

République Algérienne Démocratique et Populaire

Ministère de l'enseignement supérieur et de la recherche scientifique

Université colonel HADJ LAKHDAR –BATNA

Faculté des sciences

Département d'informatique

Mémoire

Présenté

En vue de l'obtention du diplôme :

Magister en informatique

Spécialité : Ingénierie des Systèmes Informatiques (ISI)

Par

Rédha LOUCIF

Titre :

Parallélisation d'Algorithmes d'Optimisation Combinatoire

Soutenu le : 13/01/2014

Jury

Pr : Azzedine BILAMI	(Président)	Professeur, université de Batna
Pr : Abdelmadjid ZIDANI	(Examinateur)	Professeur, université de Batna
Pr : Mohamed BATOUCHE	(Examinateur)	Professeur, université de Constantine
Pr : Nour-Eddine BOUGUECHAL	(Rapporteur)	Professeur, université de Batna
Dr : Rachid SEGHIR	(Co-Rapporteur)	Maître de conférences, université de Batna



Remerciements

Je remercie Dieu pour son immense générosité ;

Je remercie fortement mes encadreurs :

Pr : Nour-Eddine BOUGUECHAL Professeur, université de Batna pour son encadrement.

Dr : Rachid SEGHIR Maître de conférences, université de Batna pour son aide précieuse et son encouragement.

Je remercie les membres du jury qui ont accepté de juger et d'évaluer ce travail :

Pr : Azzedine BILAMI Professeur, université de Batna ;

Pr : Abdelmadjid ZIDANI Professeur, université de Batna ;

Pr : Mohamed BATOUCHE Professeur, université de Constantine ;

Un grand merci à mon cher père pour son aide.

Je remercie Mr Saci Abdellah pour son aide.

Dédicaces

Je dédie ce modeste mémoire à :

**Mes chers parents qui sans leurs prières, je n'aurais jamais achevé mon travail ;*

**ma chère épouse pour son support et sa patience ;*

**mes précieux enfants Ayoub, Sarra, Zakaria ;*

**mes frères et sœurs ;*

**mes encadreurs et enseignants ;*

**tous ceux qui me connaissent.*

المخلص :

إن تقنية التقسيم و التقييم بالإنجليزية (B&B Branch and Bound) تعتبر من الطرق الدقيقة لإستخلاص الحلول لمسائل البحث عن الاندماج الأمثل، لأنها تعتمد على التعداد "الظاهري أو الضمني" لكل الحلول الممكنة بإنشاء و تفحص شجرة التقسيم و التقييم (B&B). أما بالنسبة للمسائل الخطية فإن طريقة سمبلاكس تعتبر الأفضل لتقييم و تسيير ال(B&B).

في عملنا هذا سنوضح مختلف هذه التقنيات و تطبيقاتها "المتسلسلة أو المتوازية" لحل المسائل الخطية ذات المتغيرات بأعداد صحيحة، وسنناقش أداء خوارزميتنا المتوازية من حيث الزمن المستغرق في الحساب.

Résumé

La technique Branch and Bound (Séparation et évaluation) constitue une des méthodes exactes d'extraction de solutions de l'optimisation combinatoire, car elle se base sur l'énumération, implicite ou explicite, de toutes les solutions en construisant et parcourant l'arbre Branch and Bound (B&B). Pour résoudre un Problème Linéaire, l'algorithme du simplexe constitue une méthode forte qui est utilisée pour l'évaluation et le déroulement du B&B. Dans ce travail nous présentons ces différents algorithmes ainsi que nos implémentations séquentielle et parallèle du B&B pour résoudre un Problème Linéaire en Nombres Entiers. Nous discutons également les performances de notre algorithme parallèle en termes d'accélération du temps de calcul.

Table des Matières

Résumé	III
Table des Matières	IV
Liste des Figures.....	VIII
Liste des Tableaux :.....	X
Introduction Générale	1
Chapitre1 Optimisation combinatoire.....	3
1.1 Introduction	4
1.2 Notion d'optimisation combinatoire	4
1.3 Formulation de problèmes d'optimisation combinatoire	5
1.4 Stratégie d'extraction de solution	6
1.4.1 Solution Exacte	6
1.4.2 Les Heuristiques.....	7
1.4.3 Algorithmes hybrides.....	7
1.5 Programmation Linéaire	8
1.5.1 Programmation linéaire en nombres entiers PLNE	10
1.5.2 La méthode du simplexe.....	10
1.6 Séparation et évaluation (Branch and Bound).....	12
1.7 Source de parallélisme dans l'algorithme Branch & Bound	16
1.7.1 Stratégies pour la parallélisation	16
1.7.2 Classifications des algorithmes parallèles B&B	20
1.7.3 Anomalies dues à l'accélération	26
1.7.4 Logiciels Banch and Bound	27
1.8 Les avantages et inconvénients pour B&B Parallèle et séquentiel.....	28
1.8.1 Les points pour B&B séquentiel.....	28
1.8.2 Points pour B&B parallèle.....	28
1.9 Conclusion.....	29
Chapitre 2 Notions du parallélisme	30
2.1 Introduction	31
2.2 Algorithme parallèle	31
2.3 Les Machines Parallèles	32
2.3.1 Les machines parallèles à mémoire partagée	32
2.3.2 Les machines parallèles à mémoire distribuée.....	33

2.4	Modèles de machines parallèles.....	34
2.4.1	Le Modèle PRAM	35
2.4.2	Les modèles de bas niveau	36
2.4.3	Modèles à mémoires hiérarchique.....	37
2.4.4	Modèles réseau	37
2.5	Conception d'algorithme parallèle	37
2.5.1	Partitionnement.....	38
2.5.2	Communication.....	38
2.5.3	Agglomération.	38
2.5.4	Mapping.	38
2.6	Sources de parallélisme	39
2.6.1	Parallélisme de contrôle	39
2.6.2	Parallélisme de données.....	40
2.6.3	Parallélisme de Flux	40
2.7	Mesure de performance	41
2.7.1	Accélération	41
2.7.2	Accélération relative	42
2.7.3	Efficacité.....	42
2.7.4	Iso-efficacité.....	42
2.8	Complexité d'un algorithme parallèle	42
2.8.1	Complexité en temps.....	43
2.8.2	Complexité en communication.....	44
2.8.3	Complexité en mémoire	44
2.9	Conclusion.....	45
Chapitre 3 Présentation d'outils de parallélisation OpenMP & MPI		46
3.1	Introduction	47
3.2	Open MP (Open Multi Processing)	47
3.2.1	Historique	47
3.2.2	C'est quoi OpenMP ?	47
3.2.3	Concepts Généraux.....	48
3.2.4	Structure OpenMP	49
3.2.5	Principe d'OpenMP	50
3.2.6	Les Variables de contrôle Interne (ICVs)	52

3.2.7	Syntaxe générale d'une directive	52
3.2.8	Les Variables d'environnements.....	56
3.2.9	Construction des régions parallèles.....	57
3.3	MPI (Message Passing Interface).....	58
3.3.1	Concepts de l'échange de messages	58
3.3.2	Les objectifs de MPI	59
3.3.3	Environnement	60
3.3.4	Communication Point à point	60
3.3.5	Communication collective	61
3.3.6	Optimisations.....	62
3.3.7	Types de données dérivées.....	63
3.3.8	Topologie des processus.....	65
3.3.9	Groupes et Communicateurs.....	66
3.3.10	Installation et exécution de programme MPI (exemple MPICH).....	68
3.3.11	Les routines MPI	70
3.4	Conclusion.....	70
Chapitre 4 Nouvelle approche de parallélisation et implémentation		71
4.1	Introduction	72
4.2	Approche de parallélisation proposée.....	72
4.2.1	La stratégie de parcours de l'arbre.....	73
4.2.2	Gérer le grain de parallélisme.....	73
4.2.3	L'équilibre de charge	73
4.2.4	Gestion optimale de la mémoire.....	73
4.2.5	Assurer la communication entre les processeurs utilisés.....	73
4.2.6	Parallélisme de contrôle ou de données	74
4.3	La loi d'Amdahl	75
4.4	Complexité de l'algorithme B&B parallèle.....	76
4.4.1	Complexité en temps de calcul.....	76
4.4.2	Complexité en mémoire	77
4.4.3	Complexité en communication.....	77
4.5	Implémentation Parallèle de l'algorithme Branch and Bound	77
4.5.1	Etude expérimentale	78
4.5.2	Génération de l'ensemble de test	78

4.5.3	Résultats de l'expérimentation.....	79
4.5.4	Discutions globale.....	84
4.6	Conclusion.....	85
Conclusion et Perspectives.....		86
Bibliographie.....		88

Liste des Figures

Figure 1: Méthodes de recherche de solutions.	8
Figure 2 : Exemple de polyèdre de pleine dimension sur R^3	9
Figure 3: S Ensemble de solutions – S1 Ensemble de solutions entières.	10
Figure 4: Simplexe.	11
Figure 5: Pseudo code de l'algorithme du simplexe.	12
Figure 6: Déroulement de l'algorithme Branch and Bound.	15
Figure 7: Algorithme Séparation et évaluation (Branch and Bound).....	16
Figure 8: Modèle Multiparamétrique	23
Figure 9 : Modèle parallèle de l'exploration de l'arbre.....	24
Figure 10: Evaluation parallèles des limites.	24
Figure 11: Architecture interne Processeur Intel I7	31
Figure 12: Machine parallèle à mémoire partagée.	33
Figure 13 : Machine parallèle à mémoire distribuée.....	33
Figure 16: Différentes architectures de réseau d'interconnexion.....	34
Figure 15: Méthode pour la conception d'algorithmes parallèles.	39
Figure 16: Parallélisme de Contrôle.....	40
Figure 19: Parallélisme de flux.	41
Figure 18: Ordre de complexité.	43
Figure 19: Exécution des tâches OpenMP.	48
Figure 20 : Partage de travail OpenMP.....	49
Figure 21 : Structure d'OpenMP.....	50
Figure 22: Hiérarchie des constructeurs de types en MPI.....	65
Figure 23: schéma de communication dans MPI	67
Figure 24: Algorithme Parallèle Branch and Bound.....	75

Figure 25: Accélération avec différents type de parcours d'arbre	80
Figure 26: Parcours DeFS avant de trouver la solution optimale.	81
Figure 27: Accélération avec augmentation de threads.....	82
Figure 28: Nombre de Nœuds avec augmentation de threads.	83
Figure 29: Accélération en fonction des nombres de processeurs.	83
Figure 30: Amélioration progressive de l'accélération.....	84

Liste des Tableaux :

Tableau 1 : Paramètres des algorithmes B&B.	21
Tableau 2 : Récapitulatif des clauses OpenMP	56
Tableau 3: Type de données de MPI.	64
Tableau 4 : Temps d'exécution pour différentes stratégies de parcours de l'arbre....	79

Introduction Générale

Les problèmes d'optimisation combinatoire font partie de notre vie quotidienne implicitement plus qu'explicitement, car ces problèmes font partie intégrante de grand nombre de tâches quotidiennes tel que le calcul du plus court chemin (exemple calcul des chemins avec les logiciels de GPS) ou du sac à dos (maximum de poids et d'objets à emporter pour un voyage) ou de l'affectation quadratique (exemple la distribution des dépôts). La taille de ces problèmes ne cesse d'augmenter, et la puissance de calcul exigée pour les traiter aussi [1].

L'évolution de l'outil informatique et de la puissance de calcul en vitesse pure a atteint ses limites en raison des contraintes matérielles, la tendance actuelle est le développement en largeur en direction de la multiplication des cœurs des processeurs. Dans ce cadre, et pour une amélioration des temps d'exécution des programmes, la création d'algorithmes parallèles [2] est un souci majeur pour l'optimisation de la résolution des problèmes.

Dans ce contexte, nous nous intéressons à l'étude de la méthode Branch and Bound [3] (séparation et évaluation) qui est l'une des méthodes les plus utilisées dans la résolution exacte des problèmes d'optimisation combinatoire, tout en se concentrant sur l'aspect parallèle de l'algorithme.

Le principe de l'algorithme Branch and Bound (B&B) est de faire la division de l'ensemble des solutions selon des critères de branchements, puis l'évaluation qui consiste à calculer la valeur optimale dans les sous-ensembles. L'évaluation va nous permettre d'élaguer les branches dont la valeur optimale ne satisfait pas les critères recherchés. La solution optimale est trouvée quand tous les sous-ensembles sont élagués ou explorés.

Dans la littérature, différents travaux traitent du problème de parallélisation de l'algorithme B&B [4] [5] [6] [7]. Plusieurs classifications de ces algorithmes ont été proposées [8] [9] [10].

Dans ce mémoire, nous avons proposé une approche parallèle de l'algorithme Branch and Bound qui respecte et satisfait les critères de parallélisation sur le plan de parcours de

l'arbre de recherche et de la distribution de charge de travail sur les processeurs. Une étude expérimentale nous a permis de donner les résultats de l'accélération obtenue sur différentes tailles des problèmes traités.

Ce mémoire est organisé comme suit :

Le premier chapitre donne un aperçu sur l'optimisation combinatoire et les méthodes d'extraction de solutions, et Branch and Bound entre autres.

Le deuxième chapitre aborde différentes notions du parallélisme, tels que les différents types de machines parallèles, la méthode de conception d'algorithmes parallèles, la complexité des algorithmes et les mesures de performances des algorithmes.

Le troisième chapitre donne un aperçu des outils de parallélisation en l'occurrence, OpenMP et MPI.

Le quatrième chapitre présente notre approche de parallélisation de l'algorithme B&B basé sur différents critères de l'algorithme en plus d'une implémentation et des résultats expérimentaux.

Chapitre1

Optimisation combinatoire

1.1 Introduction

L'optimisation combinatoire est l'une des plus jeunes branches et les plus actives des mathématiques discrètes de ces dernières années. Dans ce chapitre nous allons donner quelques définitions des notions de l'optimisation combinatoire et de stratégies d'extraction de problèmes et parmi elle la méthode Branch And Bound sur laquelle nous allons nous pencher avec plus de détails.

1.2 Notion d'optimisation combinatoire

" Combinatoire " désigne la discipline des mathématiques concernée par les structures " discrètes " ou " finies ". Citons quelques branches de cette discipline : la théorie des graphes, la combinatoire énumérative, les problèmes de dénombrement, la combinatoire polyédrale, l'optimisation combinatoire, etc. Les frontières entre ces branches ne sont pas hermétiques, les différentes branches expriment plutôt des orientations méthodologiques différentes.

L'" optimisation ", ou " programmation mathématique " (Mathematical Programming) sont des termes utilisés pour recouvrir toutes les méthodes qui servent à déterminer l'optimum d'une fonction avec (ou sans) contraintes. Cette fonction modélise le choix optimal. On optimise déjà à l'école, quand on apprend comment déterminer l'optimum (minimum ou maximum) d'une fonction différentiable. Les ingénieurs, les économistes, la nature, font souvent des choix optimaux (ou quasi-optimaux), d'où l'importance de l'optimisation dans les sciences, qu'elles soient techniques, mathématiques, physiques, informatiques, économiques, naturelles, etc.

L' " **Optimisation combinatoire** " consiste à trouver le meilleur entre un nombre fini de choix. Autrement dit, minimiser une fonction, avec contraintes, sur un ensemble fini. Quand le nombre de choix est exponentiel (par rapport à la taille du problème), mais que les choix et les contraintes sont bien structurés, des méthodes mathématiques doivent et peuvent intervenir, comme dans le cas " continu ", pour permettre de trouver la solution en temps polynomial (par rapport à la taille du problème). Alors, l'optimisation combinatoire consiste à développer des algorithmes polynomiaux et des théorèmes sur des structures discrètes qui permettent de résoudre ces problèmes [11].

Ou bien encore :

L'optimisation combinatoire est une problématique consistant, pour un ensemble de variables prenant des valeurs entières, à déterminer leurs instanciations, éventuellement sous contraintes, correspondant au maximum (ou minimum) d'une fonction de ces variables, OF, généralement appelée fonction objectif. L'ensemble des combinaisons pouvant être prises par ces variables constitue l'espace de recherche du problème. Une combinaison donnée est appelée une solution potentielle ou plus simplement une solution, alors que les instanciations des variables qui maximisent OF sont appelées les solutions optimales du problème. Dans le cas d'une optimisation combinatoire sous contrainte, on définit un ensemble de contraintes sur les variables rendant non valide un sous-ensemble de l'espace de recherche [12]. L'Optimisation combinatoire se situe au carrefour de la Théorie des graphes, de la Programmation mathématique, de l'Informatique théorique (algorithmique et théorie de la complexité) et de la Recherche opérationnelle.

1.3 Formulation de problèmes d'optimisation combinatoire

Comme il existe différentes méthodes pour obtenir une solution optimale à un grand problème d'optimisation combinatoire dans un temps de calcul raisonnable, il est souvent préférable de reformuler le problème. Dans ce contexte il est souvent préférable d'augmenter le nombre de variables et de contraintes. Une bonne formulation du problème consisterait à trouver une fonction objectif qui soit le plus approprié au problème original.

Un problème d'optimisation combinatoire peut se formuler de la manière suivante : données + question.

1 - Données :

- un ensemble fini d'éléments ou configuration.
- une fonction de coût ou poids sur ces éléments.

2 - Question :

Déterminer n éléments de coût minimum ou de poids maximum, ... ?

Quelques exemples de problèmes d'optimisation combinatoire :

- 1- Plus courts chemins.
- 2- Arbres de poids minimum.
- 3- Voyageur de commerce.
- 4- Réseaux de transports.
- 5- Connexité dans un réseau de communication.

1.4 Stratégie d'extraction de solution

Les méthodes d'optimisation sont très nombreuses. Il existe plusieurs classifications possibles, la plus évidente est celle qui divise les méthodes en méthodes de solutions exactes et de solutions utilisant les heuristiques, en plus des méthodes hybrides.

1.4.1 Solution Exacte

Sa difficulté vient du fait que la région de faisabilité du problème combinatoire n'est pas un ensemble convexe. Donc nous devons chercher un trait de points, ou un ensemble de semi lignes disjointes ou des segments de lignes pour trouver une solution optimale.

1.4.1.1 Approche énumérative

La plus simple approche pour résoudre un problème de programmation entière pure est d'énumérer toutes les possibilités (efficace avec les petites instances). Parfois on peut implicitement éliminer un grand nombre de possibilités par la domination ou les arguments de faisabilité. (Le principe de la méthode Branch and Bound).

1.4.1.2 Relaxation Lagrangienne et méthodes de décomposition

Qui consiste à supprimer des contraintes difficiles en les intégrant dans la fonction objectif en la pénalisant si cette contrainte n'est pas respectée.

1.4.1.3 Méthode des plans de coupes

La méthode des plans de coupes consiste à résoudre un problème relaxé, puis à essayer d'ajouter itérativement des contraintes du problème initial, violées par la solution courante, jusqu'à ce qu'il n'y en ait plus.

1.4.2 Les Heuristiques

Trouver une solution rapide avec une certaine tolérance, pour les problèmes dont les algorithmes actuels ne sont pas capables de donner des solutions dans un temps raisonnable [13].

La recherche en utilisant les heuristiques commença avec le concept de « recherche local » où après la construction d'une solution réalisable on améliore la solution par des mouvements locaux. Et pour pouvoir s'éloigner de la solution locale, des travaux se sont inspirés de la nature (propriété de métaux, sélection naturelle, réseaux de neurones et l'apprentissage chez les animaux). Il existe des métaheuristiques qui peuvent être adaptées à plusieurs types de problèmes, on peut citer les plus connues :

- la méthode tabou,
- le recuit simulé,
- les algorithmes génétiques,
- les algorithmes aléatoires,
- les réseaux de neurones artificiels,
- les algorithmes de fourmis,

1.4.3 Algorithmes hybrides

Les heuristiques qui donnent de bonnes solutions entières réalisables, les plans de coupes qui vont lier la relaxation du programme linéaire au problème combinatoire, et tout cela est intégré dans un cadre d'application de recherche dans un arbre Branch and Bound. Ces composants vont assurer l'optimalité de la solution à la fin des calculs. La Figure 1 donne un récapitulatif des différentes méthodes de recherche de solutions.

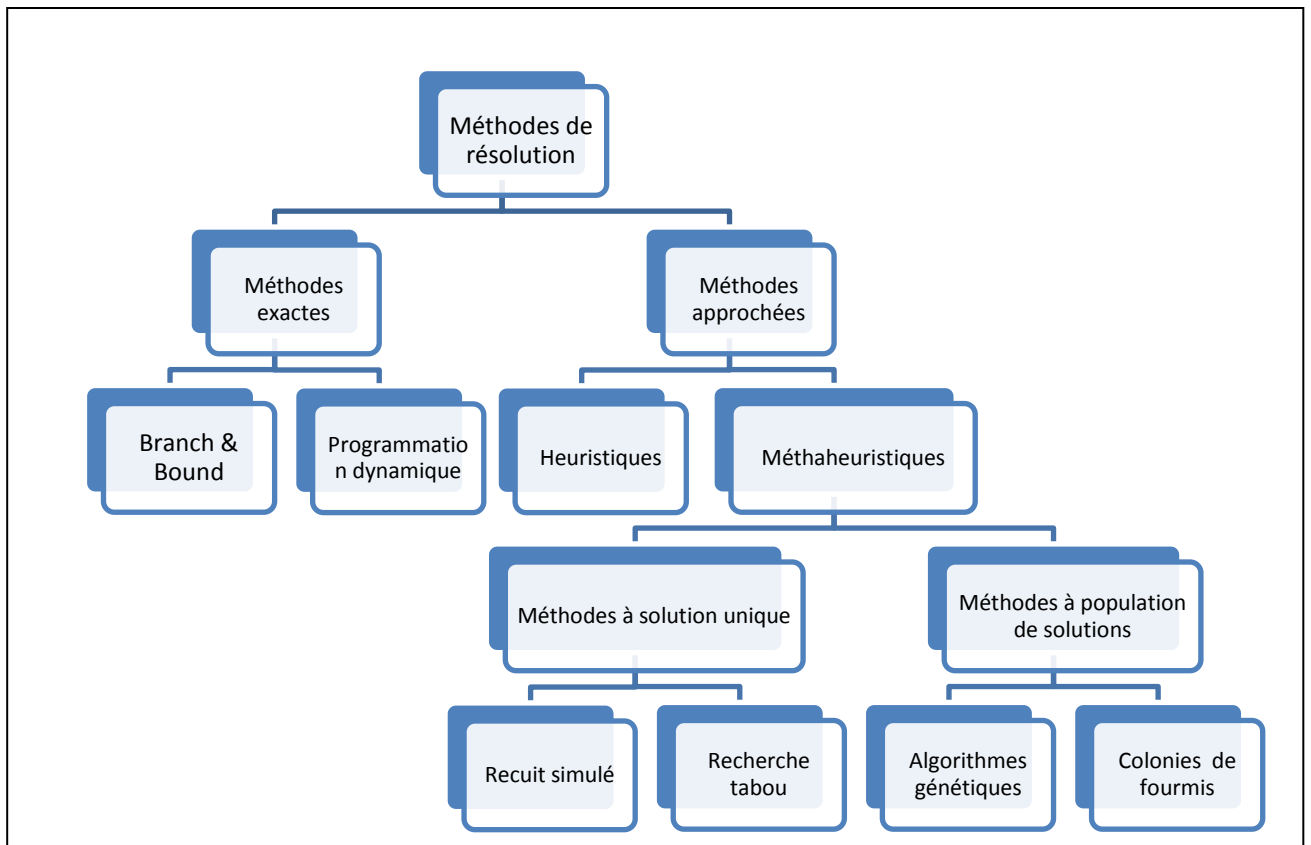


Figure 1: Méthodes de recherche de solutions.

1.5 Programmation Linéaire

"La programmation linéaire concerne la description des interrelations des composants d'un système" [14]. En d'autres termes, la programmation linéaire est l'outil mathématique qui permet d'optimiser (minimiser ou maximiser) une fonction linéaire, appelée fonction objectif, et qui est constituée d'un nombre de variables appelées variables de décision. Ces variables sont soumises à un ensemble de contraintes. On peut modéliser un problème de la programmation linéaire sous la forme :

$$\text{Maximiser } (Z) = c_1x_1 + c_2x_2 + \dots + c_nx_n$$

Soumise aux contraintes :

$$a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1$$

$$a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2$$

.

.

$$a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m$$

$$x_1 \geq 0, x_2 \geq 0 \dots x_n \geq 0$$

L'ensemble des solutions faisables du système précédent est appelé polyèdre, et il est de nature convexe (Figure 2), ce qui induit qu'il existe toujours une solution optimale qui est l'un des points extrêmes du polyèdre [15].

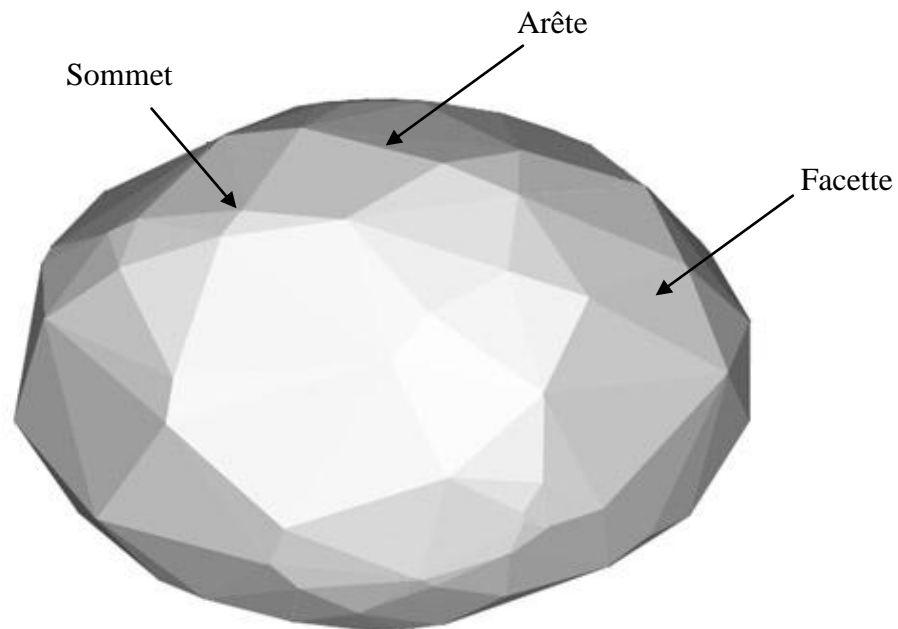


Figure 2 : Exemple de polyèdre de pleine dimension sur R^3 .

Il existe trois types principaux d'algorithmes de programmation linéaires : l'algorithme du simplexe, l'algorithme des points intérieurs et la méthode des ellipsoïdes. Les deux méthodes simplexe, et points intérieurs sont utilisées pour la résolution de problèmes linéaires pour les avantages qu'ils représentent. On peut convertir n'importe quel programme linéaire standard en introduisant des variables d'écart et des variables artificielles pour le rendre sous une forme standard [16].

1.5.1 Programmation linéaire en nombres entiers PLNE

Tous les problèmes d'optimisation combinatoire peuvent être formulés sous forme de problèmes en nombres entiers, en ajoutant que x appartient à Z^n [15].

La programmation Linéaire en Nombres Entiers PLNE est un cas spécial des problèmes d'optimisation où toutes les variables doivent avoir des valeurs entières (Figure 3) et parfois même binaires (0-1), et cela est dû à la nature que représentent ces variables [17].

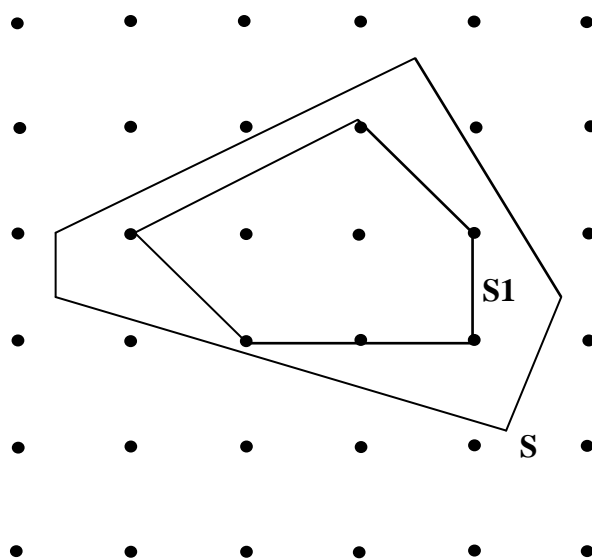


Figure 3: S Ensemble de solutions – $S1$ Ensemble de solutions entières.

1.5.2 La méthode du simplexe

Selon le créateur de la méthode du simplexe George B. Dantzig [14] :

Un simplexe à 0 dimension est un point ;

Un simplexe à 1 dimension est un segment de ligne ;

Un simplexe à 2 dimensions est un triangle et son intérieur ;

Un simplexe à 3 dimensions est un tétraèdre et son intérieur (Figure 4);

En pratique, le déroulement de l'algorithme du simplexe nous permet de détecter une solution optimale d'un programme linéaire, en partant d'une solution de base et en passant vers un autre sommet qui améliore la valeur de la fonction objectif. L'algorithme est divisé en trois étapes principales :

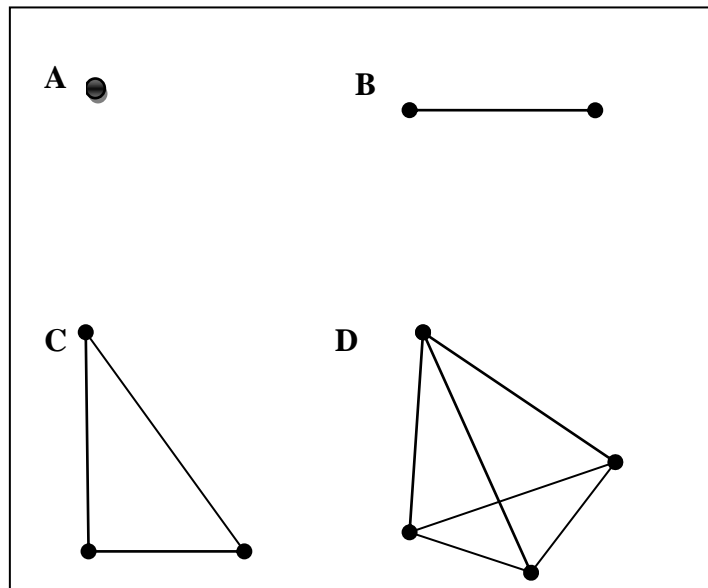


Figure 4: Simplexe.

(A) Zéro dimension. (B) Une dimension. (C) Deux dimension. (D) Trois dimension.

Déterminer une première solution de base réalisable : cette solution sert de départ au cheminement vers la solution optimale.

Si la solution obtenue précédemment n'est pas optimale, trouver une autre solution de base réalisable qui permettrait d'améliorer la fonction objectif en effectuant un pivotement.

On répète cette procédure jusqu'à ce qu'il ne soit plus possible d'améliorer la fonction objectif. La dernière solution de base réalisable constitue la solution optimale du programme

Deux techniques essentielles sont utilisées pour le déroulement de l'algorithme du simplexe : la technique des deux phases et la technique des pénalités big M.

Un pseudo code de l'algorithme simplexe est présenté dans la Figure 5

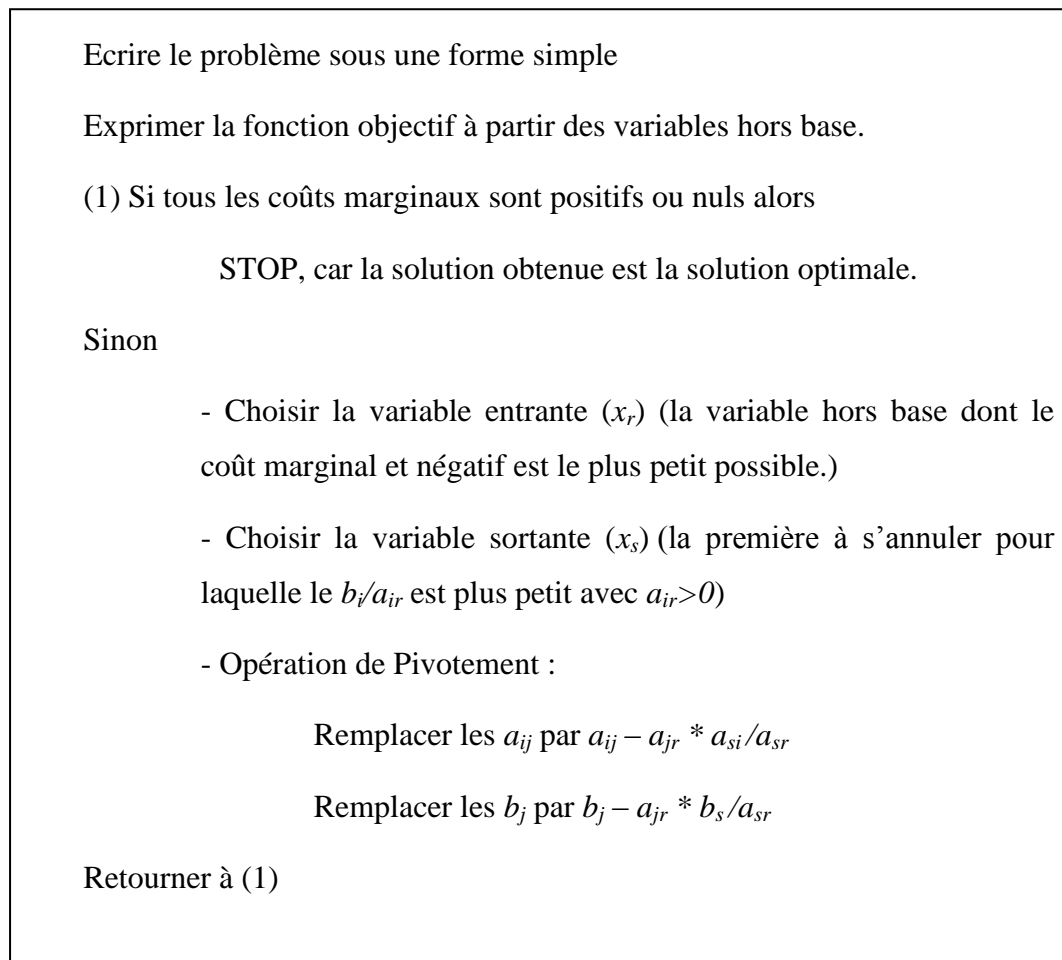


Figure 5: Pseudo code de l'algorithme du simplexe.

1.6 Séparation et évaluation (Branch and Bound)

Les méthodes par séparation et évaluation sont des méthodes de résolution exacte de problèmes d'optimisation combinatoire introduite par Land et Doig [3]. Une méthode de séparation et évaluation consiste à énumérer implicitement toutes les solutions dans S (l'espace de solutions) en examinant les sous-ensembles de S.

L'énumération des solutions du problème consiste à construire un arbre Branch and Bound dont les nœuds sont des sous-ensembles de solutions du problème, et les branches sont les nouvelles contraintes à respecter. La taille de l'arbre dépend de la stratégie utilisée pour la construire.

Rappelons que ces problèmes portent sur un ensemble de solutions réalisables (généralement) fini mais très grand. En particulier, la stratégie de recherche exhaustive

requiert un temps de calcul prohibitif. Le principe des méthodes exactes consiste souvent à ne parcourir qu'un sous-ensemble de solutions réalisables dont on a réussi à montrer qu'il contient (au moins) une solution optimale.

Principe

- Énumérer implicitement toutes les solutions [18].
- Les sous espaces de recherche S_1, \dots, S_n .
- Qualité pour chaque sous espace $v(S_1), \dots, v(S_n)$.
- Supposons : $f(s)$ à maximiser.
- Une solution réalisable a_1 connue, avec $v(a_1)=12$, et $v(S_3)=11$ implique qu'il n'est pas nécessaire d'explorer S_3 .
- (S_i) est une borne supérieure sur la qualité des solutions réalisables de S_i

Alors on peut synthétiser un algorithme Branch and Bound comme suit :

Construire l'arbre de recherche :

Un schéma de séparation divise S en sous-ensembles plus petits afin d'obtenir des problèmes qu'on sait comment résoudre. Les nœuds de l'arbre représentent les sous-ensembles, et les arcs représentent les relations qui lient les enfants aux parents.

Une stratégie de recherche ou exploration sélectionne un nœud parmi les nœuds possibles selon une priorité définie. La priorité d'un nœud ou d'un sous-ensemble S_i , $h(S_i)$, est généralement basée ou bien sur la profondeur du nœud dans l'arbre, ce qui nous amène à une stratégie de recherche en profondeur d'abord (*Depth-First*), ou bien sur sa capacité de donner de bons résultats, ce qui nous mène à une stratégie du meilleur d'abord (*Best-First*)

Elagage des branches :

Une fonction d'évaluation v donne la limite inférieure pour la meilleure solution de chaque sous-ensemble créé dans l'arbre.

L'intervalle d'exploration restreint la taille de l'arbre à construire : seulement les nœuds dont l'évaluation est dans l'intervalle sont explorés, les autres nœuds sont

éliminés. La borne supérieure est mise à jour continuellement chaque fois qu'une solution faisable est trouvée.

La relation de dominance peut être établie dans certaines applications entre les sous-ensembles S_i , ce qui va nous amener à rejeter les nœuds non dominants.

Une condition d'arrêt :

Cette condition est réalisée quand le problème est résolu, et la solution optimale est trouvée (tous les sous-problèmes ont été explorés ou élagués).

Par exemple si on a à maximiser la fonction $z = 8x_1 + 5x_2$

Soumise aux contraintes :

$$x_1 + 5x_2 \leq 15$$

$$9x_1 + 5x_2 \leq 45$$

$$x_1, x_2 \geq 0 \text{ et } x_1, x_2 \text{ entiers}$$

On aura l'arbre illustré dans la Figure 6 :

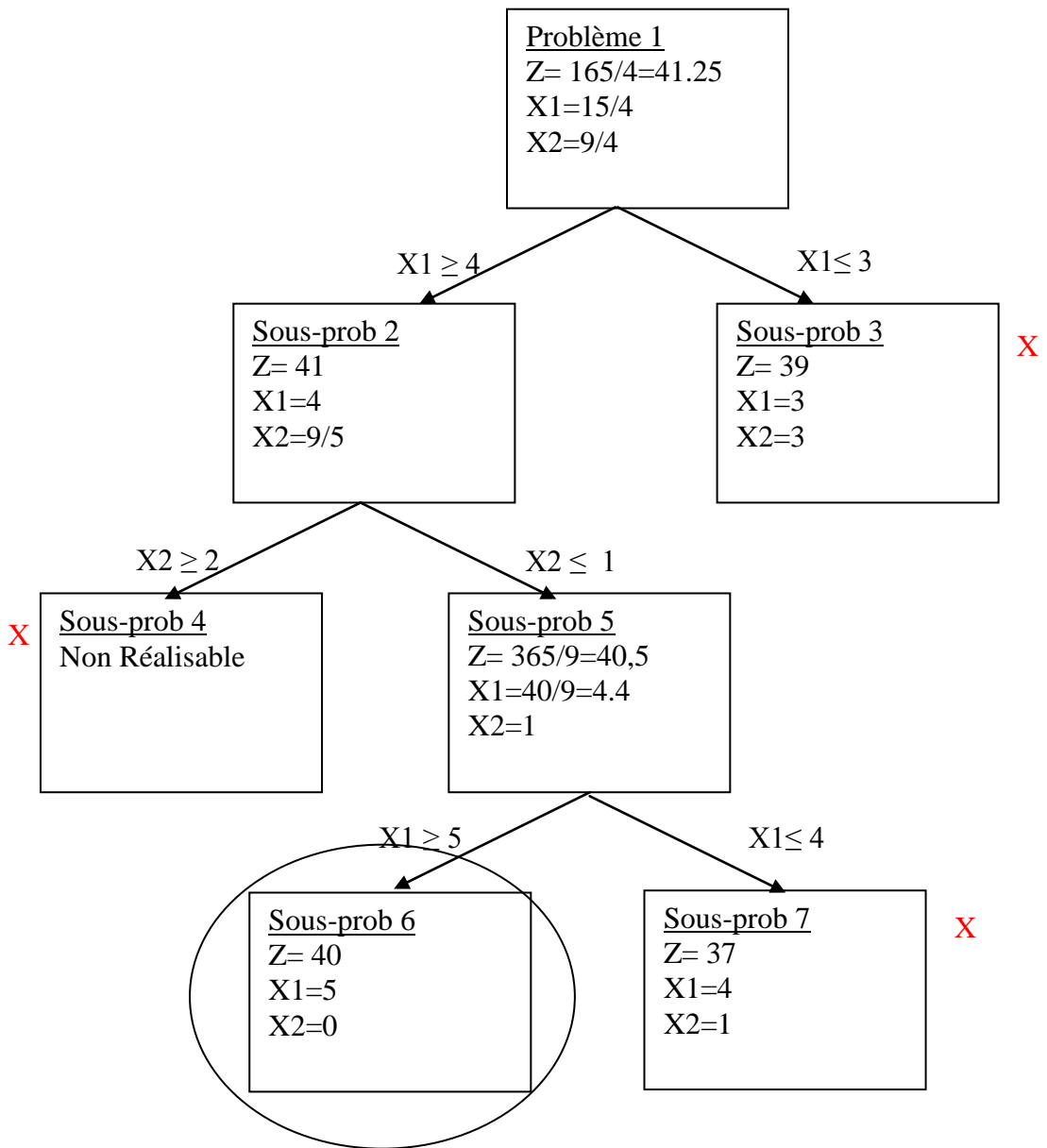


Figure 6: Déroulement de l'algorithme Branch and Bound.

Un pseudo code de l'algorithme Branch and Bound est donné dans la Figure 7 où L représente la liste d'attente, S est l'espace de recherche global.

```

Algo B&B
Tant que L ≠ ∅ faire
S = suivant (L)
S1, S2, .....Sn=séparer (S)
Pour chaque Si faire
    Si V(Si)>U ou Si non réalisable
        éliminer Si
    Sinon si Si réalisable
        Sbest =Si
        U = V(Si) (=f(Si))
    Sinon
        L=L U {Si}
Fin

```

Figure 7: Algorithme Séparation et évaluation (Branch and Bound).

1.7 Source de parallélisme dans l'algorithme Branch & Bound

1.7.1 Stratégies pour la parallélisation

La plupart des algorithmes parallèles B&B implémentent sous une forme ou une autre l'exploration parallèle de l'arbre. L'idée fondamentale, et cela dans la plupart des applications d'intérêt, est que la dimension de l'arbre B&B grandit et atteint des proportions intraitables. Ainsi, si l'exploration de l'arbre de recherche est faite plus rapidement par plusieurs processeurs, l'acquisition plus rapide de connaissance pendant la recherche (communication entre processeurs) permettra d'éliminer plus de nœuds ou d'élaguer plus de branches de l'arbre.

Pour décrire les stratégies possibles, nous démarrons à partir d'une représentation du B&B séquentiel où plusieurs opérations sont exécutées sur la structure de données qui contient le travail devant être fait (par exemple, nœuds dans plusieurs états) et l'information relative au statut de la recherche (par exemple, la valeur de la meilleure solution). Transposées dans un environnement parallèle, les stratégies de la gestion de l'arbre sont présentées comme stratégies de contrôle de la recherche qui impliquent de l'information et des stratégies de la gestion de groupe. C'est remarquable que quelques-

unes de ces stratégies induisent une exploration de l'arbre différente de celle exécutée par la méthode séquentielle.

Rappelons que le B&B séquentiel est fondamentalement une procédure récursive qui extrait un nœud (c.-à-d., un sous-problème) du groupe des nœuds en attente, donc l'effacer de la structure de données, exécute une série d'opérations (évaluation, calcul de bornes, séparation, etc.), et complète la boucle en insérant un ou plusieurs nœuds (c.-à-d., les nouveaux sous-problèmes résultants de l'opération de séparation) dans le même groupe.

Les nœuds dans le groupe sont gardés habituellement et sont accédés selon leurs priorités basées sur plusieurs attributs du nœud (par exemple, valeurs de borne inférieure et supérieure, profondeur dans l'arbre) et la stratégie de l'exploration de l'arbre de recherche (par exemple, profondeur d'abord). La priorité du nœud donc définit un ordre sur les nœuds du groupe, aussi bien que la planification séquentielle de la recherche. Une propriété évidente de la recherche B&B séquentielle est que chaque fois qu'un nœud est désigné, la décision a été prise avec une connaissance complète de l'état de la recherche, qui est la vue globale de tous les nœuds en attente produits jusqu'à présent.

Dans un environnement parallèle, les décisions de recherche et l'information de recherche peuvent être distribuées. En particulier, toutes les informations pertinentes ne peuvent pas être disponibles au moment et endroit (c.-à-d., le processeur) une décision est prise. Donc, en premier nous examinons des questions en rapport avec le stockage et la disponibilité d'informations dans une exploration parallèle de l'arbre B&B. Nous nous intéressons alors aux rôles que peuvent jouer les processeurs dans une telle exploration. La combinaison de plusieurs alternatives pour ces deux composants contient les stratégies de base pour la conception d'algorithme B&B parallèle.

Deux questions doivent être posées au moment d'examiner l'information de recherche dans un contexte parallèle :

- (1) Comment cette information est stockée ?
- (2) Quelle information est disponible au moment de la décision ?

Le volume de l'information de recherche est composé du groupe de nœuds, et les stratégies de gestion de groupe adressent la première question en ce qui les concerne : si et comment décomposer le groupe de nœuds ? Une stratégie centralisée conserve tous les

nœuds dans un groupe central. Cela implique que ce groupe unique, dans une forme ou une autre, sert tous les processeurs impliqués dans le calcul parallèle. Par alternative, dans une stratégie distribuée, le groupe est partitionné, chaque sous-ensemble est stocké par un processeur. Les autres informations pertinentes (par exemple, variables du statut global comme la valeur optimale) sont de dimension limitée. Par conséquent, la question n'est pas s'il est distribué ou pas, mais plutôt s'il est disponible et à jour lorsque les décisions sont prises.

Dans un algorithme B&B parallèle, plusieurs processeurs peuvent décider, plus ou moins simultanément de traiter un nœud. Leurs actions collectives correspondent à la planification parallèle de la recherche, et la distribution correspondante de travail sur les processeurs.

L'ordonnement des nœuds est basé sur leurs priorités. Nous définissons comme connaissance de la recherche, le groupe de nœuds avec leurs priorités, plus la meilleure solution connue et les autres variables d'état globales de la recherche. La connaissance de la recherche peut être complète ou partielle. Si la connaissance de la recherche est complète, l'ordonnement résultant est très près de l'ordonnement séquentiel. En effet, quand à chaque pas, le processeur qui doit prendre une décision d'ordonnement a une connaissance exacte et complète de tous les nœuds en attente d'être traités, sa décision est presque la même que dans le cas séquentiel. Quand seulement des informations partielles sont connues d'un processeur, la planification pourrait être vraiment différente comparée au séquentiel.

Lorsque l'information est distribuée, l'ordonnement parallèle doit inclure des prévisions spécifiques pour traiter plusieurs questions particulières :

La définition d'une phase d'initialisation parallèle et d'allocation du travail initial entre les processeurs.

La mise à jour des variables d'état globales (par exemple, la valeur de la meilleure solution connue).

La terminaison de la recherche.

La minimisation du temps de repos des processeurs.

La maximisation du travail significatif.

Les stratégies de contrôle de la recherche spécifient le rôle de chaque processeur dans la recherche parallèle, ce qui est, les décisions relatives à l'extraction et l'insertion de nœuds dans le ou les groupes, la stratégie de l'exploration, le travail exécuté (par exemple, évaluation totale ou partielle, le partitionnement, évaluation des descendants), les communications à entreprendre pour diriger les différents groupes, et la connaissance de la recherche associée.

D'un point de vue du contrôle de la recherche, nous distinguons deux types fondamentaux de processeurs : maître et esclave. Le processus maître exécute la gamme complète des tâches. Ils spécifient le travail que les processeurs esclaves doivent faire et prendre part aux communications avec les autres processus maîtres pour rendre effectif l'ordonnancement parallèle des nœuds, contrôle l'échange d'information et détermine la terminaison de la recherche. Le processeur esclave communique exclusivement avec son processeur maître assigné et exécute les tâches préspecifiées sur le sous-problème qu'il reçoit.

Les stratégies maître-esclaves classiques, ou centralisées, font usage de deux types de processeurs, un processeur maître et plusieurs processeurs esclaves. Ceci est combiné généralement à une stratégie de gestion de groupe centralisée. Donc, le processeur maître maintient la connaissance globale et le contrôle de la recherche entière, pendant que les processeurs esclaves exécutent les opérations B&B reçues du processeur maître et rendent les résultats au maître.

Une autre stratégie consiste à distribuer le contrôle tout en distribuant l'ensemble de nœuds, tous les processeurs sont maîtres, et ils sont au courant des variables globales de recherche.

La distribution de groupe et d'information de recherche résulte souvent en une charge de travail irrégulière pour les processeurs. Une stratégie d'équilibre de charge doit alors être implémentée pour indiquer comment l'information relative à l'activité du processeur circule et comment les décisions d'équilibre de charge sont prises. D'un point de vue du contrôle, deux approches sont possibles : ou la décision est collégalement distribuée sur les processeurs ou centralisée (plus ou moins). Ou bien, un des processeurs agit comme maître de partitionnement de charge. Il rassemble l'information de la charge et choisit les échanges des données à exécuter.

Ces stratégies et types de processus sont les jeux de construction de base qui peuvent être combinés pour construire des stratégies parallèles plus complexes, hiérarchiques, avec plus qu'un niveau de distribution de l'information (groupe), contrôle de la recherche et équilibre de la charge. Par exemple, un processeur pourrait dépendre d'un autre pour son propre équilibre de charge, pendant que lui il dirige l'équilibrage de charge pour un groupe de processeurs d'un niveau inférieur, aussi bien que le travail de plusieurs processeurs esclaves.

1.7.2 Classifications des algorithmes parallèles B&B

La parallélisation de B&B a bien été étudiée dans la littérature et plusieurs classifications ont été proposées [9] [19] [4] [10] et [8]. Nous allons donner un court aperçu de ces classifications dans ce qui suit :

1.7.2.1 Classification de Trienekens et al.

Trienekens et al. [9] ont proposé deux niveaux de parallélisation d'algorithmes B&B : parallélisation de bas niveau et parallélisation de haut niveau.

A. Parallélisation de bas niveau

Seule une partie de l'algorithme B&B est parallélisée de telle manière que les interactions entre la partie parallélisée et les autres parties de l'algorithme ne changent pas. Par exemple, le calcul de la fonction d'évaluation, la sélection du sous-problème à être élagué. Parce que les interactions entre les différentes parties de l'algorithme ne sont pas modifiées, la parallélisation de bas niveau ne modifie pas la sémantique globale de l'algorithme B&B. Cela signifie que le comportement global de l'algorithme B&B parallèle créé est similaire au comportement de l'algorithme séquentiel.

B. Parallélisation de haut niveau

Dans la parallélisation de haut niveau, les effets et les conséquences du parallélisme introduits ne sont pas limités à une partie particulière de l'algorithme B&B, mais influence la sémantique de l'algorithme dans son ensemble. Le travail effectué par l'algorithme parallèle ne doit pas être le même que le travail effectué par l'algorithme séquentiel. L'ordre dans lequel le travail est exécuté peut différer, et il est même possible que certaines parties du travail effectué par l'algorithme parallèle ne sont pas effectuées par l'algorithme séquentiel, ou vice versa. Par exemple, l'exploration parallèle de l'arbre

de recherche peut conduire à améliorer plus vite la meilleure solution et permet donc d'élaguer certaines branches de l'arbre qui sont explorées dans la version séquentielle de l'algorithme.

Trienekens et al. ont présenté un ensemble de paramètres à considérer pour classifier les algorithmes parallèles B&B présentés dans le Tableau 1.

Les paramètres des algorithmes B&B	
Partage de connaissances	Nombre + type des bases de connaissances
	Stratégie de mise à jour
Connaissances utilisées par les processus	Stratégie d'accès
	Stratégie de réaction
Partage de travail	Processus
	Unités de travail
	Stratégie de distribution de charges
Synchronisation	Synchronisation de chaque processus
Les quatre règles de base	Règle de division (séparation)
	Règle de limites
	Règle de sélection
	Règle d'élimination

Tableau 1 : Paramètres des algorithmes B&B.

1.7.2.2 Classification de Gendron et al.

Gendron et al. [19] Identifient trois principales approches pour la conception de l'algorithme B&B parallèle en fonction du degré de parallélisme potentiellement fourni par l'arbre de recherche :

A. Parallélisme de type 1 : introduit le parallélisme lorsque les opérations sont effectuées sur les sous-problèmes générés. Par exemple, l'exécution de l'opération de sélection en parallèle pour chaque sous-problème produit une accélération de l'exécution. Ce type de parallélisme n'a aucun impact sur la structure générale de l'algorithme B&B et il est particulier au problème à résoudre.

B. Parallélisme de type 2 : consiste à construire l'arbre de recherche en parallèle en appliquant simultanément les opérateurs B&B sur plusieurs sous-problèmes. Ce type de parallélisme peut influencer sur la conception et la sémantique de l'algorithme.

C. Parallélisme de type 3 : signifie que plusieurs arbres de recherche sont générés en parallèle. Les arbres sont explorés en utilisant différentes variantes des opérations (ramification, évaluation, et élagage), et les informations générées lors de la construction d'un arbre peuvent être utilisées pour la construction d'un autre.

1.7.2.3 Classification de Melab

Dans cette taxonomie [10], quatre modèles d'algorithmes parallèles B&B sont identifiés :

A. Modèle multiparamétrique parallèle

Le modèle parallèle multiparamétrique consiste à considérer simultanément plusieurs processus B&B qui diffèrent par un ou plusieurs opérateurs (séparation - évaluation - élagage - sélection), ou avoir les mêmes opérateurs paramétrés différemment (Figure 8). Chaque processus B&B explore un arbre qui n'est pas nécessairement le même par rapport à ses concurrents. Les algorithmes parallèles B&B peuvent varier selon l'opérateur de séparation comme dans [20], par l'opérateur de sélection [21] ou utiliser différentes limites comme dans [22]. Le principal avantage du modèle multiparamétrique est sa genericité permettant son utilisation transparente pour l'utilisateur. Toutefois, elle pourrait conduire à une exploration redondante de certains sous-problèmes qui ralentissent le temps d'exploration nécessaires pour parcourir l'arbre de recherche B&B.

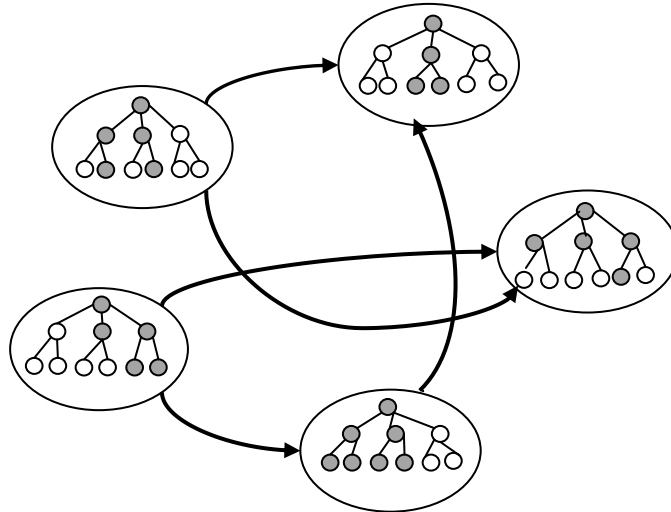


Figure 8: Modèle Multiparamétrique

B. Le modèle parallèle de l'exploration de l'arbre :

Le modèle parallèle de l'exploration de l'arbre consiste à lancer plusieurs processus de B&B à explorer simultanément des chemins différents (sous-arbres) du même arbre (

Figure 9). Les différentes opérations sélection, séparation, élagage sont exécutées en parallèle, soit dans un mode synchrone ou asynchrone. En mode synchrone, les processus B&B ont besoin d'échanger des informations globales qui permettent d'accélérer le parcours de l'arbre comme la meilleure solution courante. Dans un mode asynchrone, les processus B&B communiquent de manière imprévisible.

Comparée aux autres modèles, l'exploration parallèle de l'arbre a générée un grand intérêt et a fait l'objet de plusieurs études existantes sur des algorithmes B&B parallèles. Le degré de parallélisme de ce modèle est effectivement très élevé principalement pour les grandes instances de problèmes qui justifient son utilisation sur des architectures many-cœur et multi-cœur.

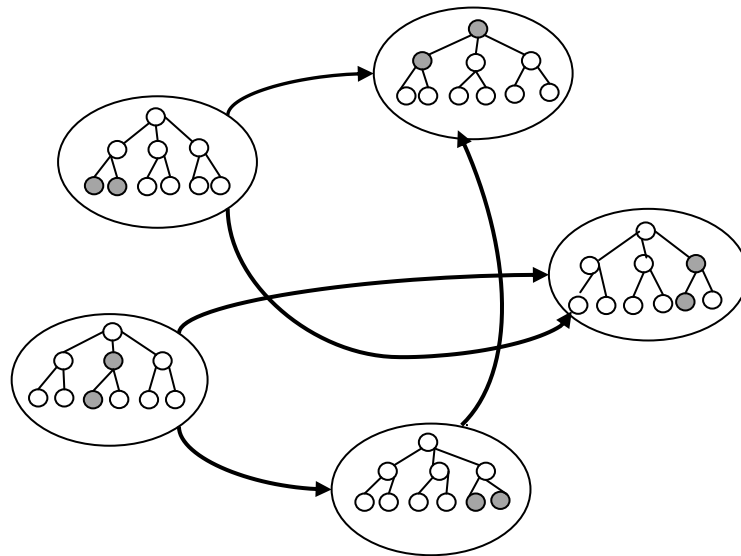


Figure 9 : Modèle parallèle de l'exploration de l'arbre.

C. L'évaluation parallèle des limites :

L'évaluation parallèle des limites consiste à lancer un seul processus B&B pendant qu'une évaluation parallèle des sous-problèmes générés par l'opérateur de séparation est effectuée (Figure 10).

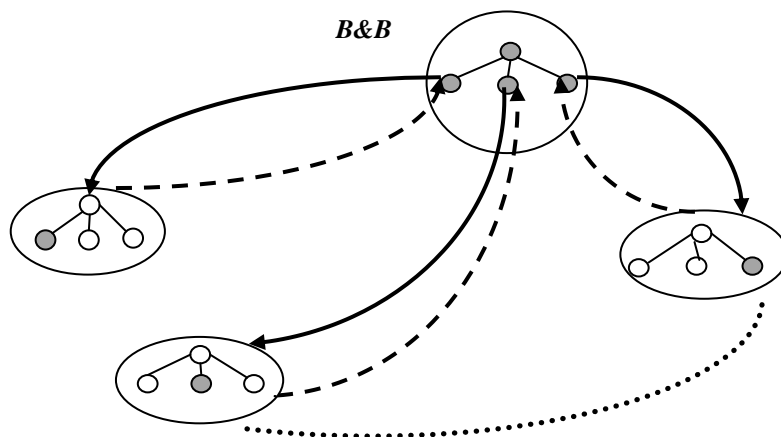


Figure 10: Evaluation parallèles des limites.

Ce modèle utilise le parallélisme de données, principalement asynchrone et de grain fin (le coût de l'évaluation de la limite). Cependant, cette parallélisation de l'évaluation

n'est portable que si l'opération d'évaluation des limites consomme énormément de temps principalement sur les architectures à forte proportion d'opérations arithmétiques.

D. Evaluation parallèle d'une seule limite / Solution

L'évaluation parallèle d'une seule limite / solution est similaire à la précédente (évaluation parallèle des limites) où un seul processus est utilisé. Dans ce modèle, un ensemble de processus évalue en parallèle la fonction limite / objectif d'un seul nœud. Il nécessite la définition de nouveaux éléments spécifiques pour le problème en cours de traitement, comme la fonction objectif partielle et la méthode pour consolider ces résultats partiels. Comme la mise en œuvre de ce modèle est naturellement synchrone, il est essentiel de mémoriser toutes les valeurs d'évaluation partielles pour les solutions en cours d'évaluation.

1.7.2.4 Classification de Crainic et al.

Deux stratégies de parallélisation ont été proposées par Crainic et al. [8].

A. **Stratégies basées nœud**, qui ont l'intention d'accélérer une opération particulière, principalement au niveau du nœud: Calcul parallèle de la borne inférieure ou supérieure, évaluation parallèle de fils, et ainsi de suite.

B. **Stratégies basées arbre**, qui visent à construire et explorer l'arbre B&B en parallèle.

Les stratégies basées nœud s'intéressent à accélérer la recherche en exécutant en parallèle une opération particulière. Ces opérations sont principalement liées au sous-problèmes, ou nœuds, tel que : évaluation, limitation, et séparation, et varie de tâches numériques "simples" (par exemple, inversion de la matrice), à la décomposition de calculer et des tâches intensives (par exemple, la génération de coupes), à la programmation mathématique parallèle (par exemple, simplexe, relaxation lagrangienne) et méta-heuristique (par exemple, les recherches tabou) les méthodes calculent des bornes inférieures et dérivent des solutions faisables. Cette classe de stratégies a aussi été identifiée comme parallélisation de bas niveau (ou type 1), parce qu'elle ne vise pas à modifier le sens de la recherche, ou la dimension de l'arbre B&B ni son exploration. L'accélérer est le seul objectif. Cependant, c'est remarquable que certaines approches basées nœud puissent modifier le sens de la recherche. Les exemples typiques sont l'utilisation de relaxation Lagrangienne parallèle ou simplexe parallèle, en particulier

lorsque les multiples optima existent ou la base est transmise par les nœuds produits par l'opération de partitionnement.

En effet, lorsque les instances du problème sont particulièrement dures et grandes, plusieurs stratégies peuvent être combinées dans un schéma algorithmique complet. Par exemple, les stratégies basées nœud pourraient commencer la recherche et rapidement produire des sous-problèmes intéressants, suivi par une exploration parallèle de l'arbre. Où, une approche de recherche multiple peut être installée, où chaque recherche B&B utilise une ou plusieurs stratégies parallèles. Les stratégies basées arbre ont fait l'objet d'efforts de recherche plus importants et plus complets.

Les stratégies basées arbre produisent des algorithmes irréguliers, ce qui génère des difficultés :

Les tâches sont créées dynamiquement en cours de l'algorithme.

La structure de l'arbre est inconnue au début.

Le graphe de dépendance entre les tâches est imprédictible. (à la compilation et à l'édition des liens).

L'affectation des tâches aux processeurs doit être faite dynamiquement.

Le partage et l'équilibre de charge de travail ainsi que la transmission des informations entre les processeurs doivent être pris en compte algorithmiquement.

1.7.3 Anomalies dues à l'accélération

Le parallélisme peut créer des tâches qui sont redondantes ou non nécessaires, ou qui dégradent les performances.

Le succès de la parallélisation de l'algorithme B&B peut être mesuré par l'accélération absolue obtenue avec p processeurs, comme étant le ratio entre le temps du meilleur algorithme séquentiel et le temps de l'algorithme parallèle utilisant p processeurs, pour une instance du problème.

Pour l'algorithme B&B parallèle basé arbre on s'attend à une accélération linéaire au maximum proche de p (efficacités près de 100%), mais parfois cette accélération est spectaculaire et supérieure à p , tandis que d'autre fois elle est inférieure à p . Ces anomalies de comportement paraissent surprenantes mais elles sont dues à la

combinaison de l'accélération et aux propriétés de l'arbre B&B où ces priorités et les fonctions de calcul des bornes sont connues postérieurement.

1.7.4 Logiciels Branch and Bound

Il existe un large éventail de logiciels B&B, cela est dû au nombre de types de problèmes, les interfaces utilisateurs, types de parallélisation et les types de machines.

Il existe plusieurs bibliothèques et applications qui sont dédiées à un type de problème, les solveurs linéaires commerciaux comme CPLEX, Xpress-MP ou open source comme GLPK ou Ip_Solve, et qui sont tous dédiés à résoudre les problèmes linéaires mixtes, mais où le B&B est caché dans une boîte noire, et ce ne serait pas pratique d'implémenter une parallélisation efficace. Toutefois on pourrait utiliser la partie solveur de ces bibliothèques et implémenter un B&B parallèle indépendant.

Il existe plusieurs cadres d'application (framework) pour les parallèles B&B, parmi eux :

1. PUBB (Univ. of Tokyo).
2. BOB++ (Univ. of Versailles).
3. PPBB-Lib (Univ. of Paderborn).
4. PICO (Sandia Lab. & Rutgers University)
5. FATCOP (University of Wisconsin)
6. MallBa (Consortium of Spanish universities)
7. ZRAM (Eth Zürich)
8. BCP (CoinOR).
9. ALPS/BiCePS (Lehigh University, IBM, Clemson University)
10. MW Framework (Lehigh University)
11. Symphony (Rice Univ.)

Ils diffèrent par le type de problèmes qu'ils résolvent, par le type de parallélisation qu'ils proposent et par les machines ciblées.

1.8 Les avantages et inconvénients pour B&B Parallèle et séquentiel

Comme utilisateurs de B&B, nous allons énumérer les avantages et inconvénients remarqués pendant le déroulement de calculs en utilisant B&B.

1.8.1 Les points pour B&B séquentiel

L'importance de trouver un bon candidat initial ne doit pas être surestimée, et le temps utilisé pour trouver un tel candidat ne représente souvent qu'un pourcentage minime de la durée totale de l'algorithme.

Dans le cas où une solution initiale est très proche de l'optimum, le choix de la stratégie de sélection de nœuds et de la stratégie de traitement fait peu de différence.

1.8.2 Points pour B&B parallèle

Deux points sont d'ordre général pour la programmation parallèle :

Ne pas utiliser de calcul parallèle pour des problèmes faciles, car cela ne vaut pas la peine. Pas d'accélération de calcul notable.

Choisir le matériel adéquat pour le problème (ou le problème pour son matériel si on n'a pas beaucoup de moyens)

En ce qui concerne B&B parallèle, il faut se méfier des principaux points suivants :

Le contrôle centralisé n'est possible que dans des systèmes avec un nombre assez limité de processeurs. Si un grand nombre de processeurs doivent être utilisés, une distribution totale ou une stratégie combinée devrait être utilisée.

Si le problème en question a une fonction d'évaluation qui fournit une limite forte, alors le nombre de sous-problèmes actif sera à tout moment très réduit. Alors seulement quelques-uns des processeurs d'un système parallèle peuvent être maintenus occupés avec un travail utile en même temps. Dans ce cas, il sera nécessaire de paralléliser aussi le calcul de la fonction d'évaluation pour exploiter les processeurs supplémentaires. Cela est généralement beaucoup plus difficile dû au fait que la résolution du problème d'optimisation fournissant la valeur de la fonction d'évaluation est souvent P-complet.

Si en revanche le calcul de la fonction d'évaluation donne lieu à de grands arbres de recherche dans le cas séquentiel, l'algorithme B&B parallèle sera probablement une très bonne méthode de résolution. Alors la répartition de la charge de travail statique peut conduire à un algorithme facilement programmable et efficace si le système utilisé est homogène.

Lorsque vous utilisez une distribution dynamique de la charge de travail, le temps passé dans la programmation, les tests et les méthodes d'optimisation sophistiquées peuvent ne pas bien payer. Souvent de bons résultats sont possibles avec les systèmes relativement simples.

Lors de la consultation de la littérature, il faut être prudent lors de la vérification des résultats des tests. Une accélération de 3,75 sur un système avec 4 processeurs peut à première vue sembler convaincante, mais d'un autre côté, une accélération aussi idéale ne peut être obtenue avec si peu de processeurs, l'algorithme sera très probablement dans des troubles graves lorsque le nombre de processeurs augmente [6].

1.9 Conclusion

L'optimisation combinatoire consiste à trouver une solution parmi un grand nombre fini de solutions, elle se situe au carrefour de la théorie des graphes, de la programmation mathématique, informatique théorique et de la recherche opérationnelle.

Deux axes principaux de recherche de solutions existent, les algorithmes exacts, et les heuristiques.

Parmi les algorithmes exacts on trouve l'algorithme séparation et évaluation (Branch and Bound) et qui consiste à énumérer implicitement toutes les solutions d'un problème en énumérant ces sous-ensembles et cela en construisant un arbre dont les nœuds sont les sous-problèmes.

Deux approches de parallélisme de B&B sont connues pour accélérer la recherche, la première basée nœuds (accélérer les calculs des bornes, évaluation parallèles des fils) et la deuxième basée arbres (construire et explorer l'arbre en parallèle).

Chapitre 2

Notions du parallélisme

2.1 Introduction

L'avancé technologique, dans tous les domaines de la vie moderne, demande de plus en plus de puissance de calculs. La course à la vitesse des processeurs a atteint ses limites en terme de réchauffement des circuits et de fréquences alors les constructeurs se sont tournés vers la parallélisation des traitements et la création de processeurs multicœurs qu'on peut facilement assimiler à des machines parallèles (Figure 11). La parallélisation des traitements est alors une tendance qui se démocratise à tous les utilisateurs par contre aux temps où il fallait des supercalculateurs pour pouvoir exécuter des programmes parallèles.

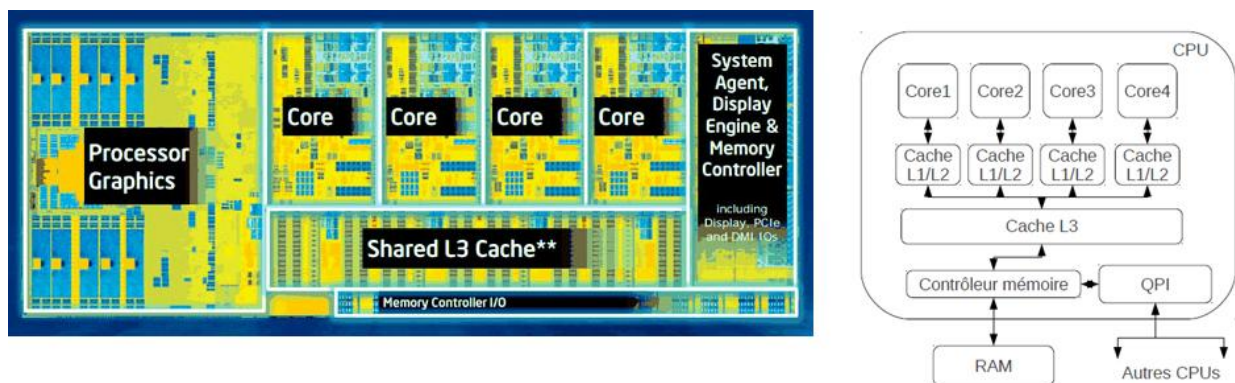


Figure 11: Architecture interne Processeur Intel I7

2.2 Algorithme parallèle

Un algorithme parallèle est conçu pour s'exécuter sur une machine parallèle pour résoudre un problème en améliorant le temps de calcul tout en respectant les contraintes de complexité en temps calcul et mémoire.

La conception d'un algorithme parallèle pour un problème donné est beaucoup plus complexe que la conception d'un algorithme séquentiel du même problème car elle demande la prise en compte de plusieurs facteurs tels que : la partie du programme qui peut être traitée en parallèle, la manière de distribuer les données, les dépendances des données, la répartition de charges entre les processeurs, les synchronisations entre les processeurs. Il y a essentiellement deux méthodes pour concevoir un algorithme parallèle, l'une consiste à détecter et à exploiter le parallélisme à l'intérieur d'un algorithme

séquentiel déjà existant, l'autre consistant à inventer un nouvel algorithme dédié au problème donné [23].

2.3 Les Machines Parallèles

Les machines parallèles sont construites autour d'un ensemble de nœuds interconnectés [24], chacun est constitué d'un ou plusieurs processeurs identiques, ou non, qui collaborent pour l'exécution d'une application. Une classification de ces machines a été introduite par M. J. Flynn (1966) [25].

SISD pour *Single Instruction, Singel Data* : c'est le modèle classique de Von Neumann où un seul processeur exécute une seule instruction sur une seule donnée (un seul flot de données) à la fois résidant dans une seule mémoire.

SIMD pour *Single Instruction, Multiple Data* : le contrôle est centralisé sur un seul processeur, les autres processeurs sont synchronisés entre chaque instruction du même algorithme qu'ils exécutent ;

MISD pour *Multiple Instruction, Single Data* : une même donnée est traitée simultanément par plusieurs processeurs.

MIMD pour *Multiple Instruction, Multiple Data* : le contrôle est réparti sur tous les processeurs et la synchronisation, lorsqu'elle est nécessaire, est réalisée par communication. Chaque processeur peut exécuter un algorithme différent ;

On distingue deux types d'architecture de la mémoire dans les machines parallèles :

2.3.1 Les machines parallèles à mémoire partagée

Ces machines sont caractérisées par une horloge indépendante pour chaque processeur, mais une seule mémoire partagée entre ces processeurs, où tous les processeurs lisent et écrivent dans le même espace d'adressage mémoire, ce qui permet de réaliser un parallélisme de données et de contrôles (Figure 12). Le programmeur n'a pas besoin de spécifier l'emplacement des données, il définit seulement la partie du programme que doit exécuter chaque processeur en plus de la gestion de la synchronisation.

Ce type d'architecture possède plusieurs avantages dont la simplicité, la scalabilité et la parallélisation de haut niveau. Son inconvénient majeur est lié principalement à la limite de la bande passante du réseau d'interconnexion

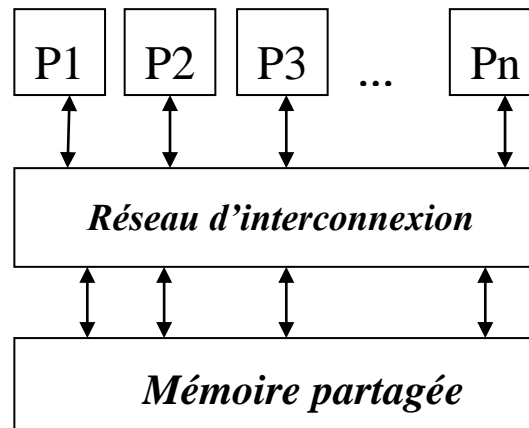


Figure 12: Machine parallèle à mémoire partagée.

2.3.2 Les machines parallèles à mémoire distribuée

Dans ce type de machine, chaque processeur possède sa propre mémoire locale, où il exécute des instructions identiques ou non aux autres processeurs. Les différents nœuds définis par l'ensemble mémoires+processeurs sont reliés entre eux par un réseau d'interconnexion (Figure 13). Le parallélisme est implémenté par échange de messages.

L'avantage principal des machines parallèles à mémoire distribuée est l'augmentation facile du nombre de processeurs avec des moyens simples, tels que les clusters. Seulement elles présentent plus de difficulté dans la programmation.

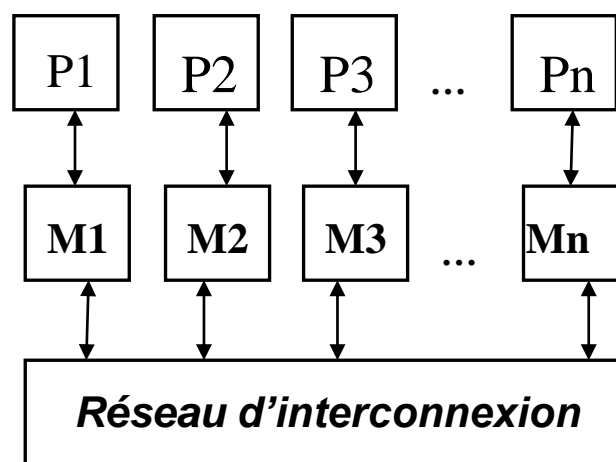


Figure 13 : Machine parallèle à mémoire distribuée.

Différents types de réseaux d'interconnexion ont été mis en place pour relier les différents nœuds des machines parallèles. On peut citer principalement : l'anneau, Grille torique, Fat Tree, hypercube et d'autres architectures hybrides (Figure 14).

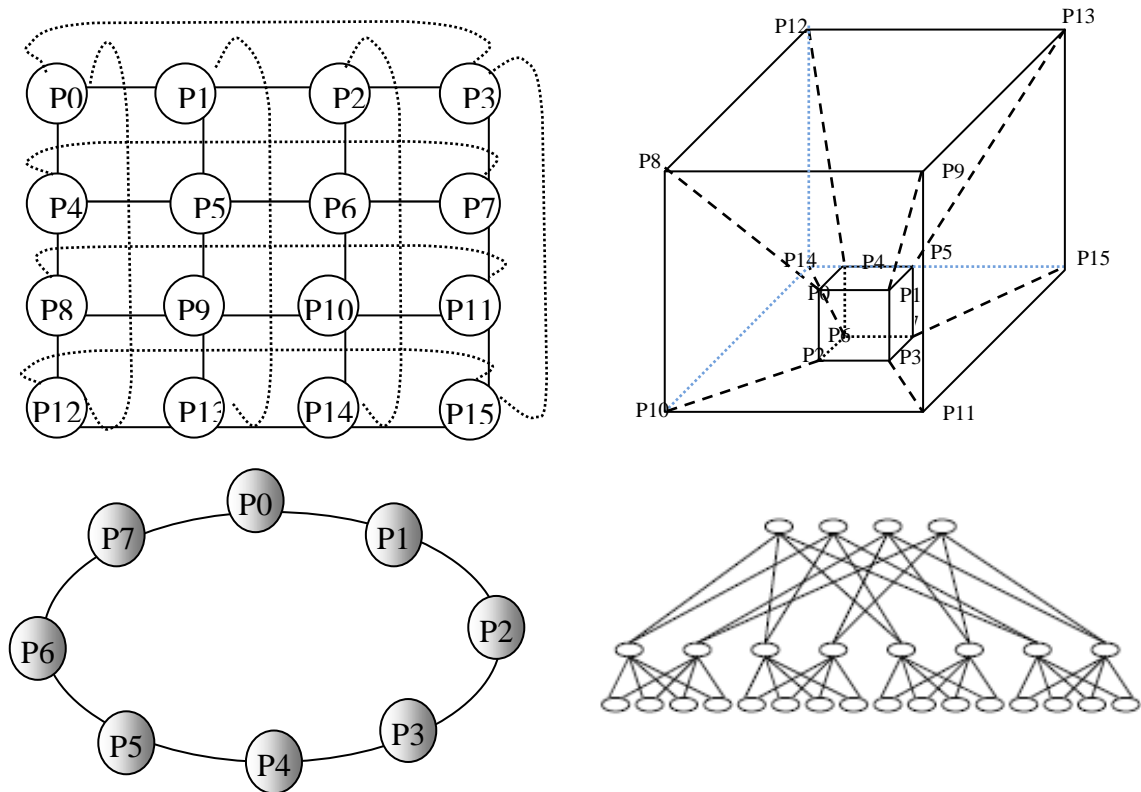


Figure 14: Différentes architectures de réseau d'interconnexion.

2.4 Modèles de machines parallèles

Des modèles de machines parallèles ont été proposés pour pouvoir mieux appréhender les caractéristiques des problèmes et des outils nécessaires pour les traiter de façon plus efficace. Le modèle RAM (*Random acces Machine*) de Von Neumann est largement répandu dans l'informatique séquentiel. Un tel modèle de calcul doit définir clairement un moteur d'exécution assez puissant pour produire une solution à une catégorie de problèmes. Ces objectifs, des notions prescriptives et descriptives, combinées pour permettre la traduction de la formulation abstraite d'un algorithme qui donne la solution souhaitée [26].

2.4.1 Le Modèle PRAM

Dans sa forme la plus simple, le modèle de la machine à accès aléatoire parallèle (PRAM), suppose qu'un ensemble de P processeurs, avec une mémoire partagée globale, exécutent un même programme de façon synchrone. Bien qu'il existe une certaine variabilité entre les définitions, la norme PRAM est un ordinateur MIMD où chaque processeur peut exécuter son propre flux d'instructions. Chaque processeur peut accéder à n'importe quel endroit de la mémoire dans un laps de temps indépendamment de l'emplacement de la mémoire. La principale différence entre les modèles PRAM est dans la façon dont ils traitent, lisent ou écrivent, les conflits d'accès à la mémoire. Différentes critiques ont été soulevées à l'encontre de ce modèle et qui concernent l'accès à la mémoire, la synchronisation, la latence et la largeur de bande et des variantes ont été proposées pour améliorer ces modèles.

2.4.1.1 L'accès à la mémoire

Parmi les variantes de base de la PRAM, le plus puissant est le CRCW PRAM [27], qui permet la lecture ou l'écriture de n'importe quel endroit de la mémoire, avec des règles pour résoudre des écritures concurrentes simultanées, comme la randomisation, la hiérarchisation, ou la combinaison. L'accès à la mémoire concurrente au coût unitaire est peut-être l'aspect le plus flagrant de ce modèle, et de nombreuses variantes ont été développées pour limiter cette idéalisation. Les variantes EREW et CREW PRAM restreignent l'accès à un emplacement de mémoire donné à un processeur à un moment, que ce soit pour la lecture ou l'écriture. Cela préserve le coût d'accès à l'unité, mais impose une certaine notion d'accès série. Une autre variante PRAM qui gère l'accès à la mémoire est le modèle de module parallèle (*Module parallel computer*) (MPC) [28]. Dans ce modèle, la mémoire globale partagée est divisée en modules m . Un seul accès à la mémoire peut se produire à l'intérieur de chaque module par intervalle de temps. Le modèle (*Queue-Read Queue-Write*) QRQW PRAM[29] donne un intermédiaire, une file d'attente, pour arbitrer et gérer les accès à la mémoire, tout en chargeant un coût d'accès à la mémoire qui est une fonction de la longueur de la file d'attente.

2.4.1.2 La Synchronisation

La norme PRAM pose un modèle d'exécution rigide dans lequel tous les processeurs sont synchronisés par une horloge globale. Une variante qui facilite cette restriction et qui

permet l'exécution asynchrone avec des points de synchronisation irréguliers est APRAM [30]. Une synchronisation périodique entre des intervalles d'exécution asynchrone est incorporée dans le XPRAM [31].

2.4.1.3 La latence

Le coût d'accès à la mémoire non locale a un effet sévère sur les performances des ordinateurs parallèles, plusieurs modèles PRAM ont été conçus pour remédier à l'idéalisation du coût d'accès à la mémoire. Il a été suggéré que le LPRAM pourrait être augmenté en facturant un coût de 1 (latence) unité pour accéder à la mémoire globale [32].

2.4.1.4 La largeur de bande

Un autre exemple d'une variante PRAM qui suppose deux classes de mémoire et comprend un mécanisme pour attribuer un coût non-unité à un accès à distance est la DRAM [33]. La DRAM est importante car elle élimine le paradigme de la mémoire partagée globale et le remplace uniquement par une mémoire distribuée privée. Alors que la topologie du réseau de communication est ignorée, la DRAM intègre la notion de bande passante limitée. Ce modèle propose une fonction de coût pour un accès à la mémoire non locale qui est basé sur l'encombrement maximum possible pour une partition donnée de données et la séquence d'exécution. Alors que la fonction est un peu compliquée, ce modèle tente de fournir des directives d'ordonnancement pour respecter l'accès limité aux données non-locales.

2.4.2 Les modèles de bas niveau

Beaucoup de modèles abstraits ont été développés qui incorporent une vision plus détaillée des composants de la machine et le comportement (*Low-Level Models*). L'objectif de ces modèles de bas niveau est souvent d'évaluer la faisabilité et l'efficacité d'une machine particulière ou de la conception de composants, parfois pour une classe particulière d'algorithmes, ou pour comprendre l'algorithme où la mise en œuvre particulière peut être plus efficace sur une machine ou sur un composant de cette machine.

2.4.3 Modèles à mémoires hiérarchique

Le stockage des données dans un ordinateur se fait à travers une variété de différentes unités physiques et médias, et où le temps d'accès est très différent entre ces médias, il y a eu un intérêt croissant pour la modélisation de ce phénomène. Dans la plupart de ces modèles, le concept de l'accès aléatoire est irrémédiablement altéré. Deux premiers exemples de modèles de mémoire hiérarchique série sont le (*Hierarchical Memory Models*) HMM [34] et le modèle (*HMM with block transfer*) BT [35]. Dans le modèle HMM il y a des niveaux K de mémoire dont chacune contient des emplacements de mémoire 2^k , l'accès à l'emplacement de mémoire x prend $f(x)$ temps pour une fonction f . Le modèle BT, (le modèle HMM avec transfert en bloc), intègre la possibilité de déplacer des données dans de grands blocs.

Généralement, ce type de modèle vise à fournir une réflexion plus raffinée et pratique des coûts d'accès à la mémoire en définissant une hiérarchie monotone pour les tailles et les coûts d'accès pour chaque niveau plus inférieur. Pour les problèmes qui impliquent le déplacement de grandes quantités de données, ces modèles peuvent être particulièrement efficaces pour produire des algorithmes efficaces.

2.4.4 Modèles réseau

Les deux classes de modèles décrits ci-dessus ignorent les impacts possibles de la topologie du réseau de communication. Le modèle de réseau de calcul parallèle reflète un sujet de préoccupation dans la première génération d'ordinateurs parallèles. Ce modèle de réseau attribue généralement une mémoire locale à chaque processeur. Le coût d'un accès à la mémoire à distance est une fonction à la fois de la topologie et de la combinaison d'accès. Les fonctions de coûts ont tendance à être complètement variables, sans coûts fixes de démarrage pour la communication. Ces modèles fournissent des incitations à la conception pour des mappings de données efficaces et les itinéraires de communication. Il y a autant de modèles qu'il y a de topologies réseau [36].

2.5 Conception d'algorithme parallèle

La conception d'algorithmes parallèles n'est pas facilement réduite à de simples astuces, en général elle demande beaucoup de créativité. Cependant, on peut bénéficier d'une approche méthodique qui maximise les options envisagées, qui prévoit des

mécanismes pour l'évaluation des alternatives, et qui réduit le coût de retour en arrière à cause des mauvais choix. Nous décrivons une telle approche et illustrons son application à une série de problèmes. Une méthode en quatre étapes a été présentée par Ian FOSTER [37] pour la conception d'algorithmes parallèles (Figure 15).

2.5.1 Partitionnement.

Le calcul qui doit être exécuté et les données opérées par ce calcul sont décomposés en petites tâches. Les questions pratiques, tel que le nombre de processeurs de l'ordinateur cible, sont ignorées, et l'attention se concentre sur la reconnaissance des possibilités d'exécution parallèle.

2.5.2 Communication.

La communication nécessaire pour coordonner l'exécution de la tâche est déterminée, en plus, des structures et des algorithmes de communication appropriés sont définis.

2.5.3 Agglomération.

Les structures de travail et de communication définies dans les deux premières étapes de la conception sont évaluées par rapport aux exigences de performance et les coûts de mise en œuvre. Si nécessaire, les tâches sont regroupées dans des tâches plus importantes pour améliorer les performances ou pour réduire les coûts de développement.

2.5.4 Mapping.

Chaque tâche est assignée à un processeur dans un souci de satisfaire les objectifs concurrents de maximiser l'utilisation du processeur et de minimiser les coûts de communication. La cartographie peut être spécifiée de manière statique ou déterminée à l'exécution par des algorithmes d'équilibrage de charge.

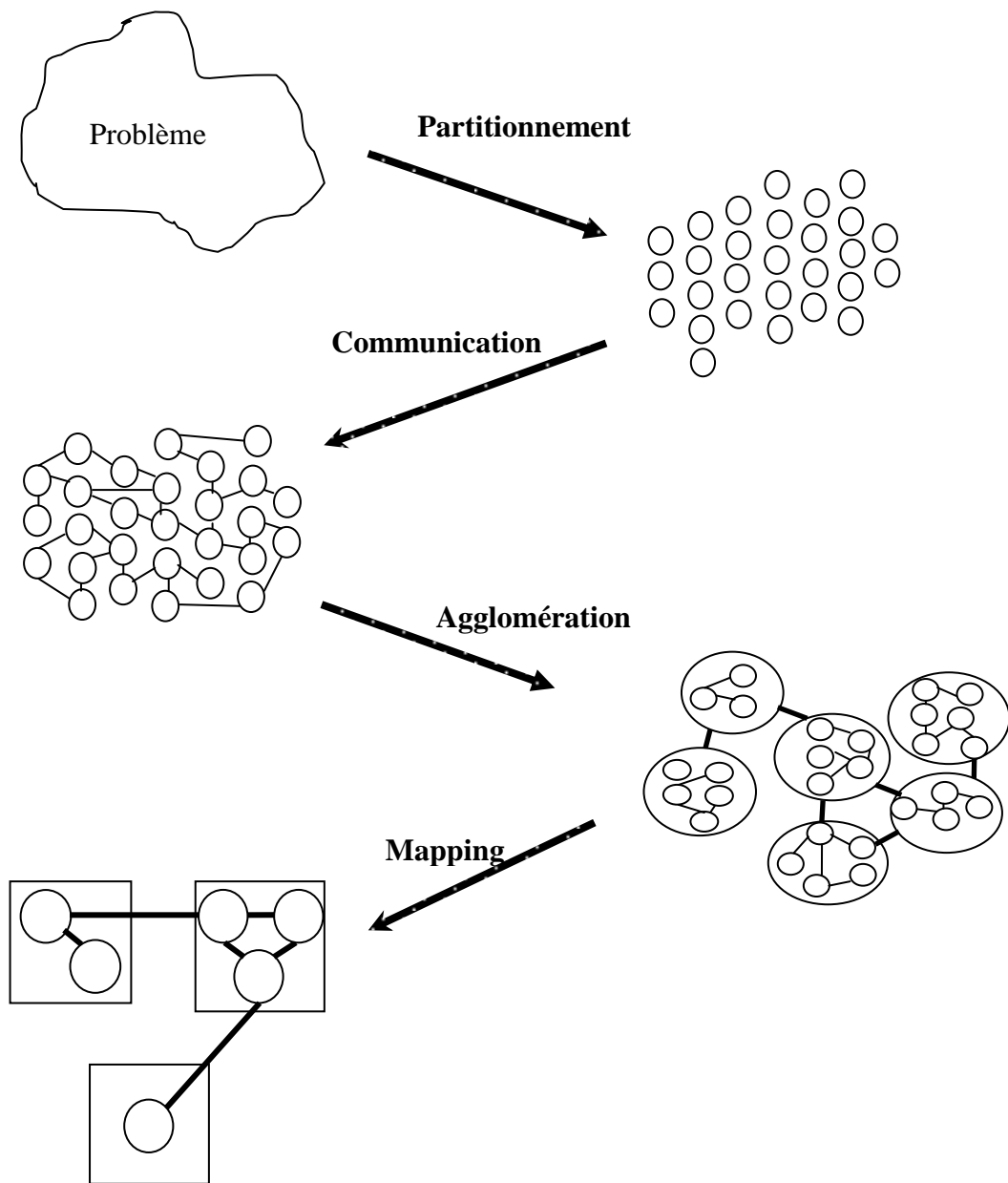


Figure 15: Méthode pour la conception d'algorithmes parallèles.

2.6 Sources de parallélisme

Trois principales sources de parallélisme sont détectées [25] :

2.6.1 Parallélisme de contrôle

Faire plusieurs choses en même temps : L'application est composée d'actions (tâches) qu'on peut exécuter en même temps (Figure 16). Les tâches peuvent être

exécutées de manière, plus ou moins, indépendante sur les ressources de calcul. Si on associe à chaque ressource de calcul une action, on aura un gain en temps linéaire. Si on exécute N actions sur N ressources on va N fois plus vite. Toutefois on remarque bien que les dépendances qui existent entre les tâches vont ralentir l'exécution parallèle.

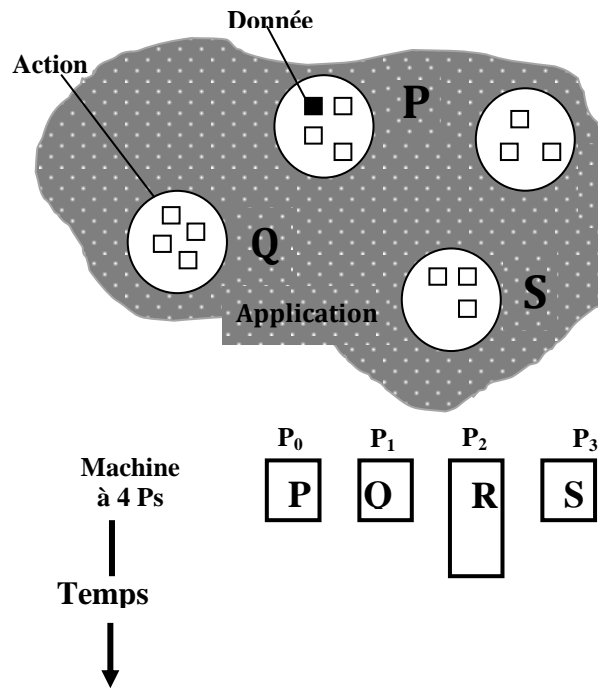


Figure 16: Parallélisme de Contrôle.

2.6.2 Parallélisme de données

Répéter une action sur des données similaires : Pour les applications qui sont composées de données identiques sur lesquelles on doit effectuer une action répétée (exemple : tableaux de données). Les ressources de calcul sont associées aux données.

2.6.3 Parallélisme de Flux

Travailler à la chaîne : Certaines applications fonctionnent selon le mode de travail à la chaîne, on dispose d'un flux de données, généralement similaires, sur lesquelles on doit effectuer une suite d'opérations en cascade. Les ressources de calcul sont associées aux actions et chaînées tels que les résultats des actions effectuées au temps T sont passés au temps $T+1$ au processeur suivant (mode de fonctionnement Pipe-Line. Figure 17).

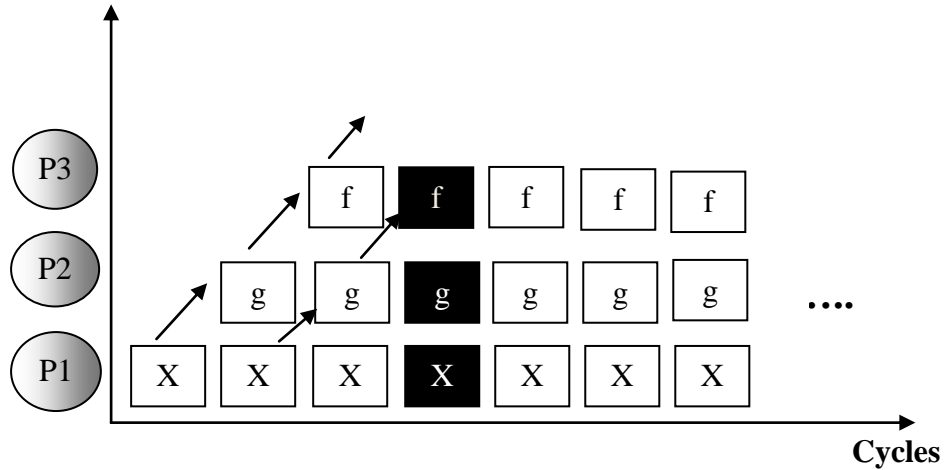


Figure 17: Parallélisme de flux.

2.7 Mesure de performance

Le but essentiel de la parallélisation est de gagner en performance de calcul, des facteurs on été mis aux points pour mesurer ces performances [38].

2.7.1 Accélération

On peut définir le temps d'exécution d'une application parallèle comme étant le temps d'exécution du processeur qui termine en dernier. On appelle accélération d'un algorithme parallèle A, le rapport entre le temps t_{seq} de l'algorithme séquentiel optimal par le temps $t_{par}(n_p)$ de l'algorithme parallèle fonction du nombre de processeurs.

$$Acc(A) = t_{seq} / t_{par}(n_p)$$

L'accélération est maximale lorsqu'elle est égale au nombre de processeurs, c'est à dire lorsque le travail réalisé par l'algorithme séquentiel optimal a été parfaitement réparti sur les processeurs et que cette répartition n'a pas induit de coût supplémentaire.

Cependant, nous pouvons remarquer que l'expression de l'accélération est dépendante du nombre de processeurs, c'est pourquoi il est nécessaire de définir une notion d'efficacité où de taux d'activité.

2.7.2 Accélération relative

En pratique, pour avoir une réelle mesure de l'accélération, on doit introduire le temps additionnel nécessaire à la conception et implémentation de l'algorithme parallèle.

$$Acc(A)_R = t_{seq} / (t_{par}(n_p) + \text{implémentation sur } p \text{ processeurs})$$

2.7.3 Efficacité

En utilisant les notations introduites précédemment, on définit l'efficacité d'un algorithme parallèle A comme étant le rapport

$$e(A) = t_{seq} / (t_{par}(n_p) * n_p)$$

L'efficacité est proche de 1 quand l'accélération est maximale. Si un programme obtient une efficacité de 0,6, cela signifie qu'il n'a réellement exécuté l'algorithme que pendant 60% du temps de l'exécution. Le reste du temps est réparti en communications, synchronisations, équilibre de charge.

2.7.4 Iso-efficacité

L'iso-efficacité d'un algorithme parallèle est définie par la quantité de travail supplémentaire nécessaire pour garantir l'efficacité parallèle quand le nombre de processeurs augmente. Dans certains cas il est possible de définir un algorithme parallèle efficace pour un certain nombre de processeurs. Mais, cette efficacité n'existe plus quand le nombre de processeurs augmente. C'est une situation de famine : il n'y a pas assez de travail pour tous les processeurs. L'iso-efficacité mesure la quantité de travail à ajouter pour maintenir l'efficacité.

2.8 Complexité d'un algorithme parallèle

L'étude de la complexité d'un algorithme consiste à évaluer le nombre d'opérations élémentaires en fonction de la taille des données. Ces évaluations nous permettent de juger de la qualité de l'algorithme. Les algorithmes peuvent être classés dans une des grandes classes de complexité (Figure 18):

- $O(1)$ complexité constante
- $O(\log(n))$ Complexité logarithmique

- $O(n)$ Complexité linéaire
- $O(n^p)$ complexité polynomiale
- $O(2^n)$ complexité exponentielle
- $O(n !)$ complexité factorielle

L'exécution d'un algorithme parallèle demande la mise à disposition d'un certain nombre de ressources (nombre de processeurs, espace mémoire,...). Le but de l'analyse de la complexité est de mesurer les différentes ressources nécessaires [24].

Le modèle classique RAM, dérivé du modèle de la machine de Turing représente le modèle fondamental des machines séquentielles. C'est à partir de ce modèle que sont définies les différentes classes de complexité en temps et en espace mémoire.

Pour le parallélisme, les mesures de complexité sont différentes suivant le modèle parallèle choisi. Pour le modèle PRAM, la notion de complexité en communication n'est pas définie, car chaque processeur peut accéder en un temps constant à toute donnée de la mémoire. Les mesures de complexités des algorithmes parallèles sont donc la complexité en temps, la complexité en communication et, comme pour le modèle RAM, la complexité en espace mémoire.

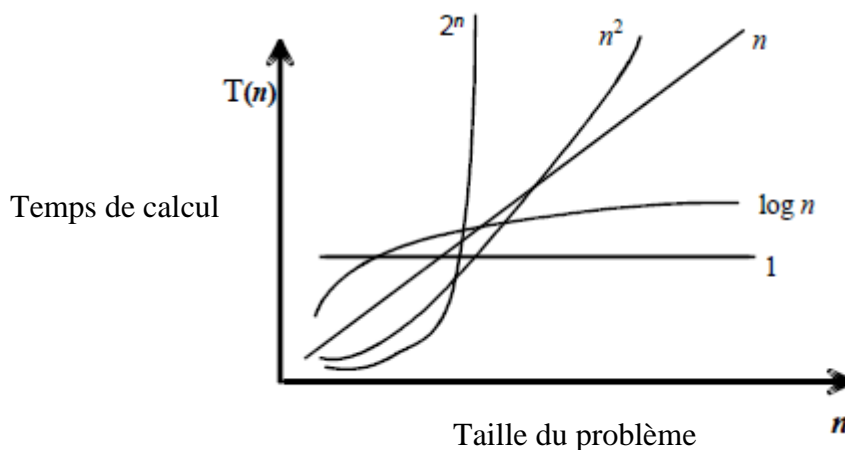


Figure 18: Ordre de complexité.

2.8.1 Complexité en temps

Le temps est souvent le premier critère considéré pour évaluer les performances d'un algorithme parallèle. La définition suivante, qui exprime le temps d'exécution, suppose que cette mesure est effectuée par un observateur extérieur au système car dans le cas

d'un système asynchrone, il n'y a pas d'horloge centralisée. Dans le cas où le système est synchrone, le temps d'exécution parallèle est déterminé par le nombre de cycles d'horloge décomptés du début de l'application à sa terminaison. La complexité en temps d'une application parallèle est égale au temps d'exécution maximale de l'algorithme pour une exécution quelconque.

2.8.2 Complexité en communication

Quand les communications sont explicites, il est possible de définir une complexité en communication. Cette notion est très importante pour la mesure de complexité du modèle DRAM.

Par contre, en PRAM, les communications sont implicites. Dans ce modèle, l'écriture ou la lecture d'une information mémoire s'effectue en un cycle d'horloge.

La définition suivante peut être proposée quand les communications sont explicites : « La complexité en communication d'une application parallèle est égale au nombre maximal de messages échangés (ou le nombre maximal d'octets échangés) par les processeurs au cours d'une exécution quelconque de l'algorithme » [39].

La complexité en communication est fondamentale dans le cas de l'algorithmique distribuée, car ce sont les communications qui déterminent les performances de l'application parallèle implique minorer le nombre de communications.

Il est parfois utile de dupliquer le calcul d'une valeur sur une partie des processeurs plutôt que d'attendre qu'un processeur diffuse ce résultat.

2.8.3 Complexité en mémoire

En algorithmique, on ne devrait pas se soucier de la méthode d'implémentation, mais si on conçoit un algorithme qui utilise de grandes structures de données, on va avoir une grande complexité en mémoire surtout pour les problèmes complexes. Parfois, un compromis entre la taille des structures de données et la complexité en calcul (temps) doit être fait pour concevoir des algorithmes optimaux.

2.9 Conclusion

Dans ce chapitre nous avons présenté différentes notions de parallélisme, tels que les algorithmes parallèles et les méthodes de leurs conceptions, et nous avons discuté des sources de parallélisme, comment mesurer les performances de ces algorithmes et enfin nous avons invoqué les facteurs de complexité des algorithmes. Ces différentes notions seront utiles pendant le reste de notre étude concernant la parallélisation de l'algorithme B&B.

Chapitre 3

Présentation d'outils de parallélisation OpenMP & MPI

3.1 Introduction

Dans ce chapitre, nous allons présenter deux outils de parallélisation qui sont OpenMP et MPI. On a choisi ces deux outils car ils sont les plus utilisés dans le domaine de la parallélisation. Après une présentation de ces outils, on va présenter une synthèse des avantages et inconvénients de chacun d'eux.

3.2 Open MP (Open Multi Processing)

3.2.1 Historique

La tentative de standardisation de PCF (*Parallel Computing Forum*) n'a jamais été adoptée par les instances officielles de normalisation. Le 28 octobre 1997, une majorité importante d'industriels et de constructeurs ont adopté OpenMP (*Open Multi Processing*) comme un standard dit < industriel >. Les spécifications d'OpenMP appartiennent aujourd'hui à l'ARB (*Architecture Review Board*), qui est le seul organisme chargé de son évolution [40].

Oct 1997: Fortran version 1.0

Oct 1998: C/C++ version 1.0

Nov 2000: Fortran version 2.0

Mar 2002: C/C++ version 2.0

May 2005: C/C++ et Fortran version 2.5

May 2008 : Spécification complète version 3.0

Juliet 2013 : Spécification complète version 4.0

3.2.2 C'est quoi OpenMP ?

Une Interface de programmation (API) qui peut être utilisée pour diriger explicitement le parallélisme multi-thread à mémoire partagée.

L'API comprend trois composants primaires :

- Directives de compilateur
- Bibliothèque d'exécution

- Variables d'environnement.

3.2.3 Concepts Généraux

Un programme OpenMP est une alternance de régions séquentielles et de régions parallèles [41] (Figure 19).

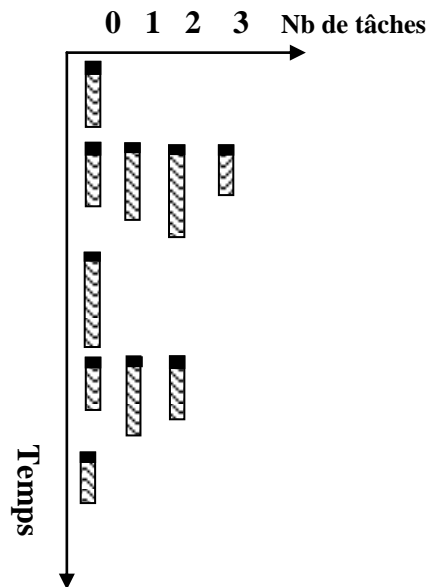


Figure 19: Exécution des tâches OpenMP.

Une région séquentielle est toujours exécutée par la tâche maître, celle dont le rang vaut 0.

Une région parallèle peut être exécutée par plusieurs tâches à la fois.

Les tâches peuvent se partager le travail contenu dans la région parallèle.

Le partage du travail consiste essentiellement à (Figure 20) :

- exécuter une boucle par répartition des itérations entre les tâches ;
- exécuter plusieurs sections de code mais une seule par tâche ;
- exécuter plusieurs occurrences d'une même procédure par différentes tâches (*orphanning*).

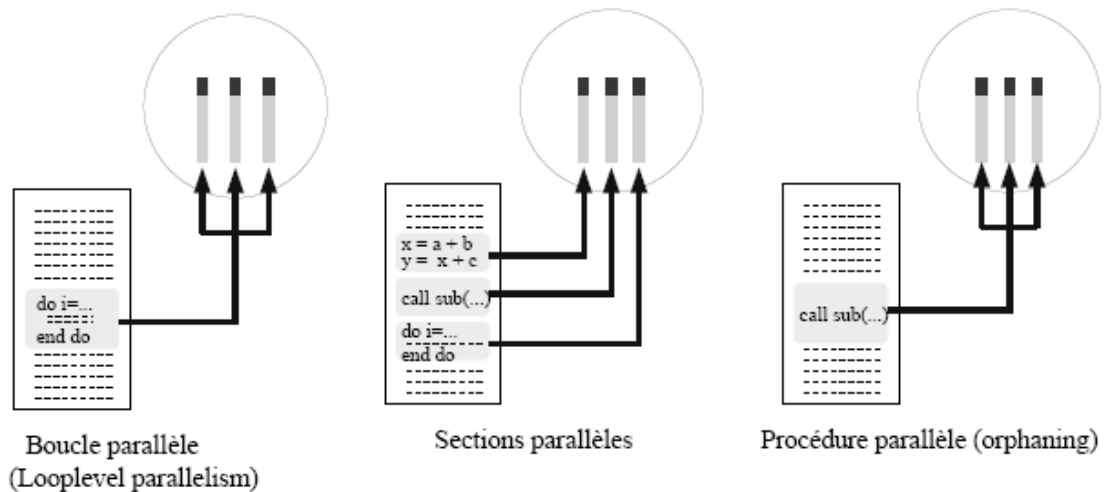


Figure 20 : Partage de travail OpenMP.

Les tâches sont affectées aux processeurs par le système d'exploitation. Différents cas peuvent se produire :

Au mieux, à chaque instant, il existe une tâche par processeur avec autant de tâches que de processeurs dédiés pendant toute la durée du travail ;

Au pire, toutes les tâches sont traitées séquentiellement par un et un seul processeur ;

En réalité, pour des raisons essentiellement d'exploitation sur une machine dont les processeurs ne sont pas dédiés, la situation est en général intermédiaire.

3.2.4 Structure OpenMP

La structure d'OpenMP est constituée de trois parties essentielles (Figure 21) qui sont :

3.2.4.1 Directives et clauses de compilation

- elles servent à définir le partage du travail, la synchronisation et le statut privé ou partagé des données ;

- elles sont considérées par le compilateur comme des lignes de commentaires à moins de spécifier une option adéquate de compilation pour qu'elles soient interprétées.

3.2.4.2 Fonctions et procédures

Ils font partie d'une bibliothèque (appelée aussi *Run-time Library*) chargée à l'édition de liens du programme.

3.2.4.3 Variables d'environnement

Une fois positionnées, leurs valeurs sont prises en compte à l'exécution.

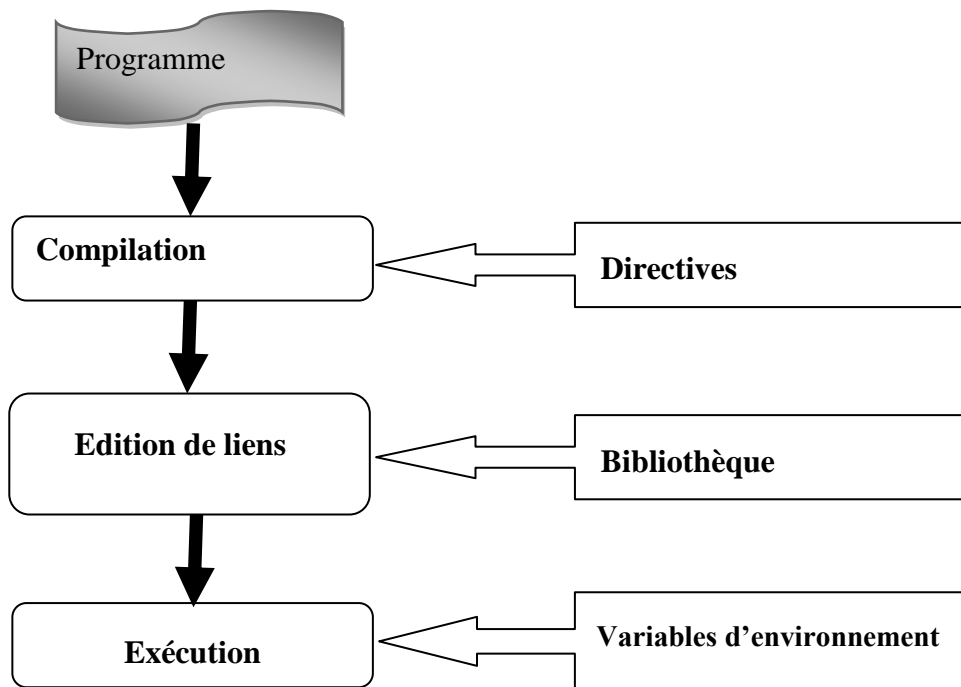


Figure 21 : Structure d'OpenMP.

3.2.5 Principe d'OpenMP

Il est à la charge du développeur d'introduire des directives OpenMP dans son code [42] (du moins en l'absence d'outils de parallélisation automatique).

A l'exécution du programme, le système d'exploitation construit une région parallèle sur le modèle <fork and join>.

A l'entrée d'une région parallèle, la tâche maître crée/active (fork) des processus <fils> (processus légers) qui disparaissent/s'assoupissent en fin de région parallèle (join)

pendant que la tâche maître poursuit seule l'exécution du programme jusqu'à l'entrée de la région parallèle suivante.

Un programme OpenMP commence comme un thread seul d'exécution, appelé le thread initial. Le thread initial s'exécute séquentiellement.

Lorsque tout thread rencontre une directive parallèle, le thread crée une équipe de lui-même et zéro ou plus thread supplémentaires et devient le maître de la nouvelle équipe. Un ensemble de tâches implicites, un par thread, est produit. Le code pour chaque tâche est défini par le code à l'intérieur de la directive parallèle. Chaque tâche est assignée à un thread différent dans l'équipe et lui est attachée; cette tâche est toujours exécutée par le thread auquel elle a été assignée initialement. La région de la tâche qui est exécutée par le thread est suspendue et chaque membre de la nouvelle équipe exécute sa tâche implicite. Il y a une barrière implicite à la fin de la directive parallèle. Au-delà de la fin de la directive parallèle, seul le thread maître continue l'exécution, en reprenant la région de la tâche qui a été suspendue à la rencontre de la directive parallèle. Les régions parallèles peuvent être arbitrairement imbriquées les unes à l'intérieur des autres.

Lorsque toute équipe rencontre une directive de partage de travail, le travail à l'intérieur de la directive est divisé entre les membres de l'équipe et exécuté d'une manière coopérative au lieu d'être exécuté par chaque thread. Il y a une barrière facultative à la fin de chaque directive de partage de travail.

Lorsque tout thread rencontre une directive de tâche, une nouvelle tâche explicite est produite. L'exécution de tâches explicitement produites est assignée à un des threads dans l'équipe courante, selon la disponibilité du thread à exécuter le travail. Donc, l'exécution de la nouvelle tâche pourrait être immédiate ou différée. Les threads sont autorisés à suspendre la tâche courante à un point de synchronisation afin d'exécuter une tâche différente [43].

Les directives de synchronisation et routines de bibliothèque sont disponibles dans OpenMP pour coordonner l'accès aux tâches et données dans les régions parallèles. De plus, les routines de bibliothèque et les variables d'environnement sont disponibles pour contrôler l'environnement et la durée des programmes OpenMP.

3.2.6 Les Variables de contrôle Interne (ICVs)

Une implémentation d'un programme avec OpenMP doit agir comme s'il y avait des variables de contrôle interne (ICVs) qui contrôlent le comportement d'un programme OpenMP. Ces ICVs stockent des informations, telles que le nombre de threads utilisables avec les régions parallèles, l'ordonnancement à utiliser pour le partage de travail, etc. Les ICVs peuvent recevoir des valeurs pendant l'exécution du programme. Elles sont initialisées par le programme lui-même et peuvent recevoir des valeurs à travers les variables d'environnement OpenMP et à travers les appels aux routines de l'API OpenMP. On peut citer :

nthreads_var qui est modifiée par la directive *omp_set_num_thread()* : définit le nombre max de threads dans un groupe. On récupère sa valeur avec la fonction *omp_get_max_threads()*.

nest-var est modifié par la directive *Omp_set_nested()* : définit si on peut avoir des régions parallèles imbriquées ou non. On récupère sa valeur avec la fonction *omp_get_nested()*.

3.2.7 Syntaxe générale d'une directive

sentinelle directive [clause [clause]...]

C'est une ligne qui doit être ignorée par le compilateur si l'option permettant l'interprétation des directives OpenMP n'est pas spécifiée. (Exemple : -fopenmp avec gcc)

La sentinelle est de la forme **#pragma** avec C et **!\$OMP** ou **C\$OMP** avec Fortran
Les directives sont :

3.2.7.1 Directive « parallel »

C'est la directive essentielle qui commence le traitement parallèle. Sa syntaxe est la suivante :

```
#pragma omp parallel [clause[ [, ]clause] ...] new-line
    structured-block
```

Où clause est l'un de ce qui suit :

```
if(scalar-expression)
num_threads(integer-expression)
```

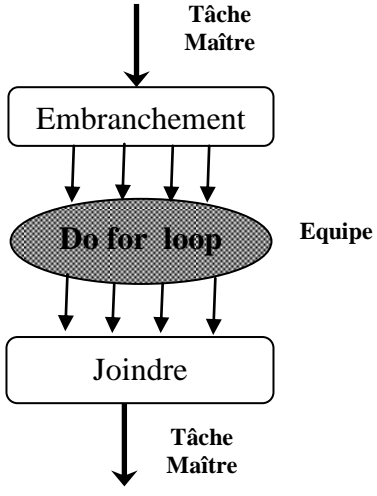
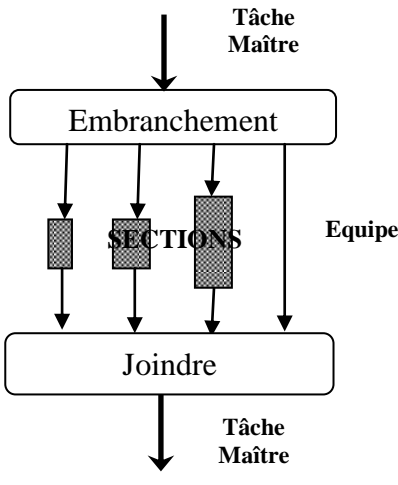
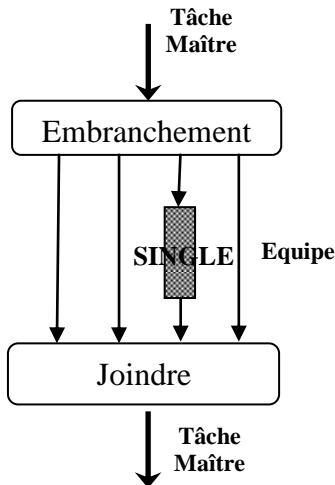
default(*shared / none*)
 private(*list*)
 firstprivate(*list*)
 shared (*list*)
 copyin(*list*)
 reduction(*operator: list*)

3.2.7.2 Directives de partage de travail

Les directives de partage de travail divisent l'exécution du code de la région, entre les membres de la même équipe qui rencontre cette région.

La directive de partage de travail ne charge pas de nouveau thread. Il n'y a aucune barrière implicite à l'entrée d'une directive de partage de travail, cependant il y a une barrière implicite à la fin d'une directive de partage du travail.

Types de directive de partage de travail :

<p>DO / for –</p> <p>Partage les itérations d'une boucle entre les éléments d'une équipe. C'est un type de « parallélisme de données ».</p>	<p>SECTIONS –</p> <p>Divise le travail en sections séparées, discrètes. Chaque section est exécutée par un thread. Qui peut être utilisé pour implémenter un type de "parallélisme fonctionnel."</p>	<p>SINGLE –</p> <p>Sérialise une section du code.</p>
		

La directive de partage de travail DO / for

La directive DO/for spécifie que les itérations de la boucle qui la suit immédiatement doivent être exécutées en parallèle par les éléments d'une équipe de thread. Ce qui sous-entend qu'une région parallèle a déjà été initialisée, sinon elles s'exécutent en série sur un seul processeur.

```
#pragma omp for [clause ...] newline
    schedule (type [,chunk])
    ordered
    private (list)
    firstprivate (list)
    lastprivate (list)
    shared (list)
    reduction (operator: list)
    nowait
for_loop
```

3.2.7.3 Directives combinées Parallèle -partage de travail

OpenMP fournit trois directives combinées :

PARALLEL DO / parallel for

PARALLEL SECTIONS

PARALLEL WORKSHARE (fortran seulement)

3.2.7.4 Directives de synchronisation

OpenMP fournit une variété de directives de synchronisation qui contrôlent l'exécution de chaque thread par rapport aux autres threads de l'équipe.

On énumère les directives suivantes :

Directive : MASTER

```
#pragma omp master newline
structured_block
```

Directive : CRITICAL

```
#pragma omp critical [ name ] newline
```

```
    structured_block
```

Directive : BARRIER

```
#pragma omp barrier newline
```

Directive : ATOMIC

```
#pragma omp atomic newline
```

```
    statement_expression
```

Directive : FLUSH

```
#pragma omp flush (list) newline
```

Directive : ORDERED

```
#pragma omp for ordered [clauses...]
```

```
    (loop region)
```

```
#pragma omp ordered newline
```

```
    structured_block
```

```
    (endo of loop region)
```

Directive THREADPRIVATE

L'utilisation de la directive THREADPRIVATE permet de privatiser une instance statique et faire que celle-ci soit persistante d'une région parallèle à une autre.

Clause	Directive					
	PARALLEL	DO/for	SECTIONS	SINGLE	PARALLEL DO/for	PARALLEL SECTIONS
IF	●				●	●
PRIVATE	●	●	●	●	●	●
SHARED	●	●			●	●
DEFAULT	●				●	●
FIRSTPRIVATE	●	●	●	●	●	●
LASTPRIVATE		●	●		●	●
REDUCTION	●	●	●		●	●
COPYIN	●				●	●
SCHEDULE		●			●	
ORDERED		●			●	
NOWAIT		●	●	●		

Tableau 2 : Récapitulatif des clauses OpenMP

Les directives suivantes n'acceptent pas de clauses :

MASTER

CRITICAL

BARRIER

ATOMIC

FLUSH

ORDERED

THREADPRIVATE

Le Tableau 2 donne un récapitulatif des clauses de OpenMP et leur utilisation.

3.2.8 Les Variables d'environnements

Open MP fournit 4 variables d'environnements pour contrôler l'exécution du code parallèle. (Elles sont en Majuscule)

OMP_SCHEDULE

Détermine comment les itérations de la boucle seront ordonnancées sur les processeurs :

```
setenv OMP_SCHEDULE "guided, 4" setenv OMP_SCHEDULE "dynamic"
```

OMP_NUM_THREADS

Définit le nombre maximal de threads à utiliser pendant l'exécution :

```
setenv OMP_NUM_THREADS 8 OMP_DYNAMIC
```

Active ou désactive l'ajustement du nombre des threads disponibles pour l'exécution d'une région parallèle. (TRUE ou FALSE) :

```
setenv OMP_DYNAMIC TRUE
```

OMP_NESTED

Active ou désactive l'imbrication des régions parallèles (TRUE ou FALSE) :

```
setenv OMP_NESTED TRUE
```

3.2.9 Construction des régions parallèles

Dans une région parallèle, par défaut, le statut des variables est partagé.

Au sein d'une même région parallèle, toutes les tâches concurrentes exécutent le même code. Il existe une barrière implicite de synchronisation en fin de région parallèle.

Il est interdit d'effectuer des < branchements > (ex. GOTO, CYCLE, etc.) vers l'intérieur ou vers l'extérieur d'une région parallèle ou de toute autre construction OpenMP.

Il est possible, grâce à la clause DEFAULT, de changer le statut par défaut des variables dans une région parallèle.

Si une variable possède un statut privé (PRIVATE), elle se trouve dans la pile de chaque tâche. Sa valeur est alors indéfinie à l'entrée d'une région parallèle.

3.3 MPI (Message Passing Interface)

L'interface de passage de message MPI est une librairie de passage de message standard basée sur le consensus du forum MPI, qui est constitué de 40 organisations de fournisseurs, chercheurs, développeurs de bibliothèques de logiciels et d'utilisateurs. Le but de l'interface de passage de message est d'établir un standard portable, efficace et flexible pour le passage de message qui sera largement utilisé pour l'écriture de programmes de passage de messages. Alors MPI est le premier standard de librairie de passage de message indépendant des fournisseurs. L'avantage de développement de logiciels qui utilisent MPI c'est d'atteindre les buts de portabilité, efficacité et flexibilité. MPI n'est pas un standard IEEE ou ISO, mais il est devenu un standard pour l'industrie pour l'écriture de programme de passage de messages sur les plateformes de Calcul haute performance (*HPC High Performance Computing*)[44].

3.3.1 Concepts de l'échange de messages

Un message est constitué de paquets de données transitant du processus émetteur au(x) processus récepteur(s). Les messages sont envoyés à des processus qui doivent les recevoir.

En plus des données (variables scalaires, tableaux, etc.) à transmettre, un message doit contenir les informations suivantes :

- l'identificateur du processus émetteur ;
- le type de la donnée ;
- sa longueur ;
- l'identificateur du processus récepteur.

Les messages échangés sont interprétés et gérés par un environnement qui peut être comparé à la téléphonie, à la télécopie, au courrier postal, à la messagerie électronique.

Le message est envoyé à une adresse déterminée.

Le processus récepteur doit pouvoir classer et interpréter les messages qui lui ont été adressés.

L'environnement en question est MPI (Message Passing Interface). Une application MPI est un ensemble de processus autonomes exécutant chacun son propre code et communiquant via des appels à des sous-programmes de la bibliothèque MPI.

Ces sous-programmes peuvent être classés dans les grandes catégories suivantes :

- environnement ;
- communications point à point ;
- communications collectives ;
- types de données dérivées ;
- topologies ;
- groupes et communicateurs.

3.3.2 Les objectifs de MPI

Le but de l'Interface de Passage de Message MPI, simplement énoncé, est de développer une norme largement utilisée pour écrire des programmes avec passage de messages. Pour cela l'interface devrait établir une norme pratique, portable, efficace et flexible pour le passage de messages [44].

On peut citer les buts de MPI suivants :

- Concevoir une API. Bien que MPI soit utilisé actuellement comme un environnement d'exécution pour les compilateurs parallèles et pour plusieurs bibliothèques, le dessin de MPI reflète à l'origine les besoins perçus par les programmeurs d'applications.
- Autoriser la communication efficace. Évitez la copie de mémoire-à-mémoire et autoriser le recouvrement de calcul et communication.
- Tenir compte des implémentations qui peuvent être utilisées dans un environnement hétérogène.
- Prévoir une utilisation avec C et Fortran 77. Aussi, la sémantique de l'interface devrait être indépendante du langage.
- Fournir une interface de communication fiable. L'utilisateur ne doit pas s'occuper des pannes de communication.

- Définir une interface pas trop différente des pratiques courantes, tel que PVM, NX, Express, p4, etc., et fournit des extensions qui autorisent une plus grande souplesse.
- Définir une interface qui peut être implémentée sur les plates-formes des différents vendeurs.

3.3.3 Environnement

Toute unité de programme appelant des sous-programmes MPI doit inclure un fichier d'en-têtes. En Fortran, il faut maintenant utiliser le module mpi introduit dans MPI-2 (dans MPI-1, il s'agissait du fichier mpif.h), et en C/C++ le fichier mpi.h.

- Le sous-programme MPI_INIT() permet d'initialiser l'environnement nécessaire.
- Réciproquement, le sous-programme MPI_FINALIZE() désactive cet environnement.
- Toutes les opérations effectuées par MPI portent sur des *communicateurs*. Le communicateur par défaut est MPI_COMM_WORLD qui comprend tous les processus actifs.
- A tout instant, on peut connaître le nombre de processus gérés par un communicateur donné par le sous-programme MPI_COMM_SIZE().
- De même, le sous-programme MPI_COMM_RANK() permet d'obtenir le rang d'un processus (i.e. son numéro d'instance, qui est un nombre compris entre 0 et la valeur renvoyée par MPI_COMM_SIZE() – 1).

3.3.4 Communication Point à point

Une communication dite point à point a lieu entre deux processus, l'un appelé processus émetteur et l'autre processus récepteur (ou destinataire).

- L'émetteur et le récepteur sont identifiés par leur rang dans le communicateur.
- Ce que l'on appelle l'enveloppe d'un message est constituée :
 - du rang du processus émetteur ;

- du rang du processus récepteur ;
 - de l'étiquette (tag) du message ;
 - du nom du communicateur qui définira le contexte de communication de l'opération.
- Les données échangées sont typées (entiers, réels, etc. ou types dérivés personnels).
 - Il existe dans chaque cas plusieurs modes de transfert, faisant appel à des protocoles différents.
 - A la réception d'un message, le rang du processus et l'étiquette peuvent être des « jokers », respectivement MPI_ANY_SOURCE et MPI_ANY_TAG.
 - Une communication avec le processus « fictif » de rang MPI_PROC_NULL n'a aucun effet.
 - Il existe des variantes syntaxiques, MPI_SENDRECV() et MPI_SENDRECV_REPLACE(), qui enchaînent un envoi et une réception.
 - On peut créer des structures de données plus complexes à l'aide de sous-programmes tels que MPI_TYPE_CONTIGUOUS(), MPI_TYPE_VECTOR(), MPI_TYPE_INDEXED() et MPI_TYPE_STRUCT()

3.3.5 Communication collective

Les communications collectives permettent de faire en une seule opération une série de communications point à point.

Une communication collective concerne toujours tous les processus du communicateur indiqué.

Pour chacun des processus, l'appel se termine lorsque la participation de celui-ci à l'opération collective est achevée, au sens des communications point-à-point (donc quand la zone mémoire concernée peut être modifiée).

Il est inutile d'ajouter une synchronisation globale (barrière) après une opération collective.

La gestion des étiquettes dans ces communications est transparente et à la charge du système. Elles ne sont donc jamais définies explicitement lors de l'appel à ces sous-programmes. Cela a entre autres pour avantage que les communications collectives n'interfèrent jamais avec les communications point à point.

Il y a trois types de sous-programmes pour la communication globale:

- 1 - celui qui assure les synchronisations globales : `MPI_BARRIER()`.
- 2 - ceux qui ne font que transférer des données :
 - diffusion globale de données : `MPI_BCAST()` ;
 - diffusion sélective de données : `MPI_SCATTER()` ;
 - collecte de données réparties : `MPI_GATHER()` ;
 - collecte par tous les processus de données réparties : `MPI_ALLGATHER()` ;
 - diffusion sélective, par tous les processus, de données réparties : `MPI_ALLTOALL()`.
- 3 - ceux qui, en plus de la gestion des communications, effectuent des opérations sur les données transférées :
 - opérations de réduction, qu'elles soient d'un type prédéfini (somme, produit, maximum, minimum, etc.) ou d'un type personnel : `MPI_REDUCE()` ;
 - opérations de réduction avec diffusion du résultat (il s'agit en fait d'un `MPI_REDUCE()` suivi d'un `MPI_BCAST()`) : `MPI_ALLREDUCE()`.

3.3.6 Optimisations

L'optimisation doit être un souci essentiel lorsque la part des communications par rapport aux calculs devient assez importante.

Calcul	Communication
--------	---------------

Calcul	Préparation	Transfert
--------	-------------	-----------

Calcul	Latence	Surcoût	Transfert
--------	---------	---------	-----------

- Latence : temps d'initialisation des paramètres réseaux.
- Surcoût : temps de préparation du message ; caractéristique liée à l'implémentation MPI et au mode de transfert.

L'optimisation des communications peut s'accomplir à différents niveaux dont les principaux sont :

- 1- Recouvrir les communications par des calculs ; (utilisation d'envoi non bloquant).
- 2- Eviter si possible la recopie du message dans un espace mémoire temporaire (buffering) et qui est souvent coûteux en mémoire et en temps.
- 3- Minimiser les surcoûts induits par des appels répétitifs aux sous-programmes de communication. (Utilisation de communications persistantes).

MPI fournit différents modes d'envoi de messages :

- 1 – Standard : il est à la charge de MPI d'effectuer ou non une recopie temporaire du message ou non.
- 2 – Synchronus : L'envoi du message ne se termine que si la réception a été terminée.
- 3 – Buffering : il est à la charge du programmeur de faire une copie temporaire, l'envoi se termine au moment de la recopie.
- 4 – Ready : l'envoi ne peut commencer que si la réception a été postée auparavant (applications client-serveur).

3.3.7 Types de données dérivées

MPI utilise ses propres types de données présentées dans le Tableau 3 :

C Data Types	Fortran Data Types
MPI_CHAR	MPI_CHARACTER
MPI_SHORT	MPI_INTEGER
MPI_INT	MPI_REAL
MPI_LONG	MPI_DOUBLE_PRECISION
MPI_UNSIGNED_CHAR	MPI_COMPLEX
MPI_UNSIGNED_SHORT	MPI_DOUBLE_COMPLEX
MPI_UNSIGNED_LONG	MPI_LOGICAL
MPI_UNSIGNED	MPI_BYTE
MPI_FLOAT	MPI_PACKED
MPI_DOUBLE	
MPI_LONG_DOUBLE	
MPI_BYTE	
MPI_PACKED	

Tableau 3: Type de données de MPI.

On peut créer des structures de données plus complexes à l'aide de sous-programmes tels que `MPI_TYPE_CONTIGUOUS()`, `MPI_TYPE_VECTOR()`, `MPI_TYPE_CREATE_HVECTOR()` (Figure 22).

Les types de données MPI sont contigus, alors que les types de données dérivées permettent de spécifier des données non contiguës de manière convenable et de les traiter comme si elles étaient contiguës.

A chaque fois que l'on crée un type de données, il faut le valider à l'aide du sous-programme `MPI_TYPE_COMMIT()`.

Si on souhaite réutiliser le même type, on doit le libérer avec le sous-programme `MPI_TYPE_FREE()`.

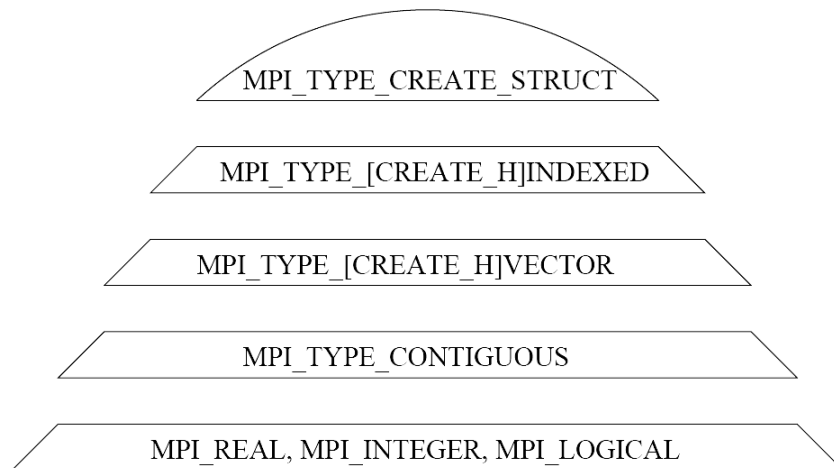


Figure 22: Hiérarchie des constructeurs de types en MPI.

3.3.8 Topologie des processus

Dans MPI, une topologie virtuelle décrit une projection topographique / ordre de processus MPI sous une "forme" géométrique.

Les deux principaux types de topologies supportés par MPI sont : Cartésiens (grille) et Graphe.

Les topologies MPI sont virtuelles - il peut n'y avoir aucune relation entre la structure réelle de la machine parallèle et la topologie du processus.

Les topologies virtuelles sont construites sur des communicateurs MPI et sur des groupes.

Les topologies doivent être implicitement implémentées par le programmeur.

Pourquoi les utiliser ?

Commodité : Les topologies virtuelles peuvent être utiles pour les applications avec des modèles de communication spécifiques - modèles qui soient similaires à une structure de topologie MPI.

Par exemple, une topologie Cartésienne peut être commode pour une application qui exige la communication avec les voisins les plus proches dans les 4-directions.

Efficacité de la communication : Certaines architectures matérielles peuvent présenter des pénalités pour les communications entre des "nœuds" distants.

Une implémentation particulière peut optimiser la projection topologique du processus basée sur les caractéristiques physiques d'une machine parallèle donnée.

Topologies de type cartésien :

- chaque processus est défini dans une grille de processus ;
- la grille peut être périodique ou non ;
- les processus sont identifiés par leurs coordonnées dans la grille.

Topologies de type graphe :

- généralisation à des topologies plus complexes.

3.3.9 Groupes et Communicateurs

Un communicateur est constitué de :

- un groupe de processus.
- un contexte de communication mis en place à l'appel du sous-programme de construction du communicateur (voir Figure 23).

Sachant que :

le groupe constitue un ensemble ordonné de processus ;

le contexte de communication permet de délimiter l'espace de communication ;

les contextes de communication sont gérés par MPI (le programmeur n'a aucune action sur eux : c'est un attribut « caché »).

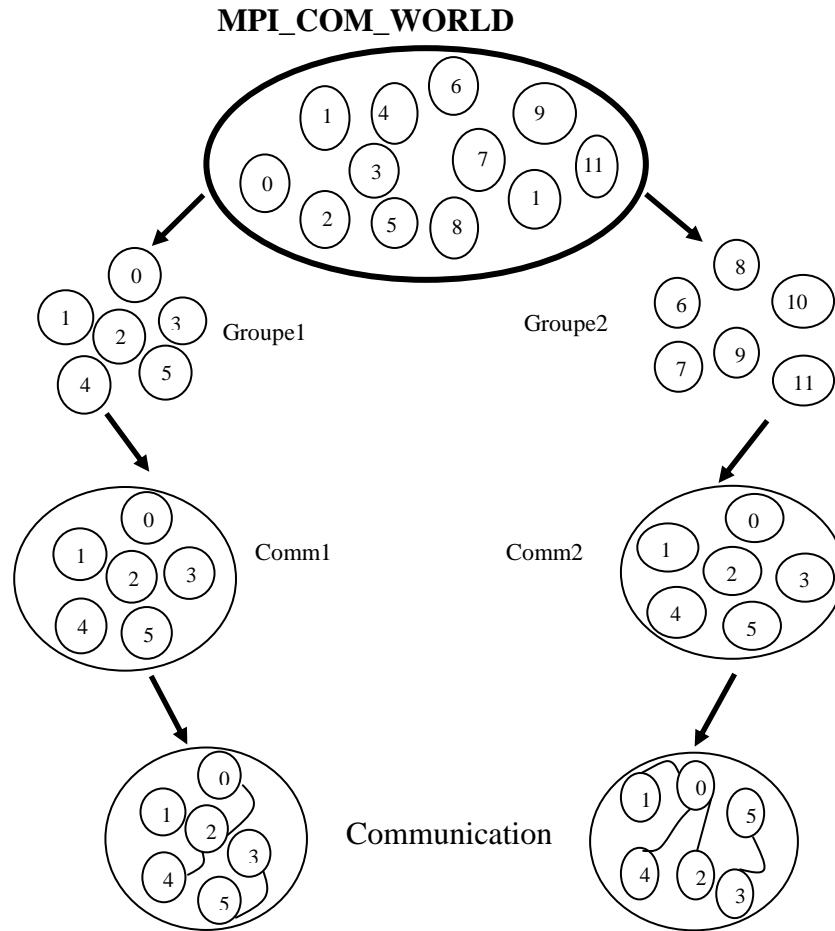


Figure 23: schéma de communication dans MPI

Les buts de l'utilisation des groupes de communications sont :

- Permettre l'organisation des tâches basées sur les fonctions en groupes de tâches.
- Activer des opérations de communication à travers un sous-ensemble de tâches.
- Fournir des bases pour l'implémentation de topologies virtuelles personnalisées.
- Fournir des communications sûres.

3.3.9.1 Programmation avec groupes et communicateurs

Les groupes et commutateurs sont dynamiques, ils peuvent être créés et détruits durant l'exécution du programme.

Un processus peut être dans plus d'un groupe/communicateur. Il a un rang unique dans chaque groupe/communicateur.

MPI fournit plus de 40 fonctions en relation avec les groupes et les communicateurs et les topologies.

Usages :

Extraction du handle du groupe global `MPI_COMM_WORLD` en utilisant `MPI_Comm_group`.

Former un nouveau groupe comme un sous-ensemble du groupe global avec `MPI_Group_incl`.

Créer un nouveau communicateur pour le nouveau groupe avec `MPI_Comm_Create`

Déterminer le nouveau rang dans le nouveau communicateur avec `MPI_Comm_Rank`

Continuer la communication en utilisant les fonctions de passage de messages de MPI.

A la fin, libérer les communicateurs et groupes (optionnels) avec `MPI_Comm_free` et `MPI_Group_free`

3.3.10 Installation et exécution de programme MPI (exemple MPICH)

Plusieurs implémentations portables de MPI existent, on peut citer :

- l'implémentation MPICH d'Argonne Laboratoire National et Mississippi Etat Université, et peut fonctionner sur plusieurs systèmes.

- L'implémentation CHIMP d'Edimbourg Parallel Computing Center.

- l'implémentation LAM de l'Ohio Supercomputing Centre, une implémentation MPI complètement standard qui utilise LAM, un environnement de la grappe de terminaux UNIX.

3.3.10.1 MPICH

On peut débiter avec MPICH qui est gratuit et répondu [45].

1 – télécharger MPICH (`mpich.tar.gz`) sur

www.mcs.anl.gov/mpi/mpich/download.html;

2 – décompresser le fichier sur une partition locale.

3 – Configurer MPICH :

```
% ./configure --prefix=/usr/local/mpich-1.2.7
```

On choisi le préfix `--with-device=` pour spécifier l'environnement de travail

`ch_shmem` : pour une machine à mémoire partagée.

`ch_p4mpd` : réseau de stations homogènes.

`ch_p4` : réseau

`globus2` = pour les grilles de calcul

Le but de cette configuration est de déterminer les caractéristiques du system et de créer le fichier Makefile qui va servir pour compiler MPICH.

4 – Compiler MPICH

```
% make |& tee make.log
```

5 – compiler des programmes avec MPICH

```
mpicc -c foo.c
```

```
mpif77 -c foo.f
```

```
mpicxx -c foo.cxx
```

```
mpif90 -c foo.f90
```

6 – édition de liens :

```
mpicc -o foo foo.o
```

```
mpif77 -o foo foo.o
```

```
mpicxx -o foo foo.o
```

```
mpif90 -o foo foo.o
```

7 – Exécution de programmes avec mpirun :

La commande mpirun est située à `/usr/local/mpich-1.2.7/bin`

Pour la plupart des systèmes on peut écrire :

```
mpirun -np 4 a.out
```

le programme a.out s'exécutera sur 4 processeurs.

3.3.11 Les routines MPI

MPI est caractérisé par l'utilisation d'un grand nombre de routines, ce qui présente un avantage pour la spécialisation et la précision de la programmation, mais un inconvénient au fait de la difficulté pour les utiliser. On peut diviser ces routines en catégories : routines de gestion de l'environnement, routines de communication point à point ou communication collective, routine de gestion de groupe et autres.

3.4 Conclusion

OpenMP offre une multitude de directives qui permet de réaliser des programmes parallèles ou de paralléliser un code existant en toute simplicité, ce qui lui donne des avantages d'être lisible et évolutif. Malgré qu'OpenMP présente des limites quand le nombre de processus augmente, il reste l'une des bibliothèques les plus utilisées pour la parallélisation.

On remarque que MPI contient un grand nombre de fonctions ce qui est handicapant pour certain, mais là aussi réside sa force car il a été créé pour être riche en fonctionnalité, ce qui est reflété par l'utilisation de types de données dérivées, l'utilisation de groupes et communicateurs et les topologies virtuelles, et en second plan pour s'adapter à la diversité et complexité des machines parallèles. L'un des buts essentiels de la création de MPI est l'écriture de programmes portables et indépendants des machines parallèles, où on compile et on exécute les programmes.

Chapitre 4

Nouvelle approche de parallélisation et implémentation

4.1 Introduction

Les chapitres précédents ont donné une vision globale des différentes classes d'algorithmes B&B parallèles avec différents critères et objectifs. Durant ce chapitre, nous allons présenter une nouvelle approche pour B&B basée sur des expérimentations qui essaient d'améliorer les performances de l'algorithme parallèle sur une machine à mémoire partagée. Différents critères du B&B parallèle vont être mis en épreuve pour converger vers un algorithme qui soit le plus performant possible.

4.2 Approche de parallélisation proposée

La littérature concernant des algorithmes parallèles pour B&B est très variée, on peut citer [46] qui étudie et synthétise le parcours de l'arbre de recherche en profondeur (DFS) ou en meilleur d'abord (BeFS) et les cas où utiliser chacun d'eux, [47] avec une variante de l'algorithme B&B "paresseux" et "impatience" (*lazy* et *eager*) qui traite l'évaluation des limites avant ou après la mise dans la file d'attente en plus du stockage des informations relatives aux nœuds, ou encore [21] qui s'est basé sur l'introduction de l'aléatoire dans le parcours parallèle. Dans [48], les auteurs ont proposé une stratégie pour la division de l'arbre de recherche sur les processeurs, et où chaque processeur implémente une stratégie différente de gestion de l'arbre et de traitement. Par ailleurs, [49] se sont basés sur une implémentation à deux étapes, la première consiste à trouver une première solution le plus vite possible en construisant plusieurs arbres de recherche, et ensuite la deuxième étape consiste à choisir un arbre de recherche et en le parcourant en parallèle.

L'élaboration d'un algorithme parallèle B&B nécessite l'exploitation de tous les critères de parallélisation, vus précédemment, qui constitue les différentes étapes de l'algorithme. Pour cela, nous devons intervenir à tous les niveaux de l'algorithme afin d'améliorer ses performances. L'apport majeur que nous proposons concerne la gestion de l'équilibre de charge en plus des autres paramètres. Dans ce qui suit, nous allons expliquer notre apport dans chacun d'entre eux :

4.2.1 La stratégie de parcours de l'arbre.

Différentes stratégies sont utilisées pour le parcours de l'arbre B&B, les plus utilisées sont BeFS (*Best First Search*) et DFS (*Depth first search*). Nous avons proposé une stratégie de parcours mixte, où un processeur parcourt l'arbre en DFS, et le reste des processeurs parcourent l'arbre en BeFS.

4.2.2 Gérer le grain de parallélisme.

Nous avons choisi un parallélisme à gros grain en raison de la nature de la fonction d'évaluation utilisée pour la résolution des problèmes linéaires, et qui est la méthode du simplexe.

4.2.3 L'équilibre de charge

L'évaluation des charges est difficile à connaître surtout avec des parcours d'arbres irréguliers [50]. Une technique efficace pour assurer un bon équilibre de charges est le lancement d'un nombre de threads supérieur au nombre de processeurs disponibles. Un autre critère que nous avons utilisé est la division en plusieurs sous-problèmes (plus que 2 sous-problèmes) si des processeurs sont au repos. Notons qu'un bon algorithme de régulation de charge ne doit pas consommer beaucoup de temps de calcul durant les différentes étapes d'évaluation et de distribution des charges.

4.2.4 Gestion optimale de la mémoire.

Une bonne gestion de la mémoire consiste en l'utilisation de structure de données adéquate pour optimiser la mémoire. Parfois il faut faire des compromis entre la taille des structures et les calculs à effectuer. En se basant sur les structures utilisées, on libère évidemment l'espace utilisé par les nœuds élagués et plus encore les nœuds parents des nœuds en cours de traitement.

4.2.5 Assurer la communication entre les processeurs utilisés

S'assurer de l'étendue des structures et des variables utilisées, pour assurer la cohérence des données partagées. La minimisation du temps de parcours des listes des nœuds actifs pour minimiser l'attente des autres processeurs concurrents.

4.2.6 Parallélisme de contrôle ou de données

On utilise une double parallélisation, de contrôle pour les processeurs et de données pour les calculs de grain fin. La parallélisation de données ne sera profitable que si la taille des problèmes dépasse un certain seuil qui profitera aux calculs parallèles des structures de données utilisées.

En respectant les critères précédents nous proposons l'algorithme décrit dans la Figure 24.

```

Lire le fichier ;
 $Z_{optim} = -inf$  ; (Meilleure valeur de  $Z$ , initialement = - infini) ;
 $Z_{ent} = -inf$  ; (Meilleure valeur de la solution entière) ;
Faire un premier calcul  $Z_0$  ;
Si (Sol faisable)
     $Z_{optim} = Z_0$  ;
Initialiser l'arbre de recherche ;
Mettre un élément dans la Liste des Eléments actifs ;
Tant Que (Liste non Vide) Faire      (parallèle)
    - Diviser les nœuds actifs sur les cœurs disponibles ;
        Séquentiel
        (
            - Choisir ( $P_i$ ) dans la liste des éléments actifs (selon la stratégie utilisée) ; (A)
        )
    - Calculer  $Z_i$  ;
    - Mettre à jour la valeur de  $Z_{optim}$  pour tous les threads actifs ;
    Si ( $Z_i > Z_{optim}$ )
         $Z_i = Z_{optim}$  (pour utilisation dans la stratégie de recherche) ;
    Si ( $Z_i > Z_{ent}$ )
        Si ( $Z_i$  est une solution entière)
             $Z_{ent} = Z_i$  ;
        Sinon      - Générer 2 Fils ;
            Séquentiel
            ( - Placer les Fils dans la liste des éléments actifs ; )

Fin TQ
Fin Si
Sinon : pas de solution possible ;

```

Figure 24: Algorithme Parallèle Branch and Bound.

4.3 La loi d'Amdahl

La loi d'Amdahl [51] est une formule utilisée pour trouver l'amélioration maximale possible par l'amélioration d'une partie particulière d'un système. Dans le calcul parallèle,

la loi d'Amdahl est principalement utilisée pour prédire l'accélération maximale théorique pour le traitement d'un programme en utilisant plusieurs processeurs.

Selon la loi d'Amdahl dans la parallélisation, si P est la proportion d'un système ou d'un programme qui peut être exécuté en parallèle, et $S = 1-P$ est la proportion qui reste en séquentiel, alors l'accélération maximale qui peut être réalisée en utilisant un nombre N de processeurs est de

$$\text{Accélération} = 1 / ((1-P)+P/N)$$

Si N tend vers l'infini, alors l'accélération maximale a tendance à $1 / (1-P)$.

Dans l'algorithme proposé, les sections séquentielles sont celles qui concernent l'accès et la gestion de la liste d'attente $S = 1 - P$. Le reste des calculs peuvent être parallélisés. Le temps d'accès à la liste d'attente est minime comparé au temps de calcul. Ce qui implique que $P \gg S$, de ce fait :

$$\begin{aligned} \text{Accélération} &= 1 / ((1-P)+P/N) \\ &= 1 / (0+1/N) \\ &= 1 / 1 / N = N \end{aligned}$$

L'accélération théorique maximale est égale à N (le nombre de processeurs) à condition de garder la portion séquentielle minimale. Dans notre cas, cette portion représente l'accès à la liste d'attente.

4.4 Complexité de l'algorithme B&B parallèle

On dit que la complexité d'un algorithme est la complexité de son meilleur algorithme connu. Il a été démontré pour certains algorithmes qu'il n'existe pas de meilleurs algorithmes, qui améliorent la complexité, que ceux déjà existants [52].

La classe des problèmes NP-complets désigne les problèmes de décision, où la solution est oui ou non. Les algorithmes NP-complets n'ont pas d'algorithme qui les résout en un temps polynomial, mais ils ont un algorithme polynomial qui vérifie si une réponse est juste ou fausse [53].

4.4.1 Complexité en temps de calcul

La résolution de problème linéaire en nombres entiers appartient à la classe des problèmes NP-complets, où le temps de résolution ne peut être estimé.

Il existe des méthodes d'estimation du temps de calcul de l'algorithme B&B. Selon [54], on peut prédire le temps de calcul d'un algorithme B&B. Cette prédiction est basée sur l'évaluation du temps moyen de calcul de nœuds après qu'on trouve la première solution, et de multiplier cette moyenne par le nombre de nœuds non traités. Cette prédiction évolue au cours de l'exécution de l'algorithme et le développement de l'arbre.

4.4.2 Complexité en mémoire

La complexité en mémoire représente la taille globale des structures utilisées à un moment donné. Les structures sont l'arbre B&B et la liste d'attente des nœuds actifs. Différentes études se sont intéressées à la prédiction de l'arbre de recherche [55], elle se base sur la proposition de Knuth [56], qui propose que l'estimation de la taille de l'arbre de recherche est la moyenne de plusieurs exécutions. La taille maximale de l'arbre B&B est $1 + d_1 + d_1 d_2 + \dots + \prod_{i=1}^k d_i$, où d_i est le nombre de successeurs du nœud choisi au niveau i , et k le niveau de profondeur atteint.

4.4.3 Complexité en communication

La communication entre les processeurs doit assurer la cohérence des informations partagées, ce qui est assuré par l'utilisation de sections critiques et d'outils de synchronisation. La section critique permet à un seul processeur de modifier certaines données en même temps. Tandis que les outils de synchronisation permettent de garder les différents processus en cours d'exécution dans un certain cadre de travail coopératif.

4.5 Implémentation Parallèle de l'algorithme Branch and Bound

Dans notre travail, nous cherchons la solution d'un problème linéaire en nombres entiers implémenté en utilisant l'algorithme de recherche Branch and Bound et en utilisant la méthode du simplexe pour évaluer la valeur de la fonction objectif. Sachant que la méthode du simplexe fournit une méthode puissante pour la résolution des PL, ce qui implique un arbre réduit de recherche. En effet, le simplexe donne une solution proche de la meilleure solution à chaque instant.

Le deuxième choix important dans notre implémentation est la stratégie de parcours de l'arbre de recherche (le choix de l'élément à traiter dans la liste d'attente des éléments

actifs). Il faut bien noter que nous avons choisi de mettre une liste indépendante de l'arbre pour ne stocker que les éléments actifs et pour pouvoir libérer de la mémoire les nœuds parents qui ont été élagués de l'arbre.

Nous avons implémenté notre algorithme avec trois stratégies différentes de parcours de l'arbre pour pouvoir faire une comparaison, la stratégie en profondeur d'abord (DeFS *Depth First Search*), le meilleur d'abord (BeFS *Best First Search*) et la stratégie en largeur d'abord (BFS *Breadth First search*).

Pour implémenter le parallélisme, nous avons utilisé OpenMP qui est une API permettant de diriger implicitement le parallélisme multithread à mémoire partagée. Elle offre une multitude de directives et de variables d'environnement qui permettent essentiellement le partage de travail et la synchronisation entre les différents threads en cours d'exécution.

4.5.1 Etude expérimentale

Dans le but de mesurer les performances de notre algorithme parallèle, nous avons effectué une multitude de tests avec différentes tailles de problèmes. Il faut toutefois noter qu'il faut s'assurer de l'intégrité des variables critiques qui sont partagées entre les différents processeurs parallèles en utilisant les directives qu'offre l'API OpenMP.

Dans nos tests, nous avons utilisé une machine dotée d'un processeur Intel I7 (3770 3.4 Ghz) doté de 4 cœurs 8 Threads et de 32GO de ram. Le système d'exploitation utilisé est Linux avec la distribution fedora 19.

4.5.2 Génération de l'ensemble de test

Pour le jeu de test, nous avons élaboré un programme qui génère une fonction objectif et un ensemble de contraintes de manière semi-aléatoire. On peut varier le nombre de contraintes, de variables et la plage de valeurs des coefficients de ces dernières. Pour chaque taille de problème (nombre de variables * nombre de contraintes), on génère plusieurs fichiers. Ces fichiers sont stockés dans un répertoire spécifique et sont utilisés comme entrée pour le programme de résolution des problèmes linéaires de manière séquentielle ou parallèle.

4.5.3 Résultats de l'expérimentation

Le but de la parallélisation est d'obtenir un bon niveau d'accélération qui soit le plus proche possible du nombre de processeurs utilisé ou bien encore plus. Durant nos travaux nous avons remarqué que l'accélération est inférieure au nombre de processeurs utilisés et cela est dû au fait que la fonction d'évaluation utilisée (le simplexe) permet, dans un grand nombre de cas, d'élaguer un grand nombre de nœuds actifs, ce qui diminue considérablement le nombre d'éléments dans la liste d'attente et affecte le rendement du parallélisme.

A travers une série d'expérimentations nous allons étudier l'introduction des améliorations successives sur les différents critères de l'algorithme et discuter des résultats.

Variables * Contraintes	Algorithme Parallèle Largeur d'abord (BFS)		Algorithme Parallèle Meilleur d'abord (BeFS)		Algorithme Parallèle Profondeur d'abord (DeFS)		Algorithme Parallèle (DeFS+BeFS)	
	1 CPU	4 CPU	1 CPU	4 CPU	1 CPU	4 CPU	1 CPU	4 CPU
200 * 200	64 Sec	29 Sec	60 Sec	27 Sec	36 Sec	17 Sec	27 Sec	15 Sec
250 * 250	251 Sec	115 Sec	244 Sec	111 Sec	215 Sec	89 Sec	87 Sec	87 Sec
300 * 300	310 Sec	152 Sec	296 Sec	135 Sec	268 Sec	112 Sec	265 Sec	108 Sec

Tableau 4 : Temps d'exécution pour différentes stratégies de parcours de l'arbre.

4.5.3.1 Parcours de l'arbre de recherche

Différentes stratégies sont implémentées pour le parcours de l'arbre B&B. Une synthèse est donnée dans le Tableau 4 sur un ensemble d'exemples. A partir de ces résultats, nous avons choisi d'utiliser surtout la méthode de parcours en profondeur d'abord en choisissant le meilleur au niveau le plus profond. Il convient à ce stade de noter que l'utilisation des parcours en largeur d'abords et meilleur d'abord accroît le nombre de nœuds en attente et l'utilisation de la mémoire en plus des calculs inutiles que cela engendre. Pour notre implémentation parallèle, on a réalisé un parcours mixte (1 processeur en DeFS et le reste en BeFS) comme précédemment décrit, et qui a donné les meilleurs

résultats. La Figure 25 illustre l'accélération moyenne obtenue en utilisant les différents types de parcours de l'arbre.

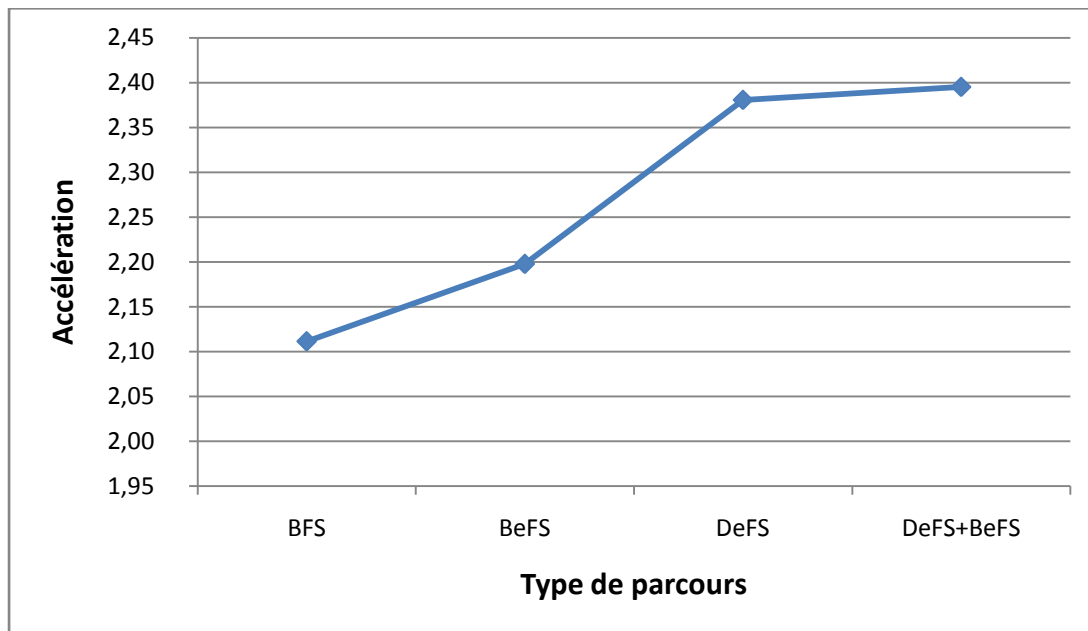


Figure 25: Accélération avec différents type de parcours d'arbre

4.5.3.2 Discussion (Parcours de l'arbre de recherche)

Nous avons choisi cette stratégie car on a constaté que l'utilisation d'un seul type de parcours de l'arbre conduit à des résultats pas très satisfaisants. Dans le cas du parcours DeFS, on trouve rapidement une solution entière satisfaisante, mais la meilleure solution entière globale n'est pas toujours près de cette solution, ce qui nous amène à parcourir tout le reste de l'arbre en DeFS (Figure 26). Tandis que l'utilisation du parcours BFS conduit à une exploration initiale qui accroît l'arbre de recherche avant de trouver une solution entière qui nous permet d'élaguer des branches, de libérer la mémoire et de réduire les calculs. Cette méthode nous permet de découvrir de nouveaux chemins en BFS, tandis que l'un des processeurs essaie de trouver chaque fois une meilleure solution locale en DeFS.

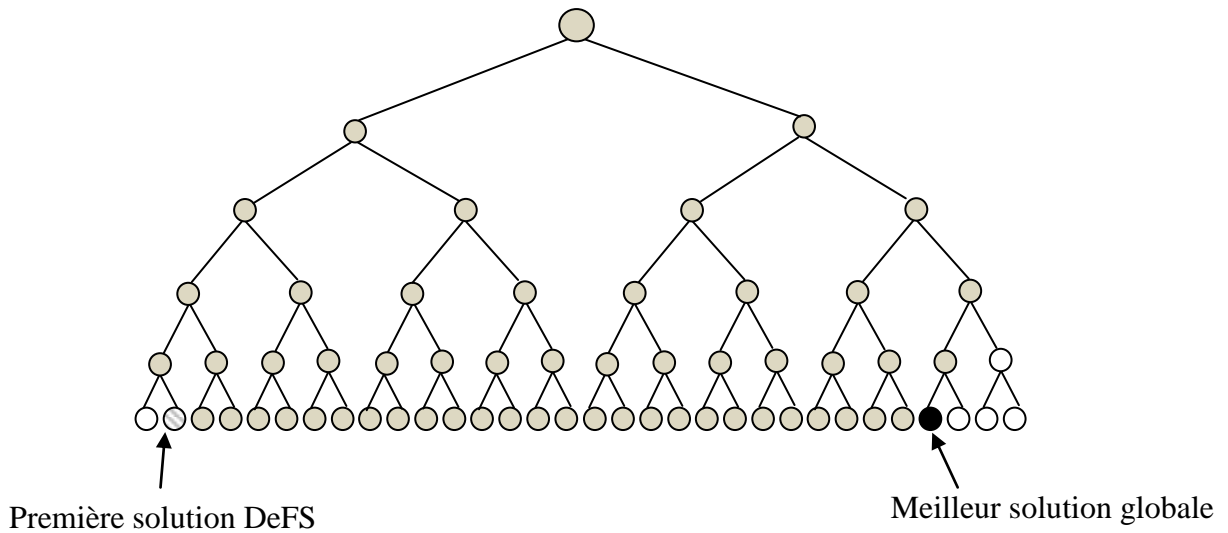


Figure 26: Parcours DeFS avant de trouver la solution optimale.

4.5.3.3 L'équilibre de charges

Durant le déroulement de l'algorithme B&B parallèle, le volume du calcul n'est pas le même pour tous les processeurs, certains processeurs terminent avant les autres. Alors nous avons choisi une augmentation en threads, ce qui consiste à utiliser plus de threads que le nombre de processeurs disponibles. Cette approche comporte une augmentation de la complexité en mémoire en raison de la génération de nœuds supplémentaires, cette augmentation est compensée par les calculs utiles supplémentaires effectués. Il faut noter que dans les processeurs multithreads, les calculs utiles se passent au niveau des cœurs (unité de calcul), alors l'utilisation de threads de processeurs peut probablement engendrer un ralentissement.

Une expérimentation a été menée pour déduire l'efficacité de cette méthode, tout en prenant en considération le nombre de nœuds supplémentaires générés. La Figure 27 résume les résultats obtenus.

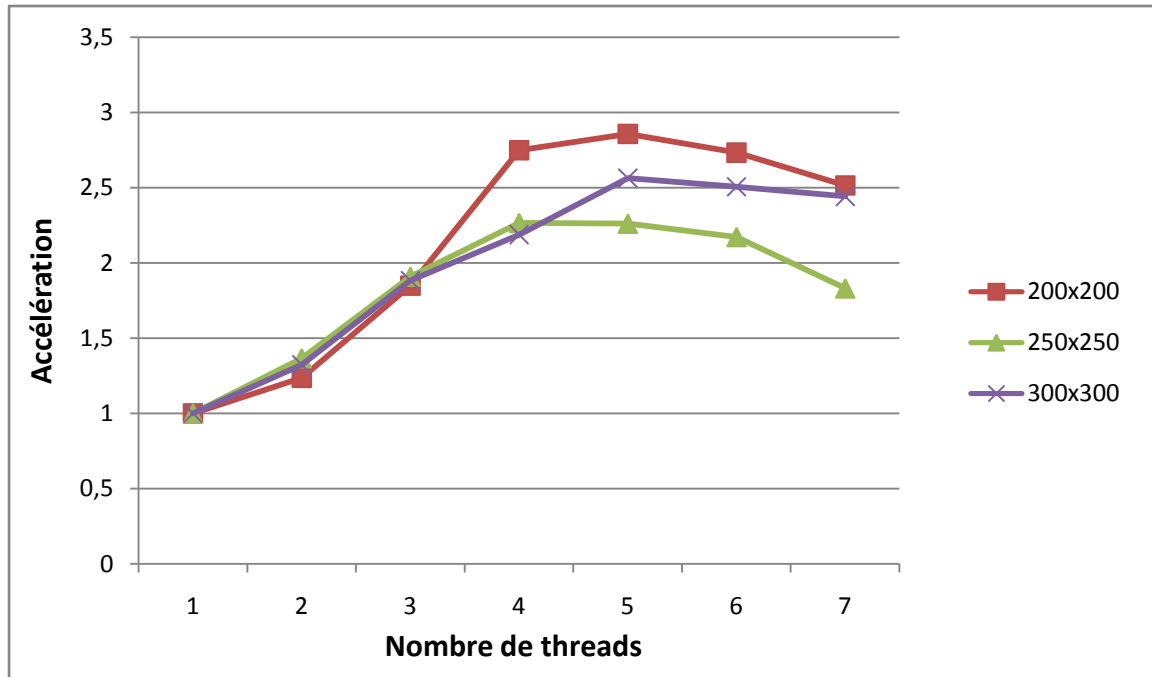


Figure 27: Accélération avec augmentation de threads.

4.5.3.4 Discussion (Equilibre de charges)

L'implémentation de l'augmentation en threads a donné des résultats satisfaisants avec un nombre réduit de threads supplémentaires. On remarque qu'avec une augmentation plus importante de threads on a un résultat inverse, où l'accélération baisse. La Figure 28 donne les résultats d'un ensemble d'expérimentations sur une machine 4 cœurs, l'augmentation avec 1 thread supplémentaire améliore l'accélération et la complexité en mémoire, tandis qu'avec 2 et plus de threads supplémentaires il y a dégradation des résultats. Alors on remarque que le nombre de nœuds augmente et l'accélération baisse.

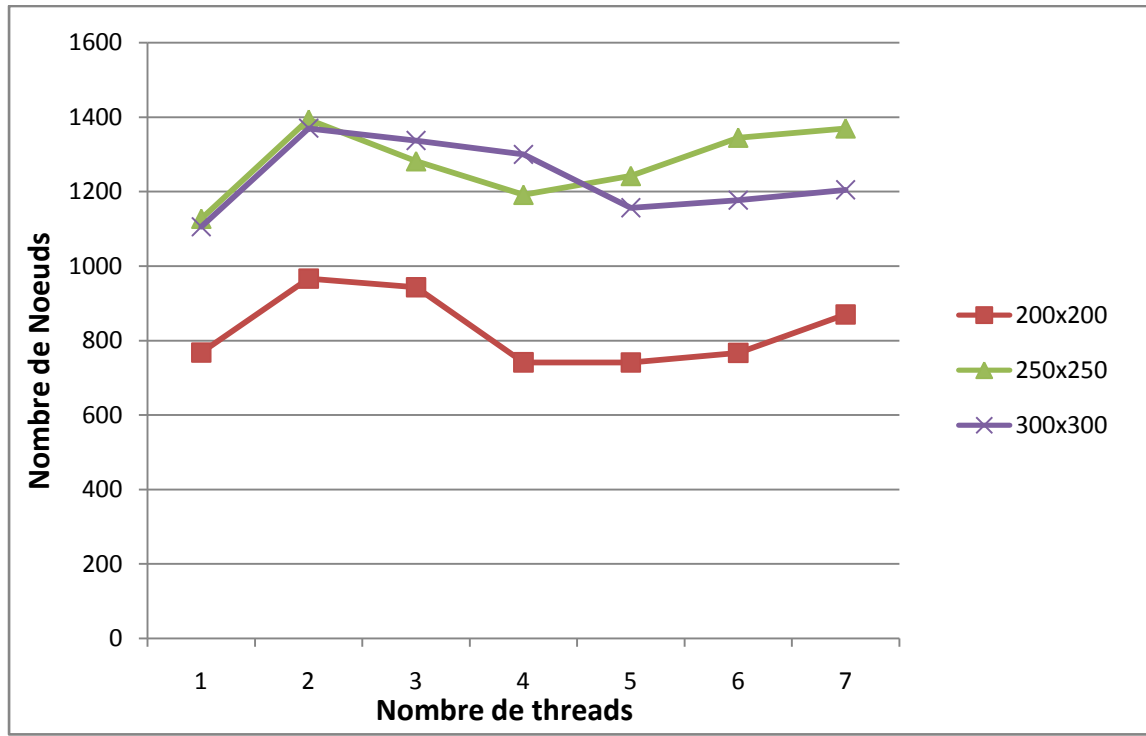


Figure 28: Nombre de Nœuds avec augmentation de threads.

A travers différentes améliorations présentés précédemment on trouve les résultats présenté dans la Figure 29.

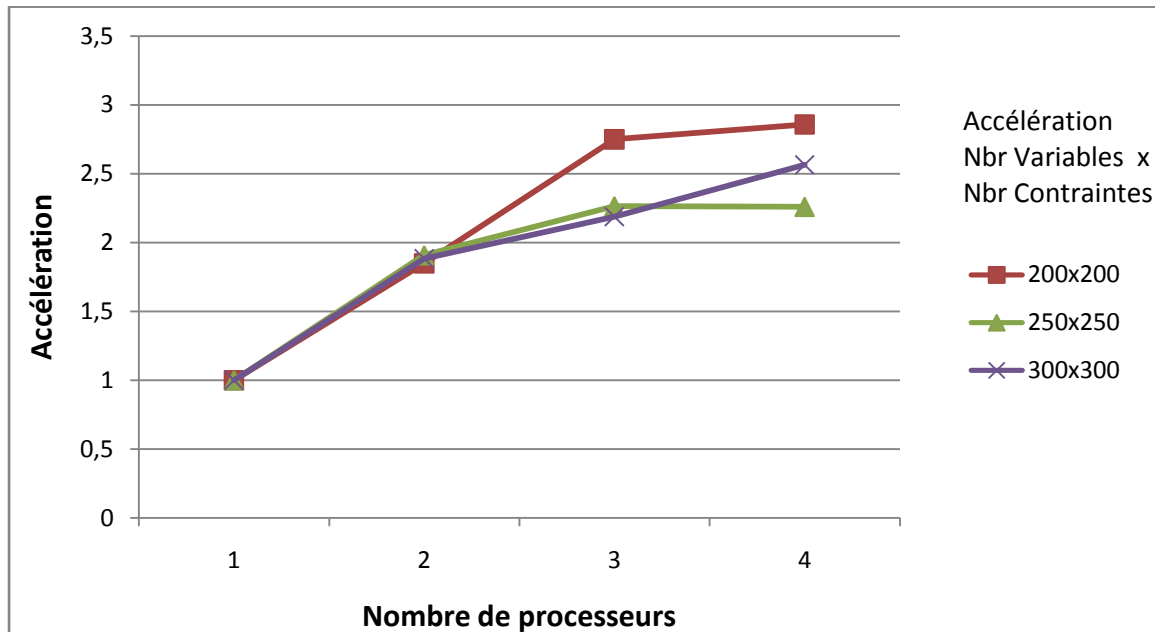


Figure 29: Accélération en fonction des nombres de processeurs.

4.5.4 Discussions globale

Dans certains cas, nous avons constaté une accélération minimale, car la séparation du problème initial donne toujours une branche significative qui prend un grand temps de calcul, et une autre branche qui soit rapidement élaguée (solution rapidement irréalisable), c'est à dire, un arbre irrégulier qui part dans une seule direction. Et dans d'autres cas, la séparation donne deux solutions réalisables et potentielles qui consomment du temps de calcul et génèrent à leur tour d'autres solutions (arbre régulier). Ce dernier cas rend le parallélisme efficace et donne une meilleure accélération. Dans la Figure 30 nous présentons la moyenne de l'accélération d'exécution de 30 problèmes pour des tailles différentes de problèmes (nombre de variables * nombre de contraintes) avec les différentes améliorations partielles obtenues avec le parcours de l'arbre mixte et avec l'augmentation en threads. Nous tenons à signaler que, durant nos expérimentations, nous avons eu une multitude d'exemples avec des accélérations plus que parfaites (accélérations supérieures au nombre de processeurs), et d'autres exemples, plus rares, où nous avons eu un ralentissement (accélérations inférieures à un).

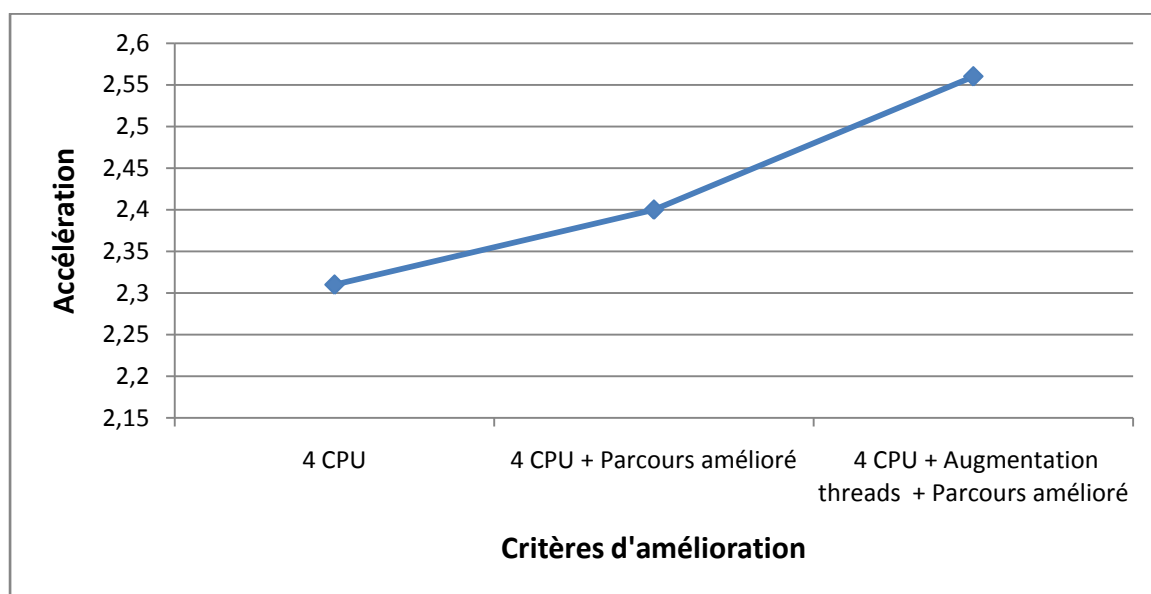


Figure 30: Amélioration progressive de l'accélération.

4.6 Conclusion

Les travaux que nous avons menés sur la résolution de problèmes d'optimisation combinatoire en utilisant la méthode du Branch and Bound et du simplexe nous ont conduits à adopter une série d'améliorations au niveau des critères de l'algorithme B&B entre autre la stratégie de parcours de l'arbre et la distribution de charges. Nous avons constaté qu'une stratégie mixte de parcours de l'arbre est la plus appropriée surtout pour ce type de problème, et que l'implémentation parallèle a donné des résultats satisfaisants suivant le nombre de processeurs de la machine parallèle.

Il faut noter que, chaque fois, ces résultats d'accélération sont une moyenne de plusieurs exemples, car les solutions de problèmes varient selon la taille, la nature et les valeurs des contraintes et de la fonction objectif.

Conclusion et Perspectives

Dans ce mémoire, nous avons abordé le sujet de parallélisation d'algorithmes d'optimisation combinatoire, en l'occurrence, Branch and Bound, qui est l'un des algorithmes les plus utilisés dans le domaine de recherche de solutions. Nous avons pris soin de donner des notions globales du parallélisme, puis nous nous sommes concentrés sur l'étude des algorithmes parallèles, les méthodes de leur conception et les mesures de leurs performances.

Différents algorithmes de parallélisation de l'algorithme B&B ont été proposés durant les trois dernières décennies, nous avons donné un aperçu des différentes classifications de ces algorithmes, et quelques uns de ces algorithmes parallèles.

Nous avons, par la suite, présenté notre nouvelle approche de parallélisation en tenant compte des différentes contraintes, et en apportant des améliorations sur différents niveaux. La réalisation d'une implémentation et les résultats de tests confirment l'efficacité de l'approche de parallélisation proposée.

En perspectives, nous envisageons l'expérimentation sur des HPC pour aborder des problèmes plus grands et discuter les résultats avec un plus grand nombre de processeurs. Comme nous espérons avoir des jeux de tests représentant des problèmes réels pour avoir des résultats plus significatifs.

On n'oublie pas de mentionner les contraintes qui seront soulevées par l'éloignement des sites de calcul et entre les mémoires et les processeurs dans les architectures et topologies hétérogènes.

Bibliographie

- [1] George B. DANTZIG and Mukund N THAPA, "Linear Programming: 2: Theory and Extensions," *Phil. Trans. Roy. Soc.*, vol. A247, pp. 529-551, 1955.
- [2] Vipin Kumar, Ananth Grama, Anshul Gupta, and George Karypis, *Introduction to parallel computing.*: Redwood City : Benjamin/Cummings, 1994.
- [3] Ailsa H Land and Alison G Doig, "An automatic method of solving discrete programming problems," *Econometrica: Journal of the Econometric Society*, pp. 497--520, 1960.
- [4] C. Roucairol., "parallel processing for difficult combinatorial optimization problems," *European Journal of Operational Research*, vol. 92, no. 3, pp. 573-590, 1996.
- [5] Ten-Hwang Lai and Sartaj Sahni, *Anomalies in parallel branch-and-bound algorithms.*: Communications of the ACM, 1984, vol. 27, no 6, p. 594-602.
- [6] Jens Clausen, "Parallel search-based methods in optimization," in *Applied Parallel Computing Industrial Computation and Optimization.*: Springer, 1996, pp. 176-185.
- [7] Jonathan. ECKSTEIN, "Parallel branch-and-bound methods for mixed integer programming.," *Applications on Advanced Architecture Computers*, vol. 3, p. 141, 1996.
- [8] Teodor Gabriel. CRAINIC, Bertrand , Bertrand LE CUN, and Catherine ROUCAIROL, "Parallel branch-and-bound algorithms.," *Parallel Combinatorial Optimization*, pp. 1-28, 2006.
- [9] Harry WJM. TRIENEKENS and A. de. BRUIN, *Towards a taxonomy of parallel branch and bound algorithms.*: Erasmus School of Economics (ESE),

- 1992.
- [10] Nouredine. MELAB, *Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul.*: Thèse de doctorat., 2005.
- [11] Alexander Schrijver, *Combinatorial optimization: polyhedra and efficiency.*: Springer, 2003.
- [12] David. HERNANDEZ, *Stratégies d'optimisation combinatoire pour le problème de l'alignement local multiple sans indels, et application aux séquences protéiques.*: Thèse de doctorat., 2005.
- [13] Patrick SIARRY, Johann DRÉO, Alain PÉTROWSKI, and Eric Taillard, "Métaheuristiques pour l'optimisation difficile," *Eyrolles, Paris*, vol. 7, no. 2, 2003.
- [14] George B Danzig, *Linear Programming and Extensions.*: Princeton University Press, Princeton , NY, 1963.
- [15] B. Bernhard H. KORTE and Jens. VYGEN, *Combinatorial optimization.*: Springer, 2012.
- [16] Stephen J. WRIGHT, *Primal-dual interior-point methods.*: Siam, 1997.
- [17] Alexander. SCHRIJVER, *Theory of linear and integer programming.*: Wiley.com, 1998.
- [18] David A Bader, William E Hart, and Cynthia A Phillips, *Parallel algorithm design for branch and bound. In : Tutorials on Emerging Methodologies and Applications in Operations Research.*: Springer New York, 2005., vol. p. 5-1-5-44.
- [19] Bernard et CRAINIC, Teodor Gabriel. GENDRON, "Parallel Branch-and-Branch Algorithms: Survey and synthesis.," *Operations Research - INFORMS*,

- vol. 42, no. 6, pp. 1042-1066, 1994.
- [20] Donald L. MILLER and Joseph F. PEKONY, "Commentary—The Role of Performance Metrics for Parallel Mathematical Programming Algorithms.," *ORSA Journal on Computing*, vol. 5, no. 1, pp. 26-28, 1993.
- [21] Virendra K Janakiram, Edward F Gehring, Dharma P Agrawal, and Ravi Mehrotra, "A randomized parallel branch-and-bound algorithm.," *International Journal of Parallel Programming*, vol. 17, no. 3, pp. 277-301, 1988.
- [22] Vipin KUMAR and Laveen N. KANAL, "Parallel branch-and-bound formulations for AND/OR tree search. Pattern Analysis and Machine Intelligence," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, no. 6, pp. 768-778., 1984.
- [23] K. Mani. CHANDY, *Parallel program design.*: Springer US, 1989.
- [24] Behrooz. PARHAMI, *Introduction to parallel processing: algorithms and architectures.*: Springer, 1999.
- [25] Aad J Van der Steen and Jack J Dongarra, *Overview of recent supercomputers.*: NCF, 1995.
- [26] Bruce M Maggs, Lesley R Matheson, and Robert Endre Tarjan, "Models of parallel computation: A survey and synthesis," *Proceedings of the Twenty-Eighth Hawaii International Conference on System Sciences*, vol. 2, pp. 61--70, 1995.
- [27] Joseph JAJA, *An introduction to parallel algorithms.*: Addison Wesley Longman Publishing Co., Inc., 1992.
- [28] Kurt Mehlhorn and Uzi Vishkin, "Randomized and deterministic simulations of PRAMs by parallel machines with restricted granularity of parallel memories," *Acta Informatica - Springer*, vol. 21, no. 4, pp. 339--374, 1984.

- [29] Phillip B Gibbons, Yossi Matias, and Vijaya Ramachandran, "The QRQW PRAM: Accounting for contention in parallel algorithms," in *Proceedings of the fifth annual ACM-SIAM symposium on Discrete algorithms*, 1994, pp. 638--648.
- [30] Richard Cole and Ofer Zajicek, "The APRAM: Incorporating asynchrony into the PRAM model," in *Proceedings of the first annual ACM symposium on Parallel algorithms and architecture*, 1989, pp. 169--178.
- [31] Jan Leeuwen, *Handbook of Theoretical Computer Science: Algorithms and complexity. Volume A*, Access Online via Elsevier, Ed., 1990.
- [32] Alok Aggarwal and Ashok K Chandra, *Communication complexity of PRAMs*, Springer, Ed., 1988.
- [33] Charles E Leiserson and Bruce M Maggs, "Communication-efficient parallel algorithms for distributed random-access machines," *Algorithmica - Springer*, vol. 3, no. 1-4, pp. 53--77, 1988.
- [34] Alok Aggarwal, Bowen Alpern, Ashok Chandra, and Marc Snir, "A model for hierarchical memory," *Proceedings of the nineteenth annual ACM symposium on Theory of computing - ACM*, pp. 305--314, 1987.
- [35] Alok Aggarwal, Ashok K Chandra, and Marc Snir, "Hierarchical memory with block transfer," *28th Annual Symposium on Foundations of Computer Science - IEEE*, pp. 204--216, 1987.
- [36] Frank Thomson Leighton, *Introduction to parallel algorithms and architectures*, Francisco, Morgan Kaufmann San ed., 1992.
- [37] Ian FOSTER, *Designing and building parallel programs.:* Addison-Wesley Reading, 1995.
- [38] Ivan Lavalée and Christian Lavault, *Algorithmique parallèle et distribuée.*, 1985.

- [39] Jean-Pierre BARTHÉLEMY, Gérard COHEN, and Antoine. LOBSTEIN, *Complexité algorithmique et problèmes de communications.*: Masson, 1992.
- [40] ARB OpenMP, *OpenMP Application Program Interface, v. 4.0*, May, Ed., 2013.
- [41] Jalel CHERGUI and Pierre-François LAVALLÉE, *OpenMP.*, 2008.
- [42] Barbara M. CHAPMAN and Dieter. AN MEY, "The Future of OpenMP in the Multi-Core Era.," in *PARCO*, 2007, pp. 571-572.
- [43] Ramesh Menon, Leo Dagum, David Kohr, Dror Maydan, and Jeff McDonald, *Parallel Programming in OpenMP.*: Morgan Kaufmann., 2000.
- [44] Marc Snir, Steve W Otto, David W Walker, Jack Dongarra, and Steven Huss-Lederman, *MPI—the complete reference.*: The MIT Press Cambridge, Massachusetts - London, England, 1996.
- [45] William GROPP and Ewing. LUSK, "Installation guide for mpich, a portable implementation of MPI," Technical Report ANL-96/5, Argonne National Laboratory, 1996.
- [46] Weixiong ZHANG and Richard E. KORF, "Depth-first vs. best-first search: New results.," *AAAI*, pp. 769-775, 1993.
- [47] Jens. CLAUSEN, "Branch and bound algorithms-principles and examples.," *Department of Computer Science, University of Copenhagen* , pp. 1-30, 1999.
- [48] Benoît BOURBEAU , Teodor GABRIEL CRAINIC, and Bernard. GENDRON, "Branch-and-bound parallelization strategies applied to a depot location and container fleet management problem.," *Parallel Computing*, vol. 26, no. 1, pp. 27-46., 2000.
- [49] G Mitra, I Hai, and MT Hajian, "A distributed processing algorithm for solving integer programs using a cluster of workstations.," *Parallel Computing*, vol. 23,

no. 6, pp. 733-753., 1997.

- [50] Thierry GAUTIER, Jean-Louis ROCH, and Gilles. VILLARD, "Regular versus irregular problems and algorithms. In :," *Parallel Algorithms for Irregularly Structured Problems. Springer Berlin Heidelberg*, pp. 1-25, 1995.
- [51] Gene M Amdahl, "Validity of the single processor approach to achieving large scale computing capabilities," *Proceedings of the April 18-20, 1967, spring joint computer conference-ACM*, pp. 483--485, 1967.
- [52] Pascal. REBREYEND, "Algorithmes génétiques hybrides en optimisation combinatoire.," Ecole normale supérieure de lyon-ENS LYON., Thèse de doctorat. 1999.
- [53] Stephen A. COOK, "The complexity of theorem-proving procedures. ," *Proceedings of the third annual ACM symposium on Theory of computing. ACM*, pp. 151-58, 1971.
- [54] Osman Y. ÖZALTIN, Brady, HUNSAKER, and Andrew J SCHAEFER, "Predicting the solution time of branch-and-bound algorithms for mixed-integer programs.," *INFORMS Journal on Computing*, vol. 23, no. 3, pp. 392-403, 2011.
- [55] Gérard CORNUÉJOLS, Miroslav KARAMANOV, and Yanjun. LI, "Early estimates of the size of branch-and-bound trees.," *INFORMS Journal on Computing*, vol. 18, no. 1, pp. 86-96, 2006.
- [56] Donald E Knuth, "Estimating the efficiency of backtrack programs," *Mathematics of computation*, vol. 29, no. 129, pp. 122-136, 1975.