l'arbre courant de retour peuvent décider si jamais ils doivent générer un nouveau fils, être élagués, ou rester inchangés. Basé sur ces décisions, l'algorithme met à jour l'arbre courant de retour et *distribue des informations globales* en O(*log*m)

pas, où *m* est la taille de l'arbre courant de retour. Cette méthode inclut aussi un *mécanisme d'allocation dynamique* pour l'allocation des processus de recherche aux processeurs, produisant un *équilibrage optimal des tâches.* Même dans le cas où des changements très importants se présentent sur l'arbre courant de retour, la performance de notre mécanisme d'équilibrage de tâches ne se détériore pas.

Il faut également remarquer que le temps en O(*log*m) mentioné ci-dessus est mesuré en considérant seulement des messages de longueur constante, échangés entre deux processeurs voisins, comme des opérations en O(1), selon le modèle défini au Chapitre 1. Sur un tel modèle, la complexité de l'opération "PREF" (parallel inter-processor read) en *LISP de la Connection Machine peut représenter un coût plus important que les O(*log*m) étapes que nous proposons..

4.1.2 BRANCH AND BOUND SÉQUENTIEL

La technique de B&B s'est beaucoup développée au cours des dernières années. Les algorithmes A*, Alpha-Beta, Hill-Climbing, Best-First et plusieurs autres ne sont que des instances particulières de B&B ([Win84]).

Au lieu de présenter des parallélisations spécifiques pour chacun de ces algorithmes, nous partons d'une formulation très générale de l'algorithme de B&B, ayant pour objectif la proposition d'une solution qui peut être appliquée à un maximum d'instances existantes. Ainsi on définit cinq règles pour le B&B qui, implantées de façon diverse, nous permettent de retrouver la plupart des algorithmes basés sur le B&B ([And87]).

Règle de coût : étant donné un nœud feuille de l'arbre de recherche et le coût de son père, cette règle définit le coût de la solution jusqu'à cette feuille, en l'incluant.

Règle d'évaluation : étant donné un nœud de l'arbre de recherche, cette règle retourne *vrai* si ce nœud n'est plus plausible (i.e., doit être supprimé), sinon retourne *faux*.

Règle de sélection : étant donné un nœud de l'arbre de recherche, cette règle retourne un entier réprésentant le nombre d'enfants qui doivent être générés par ce nœud pendant l'itération courante.

Règle d'expansion : étant donnés un nœud de l'arbre de recherche, le nombre d'enfants à être générés et un pointeur vers la position où les créer, cette règle permet la génération de nouveaux enfants.

Règle de terminaison : étant donné un ensemble composé d'informations globales, cette règle retourne *vrai* si une solution satisfaisante a été trouvée ou si toutes les possibilités ont été explorées, sinon retourne *faux*.

A l'aide de ces cinq règles on décrit l'algorithme de B&B suivant :

Algorithme *B&B séquentiel*

```
1   Initialiser l'arbre courant de retour avec un nœud racine
2   Tant que la règle de terminaison retourne faux faire
3       Calculer le coût du nœud en utilisant la règle de coût
4       Si la règle d'évaluation retourne vrai
5           alors supprimer le nœud
6           sinon mettre à jour l'arbre de retour
7       Engendrer un nouveau nœud en utilisant les
            règles de sélection et d'expansion
8   Rapporter le meilleur nœud et son chemin de retour
```

Figure 4.1.1 : *L'algorithme de B&B séquentiel*

4.1.3 PARALLELISATION EN GRAIN FIN

Un algorithme de Branch and Bound recherche dans l'espace de toutes les solutions plausibles pour un problème donné. Ces solutions plausibles constituent un arbre de recherche S sur l'espace solution. Pour les algorithmes de B&B en parallèle, la recherche dans S d'une solution optimale, ou même satisfaisante, est exécutée simultanément sur plusieurs nœuds de S. Ces nœuds sont alors appelés *nœuds actifs*. Le sous-arbre T de S, défini par l'union des chemins de retour de tous les nœuds actifs, est l'arbre courant de retour introduit auparavant. Noter que n'importe quel nœud de l'arbre courant de retour peut être un nœud actif - et non pas nécessairement que ses feuilles -, ce qui permet l'implantation de méthodes comme la recherche meilleur d'abord, par exemple. Dorénavant, *m* sera la taille de l'arbre courant de retour.

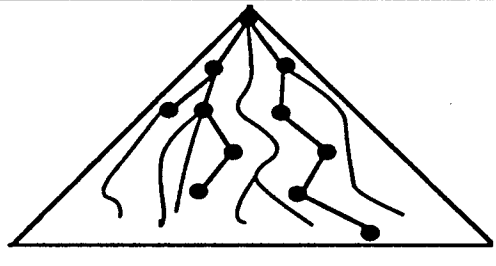

Figure 4.1.2 : *Un arbre courant de retour (nœuds en gras)*

Notre algorithme utilise des versions généralisées des opérations de routage sur un hypercube décrites dans le paragraphe 1.6.2. Ces opérations de base sont présentées en détail dans le paragraphe 4.1.5, à la fin de cette section.

Outre les registres énumérés dans la suite, l'implantation effective de cet algorithme peut requérir un nombre constant de registres auxiliaires.

Sur un hypercube avec N=2q processeurs, pour tout registre A disponible sur chaque processeur, A(i) fait référence au registre A du processeur PE(i).

On suppose que chaque processeur PE(i) a un registre de taille constante **n**(i) pour garder un *nœud*(i) de l'arbre courant de retour T. Le registre n(i) contient des champs **n.data**(i), **n.cost**(i), **n.parent**(i), **n.children**(i), **n.firstChild**(i), **n.level**(i) et **n.newChildren**(i). Chacun de ces champs garde une quantité constante de données associée au nœud(i) ; ses coût, parent, nombre d'enfants, position du premier enfant, niveau, et nombre de nouveaux enfants, respectivement. On admet aussi que chaque processeur a une copie de chaque règle utilisée dans l'algorithme de B&B séquentiel de la figure 4.1.1.

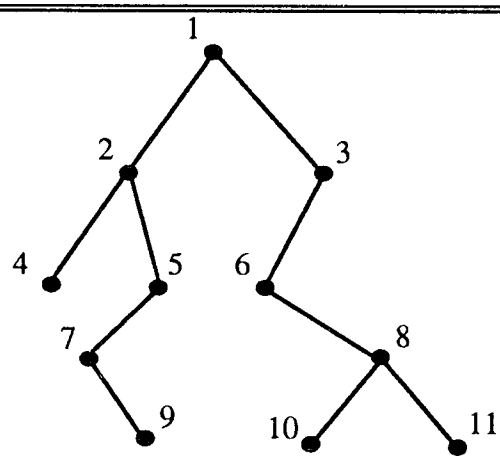

Figure 4.1.3 : *L'ordonnancement par niveau d'un arbre courant de retour*

On considère l'ordonnancement par niveau des nœuds de T comme sur la figure 4.1.3. L'arbre courant de retour T est stocké sur l'hypercube de telle façon que chaque nœud v de T est stocké dans le registre n(i) du

processeur PE(i), où i est l'indice de v dans l'ordonnancement par niveau de T.

La structure globale de l'algorithme parallèle de B&B est décrite par la suite. L'arbre de retour T débute comme un unique nœud racine, stocké dans le registre n(0) du processeur PE(0). La boucle principale de l'algorithme itère jusqu'à ce que la règle de terminaison retourne *vrai*.

Algorithme *B&B parallèle*

A Initialiser l'arbre courant de retour comme la racine dans le processeur PE(0). Initialiser n.data(0) et mettre tous les autres champs de n(0) à 0. Générer aussi un nœud factice dans PE(1) avec n.parent(1) = ∞.

B Tant que la *règle de terminaison* retourne *faux* faire

1 Tout PE(i) avec un nœud qui vient d'être créé utilise la *règle de coût* pour calculer n.cost(i). Le coût minimum global est ensuite distribué à tous les processeurs.

2 Tout PE(i) utilise la *règle d'évaluation* pour mettre alive(i) à 1 si le nœud est encore plausible, ou à 0 sinon.

3 Tout PE(i) utilise la *règle de sélection* pour faire n.newChildren(i) = nombre d'enfants à être générés.

4 On appelle la procédure UpDateTree qui génère, à partir de l'ancienne version de l'arbre de retour, un nouvel arbre avec tous les nœuds non-plausibles déjà supprimés et des nouveaux nœuds insérés.

5 Tout PE(i) utilise la *règle d'expansion* pour affecter les données correspondantes aux n.newChildren(i) nouveaux enfants de chaque nœud n(i).

C Rapporter le meilleur nœud et le chemin qui mène à lui à partir de la racine.

Figure 4.1.4 : *L'algorithme parallèle de B&B*

Analyse :

A chaque itération l'algorithme met à jour l'arbre de retour au moyen de l'insertion des nouveaux nœuds de la façon prescrite par les règles de sélection et d'expansion, tandis que les règles de coût et d'évaluation sont utilisées pour l'élagage de l'arbre.

Un passage par la boucle principale est composé de cinq pas. D'abord, **au pas 1**, la fonction de coût est calculée pour tous les nœuds qui ont été générés à la dernière itération. Cette même règle de coût peut aussi maintenir la connaissance globale en ce qui concerne le développement de la recherche. Par exemple, après avoir affecté de nouveaux coûts aux nœuds, la règle de coût peut calculer le coût minimum global, qui sera utilisé par les règles d'évaluation et terminaison.

La maintenance d'une telle connaissance globale n'augmente pas la complexité de l'algorithme, puisqu'on peut en calculer en utilisant des procédures classiques de minimisation globale et diffusion ([Jo89]) en temps O(*log*m). En effet, on peut utiliser n'importe quelle information globale dont les coûts de calcul et diffusion sur un hypercube sont O(*log*m).

Dans **le pas 2** la règle d'évaluation identifie les nœuds à supprimer dans cette itération. Aucune implantation spécifique de cette règle ne doit jamais supprimer un nœud père sans que ses enfants ne soient, eux aussi, supprimés. L'évaluation en parallèle d'un grand nombre de nœuds est l'un des aspects positifs de cet algorithme de B&B en grain fin, de par le grand nombre de nœuds de l'arbre de retour qui peuvent être supprimés d'un seul coup. La structure de données peut ainsi être compactée à un coût très bas d'un point de vue de la complexité temporelle.

La règle de sélection est utilisée dans **le pas 3** pour l'affectation d'un entier n.newChildren à chaque nœud(i), indiquant le nombre d'enfants à

être créés par ce nœud, dans cette itération. La règle de sélection peut aussi utiliser de l'information globale pour déterminer les nœuds à générer.

La procédure UpDateTree, appelée **au pas 4** de l'algorithme, en est la partie la plus importante. Dans les étapes précédentes toutes les informations nécessaires à l'expansion de l'arbre de retour ont été collectionées et à ce moment l'arbre doit être mis à jour par la suppression de certains nœuds et par la création d'espace pour les nouveaux nœuds. La procédure UpDateTree sera décrite en détail dans le paragraphe suivant.

Finalement, **au pas 5**, chaque nœud connaît le nombre d'enfants à créer, aussi bien que les adresses des processeurs libres alloués pour ses enfants par la procédure UpDateTree. Par conséquent, les nœuds pères doivent envoyer leurs données aux processeurs qui auront à stocker leurs enfants, pour que les processeurs respectifs calculent en parallèle, à l'aide de la règle d'expansion, pour chaque enfant son propre ensemble de données. Une telle opération peut être implantée en O(logm) à l'aide de la procédure RouterEtCopier décrite en 4.1.5.

4.1.4 MISE À JOUR DE L'ARBRE DE RETOUR

Le cœur de cet algorithme de B&B en parallèle est la procédure UpDateTree qui met à jour l'arbre de retour après que les nœuds non plausibles aient été supprimés et de nouveaux nœuds aient été ajoutés au pas précédent. Le problème principal qui se pose ici vient du fait que, pour le nouvel arbre, les nœuds doivent encore être stockés sous un ordonnancement par niveau (pour des raisons liées au stockage et à la performance de l'algorithme). De ce fait, il est nécessaire de calculer la nouvelle adresse de chaque nœud du nouvel arbre de retour et de les réaménager selon le schéma de stockage correct. Un tel réaménagement des

nœuds de l'arbre fournit, en outre, un mécanisme d'allocation optimale de tâches pour résoudre le problème d'équilibrage de tâches.

Les paramètres pour cette procédure sont gardés dans les registres alive(i) et n.newChildren(i) de chaque processeur. Le registre alive(i) est mis à 1 si le nœud ne doit pas être supprimé, ou à 0 sinon. D'autre part, n.newChildren(i) reçoit le nombre de nouveaux enfants du nœud n(i).

Pour illustrer l'idée sousjacente à cette procédure, on considère le nœud *p* de la figure 4.1.5. L'adresse (numéro de processeur) du premier enfant de *p*, après la mise à jour de l'arbre, sera la somme de deux nombres, x et y, définis comme suit : x est le nombre de nœuds dans l'arbre jusqu'au niveau de *p* (en l'incluant), tandis que y est le nombre d'enfants des nœuds qui sont à gauche de *p* dans son niveau. La somme de ces deux nombres indique la position du premier enfant de p dans le nouvel arbre.

La procédure UpDateTree calcule où chaque nœud placera ses enfants sur l'hypercube après la mise à jour de l'arbre de retour. Chaque nœud père doit diffuser ses informations à tous ses enfants. Cependant, comme la totalité des nœuds changera de place, tous les nœuds père diffusent à leurs enfants aussi leur propre nouvelle adresse. Ainsi les enfants peuvent mettre à jour leur pointeur sur le père.

Au moment où ces étapes seront finies, tous les nœuds qui n'ont pas été supprimés sont routés vers leurs nouvelles places. Cette opération de routage laisse de l'espace nécessaire pour les nouveaux enfants, créés par la règle d'expansion appelée dans la procédure principale.

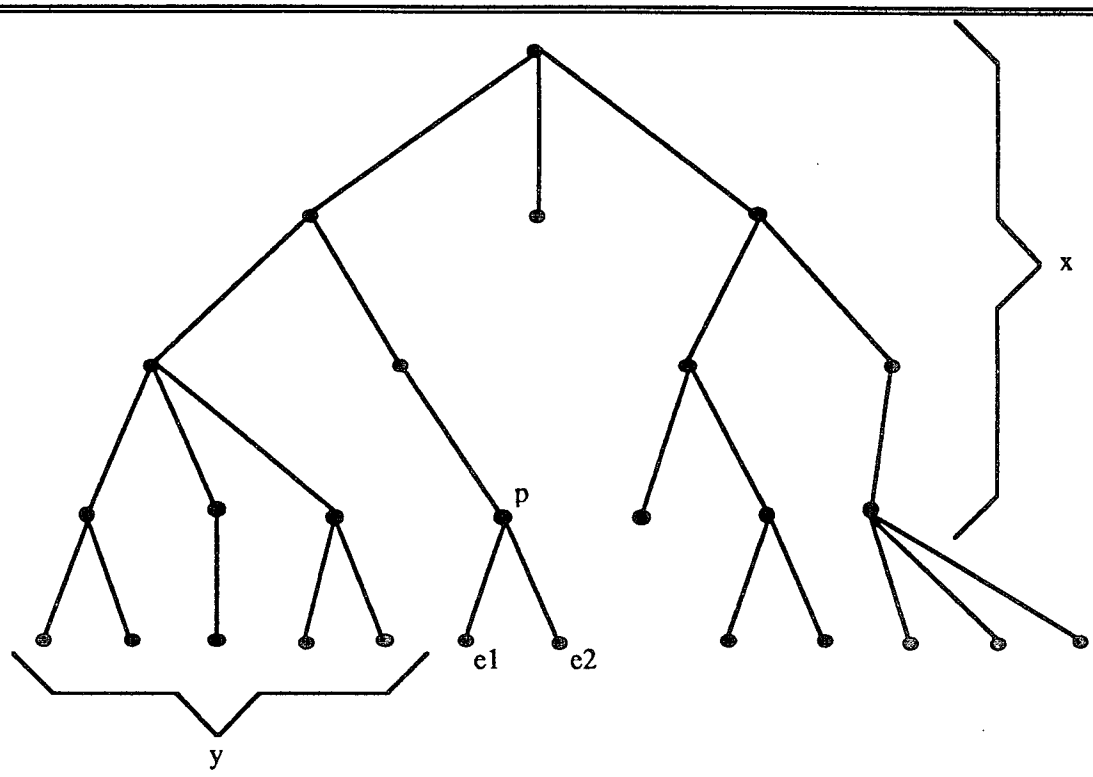

Figure 4.1.5 : *Illustration du calcul des adresses pour le nouvel arbre courant de retour*

Procédure *UpDateTree*

1 BlocPsum(alive(i), alive'(i), n.parent(i))
Router(alive'(i), n.children(i), n.parent(i), n.parent(i) ≠ n.parent(i+1))
Tout PE(i) : n.oldChildren(i) := n.children(i)
Tout PE(i) : n.children(i) := n.children(i) + n.newChildren(i)

2 Psum(n.children(i), x')
Tout PE(i) : x'(i) := x'(i) + 1
IdentifierBloc(n.level(i), endOfLevel(i))
Tout PE(i) : nextEndOfLevel(i) := endOfLevel(i+1)
RouterEtCopier(x'(i), x(i), nextEndOfLevel(i), i = endOfLevel(i))

3 BlocPsum(n.children(i), y(i), level(i))
Tout PE(i) : y(i) := y(i) - n.children(i)

```
4    Tout PE(i) : newFirstChild(i) := x(i) + y(i)
     Tout PE(i) : lastChild(i) := n.firstChild + n.oldChildren(i) - 1
     RouterEtCopier(newFirstChild(i), firstSibling(i), lastChild(i),
                                              n.oldChildren(i) > 0)
     Tout PE(i) : newAddress(i) := firstSibling(i) + alive'(i) - 1

5    RouterEtCopier(newAddress(i), n.parent(i), lastChild(i),
                                              n.oldChildren(i) > 0)
     Tout PE(i) : n.firstChild := newFirstChild(i)

6    Router(n(i), n(i), newAddress(i), alive(i) = 1)
```

Figure 4.1.6 : *La procédure de mise à jour de l'arbre de retour*

Analyse :

La procédure UpDateTree est composée de six étapes. Dans **l'étape 1**, le nombre d'enfants de chaque nœud est recalculé pour tenir compte des nœuds qui seront supprimés et des nouveaux nœuds qui ont été ajoutés. Cette opération est accomplie à l'aide d'une somme partielle des nœuds qui sont encore vivants dans chaque bloc de nœuds qui se partagent le même père. Le résultat de la somme partielle est alors envoyé à chaque père par son dernier enfant. Cette étape est completée par l'affectation du nouveau nombre d'enfants (enfants non supprimés plus enfants générés) au registre n.children(i) de chaque PE(i).

Pour chaque nœud au niveau k, la valeur de x est 1 plus le nombre d'enfants de tous les nœuds jusqu'au dernier nœud du niveau k-1 (en l'incluant). A **l'étape 2**, cette valeur est calculée par une somme partielle sur le nombre d'enfants, n.children(i), de chaque nœud. Une telle somme est gardée dans le registre x'(i) qui est incrémenté de 1 par la suite pour tenir

compte de la racine de l'arbre. Finalement, le dernier nœud de chaque bloc de nœuds partageant le même père envoie x' à son père.

L'étape 3 calcule la valeur de y, à l'aide d'une somme partielle sur le registre n.children(i) pour tous les nœuds d'un même niveau. Pour éviter de compter les enfants d'un nœud dans son propre y(i), n.children(i) doit être enlevé de la valeur fournie par la somme partielle.

Pendant **l'étape 4**, la nouvelle adresse du premier enfant de chaque nœud est calculée et stockée dans le registre newFirstChild(i), en utilisant les valeurs x(i) et y(i) calculées précédément. Cette adresse est alors diffusée à tous les enfants de chaque nœud. De ce fait, chaque enfant peut calculer sa nouvelle adresse selon l'ordonnancement par niveau dans l'hypercube, puisqu'elle est égale à l'adresse du premier enfant de son père plus le nombre de ses frères qui sont placés avant lui.

A ce point de la procédure tous les nœuds connaissent leur nouvelle adresse. Par conséquent, tous les pères communiquent les leurs à leurs enfants, **à l'étape 5**, pour que ces enfants, à leur tour, puissent mettre à jour leur pointeur pour le père.

Pour clôturer la procédure UpDateTree il ne manque que l'aménagement des nœuds à leurs nouvelles positions, ce qui est exécuté pendant **l'étape 6**.

Toutes les opérations globales utilisées dans la procédure UpDateTree ont une complexité en O(*log*m), où m est la taille de l'arbre courant de retour (*cf.* 4.1.5). Par conséquent :

Lemme 4.1.1 La complexité temporelle de la procédure UpDateTree est O(*log*m). ♦

Et comme implication directe on a :

Théorème 4.1.2 L'algorithme parallèle de B&B décrit dans la figure 4.1.4 prend O(*log*m) pas pour la mise à jour - suppression et création de nœuds - et la réallocation de l'arbre courant de retour. ♦

Ce théorème montre que davantage de nœuds de l'arbre de recherche peuvent être visités, avec un "overhead" de à peine O(*log*m). Malheureusement, à cause des anomalies qui peuvent se produire ([Lai84], [Li86], [Wah85]), il est très difficile de comparer notre solution aux algorithmes séquentiels existants, en ce qui concerne leurs complexités asymptotiques.

Par conséquent, cet algorithme pour la parallélisation de la méthode de B&B est en phase d'implantation sur la Connection Machine et des résultats de performances empiriques seront disponibles dans un futur proche. Dans un premier temps on implante des algorithmes pour la résolution de problèmes de simple description, du type Voyageur de Commerce, pour ensuite essayer l'exécution d'algorithmes de jeux ([Akl89]).

4.1.5 OPÉRATIONS DE BASE

Psum(source(i),result(i)) : les processeurs ont une valeur stockée dans le registre **source**(i). Cette opération calcule **result**(i) := **source**(0) + **source**(1) + ... + **source**(i) pour chaque PE(i). Il s'agit ici de l'opération de sommes partielles standard sur un hypercube, qui peut être implantée en O(*log*N) pas (*cf.* Chapitre 1).

Router(reg$_1$(i),reg$_2$(i),dest(i),cond(i)) : tout processeur PE(i) possède deux registres de données **reg$_1$**(i), **reg$_2$**(i), un registre de destination **dest**(i), et un registre booléen de condition **cond**(i). On suppose que les destinations **dest**(i) sont monotones, i.e., si i < j alors **dest**(i) < **dest**(j). Cette opération route, pour tout processeur PE(i) pour lequel **cond**(i) = *vrai*, le contenu du registre **reg$_1$**(i) pour le registre **reg$_2$**(dest(i)) du processeur PE(**dest**(i)). Elle peut être implantée à l'aide d'un appel de *Rank* (avec **cond**(i) = vrai comme critère de sélection), suivi d'un appel de *Concentrate* et ensuite de *Distribute* (suivant les destinations **dest**(i)). Son coût est alors O(*log*N) (*cf.* Chapitre 1).

RouterEtCopier(reg$_1$(i),reg$_2$(i),dest(i),cond(i)) : sous les mêmes hypothèses que pour l'opération Router, cette opération route, pour tout processeur PE(i) pour lequel **cond**(i) = *vrai*, une copie de **reg$_1$**(i) pour les registres **reg$_2$**(**dest**(i-1) + 1), ..., **reg$_2$**(**dest**(i)). Son implantation suit les pas utilisés pour l'opération *Router*, sauf pour l'appel de *Distribute*, qui est substitué par un appel de *Generalize*. Sa complexité sera donc en O(*log*N).

IndentifierBloc(bloc(i),endOfBloc(i)) : un bloc de processeurs est défini par des PE(i) consécutifs qui ont la même valeur dans le registre **bloc**(i). Pour chaque PE(i), cette opération affecte à **endOfBloc**(i) le plus grand j tel que **bloc**(j) = **bloc**(i). Sa complexité temporelle est aussi en O(*log*N), étant implantée de la façon suivante :

```
Tout PEi : lire bloc(i+1)
Si bloc(i+1) ≠ bloc(i) alors select(i) := vrai
RouterEtCopier(i,endOfBloc(i),i,select(i))
```

BlocPsum(source(i),result(i),bloc(i)) : cette opération calcule des sommes partielles uniquement dans des blocs. Tout PE(i) garde en **result**(i) la somme partielle des valeurs stockées dans le registre **source**(j) des PE(j)

qui appartiennent à son bloc. Il est évident que la valeur finale de la variable **result**(i) est la valeur de la somme partielle globale jusqu'à PE_i, moins celle de la somme partielle jusqu'au dernier processeur du bloc antérieur. Cette opération aussi, peut être implantée en O(*log*N) :

```
PSum(source(i),result(i))
IdentifierBloc(bloc(i),endOfBlocAux(i))
Tout PEi : lire endOfBlocAux(i+1)
Si endOfBlocAux(i) = i alors select(i) = vrai
RouterEtCopier(result(i),aux(i),endOfBlocAux(i+1),select(i))
Tout PEi : result(i) := result(i) - aux(i)
```

4.2 RECUIT SIMULÉ

Toujours intéressés par des techniques qui trouvent un minimum global d'une fonction coût parmi un ensemble d'états, nous abordons dans cette section l'algorithme de Recuit Simulé (dorénavant appelé RS). Un tel algorithme commence la recherche sur un état de l'espace solution, calcule sa fonction coût associée et à chaque étape choisit un nouvel état aléatoirement. Si ce nouvel état induit une meilleure évaluation de la fonction coût alors il devient le nouvel état pour l'algorithme. Sinon il peut être accepté selon une certaine probabilité, dépendant d'un paramètre - normalement appelé température - qui décroît en fonction du temps ([Kir83]).

Si l'on suppose que rien n'est connu sur la structure de l'ensemble d'états, l'algorithme RS est asymptotiquement moins bon qu'une simple répétition de la technique de Recherche Aléatoire Locale ([Zer88]). Avec la même hypothèse, nous prouvons une borne supérieure pour la probabilité que RS trouve, dans un nombre fini d'itérations, des solutions quasi-optimales. Nous montrons aussi que, pour ce genre de solution, l'algorithme

de Recherche Aléatoire Locale (RAL) a un meilleur comportement asymptotique que RS.

Le grand inconvénient de l'utilisation du Recuit Simulé dans la résolution de problèmes d'optimisation combinatoire est son temps de calcul très élévé ([Rom84]). Ainsi l'emploi du parallèlisme pour son exécution en est une conséquence directe, même si, pour la plupart de différentes approches proposées, rien n'a pu être prouvé par rapport à la convergence de l'algorithme ([Laa87]).

Pour tenir compte de la diversité de telles approches, nous introduisons un *algorithme parallèle de recuit*, très général, qui contient une grande classe des parallélisations existantes de la technique de Recuit Simulé ([Bon84], [Bra88], [Don88], [Fel85]). Ensuite nous démontrons que cet algorithme parallèle de recuit est asymptotiquement moins bon que la parallélisation triviale de RAL.

4.2.1 QUELQUES DÉFINITIONS ET NOTATIONS

Ci-dessous on présente une description formelle du problème générique d'optimisaton combinatoire que l'on veut résoudre :

Etant donnés :

- un ensemble fini d'états S,
- un mécanisme de perturbation aléatoire qui affecte à chaque état s de S, un autre état s' de S, designé comme un *voisin* de s,
- une fonction coût qui affecte une valeur c(s) à chaque état s.

On veut obtenir un état s* de S dont la valeur c(s*) est minimum en S.

Outre ces hypothèses, on suppose que rien n'est connu à priori sur la structure de l'ensemble S d'états et on n'a donc pas de bon candidat comme état initial : RS doit commencer à partir d'un état aléatoire.

Recuit Simulé

L'algorithme de RS utilisé pour résoudre ce problème générique est composé de deux boucles :

Algorithme *RS*

```
1   calculer un état aléatoire initial s ;
2   fixer la température initiale T;
3   répéter -- boucle externe
4       répéter -- boucle interne
5           calculer un voisin aléatoire s' de s ;
6           si c(s') < c(s) alors
7               transformer s' en le nouvel état s
8           sinon
9               transformer s' en le nouvel état s
                    selon une probabilité p(T)
10      jusqu'à l'équilibre
11      decroître T
12  jusqu'à cristalisation.
```

Figure 4.2.1 : *L'algorithme général de Recuit Simulé*

La façon dont on définit les expressions en gras dans l'algorithme est très importante en ce qui concerne son comportement ([Bon84], [Laa87]). On reviendra sur ce point plus loin dans le texte. Maintenant on décrit superficiellement la convergence d'un tel algorithme basée sur l'étude de comportements infinis sous une situation théorique idéale. Une étude plus approfondie peut être trouvée dans ([Laa87]).

L'évolution du système est formulée par une série de variables aléatoires, dont les valeurs se trouvent dans l'ensemble d'états S. Une telle série est une chaine de Markov. La température est représentée par T, un paramètre dont la chaine dépend. Si le voisinage possède une symétrie correcte, et si ce même voisinage est tel que chaque état peut être atteint à partir de n'importe quel état, alors la convergence asymptotique du système est assurée quand le critère de Boltzmann est choisi pour calculer la probabilité d'acceptation d'un nouvel état pour lequel l'évaluation de la fonction coût n'est pas meilleure que celle de l'état courant. De ce fait on choisit la probabilité comme étant définie par

$$p(T) = e^{(c(s)-c(s'))/T}$$

et la série convergera asymptotiquement vers la distribution de Boltzmann. Il faut remarquer que la probabilité dépend de la fonction coût du voisin choisi et de la température courante.

Si l'on observe la façon dont la fonction coût décroit pendant l'exécution de l'algorithme, on remarque des intervalles de T pour lesquels cette fonction chute rapidement. Ce phenomène, interprété comme un changement de phase de la procédure, indique qu'on a atteint des régions critiques. D'autre part, la notion d'équilibre pour une température donnée correspond à une chaine de Markov complète ; qui est, en fait, infinie.

Dans la pratique, plusieurs moyens ont été proposés aussi bien pour éviter la dépense de temps de calcul dans des intervalles inapropriés de T que pour essayer de réproduire des chaines de Markov avec un nombre fini de pas dans la boucle interne. On suppose ici que la température T est baissée selon un plan prédéfini :

$$(T_1,m_1), (T_2,m_2), \ldots, (T_k,m_k), \ldots$$

qui signifie exécuter m_1 pas avec la température T_1, ..., exécuter m_k pas avec la température T_k, ...

La valeur finale de T est l'analogue de la température de cristalisation dans le processus physique. Comme on l'a vu précédemment, T doit tendre vers 0 quand le nombre de pas tend vers l'infini. En outre il faut qu'elle soit baissée très lentement pour assurer que la solution optimale globale soit trouvée, asymptotiquement, avec une probabilité 1.

Recherche Aléatoire Locale

La technique de Recherche Aléatoire Locale est très simple et consiste en la répétition de recherches locales (optimisation locale) sur des solutions initiales choisies aléatoirement :

Algorithme *RAL*

```
1   répéter -- boucle externe
2       générer un état aléatoire initial s ;
3       répéter -- boucle interne
4           trouver le meilleur voisin s' de s ;
5           transformer s' en le nouvel état s
6       jusqu'à ce que il n'y ait plus de meilleur voisin
7   jusqu'à la limite du nombre de pas.
```

Figure 4.2.2 : *L'algorithme général de Recherche Aléatoire Locale*

Pour étudier et pouvoir comparer le comportement des algorithmes présentés aux figures 4.2.1 et 4.2.2, nous avons besoin d'introduire une notation spécifique à ce problème. On prend comme unité de temps le coût du choix aléatoire d'un état voisin dans la boucle interne de RS.

Soient alors :

N, le nombre d'états de S.

K_1, le nombre d'états de S à partir desquels **tous** les chemins strictement décroissants finissent en un minimum global.

K, le nombre d'états de S à partir desquels **au moins un** des chemins strictement décroissants finit en un minimum global. Clairement $K \geq K_1$. On suppose $0 < K \leq K_1 < N$ pour éviter les cas triviaux.

R, la longueur du plus long chemin strictement décroissant vers un minimum local arbitraire.

w, le coût - en unités de temps - pour générer aléatoirement chaque état initial dans la boucle externe de RAL.

d, le nombre maximum de voisins qu'un état de S possède.

q_i, la plus petite probabilité que RS, à la température T_i et calculé sur tous les états, n'accepte pas un nouvel état qui soit moins bon que l'état courant. Il s'agit de la plus petite probabilité de "**ne pas monter**" en RS. Remarquer que q_i tend vers un quand i tend vers l'infini, par la définition de *p(T)* donnée précédemment.

Soit SRAL(n) la probabilité que l'algorithme RAL en séquentiel aboutisse (i.e., trouve une solution optimale) en n pas, où un pas équivaut à une unité de temps. On rappelle que chaque passage complet par la boucle externe de RAL coûte au plus w + R*d. De façon similaire, soit SRS(n) la probabilité que l'algorithme RS en séquentiel aboutisse en n pas. Cependant, contrairement au cas de RAL, on dit que RS a aboutit quand on atteint un état s à partir duquel au moins un chemin strictement décroissant finit dans un minimum global, ce qui veut dire qu'un tel état s est compté dans K.

Ainsi, Zerovnik ([Zer88]) a prouvé que

$$SRAL(n) \geq 1 - \left(\frac{N - K1}{N}\right)^{\left\lceil \frac{n}{w + R.d} \right\rceil}$$

et

$$SRS(n) \leq 1 - q_i^{\widetilde{m_i}} q_{i-1}^{m_i - 1} \ldots q_2^{m_2} q_1^{m_1} \left(1 - \frac{K}{N}\right)$$

où

$$0 < \widetilde{m_i} \leq m_i \text{ and } n = \widetilde{m_i} + \sum_{k=1}^{i-1} m_k.$$

ce qui a conduit au théorème suivant :

Théoreme 4.2.1([Zer88]) Il existe une constante n_0 telle que $n > n_0$ implique $SRS(n) < SRAL(n)$. ♦

Ce théorème montre que, pour le mode séquentiel et dans le cas où on veut trouver la solution optimale, le comportement asymptotique de la technique de Recuit Simulé est moins bon que celui de Recherche Aléatoire Locale. Dans les sections suivantes nous montrerons comment ce résultat peut être utilisé à la fois pour obtenir des bornes supérieures pour le comportement de RS sur des applications pratiques où on recherche une solution quasi-optimale pendant un nombre fini d'itérations, et aussi pour étudier le comportement asymptotique de RS, face à celui de RAL, pour le mode parallèle de calcul.

4.2.2 UNE BORNE POUR LE RECUIT SIMULÉ

Pour prouver que les résultats montrés dans la section précédente sont valables aussi quand on recherche une solution quasi-optimale, nous introduisons de nouvelles définitions. Soient :

c_0, le coût d'un état optimal pour un problème donné et $\varepsilon > 0$. Si le coût c d'un état est tel que $c < (1+\varepsilon)c_0$, alors on l'appelle *état ε-optimal.*

K'_1, le nombre d'états à partir desquels **tous** les chemins strictement décroissants finissent en un des états ε-optimals.

K', le nombre d'états à partir desquels il existe **au moins un** chemin strictement décroissant qui mène à une solution ε-optimal.

QRS(n), la probabilité que *RS quasi-optimal* aboutisse (i.e., RS atteigne un état compté dans K') en n pas.

QRAL(n), la probabilité que RAL trouve une solution ε-optimal en n pas.

Proposition 4.2.2 Il existe une constante n_0 telle que $n > n_0$ implique $QRS(n) < QRAL(n)$.

Preuve :

Cette démonstration suit le même plan que celle du théorème 4.2.1. Il est évident à partir des définitions ci-dessus que

$$QRAL(n) \geq 1 - \left(\frac{N - K'1}{N}\right)^{\left\lceil \frac{n}{w + R.d} \right\rceil}$$

et

$$QRS(n) \leq 1 - q_i^{\widetilde{m_i}} q_{i-1}^{m_{i-1}} \ldots q_2^{m_2} q_1^{m_1} \left(1 - \frac{K'}{N}\right)$$

où

$$0 < \widetilde{m_i} \leq m_i \text{ and } n = \widetilde{m_i} + \sum_{k=1}^{i-1} m_k.$$

d'où on se retrouve dans les mêmes conditions du théorème 4.2.1 et, par conséquent, pour des valeurs suffisamment grandes de n il est vrai que

$$QRS(n) < QRAL(n). \quad \blacklozenge$$

La démonstration de cette proposition donne lieu à un résultat très intéressant sur le comportement de RS pour le cas où on recherche une solution ε-optimale dans un nombre fini d'itérations :

Corollaire 4.2.3 Le comportement fini de RS quasi-optimal est borné, i.e., la probabilité que RS quasi-optimal aboutisse en $(\widetilde{m_i} + \Sigma m_j)$ pas, $1 \leq j < i$ et $0 < \widetilde{m}_i \leq m_i$, est au plus $1 - q_i^{\widetilde{m_i}} q_{i-1}^{m_{i-1}} \ldots q_2^{m_2} q_1^{m_1} \left(1 - \frac{K'}{N}\right)$. ◆

La plus importante conséquence de ce corollaire est le fait que, pour toute application de l'algorithme RS, la probabilité qu'il produise un solution ε-optimal est bornée supérieurement. De plus, cette borne, même si elle n'est pas serrée (noter, e.g., la définition relâchée de l'aboutissement de la recherche), ne dépend que de l'instance particulière du problème d'optimisation traité, pouvant ainsi être calculée pour des problèmes de petite taille. Une telle démarche pourrait se révéler fructueuse pour nous permettre de mieux comprendre le vrai comportement de l'algorithme RS.

Au vue des résultats qu'on vient d'étudier, nous sommes enclins à dire que RS avec un refroidissement plus rapide et plusieurs départs pourrait être plus performant que l'algorithme standard qui recuit une seule solution initiale. Par contre, cette technique serait proche de celle de recherche locale, dont la faiblesse est d'être passible de rester prise au piège sur un minimum local.

4.2.3 EXTENSION AU MODE PARALLELE

Deux modèles principaux de parallélisations de l'algorithme RS ont été proposées dans la litérature. Le premier est basé sur le partitionnement de données, i.e., les données sont partagées entre les processeurs ([Bon84], [Bra88], [Don88], [Fel85]). La convergence de l'algorithme dans ce cas est

garantie par une répartition dynamique des données, ce qui permet la génération de toute solution plausible ([Fel85]). Le second modèle pour paralléliser la technique de Recuit Simulé est obtenu à travers la synchronisation des processeurs : à un certain moment de l'algorithme les processeurs se synchronisent pour choisir un nouvel état de départ. Comme ce choix est habituellement basé sur le coût des états proposés par chaque processeur, plusieurs critères pour orienter le choix global du nouvel état ont été étudiés. Entre autres on peut citer le choix de l'état dont l'évaluation de la fonction coût est la meilleure parmi tous les états proposés ([Don88]), le choix de x états avec les y meilleures valeurs ([Don88]), le choix d'un état selon la distribution de Boltzmann ([Laa87]), etc...

Avec l'intention d'aborder un problème plus général, on suppose que p processeurs exécutent indépendamment un *Algorithme Général Parallèle de Recuit* (RP, en court) selon un *tableau de synchronisation* (0, v_1, v_2,..., v_k,...), qui indique que tous les processeurs se synchronisent au début de l'algorithme, se resynchronisent pour choisir un nouvel état et/ou échanger des données après avoir exécuté v_1 pas, se resynchronisent à nouveau au pas $v_1 + v_2$, et ainsi de suite.

Algorithme *Général de RP*

```
1    i := 0 ;
2    calculer un état aléatoire initial s ;
3    répéter -- boucle externe --
4        répéter -- boucle interne --
5            calculer aléatoirement un état voisin s' ;
6            i := i + 1 ;
7            si c(s') < c(s) alors
8                transformer s' en le nouvel état s
```

```
9                   sinon
10                          transformer s' en le nouvel état s selon
                                une probabilité p(i)
11              jusqu'à synchronisation ;
12              choix global d'un nouvel état de départ
                        et/ou échange de données
13      jusqu'à la limite du nombre de pas.
```

Figure 4.2.3 : *L'algorithme général parallèle de recuit*

Il est important de noter que les deux approches pour la parallélisation de RS, décrites auparavant, ne sont jamais que des cas spéciaux de l'algorithme général présenté ci-dessus, où le nombre de pas tient le rôle de la température.

La probabilité de **monter** au cours d'une itération de RP (i.e., accepter un nouvel état moins bon que l'état courant) dépend maintenant du nombre de pas exécutés par chaque processeur. Pour prouver les résultats qui suivront, on garde les définitions du paragraphe 4.2.1, où chaque mention à RS doit être changée en RP. La seule modification porte sur la probabilité de succès de l'algorithme :

q_i = *min* {probabilité, sur tous les processeurs, de RP ne pas **monter** à l'étape v_i} ;

où q_i est calculé sur tous les états de l'espace solution, pour chaque passage dans la boucle interne de RP au courant de l'étape v_i.

Dans ce cas, la seule hypothèse requise est que les probabilités $q_{i's}$ soient croissantes, avec i, dans le comportement global du modèle parallèle : $q_i < q_{i+1}$.

De façon similaire au cas séquentiel, on considère que RP a abouti si au moins un des p processeurs a atteint un état à partir duquel il existe **au**

moins un chemin strictément décroissant qui mène à un état d'optimum global. On note PRP_i la probabilité de l'évenement

A_i = {RP a abouti en au plus $(v_1 + v_2 + \ldots + v_i)$ pas}.

On peut à nouveau appliquer le raisonnement utilisé pour la démonstration des résultats pour le mode séquentiel : si RP aboutit au cours des $(v_1 + v_2 + \ldots + v_i)$ premiers pas, soit il avait déjà abouti au cours de l'étape précédente de synchronisation, soit il est **monté** au moins une fois pendant l'étape courante. D'où

$$PRP_i \leq PRP_{i-1} + (1 - PRP_{i-1})\left(1 - q_i^{p.v_i}\right) = 1 - q_i^{p.v_i}(1 - PRP_{i-1})$$

De plus, pour la première synchronisation on a

$$PRP_1 \leq \frac{K}{N} + \left(1 - \frac{K}{N}\right)\left(1 - q_1^{m1}\right) = 1 - q_1^{m1}\left(1 - \frac{K}{N}\right)$$

Par conséquent la probabilité de succès de RP en *n* pas peut être bornée par la formule suivante :

$$PRP(n) \leq 1 - q_i^{p.\tilde{v}_i}\, q_{i-1}^{p.v_{i-1}} \ldots q_2^{p.v_2}\, q_1^{p.v_1}\left(1 - \frac{K}{N}\right)$$

où

$$0 < \tilde{v}_i \leq v_i \text{ and } n = \tilde{v}_i + \sum_{k=1}^{i-1} v_k.$$

D'autre part, la probabilité PRAL(n) que p exécutions indépendantes de l'algorithme RAL trouvent un optimum global en n pas parallèles est bornée par la même borne inférieure que RAL(p.n) :

$$PRAL(n) = 1 - (1 - SRAL(n))^p \geq 1 - (1 - (1 - Q))^p = 1 - Q^p$$

où

$$Q = \left(\frac{(N - K_1)}{N}\right)^{\left\lceil \frac{n}{(Rd+w)} \right\rceil}$$

et, par conséquent

Théorème 4.2.4 Il existe une constante n_0 telle que $n > n_0$, implique $PRP(n) < PRAL(n)$. ♦

4.2.4 REMARQUES

Principalement à cause de son apparente simplicité, la technique de Recuit Simulé est intensivement utilisée dans la résolution de problèmes d'optimisation combinatoire. Cet algorithme est apparu à partir d'une modélisation de la physique mécanique et, en ce qui concerne les applications courantes, RS est très portable de par la généralité des notions importantes de l'algorithme, comme celles de 'température' et 'cristalisation', par exemple.

Cependant, à cause de cette grande généralité, la phase de réglage de l'algorithme - où tous les paramètres doivent être définis une fois pour toutes -, demande beaucoup d'effort de la part des analystes qui travaillent sur un problème spécifique donné (et qui coûtent davantage que le temps CPU d'une machine).

Prouver qu'une simple Recherche Aléatoire Locale est asymptotiquement meilleure que Recuit Simulé, aussi bien en séquentiel qu'en parallèle, ne veut pas pour autant dire que pour des problèmes particuliers l'algorithme RS n'est pas une bonne approche. Toujours est-il, pourtant, que la borne supérieure introduite au corollaire 4.2.3 pour la probabilité de succès de Recuit Simulé peut apporter de nouveaux arguments à la discussion sur la puissance d'une telle technique.

CONCLUSION

Nous venons de présenter des contributions à l'étude du problème de la recherche d'un élément dans un ensemble. Nous avons démontré des bornes inférieures, proposé des algorithmes séquentiels et étudié des solutions parallèles pour des problèmes de recherche, soit dans le domaine des ensembles partiellement ordonnés, soit concernant l'étude des problèmes de décision ou d'optimisation combinatoire. Les résultats que nous avons trouvés ont engendré de nouvelles voies de recherches que nous essayons de rappeler par la suite.

Pour les matrices définies par la somme cartésienne $X + Y$ (triés) nous avons établi la complexité du problème de la recherche ($\Theta(n)$) et du problème de la sélection du k-ième plus petit élément ($\Theta(\sqrt{k})$). Puis nous avons fait un tour d'horizon sur les résultats existants pour $X + Y$ (triés) et pour des multi-ensembles, définis comme étant de la forme $\sum X_i$, la somme de vecteurs triés. Ayant par but l'accélération de la recherche dans de tels ensembles, nous avons étudié le problème de la sélection du successeur d'un élément quelconque, donné, dans $X + Y$ (triés), pour lequel nous avons démontré que la complexité est $\Theta(n)$, dans le pire cas.

En outre, pour des multi-ensembles, nous avons introduit des algorithmes et avons démontré des résultats de complexité telle que, par exemple, l'appartenance à la classe NP-Complet du problème de la recherche dans $\sum X_i$. Un problème très intéressant qui reste ouvert est celui de la recherche dans $X + Y + W$ (triés). Nous avons montré que la meuilleure borne inférieure connue est en $\Omega(n)$, tandis que sa borne supérieure est toujours en $O(n^2)$. Un autre problème ouvert relatif à $X + Y$

était celui de trier les n^2 éléments à l'aide de seulement $O(n^2)$ comparaisons. Apparemment une solution à ce problème, ouvert depuis une quinzaine d'années, sera présentée à la Conférence STACS 90, à Rouen, France.

Nous avons aussi étudié la parallélisation de la recherche dans X + Y (triés) sur divers modèles de calcul parallèle. Nous avons proposé quatre algorithmes basés sur différentes stratégies, la démonstration de leurs optimalités étant dépendante du nombre de processeurs et des modèles utilisés.

A partir des idées développées pour la recherche dans X + Y, nous avons conçu plusieurs algorithmes parallèles pour le problème de décision du Sac-à-Dos, sur différents modèles de parallélisme, qui présentent des améliorations importantes au niveau des compromis temps/processeurs et temps/matériel.

Pour les machines à mémoire partagée, nous avons d'abord proposé des algorithmes qui utilisent strictement moins de $O(n2^{n/2})$ matériel pour une complexité temporelle aussi strictement inférieure à $O(n2^{n/2})$. Ensuite nous avons introduit un algorithme qui présente une accélération optimale, induisant des compromis temps/processeur et temps/matériel en $O(n2^{n/2})$.

Enfin, nous avons montré que les meilleures solutions existantes pour le problème du Sac-à-Dos pouvaient être réduites au problème du tri. Par conséquent nous en avons déduit des solutions à accélération optimale pour les principales architectures des machines à mémoire distribuée. Malheureusement, une instance de ce problème avec 100 variables reste toujours hors de portée des ordinateurs existants. Pour pouvoir résoudre un tel problème il est nécessaire qu'une parallélisation à accélération optimale de l'algorithme des deux-listes quatre-tables sur des machines à mémoire distribuée soit proposée.

Dans le domaine de l'optimisation combinatoire nous avons présenté un algorithme de Branch and Bound sur un hypercube à granularité fine dont la principale caractéristique est l'accès des processeurs à quelques informations sur l'état global du système, ce qui conduit à un équilibrage optimal des tâches, obtenu grâce à un mécanisme d'allocation dynamique pour les processeurs. Visant la résolution de problèmes d'optimisation combinatoire sur des architectures à grain fin, cet algorithme est en phase finale d'implantation sur une Connection Machine.

La parallélisation d'une telle méthode sur des architectures à gros grain pose des difficultés diverses causées par une presque stricte dépendence entre deux itérations consécutives. La connaissance par la totalité des processeurs des informations obtenues par l'un d'entre eux peut être fondamentale pour l'efficacité de l'algorithme, puisque, selon l'équilibrage des tâches, quelques processeurs peuvent travailler d'avantage que d'autres.

Finalement, nous avons abordé la technique du Recuit Simulé, aussi très répandue en optimisation combinatoire. Nous avons prouvé que la convergence asymptotique vers une solution optimale de cet algorithme est moins rapide que celle de la technique - beaucoup plus simple - de Recherche Aléatoire Locale. Nous avons aussi démontré que, pour des variables bien définies, ne dépendant que de l'application en vue, la probabilité d'aboutissement de l'algorithme de Recuit Simulé dans un nombre fini d'itérations (i.e., dans tous les cas pratiques) est bornée supérieurement. Il serait très intéressant de calculer exactement cette borne pour des applications de petite taille en vue d'une meilleure compréhension de cette méthode.

Nous avons étendu ces résultats au mode parallèle puisque, comme pour le Branch and Bound, beaucoup d'études portent sur l'utilisation de machines parallèles pour la diminution du temps de calcul, assez important, de l'algorithme de Recuit Simulé.

REFERENCES

[Abd88] : T.S.Abdelrahman et T.N.Mudge, "Parallel branch and bound algorithms on hypercube multiprocessors", dans *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox (Ed.), ACM Press, pp 1492-1499

[Aho74] : A.Aho, J.Hopcroft et J.Ullman, *The Design and Analysis of Computer Algorithms*, Addison-Wesley Publishing Co. (1974)

[Akl84a] : S.G.Akl, "Optimal parallel algorithms for computing convex hulls and for sorting", *Computing* 33 (1984), pp 1-11

[Akl84b] : S.G.Akl, "An optimal algorithm for parallel selection", *Information Processing Letters* 19 (1984), pp 47-50

[Akl85] : S.G.Akl, *Parallel Sorting Algorithms*, Academic Press, Inc. (1985)

[Akl87] : S.G.Akl et N.Santoro, "Optimal parallel merging and sorting without memory conflicts", *IEEE Trans. on Comp.*, vol. c-36, n° 11, Novembre 1987, pp 1367-1369

[Akl89] : S.G.Akl, *Design and Analysis of Parallel Algorithms*, Prentice-Hall International, Inc. (1989)

[And87] : S.Anderson et M.C.Chen, "Parallel branch and bound algorithms on the hypercube", dans *Hypercube Multiprocessors 1987*, M.T.Heath (Ed.), SIAM Press, Philadelphia, PA, pp 309-317

[Blu73] : M.Blum, R.W.Floyd, V.Pratt, R.L.Rivest et R.E.Tarjan, "Time bounds for selection", *JCSS*, 7 (1973), pp 448-461

[Bon84] : E.Bonomi et J.-L.Lutton, "The N-City travelling salesman problem : statistical mechanics and the Metropolis algorithm", *SIAM Revue*, 26 (1984) 551-568

[Bra88] : B.Braschi, "Solving the traveling salesman problem with simulated annealing techniques on a concurrent supercomputer", *Rapport de Recherches* RR 752-I, Novembre 1988, IMAG, Grenoble (France)

[Bra89a] : B.Braschi, A.G.Ferreira et J.Zerovnik, "On the behavior of simulated annealing", *Rapport de Recherches* RR 89-10, Septembre 1989, LIP-IMAG, Lyon (France)

[Bra89b] : B.Braschi, A.G.Ferreira, J.Zerovnik, "On the asymptotic behavior of parallel simulated annealing", dans *Proceedings of the International Conference on Parallel Computing 89*, à paraître

[Col88] : R.Cole, "Parallel merge sort", *SIAM J. Comput.*, Vol. 17, n° 4, Août 1988, pp 770-785

[Coo82] : S.Cook et C.Dwork, "Bounds on the time for parallel RAM's to compute simple functions", dans *Proceedings of the 14th Ann. ACM Symp. on Theory of Computing*, 1982, pp 231-233

[Cos88] : M.Cosnard, A.G.Ferreira et H.Herbelin, "The two-list algorithm for the knapsack problem on a FPS T20", *Parallel Computing* 9 (1988/89), pp 385-388

[Cos89a] : M.Cosnard, J.Duprat et A.G.Ferreira, "Complexity of selection in X + Y", *Theoretical Computer Science* 67 (1989), pp 115-120

[Cos89b] : M.Cosnard et A.G.Ferreira, "Parallel algorithms for searching in X + Y", dans *Proceedings of the International Conference on Parallel Processing 89*, Vol. 3 - Algorithms & Applications, F.Ris et P.M.Kogge (Eds.), Penn State University Press, pp 16-19

[Cos89c] : M.Cosnard, J.Duprat et A.G.Ferreira, "The complexity of searching in X + Y and other multisets", *Information Processing Letters*, à paraître

[Cos89d] : M.Cosnard, J.Duprat et A.G.Ferreira, "Known and new results on selection, sorting and searching in X + Y and sorted matrices", *Rapport de recherches* RR 89-9, Septembre 1989, LIP-IMAG, Lyon (France), *soumis pour publication*

[Deh89a] : F.Dehne, A.G.Ferreira et A.Rau-Chaplin, "Parallel branch and bound on fine grained hypercube multiprocessors", dans *Proceedings of IEEE International Workshop on Tools for Artificial Intelligence*, Herndon, VA (USA), 1989, pp 616-622

[Deh89b] : F.Dehne, A.G.Ferreira et A.Rau-Chaplin, "Parallel fractional cascading on a hypercube multiprocessor", dans *Proceedings of the 27th Ann. Allerton Conference on Communication, Control and Computing*, 09/89, Allerton (Ill), USA, à paraître

[Don88] : J.G.Donnet et D.B.Skillicorn, "Code partitioning by simulated annealing ", *Parallel Processing and Applications*, E.Chiricozzi et A.D'Amico (Eds.), North-Holland (1988), pp 303-308

[Fel85] : E.Felten, S.Karlin et S.W.Otto, "The travelling salesman problem on a hypercubic, MIMD computer", dans *Proceedings of the 1985 International Conference on Parallel Processing*, Août 20-23, 1985, pp 6-10

[Fel88] : E.W.Felten, "Best-first branch and bound on a hypercube", dans *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox, ed., ACM Press, pp 1500-1504

[Fer88] : A.G.Ferreira, "Efficient parallel algorithms for the knapsack problem", dans *Parallel Processing*, M. Cosnard, M.H.Barton et M.Vanneschi, (Eds.), IFIP - North Holland, 1988, pp 169-179

[Fer89a] : A.G.Ferreira, "The knapsack problem in parallel architectures", dans *Parallel & Distributed Algorithms*, M.Cosnard et al. (Eds.), North-Holland 1989, pp 145-152

[Fer89b] : A.G.Ferreira, "A parallel time/hardware tradeoff T.H = $O(2^{n/2})$ for the knapsack problem", *IEEE Trans. on Comp.*, à paraître

[Fer89c] : A.G.Ferreira et J.Zerovnik, "A bound for the probability of success of simulated annealing", *Preprint Series Math. Dept.*, University E.K. Ljubljana, **27** (1989), n° 272, pp 81-87

[Fer89d] : A.G.Ferreira et M.Gastaldo, "Implementing sorting on a hypercube", dans *Hypercube and Distributed Computers*, F.André et J.-P.Verjus (Eds.), INRIA-North Holland 1989, pp 359-360

[Fer89e] : A.G.Ferreira et M.Gastaldo,"Expérimentations de deux algorithmes de tri sur l'hypercube FPS T-20", *Rapport de Recherches* RR 89 -11, Septembre 1989, LIP-IMAG, Lyon (France), à paraître dans *La Lettre du Transputeur*

[Fly66] : M.J.Flynn, "Very high-speed computing systems", *Proceedings IEEE* 54 (12), 1966, pp 1901-1909

[Fr75] : M.L.Fredman, "Two applications of a probabilistic search technique : sorting X + Y and building balanced search trees", dans *Proc. 7-th Annual ACM Symp. on Theory of Computing*, (Mai 1975), ACM, 1975, pp 240-244

[Fre82] : G.N.Frederickson et D.B.Johnson, "The complexity of selection and ranking in X + Y and matrices with sorted columns", *J. CSS* 24 (1982), pp 197-208

[Fre84] : G.N.Frederickson et D.B.Johnson, "Generalized selection and ranking : sorted matrices", *SIAM J. Comput.* 13 (1), Fevrier 1984, pp 14-30

[Gar79] : M.R.Garey et D.S.Johnson, *Computers and Intractability : A Guide to the Theory of NP-Completeness*, W.H.Freeman (1979)

[Har75] : L.H.Harper, T.H.Payne, J.E.Savage et E.Straus, "Sorting X+Y", *Comm. ACM* ( ) (1975), pp 347-349

[Hil87] : D.Hillis, "Un ordinateur parallèle : la Connection Machine", *Pour la Science*, Août 1987, pp 86-94

[Hor74] : E.Horowitz et S.Sahni, "Computing partitions with applications to the knapsack problem", *J. of the ACM*, vol 21, n° 2, Avril 1974, pp 277-292

[Hua89] : S.R.Huang et L.S.Davis, "Parallel iterative A* search : an admissible distributed heuristic search algorithm", *International Joint Conference on Artificial Intelligence 89*, preprint

[Jo89] : S.L.Johnsson et C.T.Ho, "Optimum broadcasting and personalized communication in hypercubes", *IEEE Trans. on Computers*, vol c-38, n° 9, Septembre 1989, pp 1249-1268

[Joh78a] : D.B.Johnson et S.D.Kashdan, "Lower bounds for selection in X+Y and other multisets", *J. ACM* 25 (5) (1978), pp 556-570

[Joh78b] : D.B.Johnson et T.Mizoguchi, "Selecting the k-th element in X+Y and X1+X2+...+Xm", *SIAM J. Comput.* 7 (2) (1978), pp 147-153

[Ka84] : E.D.Karnin, "A parallel algorithm for the knapsack problem", *IEEE Transactions on Computers*, vol c-33, n° 5, Mai 1984, pp 404-408

[Kar72] : R.Karp, "Reducibility among combinatorial problems", dans *Complexity of Computer Computations*, R.E.Miller et J.W.Thather (Eds.), Plenum Press (1972), pp 85-104

[Kir83]: S.Kirkpatrick, C.D.Gelatt et M.P.Vecchi, "Optimization by simulated annealing", *Science*, Mai 1983, vol. 220, N. 4598, pp 671-680

[Knu72] : D.E.Knuth, *The Art of Computer Programming, vol. III, Sorting and Searching*, Addison-Wesley Publishing Co. (1972)

[Kru83] : C.P.Kruskal, "Searching, merging and sorting in parallel computation", *IEEE Trans. on Comp.*, c-32, 1983, pp 942-946

[Kru85] : C.P.Kruskal, *The architecture of parallel computers*, NATO ASI Series, vol. F14, M.Broy (ed.), Springer Verlag (1985)

[Laa87] : P.J.M.Laarhoven et E.H.L.Aarts, *Simulated Annealing, Theory and Applications*, D. Reidel Publishing Company (1987)

[Lai84] : T.H.Lai et S.Sahni, "Anomalies in parallel branch-and-bound algorithms, *Communications of the ACM*, Vol. 27, n° 6, Juin 1984, pp 594-602

[Lei84] : F.T.Leighton, "Tight bounds on the complexity of parallel sorting", *Proc. 16th Annual ACM Symp. Theory of Computing*, Washington, D.C., Mai 1984, pp 71-80

[Li86] : G.J.Li et B.W.Wah, "Coping with anomalies in parallel branch and bound algorithms", *IEEE Trans. on Comp.*, vol. c-35, n° 6, Juin 86, pp 568-573

[Lin85] : N.Linial et M.Saks, "Searching ordered structures", *Journal of Algorithms* 6 (1985), pp 86-103

[Lut86] : J.-L.Lutton et E.Bonomi , "Simulated annealing algorithm for the minimum weighted perfect euclidean matching problem", *R.A.I.R.O. Recherche Opérationnelle*, vol. 20, No 3, Août 1986, pp 177-197

[Mar82] : A.Marsland et M.Campbell, "Parallel search of strongly ordered game trees", *Computing Surveys*, vol. 14, n° 4, Décembre 82, pp 533-551

[Mir84] : A.Mirzaian, "Channel routing in VLSI", dans *Proceedings of the 16-th Annual ACM Symposium on Theory of Computing*, Washington D.C. (USA), Avril 1984, pp 101-107

[Mir85] : A.Mirzaian et E.Arjomandi, "Selection in X+Y and matrices with sorted rows and columns", *Inf. Proc. Letters* 20 (1985), pp 13-17

[Nas81] : D.Nassimi et S.Sahni, "Data broadcasting in SIMD computers", *IEEE Trans. on Comp.*, Vol. c-30, n° 2, 1981, pp 101-106

[New88] : M.Newborn, "Unsynchronized iteratively deepening parallel alpha-beta search", *IEEE Trans. on PAMI*, vol. 10, n° 5, Septembre 88, pp 687-694

[Nil80] : N.J.Nilsson, *Principles of Artificial Intelligence*, Tioga Publishing Co. (1980)

[Par88] : R.P.Pargas et D.E.Wooster, "Branch and bound algorithms on a hypercube", dans *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox (Ed.), ACM Press, pp 1514-1519

[Pre78] : F.P.Preparata, "New parallel-sorting schemes", *IEEE Trans. on Comp.*, vol. c-27, n° 7, Juillet 1978, pp 669-673

[Qui87] : M.J.Quinn, "Implementing best-first branch and bound algorithms on hypercube multicomputers", dans *Hypercube*

Multiprocessors 1987, M.T.Heath (Ed.), SIAM Press, Philadelphia, PA, pp 318-326

[Rom84] : F.Romeo, A.Sangiovanni-Vincentelli et C.Sechen, "Research on simulated annealing at Berkeley", dans *Proceedings of the International Conference on Computer Design*, Octobre 1984, pp 652-657

[Rou87] : C.Roucairol, "Du séquentiel au parallèle : la recherche arborescente et son application à la programmation quadratique en variables 0.1", *Thèse d'Etat*, Université Paris 6, Juin 1987

[Sch81] : R.Schroeppel et A.S.Shamir, "A $T=O(2^{n/2})$, $S=O(2^{n/4})$ algorithm for certain NP-Complete problems", *SIAM J. Comput.*, 10 (3), 1981, pp 456-464

[Scw88] : K.Scwan, J.Gawkowski et B.Blake, "Process and workload migration for a parallel branch and bound algorithm on a hypercube multicomputer", dans *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox (Ed.), ACM Press, pp 1520-1530

[Sie77] : H.J.Siegel, "The universality of various types of SIMD machine interconnection networks", dans *Proceedings of the Fourth Annual Symposium on Computer Architecture*, Silver Spring, Md., Mars 1977

[Sni82] : M.Snir, "On parallel searching", dans *Proceedings of the ACM Symp. on Principles of Distributed Computing*, 1982, pp 242-253

[Ste88] : R.E.Stearns et H.B.Hunt III, "Power indices and easier NP-Complete problems", *Rapport de recherches* RR 88 - 27, 1988, University at Albany, SUNY

[Sto71] : H.S.Stone, "Parallel processing with the perfect shuffle", *IEEE Trans. on Comp.*, Vol. c-20, n° 2, Fevrier 1971, pp 153-161

[Sto87] : H.S.Stone, *High-Performance Computer Architecture*, Addison-Wesley Publishing Co., (1987)

[Tsu88] : F.S.Tsung et M.H.Ma, "A dynamic load balancer for a parallel branch and bound algorithm", dans *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications*, 1988, G.Fox (Ed.), ACM Press, pp 1505-1513

[Usu87] : H.Usui, M.Yamashita, M.Imai et T.Ibaraki, "Parallel searches of game trees", *Systems and Computers in Japan*, vol. 18, n° 8, 1987, pp 97-109

[Vys88] : J.Vyscoc, *communication personnelle*, Juillet 1988

[Wah85] : B.W.Wah, G.J.Li et C.F.Yu, "Multiprocessing of combinatorial search problems", *Computer*, Juin 85, pp 93-108

[Win84] : P.H.Winston, *Artificial Intelligence*, Addison-Wesley Publishing Co. (1984)

[Zer88] : J.Zerovnik, "A comparison of asymptotic behavior of two randomized heuristic approaches", French-Israeli Binational Conference on Combinatorics and Algorithms, Jerusalem (Is), 14-17/11/88, *soumis pour publication*

INPG département des études doctorales

AUTORISATION de SOUTENANCE

VU les dispositions de l'Arrêté du 23 novembre 1988 relatif aux Etudes doctorales

VU les rapports de présentation de

- Monsieur Selim AKL
- Monsieur Robert CORI

Monsieur GALVAO FERREIRA Afonso

est autorisé(e) à présenter une thèse en soutenance en vue de l'obtention du diplôme de DOCTEUR de L'INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE, spécialité "Informatique"

Fait à Grenoble, le 10 Janvier 1990

Pour le Président de l'I.N.P.-G.
et par délégation,
le Vice-Président
P. VENNEREAU

Résumé :

Le problème de la recherche d'un élément dans un ensemble donné est un des problèmes fondamentaux en informatique non-numérique, lié, par exemple, aux bases de données, à la compilation, à l'optimisation combinatoire, etc ... Dans cette thèse nous abordons le problème de la recherche dans des ensembles ordonnés et quelques problèmes qui lui sont liés.

Une matrice est dite triée si elle possède une relation d'ordre total sur chaque ligne et chaque colonne. Un cas particulier est l'ensemble des matrices définies par X + Y, la somme cartésienne de deux vecteurs triés. Après un tour d'horizon des problèmes de la sélection, de la recherche et du tri sur des ensembles de la forme X + Y, nous démontrons des bornes inférieures et introduisons des bornes supérieures pour les problèmes de la recherche et de la sélection. Nous proposons, en outre, plusieurs algorithmes parallèles pour la recherche dans X + Y.

Ensuite nous montrons comment appliquer ces résultats à la résolution en parallèle du problème de décision du Sac-à-Dos, où, étant donnés plusieurs objets et un sac de capacité définie à remplir, on veut décider s'il existe une combinaison des objets qui remplit exactement le sac. Des algorithmes avec une accélération optimale sont introduits pour divers modèles de machines parallèles à mémoire partagée et distribuée. Pour ce problème les approches existantes se ramènent soit à la recherche dans X + Y, soit à la traversée d'un espace combinatoire à l'aide de la technique du Branch & Bound.

Cette dernière technique est aussi discutée dans ce travail, où nous présentons un algorithme de Branch & Bound en vue de la résolution de problèmes d'optimisation combinatoire sur la Connection Machine. Enfin, toujours dans le domaine de l'optimisation combinatoire, nous étudions le comportement de l'algorithme de Recuit Simulé. Nous prouvons que, malgré la confiance qu'il inspire et sa grande utilisation pour des applications en recherche et développement, sa convergence vers une solution est moins rapide que celle de la technique - beaucoup plus simple - de Recherche Aléatoire Locale. De plus, nous donnons une borne supérieure pour la probabilité que la recherche basée sur le Recuit Simulé aboutisse.

Mots-clés : Recherche, Recherche en Parallèle, X + Y, Optimisation Combinatoire en Parallèle, Problème du Sac-à-Dos, Modèles Parallèles

Abstract :

The problem of searching a given set for a given element is one of the most important problems in Computer Science. Its applications can be found, for instance, in data-base theory, compilers, combinatorial optimisation, etc... In this thesis we deal with the searching problem in sorted and partial sorted sets and with some other related problems.

A matrix is named sorted matrix if there is a total order relation over its rows and columns. One particular case of such matrices is the set of matrices defined by X + Y, the cartesien sum of two sorted vectors. After a survey on the selection, searching and sorting problems in X + Y, we prove some lower bounds and show some upper bounds for the searching and selection problems in X + Y. Furthermore, we propose several parallel algorithms for searching in X + Y.

Then we show how to use such results to solve the Knapsack Decision Problem, where several objects are available to fill a bounded knapsack, and we want to decide whether there exists a combination of the objects that fill the Knapsack exactly. Optimal speedup algorithms are given that run in different models of shared and distributed memory parallel machines. For such a problem, existing approaches reduce whether to searching in X + Y, or to the traversal of a combinatorial space with the help of the Branch & Bound technique.

Such a technique is also discussed in this thesis. We present a parallel Branch & Bound algorithm looking forward to solving combinatorial optimisation problems in the Connection Machine. At last, still related to combinatorial optimisation, we study the behavior of the Simulated Annealing algorithm. We prove that, in spite of its widespread acceptance and use, its convergence towards a solution is less rapid than the convergence of Random Local Search. Moreover, we introduce an upper bound for the likelyhood that Simulated Annealing searching is successful.

Key-words : Searching, Parallel Searching, X + Y, Parallel Combinatorial Optimisation, Knapsack Problem, Parallel Models