UNIVERSITÉ DU QUÉBEC

MÉMOIRE PRÉSENTÉ À
L'UNIVERSITÉ DU QUÉBEC À CHICOUTIMI
COMME EXIGENCE PARTIELLE
DE LA MAÎTRISE EN INFORMATIQUE

par
Pierre Delisle

Parallélisation d'un algorithme d'Optimisation par Colonies de Fourmis pour la résolution d'un problème d'ordonnancement industriel

6 juin 2002

Bibliothèque
Paul-Émile-Boulet

UQAC
Université du Québec
à Chicoutimi

Mise en garde/Advice

Afin de rendre accessible au plus grand nombre le résultat des travaux de recherche menés par ses étudiants gradués et dans l'esprit des règles qui régissent le dépôt et la diffusion des mémoires et thèses produits dans cette Institution, **l'Université du Québec à Chicoutimi (UQAC)** est fière de rendre accessible une version complète et gratuite de cette œuvre.

Motivated by a desire to make the results of its graduate students' research accessible to all, and in accordance with the rules governing the acceptation and diffusion of dissertations and theses in this Institution, the **Université du Québec à Chicoutimi (UQAC)** is proud to make a complete version of this work available at no cost to the reader.

L'auteur conserve néanmoins la propriété du droit d'auteur qui protège ce mémoire ou cette thèse. Ni le mémoire ou la thèse ni des extraits substantiels de ceux-ci ne peuvent être imprimés ou autrement reproduits sans son autorisation.

The author retains ownership of the copyright of this dissertation or thesis. Neither the dissertation or thesis, nor substantial extracts from it, may be printed or otherwise reproduced without the author's permission.

RÉSUMÉ

Les problèmes d'optimisation combinatoire peuvent être retrouvés, sous différentes formes, dans un grand nombre de sphères d'activité économique au sein de notre société. Ces problèmes complexes représentent encore un défi de taille pour bon nombre de chercheurs issus de domaines scientifiques variés tels les mathématiques, l'informatique et la recherche opérationnelle, pour ne citer que quelques exemples. La nécessité de résoudre ces problèmes de façon efficace et rapide a entraîné le prolifération de méthodes de résolution de toutes sortes, certaines étant plus spécifiques à un problème et d'autres étant plus génériques.

Ce mémoire réunit différentes notions du parallélisme et des métaheuristiques afin d'apporter une méthode de résolution performante à un problème d'optimisation combinatoire réel. Il démontre que l'introduction de stratégies de parallélisation à un algorithme d'Optimisation par Colonies de Fourmis permet à ce dernier d'améliorer considérablement ses facultés de recherche de solutions. Le succès de cette approche dans la résolution d'un problème d'ordonnancement industriel rencontré dans une entreprise de fabrication d'aluminium montre l'intérêt pratique de ces méthodes et leurs retombées économiques potentielles.

Ce travail de recherche, loin d'être une fin en soi, représente plutôt une première exploration des possibilités offertes par deux domaines fort prometteurs de l'informatique et de la recherche opérationnelle. L'union de méthodes d'apprentissage intelligentes et d'une puissance de calcul imposante pourrait fort bien se révéler un outil performant pour la résolution de problèmes d'une telle envergure.

REMERCIEMENTS

Je tiens tout d'abord à exprimer ma reconnaissance à monsieur Marc Gravel, mon directeur de recherche, pour son implication à la réalisation de ce travail de recherche et pour le support qu'il m'a apporté. Sa patience, sa disponibilité et la pertinence de ses conseils m'ont été d'une aide précieuse tout au long de ce travail.

Je remercie également madame Caroline Gagné pour m'avoir fait bénéficier de ses connaissances, ainsi que pour ses conseils et encouragements lors de moments plus difficiles. Mes remerciements vont aussi à monsieur Michaël Krajecki pour son accueil et sa supervision lors de mon séjour en France. Sa présence a rendu ces quatre mois enrichissants et agréables.

Mes remerciements vont aussi aux étudiants et professionnels du laboratoire du Groupe de Recherche en Informatique de l'UQAC. Leur présence durant ces deux dernières années a été fort appréciée.

Mentionnons également le support du Centre Informatique National de l'Enseignement Supérieur (CINES) et du Centre Lorrain de Calcul à Hautes Performances (Centre Charles Hermite : CCH) pour l'utilisation des ordinateurs parallèles.

Je me dois également de remercier mes parents qui, par leurs encouragements et leur soutien, m'ont permis d'aller de l'avant dans mes projets.

Finalement, mes remerciements les plus sincères vont à ma conjointe Alexandra, pour son amour et sa patience malgré mes horaires de travail imprévisibles et mes nombreuses nuits blanches.

TABLE DES MATIÈRES

RÉSUMÉ ii

REMERCIEMENTS iii

TABLE DES MATIÈRES iv

LISTE DES TABLEAUX vii

LISTE DES FIGURES ix

CHAPITRE 1 - INTRODUCTION 1

CHAPITRE 2 – LE PARALLÉLISME 6

2.1 Introduction 7

2.2 Les architectures parallèles 8
2.2.1 Structure d'un ordinateur séquentiel conventionnel 9
2.2.2 Classification des architectures parallèles 10

2.3 Les algorithmes parallèles 15
2.3.1 Définition d'un algorithme parallèle 16
2.3.2 La conception d'algorithmes parallèles 17
2.3.3 L'analyse des algorithmes parallèles 23
2.3.3.1 L'accélération 23
2.3.3.2 L'efficacité 25
2.3.3.3 L'iso-efficacité 26
2.3.4 Facteurs de performance des algorithmes parallèles 26
2.3.4.1 Le grain de parallélisme 27
2.3.4.2 Les communications 28
2.3.4.3 La synchronisation 29
2.3.4.4 Le placement des tâches 30

2.4 Les modèles de programmation 32
2.4.1 Les paradigmes nouveaux 33
2.4.2 Les extensions à des langages séquentiels existants 33

2.4.3 Les langages data-parallèles 34
2.4.4 Les approches basées sur les bibliothèques 35

2.5 Conclusion 36

CHAPITRE 3 – LES MÉTAHEURISTIQUES PARALLÈLES 37

3.1 Introduction 38

3.2 Les métaheuristiques 39
3.2.1 Les algorithmes génétiques 40
3.2.2 La recherche avec tabous 41
3.2.3 Le recuit simulé 43
3.2.4 Le GRASP (Greedy Randomized Adaptive Search Procedures) 44
3.2.5 L'Optimisation par Colonies de Fourmis (OCF) 45

3.3 Le parallélisme et les métaheuristiques 47
3.3.1 Classification des métaheuristiques parallèles 48
3.3.1.1 La classification de Crainic et Toulouse 48
3.3.1.2 La classification de Cung *et al.* 51
3.3.2 Mesures de performance des métaheuristiques parallèles 54
3.3.3 La coopération 56
3.3.3.1 La nature des informations à partager 58
3.3.3.2 Le moment où l'information est partagée 58
3.3.3.3 Les processus entre lesquels l'information est partagée 59
3.3.3.4 La co-évolution 60
3.3.4 Quelques exemples de métaheuristiques parallèles 61
3.3.4.1 Algorithmes génétiques parallèles 61
3.3.4.2 Recherche avec tabous parallèle 63
3.3.4.3 Optimisation par Colonies de Fourmis parallèle 65
3.3.4.4 Approches hybrides 67

3.4 Objectifs de la recherche 68

3.5 Conclusion 69

CHAPITRE 4 - PARALLÉLISATION D'UN ALGORITHME D'OPTIMISATION PAR COLONIES DE FOURMIS DANS UN CONTEXTE D'ORDONNANCEMENT INDUSTRIEL 70

4.1 Introduction 71

4.2 Le problème d'ordonnancement industriel 73

4.3 Description des versions séquentielles de l'OCF 75
4.3.1 Algorithme de base (OCF_{S1}) 75
4.3.2 Version améliorée de l'algorithme (OCF_{S2}) 78

4.4 Parallélisation de l'OCF 81
4.4.1 Parallélisation interne 82
4.4.1.1 Parallélisation de l'OCF sur une architecture à mémoire partagée 83
4.4.1.1.1 Le parallélisme « naturel » de l'ACO 84
4.4.1.1.2 Modèle à passage de messages vs. modèle à mémoire partagée 84
4.4.1.1.3 Synchronisation pour la mise à jour de la matrice τ_{ij} 87
4.4.1.1.4 La répartition de la charge entre les processeurs 87
4.4.1.1.5 La mise à jour de la matrice $\Delta\tau_{ij}$ en parallèle 88
4.4.1.1.6 La mise à jour du quadtree en parallèle 89
4.4.1.2 Essais numériques et résultats obtenus 90
4.4.1.3 Sommaire 95
4.4.2 Parallélisation co-évolutive 96
4.4.2.1 Parallélisation à processus multiples indépendants 98
4.4.2.2 Stratégies d'échange d'information 102
4.4.2.2.1 Échange de la meilleure solution globale (P1) 102
4.4.2.2.2 Échange circulaire des meilleures solutions locales (P2) 105
4.4.2.2.3 Échange circulaire de migrateurs (P3) 107
4.4.2.2.4 Échange circulaire des meilleures solutions locales et des migrateurs (P4) 108
4.4.2.3 Les paramètres de l'algorithme parallèle 109
4.4.2.4 Diversification de la recherche de solutions 110
4.4.2.5 Essais numériques et résultats obtenus 111
4.4.2.5.1 Parallélisation en processus multiples indépendants 112
4.4.2.5.2 La fréquence des échanges d'information 114
4.4.2.5.3 La quantité d'information échangée 116
4.4.2.5.4 Visibilité fixe et visibilité variable 117
4.4.2.5.5 La contribution des processeurs lors de la recherche 123
4.4.2.6 Sommaire 126
4.5 Conclusion 127

CHAPITRE 5 - CONCLUSION *129*

BIBLIOGRAPHIE *135*

LISTE DES TABLEAUX

Tableau 4.1 Résultats obtenus par l'algorithme OCF_P pour un carnet de 50 commandes avec 100 cycles et 1000 fourmis 92

Tableau 4.2 Résultats obtenus par l'algorithme OCF_P pour un carnet de 50 commandes avec 200 cycles et 1000 fourmis 92

Tableau 4.3 Résultats obtenus par l'algorithme OCF_P pour un carnet de 80 commandes avec 100 cycles et 1000 fourmis 92

Tableau 4.4 Résultats obtenus par l'algorithme OCF_P pour un carnet de 80 commandes avec 200 cycles et 1000 fourmis 93

Tableau 4.5 Résultats comparatifs des versions de OCF_{S2} et OCF_{P0} lorsque le "retard" est l'objectif à minimiser 113

Tableau 4.6 Tests paramétriques t d'égalité sur la moyenne des rangs entre la version OCF_{S2} et OCF_{P0} pour les huit problèmes tests 114

Tableau 4.7 Retard moyen et écart type obtenus pour 10 essais à l'aide des 4 stratégies d'échange d'information sur le problème de taille 70 lorsque la fréquence d'échange varie 115

Tableau 4.8 Tests paramétriques t d'égalité sur la moyenne des rangs pour chacune des quatre stratégies lorsque la fréquence d'échange diffère 116

Tableau 4.9 Tests paramétriques t d'égalité sur la moyenne des rangs entre la version parallèle de base (OCF_{P0}) et les quatre versions basées sur des stratégies d'échange d'information sur le problème de taille 70 lorsque les paramètres de visibilité sont identiques pour tous les processeurs 117

Tableau 4.10 Tests paramétriques t d'égalité sur la moyenne des rangs pour deux stratégies de visibilité pour chacune des quatre versions utilisant des stratégies d'échange d'information sur le problème de taille 70 119

Tableau 4.11 Retard moyen et écart type obtenus à l'aide des 4 stratégies d'échange d'information sur les problème de taille 10 à 80 120

Tableau 4.12 Tests paramétriques t d'égalité sur la moyenne des rangs entre la version parallèle de base (OCF_{P0}) et les quatre versions basées sur des stratégies d'échange d'information sur les 8 problèmes tests _____ 123

LISTE DES FIGURES

Figure 1.1 Architecture d'un ordinateur séquentiel conventionnel 9

Figure 1.2 Structure SIMD 12

Figure 1.3 Structure MIMD 13

Figure 1.4 Schéma de l'organisation d'une machine à mémoire partagée 14

Figure 1.5 Schéma de l'organisation d'une machine à mémoires distribuées 14

Figure 1.6 Algorithme de calcul de la somme sur le modèle PRAM 19

Figure 1.7 Schéma du calcul de la somme sur un modèle PRAM quand n = 8. Chaque nœud interne représente une opération d'addition. 20

Figure 1.8 Un hypercube de dimension d = 3 21

Figure 1.9 Algorithme de calcul de la somme sur un hypercube 22

Figure 3.1 Un algorithme génétique 41

Figure 3.2 Un algorithme de recherche avec tabous 43

Figure 3.3 Un algorithme de recuit simulé 44

Figure 4.1 Le processus de coulée horizontale 73

Figure 4.2 Version séquentielle de l'Optimisation par Colonies de Fourmis (OCF_{S1}). 78

Figure 4.3 Version séquentielle améliorée de l'Optimisation par Colonies de Fourmis (OCF_{S2}). 80

Figure 4.4 OCF parallèle dans un modèle à passage de messages 85

Figure 4.5 Version parallèle de l'Optimisation par Colonies de Fourmis (OCF_P) dans un modèle à mémoire partagée 90

Figure 4.6 Temps d'exécution avec des carnets de 50 et 80 commandes (100 cycles et 1,000 fourmis) 93

Figure 4.7 Efficacité avec 4 processeurs, 100 cycles, 1000 fourmis et des carnets de commandes variant entre 10 et 80 94

Figure 4.8 Parallélisation de l'OCF_{S2} en processus multiples indépendants (OCF_{P0}) 99

Figure 4.9 Utilisation du quadtree en mémoire partagée 100

Figure 4.10 Diffusion de la meilleure solution globale dans un modèle à mémoire partagée 104

Figure 4.11 OCF_{S2} parallèle avec stratégie d'échange de la meilleure solution globale (OCF_{P1}) 104

Figure 4.12 Échange circulaire des meilleures solutions locales dans un modèle à mémoire partagée 106

Figure 4.13 OCF_{S2} parallèle avec stratégie d'échange circulaire des meilleures solutions locales (OCF_{P2}) 106

Figure 4.14 OCF_{S2} parallèle avec stratégie d'échange circulaire des meilleures solutions locales (OCF_{P3}) 108

Figure 4.15 OCF_{S2} parallèle avec stratégie d'échange circulaire des meilleures solutions locales et de migrateurs (OCF_{P4}) 109

Figure 4.16 Contribution des processeurs lors de l'exécution de l'algorithme 125

CHAPITRE 1

INTRODUCTION

Depuis quelques années, les ordinateurs parallèles sont devenus de plus en plus abordables et certains efforts ont été faits dans le but de les rendre encore plus accessibles et conviviaux. Associées généralement à la haute technologie, à la science et la défense, ces puissantes machines de calcul pourraient fort bien, dans un futur rapproché, devenir une partie intégrante des systèmes informatiques de plusieurs organisations. Ces dernières pourraient alors bénéficier des opportunités offertes par le parallélisme pour la résolution de problèmes de nature et de taille variées.

Les problèmes d'optimisation combinatoire rencontrés dans une multitude de domaines d'activité, autant académiques qu'industriels, représentent encore aujourd'hui un défi de taille pour bon nombre de chercheurs et de praticiens. La plupart de ces problèmes, dits NP-Difficiles, ne peuvent être résolus de façon optimale par des algorithmes exacts et ce, peu importe la technologie utilisée. La nécessité de trouver rapidement des solutions acceptables à plusieurs de ces problèmes a entraîné le développement d'algorithmes d'approximation dont font partie les métaheuristiques.

L'efficacité des métaheuristiques à traiter ces problèmes a été démontrée dans plusieurs travaux de recherche fondamentale et appliquée. Le succès de ces méthodes s'explique par un ensemble de facteurs, notamment par leur potentiel d'adaptation aux différentes contraintes associées à des problèmes spécifiques, par leur facilité d'implantation dans des programmes d'application divers et par la qualité des solutions qu'elles peuvent produire dans un temps relativement court. Le domaine des métaheuristiques est toutefois encore jeune et de nombreux efforts sont présentement mis en oeuvre dans le but d'améliorer la performance de ces méthodes de résolution.

Quelques travaux récents (Crainic et Toulouse (1998), Cung *et al.* (2001), Eksioglu *et al.* (2001)) ont identifié et abordé le calcul parallèle comme étant une avenue très prometteuse pour l'amélioration de la performance des métaheuristiques. Cependant, les comportements et les effets des mécanismes parallèles dans ce cadre sont encore peu étudiés.

Les objectifs généraux de ce travail de recherche sont de deux ordres. Le premier est d'étudier les relations entre le calcul parallèle et les métaheuristiques afin de mieux comprendre les bénéfices que peuvent apporter les métaheuristiques parallèles dans la résolution de problèmes d'optimisation combinatoire. Le deuxième est d'identifier et de mettre en oeuvre des solutions parallèles performantes à un problème d'optimisation combinatoire pratique, c'est-à-dire un problème d'ordonnancement industriel rencontré dans une entreprise de production d'aluminium. L'atteinte de ces objectifs passe alors par une connaissance adéquate du parallélisme, des métaheuristiques et du problème d'ordonnancement industriel.

Dans le chapitre 2, une revue de la littérature sur le parallélisme sera présentée en détaillant les concepts du domaine les plus pertinents pour la réalisation de ce travail. Divers modèles d'architectures, d'algorithmes et de programmation seront discutés, de même que certaines relations unissant ces modèles entre eux.

Le chapitre 3 portera sur les métaheuristiques séquentielles et parallèles. Dans un premier temps, une revue sommaire des principales métaheuristiques sera réalisée en mettant l'accent sur la métaheuristique utilisée dans cette étude. Par la suite, le parallélisme sera introduit dans le monde des métaheuristiques et une description des principales

stratégies de parallélisation de métaheuristiques présentes dans la littérature sera effectuée. Finalement, quelques travaux pertinents à ce sujet seront soulignés.

Le chapitre 4 présentera les travaux réalisés dans cette étude pour la résolution d'un problème d'ordonnancement industriel par la parallélisation d'une métaheuristique. Dans un premier temps, le problème d'ordonnancement industriel sera énoncé et la méthode de résolution métaheuristique implantée dans l'entreprise sera décrite en soulignant ses principaux éléments pertinents. Par la suite, nous présenterons la démarche et les résultats de la première partie de l'étude portant sur les possibilités offertes par le parallélisme en ce qui concerne l'accélération des calculs au cours du processus de recherche d'une métaheuristique. Nous montrerons comment les coûts associés aux communications et aux synchronisations peuvent être réduits sur une architecture à mémoire partagée, de même que les limites relatives au nombre de processeurs. Finalement, l'intégration des concepts présentés aux chapitres deux et trois sera effectuée en adaptant certaines stratégies de parallélisation à la métaheuristique séquentielle et en les appliquant à la résolution du problème d'ordonnancement. Dans cette deuxième partie, nous montrerons alors comment la coopération d'agents engagés dans la résolution d'un problème d'ordonnancement industriel peut permettre de résoudre ce dernier plus efficacement qu'un agent seul ou qu'un groupe d'agents opérant isolément les uns des autres. Afin de vérifier la validité et la pertinence de l'approche, les résultats obtenus par la parallélisation seront comparés avec ceux de l'algorithme séquentiel.

La parallélisation étant étudiée dans le cadre d'une architecture réelle à mémoire partagée et d'un environnement de programmation novateur, l'approche n'a pas été exploitée dans la littérature à notre connaissance et présente des éléments nouveaux.

Il faut également souligner que ce travail fait suite à une série de travaux effectués dans le but d'établir des méthodes de résolution performantes à des problèmes d'ordonnancement industriel (Gagné (2001), Gagné *et al.* (2001a), Gagné *et al.* (2001b), Gravel *et al.* (2000), Gravel *et al.* (2002)). Il permettra d'évaluer l'intérêt et le potentiel des métaheuristiques parallèles dans ce contexte. La conclusion de ce mémoire portera sur l'appréciation générale des résultats obtenus par la parallélisation et sur l'identification de futures avenues de recherches pertinentes.

CHAPITRE 2

LE PARALLÉLISME

2.1 Introduction

Le concept de parallélisme est apparu à peu près en même temps que la science informatique elle-même. Cependant, ce n'est qu'à partir des années quatre-vingt, avec l'apparition des premières machines parallèles à usage général, que le domaine du calcul parallèle devint accessible à des disciplines autres que la science, l'ingénierie et la défense. Aujourd'hui, avec la disponibilité d'ordinateurs parallèles à haute performance à des coûts relativement bas, il est possible de s'attaquer à des problèmes qui demandent une charge de calcul trop importante pour les ordinateurs séquentiels conventionnels. Ces problèmes proviennent de domaines aussi variés que la science, l'ingénierie, l'économie, l'environnement et la société, pour ne citer que quelques exemples.

Dans l'histoire de l'informatique, l'augmentation de la puissance de calcul s'est effectuée principalement en améliorant les circuits électroniques et le matériel des ordinateurs séquentiels. Même si les limites physiques de cette approche ne semblent pas encore avoir été atteintes, on peut déjà constater que les améliorations futures sur ce plan nécessiteront des efforts scientifiques et financiers de plus en plus importants (Plateau et Trystram (2000)). Une autre façon d'augmenter la puissance de calcul est offerte par le traitement parallèle et distribué.

Le but premier du parallélisme est d'utiliser plusieurs processeurs de façon concurrente pour effectuer des calculs plus rapidement qu'avec un seul processeur. Cependant, la poursuite de ce but de façon efficace est un exercice complexe qui nécessite la considération d'un ensemble d'éléments de nature conceptuelle et technique. Les modèles

d'architecture permettent de simplifier les mécanismes complexes des systèmes parallèles afin de dégager les paramètres principaux de leur comportement global. Des modèles algorithmiques permettent de résoudre différents problèmes par l'entremise du calcul parallèle et d'analyser la performance des algorithmes parallèles en relation avec différentes architectures. Des modèles de programmation sont nécessaires à l'implémentation de ces algorithmes. Il n'y pas de limites bien définies entre tous ces modèles et il n'y a pas de modèles universels qui font l'unanimité actuellement. Une connaissance générale des différents paradigmes du parallélisme est donc nécessaire avant d'aborder la conception et l'implémentation d'algorithmes parallèles.

Le parallélisme est un vaste domaine dont un état de l'art relativement complet peut être trouvé dans les ouvrages suivants : Codenotti et Leoncini (1993), Cosnard et Trystram (1995), Gentler *et al.* (1996), Leopold (2001). Ce chapitre présente les principaux concepts nécessaires à la compréhension des chapitres suivants et de certains enjeux impliqués par la résolution parallèle d'un problème donné. Une approche en trois niveaux d'abstraction est utilisée. Tout d'abord, une classification des architectures parallèles sera présentée. Ensuite, des notions algorithmiques seront abordées et, finalement, ce chapitre se terminera au niveau des modèles de programmation, celui-ci étant plus pratique et plus près de l'implémentation.

2.2 Les architectures parallèles

Augmenter le nombre de processeurs dans un ordinateur altère considérablement sa structure de base. Les problèmes d'accès à la mémoire deviennent cruciaux en raison de la

nécessité de fournir un flot de données suffisant à l'ensemble des processeurs. De plus, les problèmes de communication entre les processeurs deviennent importants et croissent avec l'augmentation du nombre de processeurs. Plusieurs solutions à ces problèmes ont été proposées et différentes architectures ont été réalisées (Cosnard et Trystram (1995)). Dans cette section, une méthode de classification des architectures parallèles, proposée par Flynn (1966), est présentée. Auparavant, la structure d'un ordinateur séquentiel classique est brièvement expliquée.

2.2.1 Structure d'un ordinateur séquentiel conventionnel

Un ordinateur séquentiel classique (machine de von Neumann), dont la structure est illustrée à la Figure 1.1, comporte 4 éléments principaux : l'unité centrale de traitement ou processeur (CPU), l'unité de contrôle, la mémoire et les périphériques d'entrée-sortie (I/O) (Codenotti et Leoncini (1993), Cosnard et Trystram (1995)).

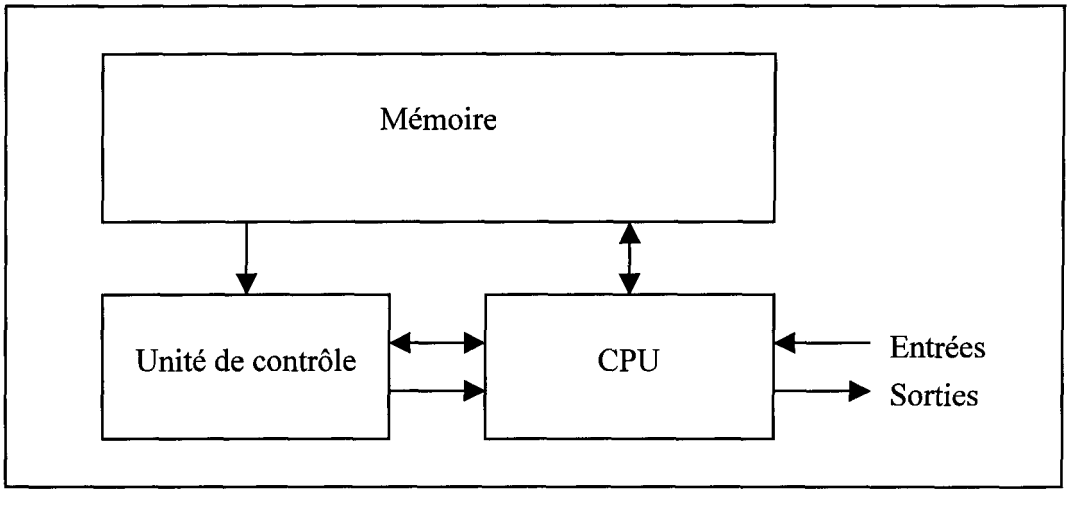

Figure 1.1 Architecture d'un ordinateur séquentiel conventionnel

L'unité de contrôle, le processeur et la mémoire représentent le cœur de l'ordinateur et déterminent son comportement. La mémoire contient un programme qui spécifie la séquence d'instructions à être exécutée et les données qui seront utilisées. À chaque étape du programme, l'unité de contrôle obtient de la mémoire une instruction qui opère sur un ensemble de données et l'envoie au processeur. Par exemple, cette instruction pourrait dicter au processeur d'effectuer une opération arithmétique ou logique sur deux nombres et de retourner le résultat à la mémoire. Le processeur dispose d'un certain nombre de registres pour effectuer ses calculs et est connecté à des unités d'entrée et de sortie pour communiquer avec l'extérieur.

Ce modèle relativement simple représente, à peu de choses près, la plupart des ordinateurs séquentiels d'aujourd'hui. Le domaine de l'algorithmique séquentielle est basé sur celui-ci. La prochaine section montrera qu'il en va tout autrement dans le domaine du parallélisme, la diversité des architectures rendant la conception d'algorithmes plus complexe.

2.2.2 Classification des architectures parallèles

Un modèle d'architecture parallèle est une description abstraite d'un ordinateur parallèle dont le but est de représenter les particularités les plus importantes de la machine (Leopold (2001)). Avec de tels modèles ignorant les détails d'implémentation superflus, il est possible de concentrer les efforts sur les aspects les plus importants dans la recherche de solutions parallèles à différents problèmes.

Plusieurs systèmes de classification des ordinateurs parallèles sont proposés dans la littérature. Celui de Flynn (1966), basé sur l'organisation des flots d'instructions (séquences d'instructions transmises à partir d'une unité de contrôle vers un ou plusieurs processeurs) et des flots de données (séquences de données provenant de la mémoire et se dirigeant vers un processeur ou provenant d'un processeur et se dirigeant vers la mémoire), est le plus populaire.

Une première classe d'architecture, selon le modèle de Flynn, est la classe SISD (***Single Instruction stream, Single Data stream***), qui est celle des ordinateurs séquentiels classiques dont la structure a été définie à la section précédente. Ils utilisent un unique flot d'instructions et opèrent sur un unique flot de données.

Dans une machine SIMD (***Single Instruction stream, Multiple Data stream***), plusieurs processeurs sont supervisés par la même unité de contrôle tel qu'illustré à la Figure 1.2. Les p processeurs (notés P_i) reçoivent la même instruction de l'unité de contrôle, mais opèrent sur des données distinctes qui proviennent de différents flots de données. Ils opèrent de façon synchrone, chaque processeur exécutant la même instruction à chaque unité de temps.

La classe de machines MISD (***Multiple Instruction stream, Single Data stream***) applique, pour sa part, plusieurs flots d'instructions à un unique flot de données. Certains auteurs considèrent que cette classe ne correspond pas à un mode de fonctionnement réaliste (Codenotti et Leoncini (1993)). D'autres voient plutôt dans cette classe le mode de fonctionnement en pipeline (Cosnard et Trystram (1995)), c'est-à-dire le travail à la chaîne, dans la mesure où il existe une unique entrée des données et plusieurs unités de calcul.

Toutefois, le pipeline peut également être vu comme un mode MIMD dans la mesure où il y a plusieurs unités de calcul et plusieurs flots de données, chaque étage du pipeline recevant des données différentes (Gentler *et al.* (1996)).

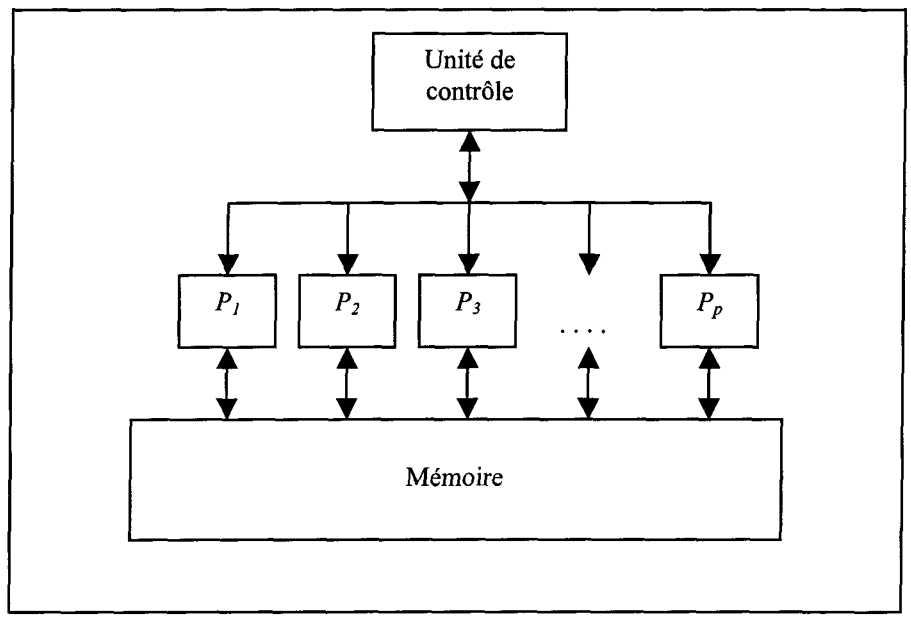

Figure 1.2 Structure SIMD

Dans une machine fonctionnant en mode MIMD (***Multiple Instruction stream, Multiple Data stream***), les processeurs sont indépendants les uns des autres et fonctionnent de façon asynchrone, chacun possédant sa propre unité de contrôle (notée UC_i). Un schéma d'architecture MIMD est illustré à la Figure 1.3. Comme en SIMD, chacun utilise ses propres données mais y applique, en plus, ses propres flots d'instructions.

On peut également classifier les ordinateurs parallèles en fonction de l'organisation de leur mémoire. Cette forme de classification est en quelque sorte complémentaire à celle de Flynn (Gentler *et al.* (1996)).

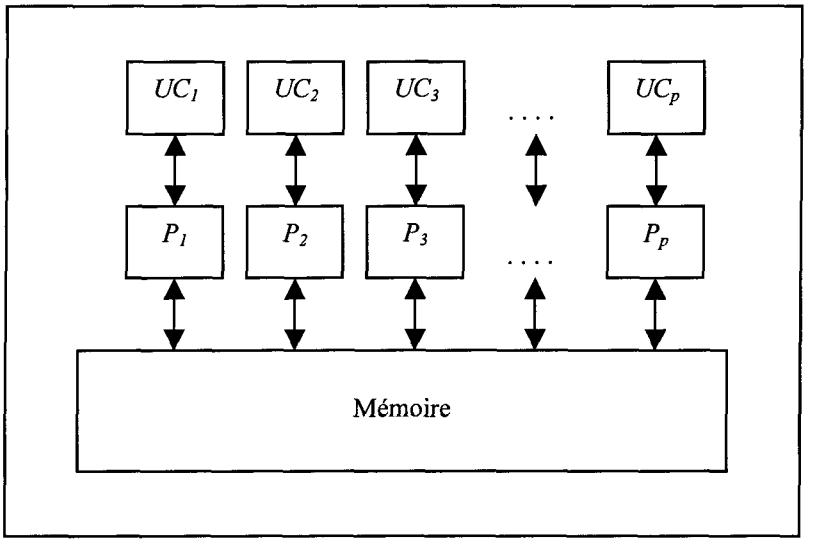

Figure 1.3 Structure MIMD

Les machines parallèles à mémoire partagée (*Shared Memory (SM)*), dont la structure est illustrée à la Figure 1.4, possèdent une mémoire commune accessible par tous les processeurs. La communication entre processeurs se fait par des lectures et écritures successives dans la mémoire de sorte qu'une écriture faite par un processeur peut ultérieurement être lue par un autre. Le lien entre les processeurs et la mémoire se fait par un réseau d'interconnexion ou un bus. Cette organisation entraîne toutefois un problème de gestion des conflits d'accès à la mémoire. Celui-ci peut être minimisé, entre autres, par la division de la mémoire en bancs accessibles par un seul processeur à un instant donné.

Dans la classe des machines parallèles à mémoires distribuées (*Distributed Memory (DM)*), dont un schéma est donné à la Figure 1.5, chaque processeur possède sa propre mémoire locale et en a l'accès exclusif. La communication entre processeurs ne peut donc plus se faire par le biais de la mémoire. Elle se fait plutôt par l'envoi de messages entre

processeurs à travers un réseau d'interconnexion qui relie les processeurs entre eux plutôt que de les relier à la mémoire comme dans le cas des machines SM.

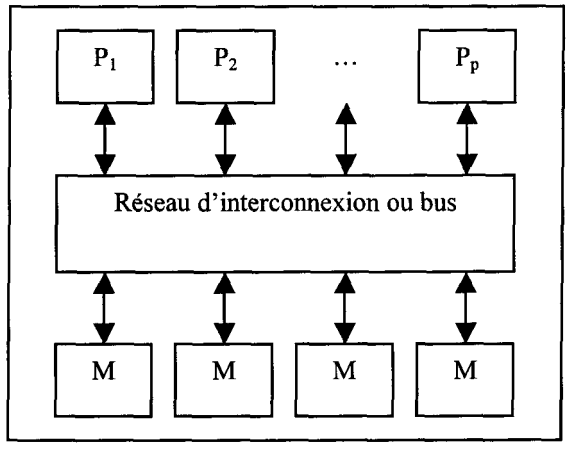

Figure 1.4 Schéma de l'organisation d'une machine à mémoire partagée

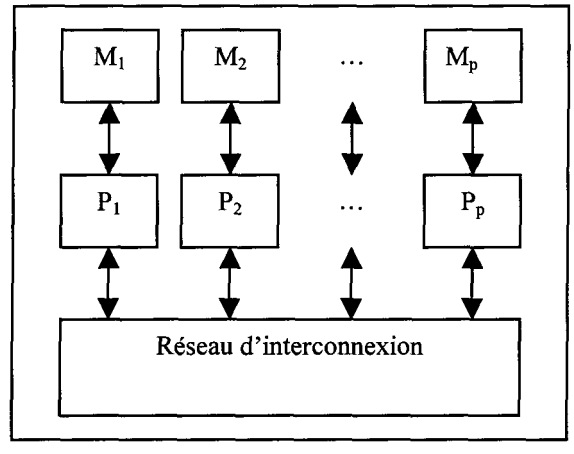

Figure 1.5 Schéma de l'organisation d'une machine à mémoires distribuées

Une classification largement acceptée (Germain-Renaud et Sansonnet (1992)) combine, d'une part, les classes SIMD et MIMD de Flynn et, d'autre part, les classes SM et DM. Cette classification considère le contrôle des séquences d'instructions à exécuter comme étant centralisé ou distribué. S'il est centralisé, il est unique (SI), et s'il est distribué, il est multiple (MI). Les données sont toujours multiples (MD) et peuvent se trouver dans une mémoire partagée (SM) ou dans des mémoires distribuées (DM). Pour nommer ces classes, on utilise les sigles SIMD-SM, SIMD-DM, MIMD-SM et MIMD-DM. Le modèle théorique de machine parallèle le plus largement reconnu est le modèle PRAM (***Parallel Random Access Machine***) qui appartient à la classe SIMD-SM (Gentler *et al.* (1996)). Il sera présenté de façon plus détaillée dans la section sur les algorithmes parallèles.

La plupart des architectures, qu'elles soient à mémoires partagées ou à mémoires distribuées, sont basées sur un réseau d'interconnexion de topologie statique et régulière qui relie les processeurs entre eux ou à des bancs mémoire. Les topologies les plus populaires sont les grilles, les tores, les hypercubes, les anneaux et les arbres (Cosnard et Trystram (1995)).

Cette section sur les architectures parallèles a montré qu'une des différences les plus frappantes entre les approches séquentielle et parallèle est la grande diversité des modèles parallèles disponibles. Différentes approches algorithmiques peuvent être utilisées afin de résoudre un même problème sur des modèles d'architecture différents et certaines approches seront plus efficaces que d'autres selon le contexte. Dans la prochaine section, traitant des algorithmes parallèles, un certain nombre de ces approches seront présentées.

2.3 Les algorithmes parallèles

La conception et l'analyse d'algorithmes parallèles est une pierre angulaire du domaine du parallélisme. Des algorithmes parallèles rapides sont nécessaires si l'on veut atteindre une réduction significative des temps de calcul dans la résolution de problèmes complexes sur des ordinateurs parallèles.

Dans cette section, la notion d'algorithme parallèle sera définie. Ensuite, les principaux thèmes relatifs à l'algorithmique parallèle, tels que la conception, l'analyse et les facteurs de performance des algorithmes parallèles, seront abordés.

2.3.1 Définition d'un algorithme parallèle

La définition de la notion d'algorithme diffère sensiblement d'une référence à une autre. Cormen *et al.* (1994) définissent un algorithme comme étant une procédure de calcul bien définie qui prend en entrée une valeur ou un ensemble de valeurs et qui produit en sortie une valeur ou un ensemble de valeurs. Selon ces auteurs, un algorithme est donc une séquence d'étapes de calcul permettant de passer de la valeur d'entrée à la valeur de sortie. On dit qu'on algorithme est correct s'il se termine avec une sortie correcte pour chaque instance d'entrée. L'algorithme résout alors le problème posé.

Cosnard et Trystram (1995) définissent un algorithme séquentiel comme un ensemble d'opérations définies d'une façon rigoureuse et non ambiguë, de sorte que chacune des opérations puisse être effectuée par un ordinateur RAM. Dans le modèle RAM, les instructions d'un algorithme sont exécutées l'une après l'autre, sans opérations simultanées. Dans le cas des algorithmes parallèles, plusieurs opérations sont effectuées simultanément sur plusieurs processeurs. On peut définir un algorithme parallèle comme *tout algorithme dans lequel les séquences de calcul peuvent être effectuées simultanément* (Academic Press dictionary of science and technology (1992)) ou comme *un algorithme dans lequel plusieurs opérations sont effectuées simultanément* (McGraw-Hill dictionary of scientific and technical terms (1994)).

Akl (2000) définit un algorithme parallèle comme étant une méthode pour résoudre un problème de calcul sur un modèle parallèle de calcul. Cette définition semble la plus appropriée pour ce travail et sera celle retenue. En effet, dans la prochaine section, nous

verrons de quelle façon il est possible de résoudre un problème donné sur différents modèles parallèles.

2.3.2 La conception d'algorithmes parallèles

Un modèle généralement accepté pour l'étude des algorithmes séquentiels consiste en une unité centrale de traitement avec une mémoire à accès aléatoire qui y est rattachée (modèle RAM) (Cosnard et Trystram (1995)). Le succès de ce modèle est dû à sa simplicité et à sa facilité à reproduire la performance des algorithmes séquentiels sur les machines séquentielles réelles. En calcul parallèle, la performance des algorithmes est reliée à un ensemble d'éléments dépendants de la machine utilisée tels la concurrence des calculs, le découpage et l'allocation des tâches aux divers processeurs, la communication et la synchronisation. Ceci explique pourquoi il est plus difficile de définir un modèle algorithmique largement accepté (JàJà (1992)). Le modèle PRAM (***P**arallel **R**andom **A**ccess **M**achine*) est le modèle le plus populaire. Même si aucune machine réelle ne correspond à ce modèle, il permet au concepteur d'algorithmes de se concentrer sur les propriétés structurelles du problème en faisant abstraction des détails moins importants.

Une machine PRAM, qui est de type SIMD, est constituée de n processeurs P_1, P_2, ..., P_n, où $n \geq 2$, qui partagent une mémoire de m emplacements U_1, U_2, ..., U_m, où $m \geq 1$. Chaque processeur possède une petite mémoire locale (un nombre déterminé de registres) et des circuits qui permettent l'exécution d'opérations arithmétiques et logiques sur les données contenues dans les registres. Les processeurs ont accès aux emplacements mémoire et opèrent façon synchrone, c'est-à-dire en effectuant la même séquence

d'instructions sur des données différentes. Chaque étape de calcul d'une machine PRAM consiste en trois phases. La première phase en est une de lecture où chaque processeur copie une donnée de la mémoire dans un de ses registres. Ensuite vient la phase de calcul où chaque processeur effectue une opération arithmétique ou logique sur les données contenues dans les registres. Finalement, dans la phase d'écriture, chaque processeur copie une donnée provenant d'un de ses registres dans un emplacement de la mémoire partagée.

Dans la phase de lecture, les processeurs peuvent avoir accès au même emplacement mémoire sans qu'il n'y ait de conflit. Dans la phase d'écriture, il faut spécifier, selon le problème, le comportement de l'algorithme quand deux ou plusieurs processeurs tentent d'écrire dans la même case mémoire. Il serait, par exemple, possible de spécifier que la somme des valeurs écrites par les processeurs soit enregistrée, ou encore le minimum de ces valeurs, etc. Notons à ce sujet qu'il existe différentes variantes du modèle PRAM dans lesquelles les possibilités de gestion des accès concurrents à la mémoire sont déterminées. Par exemple, la machine EREW (***Exclusive Read Exclusive Write***) PRAM ne permet pas l'accès concurrent à un emplacement mémoire précis, que ce soit en lecture ou en écriture, la machine CREW (***Concurrent Read Exclusive Write***) PRAM ne permet que la lecture concurrente tandis que la machine CRCW (***Concurrent Read Concurrent Write***) PRAM permet les deux.

Le modèle PRAM suppose que l'accès à la mémoire se fait en temps constant quel que soit le nombre de processeurs. Même si cette supposition n'est pas réaliste, il demeure que ce modèle est le plus adapté pour exprimer un algorithme parallèle et pour évaluer ses performances sur un plan théorique. Les machines à mémoire virtuellement partagée

(SMP, CC-NUMA) sont celles qui se rapprochent le plus de ce modèle théorique (Leopold (2001)).

Une étude approfondie des algorithmes parallèles sur le modèle PRAM peut être trouvée dans le travail de JàJà (1992). Dans la suite de cette section, nous verrons brièvement comment un problème simple, la somme de n nombres, peut être traité différemment selon le modèle parallèle choisi.

Le problème du calcul de la somme de n nombres peut être énoncé de la façon suivante : soit un tableau A de $n = 2^k$ nombres (pour une constante k quelconque) dont il faut calculer la somme $S = A(1) + A(2) + \ldots + A(n)$. Deux méthodes seront présentées pour montrer de quelle façon on peut le résoudre de façon parallèle sur deux modèles d'architectures, c'est-à-dire sur une machine PRAM et sur un hypercube.

La Figure 1.6 présente un algorithme qui résout ce problème sur une machine PRAM avec n processeurs $\{P_1, P_2, \ldots, P_n\}$. Le programme prend en entrée le tableau A qui est en mémoire partagée.

```
POUR i = 1 à n FAIRE EN PARALLÈLE
B(i) = A(i);
POUR h = 1 à log n FAIRE
    POUR i = 1 à n/2^h FAIRE EN PARALLÈLE
        B(i) = B(2i-1) + B(2i);
S = B(1);
```

Figure 1.6 Algorithme de calcul de la somme sur le modèle PRAM

La Figure 1.7 illustre de façon arborescente le comportement de l'algorithme quand n est égal à 8. Durant la première boucle « POUR », une copie B de A est effectuée et

stockée en mémoire partagée. Dans la première itération de la deuxième boucle « POUR », un traitement est effectué en parallèle, c'est-à-dire que les processeurs numérotés *i,* pour *i* allant de 1 à 4, vont effectuer la somme des éléments $(2i - 1)$ et $(2i)$ du tableau et placer le résultat à l'emplacement *i*. Durant la deuxième itération de la boucle « POUR », les deux premiers processeurs travailleront avec les sous-totaux trouvés à l'itération précédente et, à la dernière itération, la somme de tous les éléments se trouvera à la case 1.

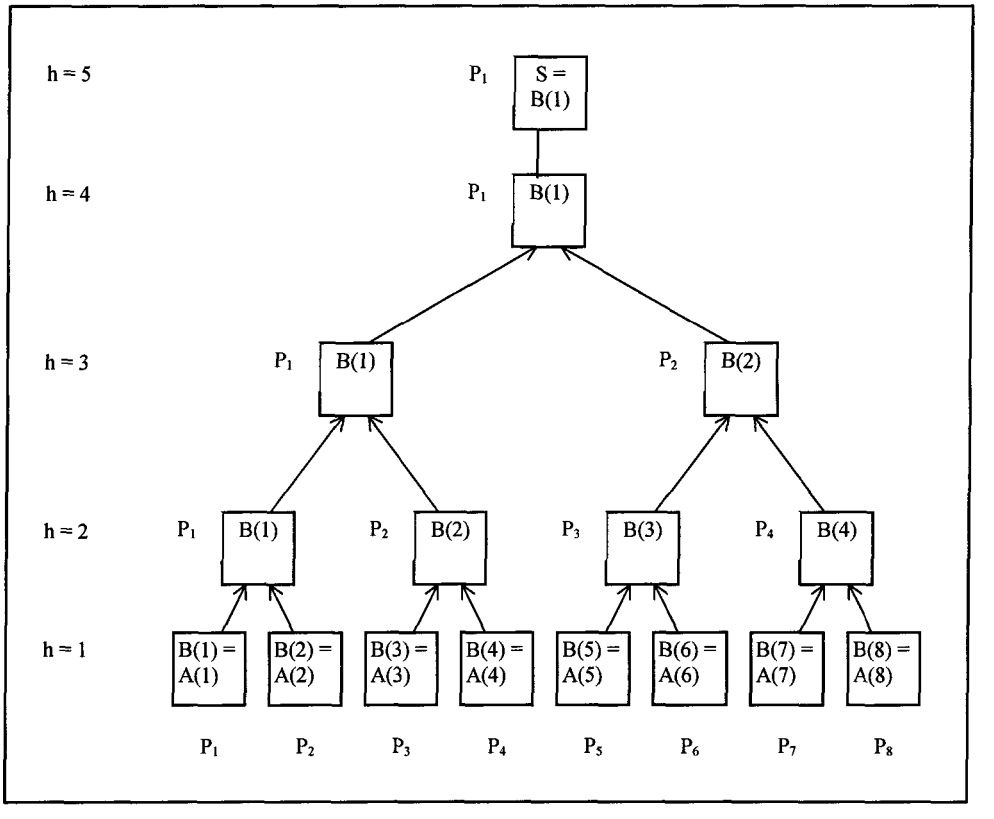

Figure 1.7 Schéma du calcul de la somme sur un modèle PRAM quand n = 8. Chaque nœud interne représente une opération d'addition.

Cet exemple très simple avait pour but de montrer de quelle façon il est possible d'exprimer le parallélisme d'un algorithme sur le modèle PRAM. Dans le cas où un

modèle d'architecture différent serait utilisé, ce même problème nécessiterait un autre algorithme parallèle.

Dans un modèle d'hypercube de dimension d, modèle dans lequel la mémoire est distribuée, il y a $n = 2^d$ processeurs qui sont numérotés de 0 à $(n - 1)$. Dans ce modèle, deux processeurs sont reliés entre eux si et seulement si leurs indices représentés en notation binaire ne diffèrent que d'un seul bit. La Figure 1.8 illustre un hypercube de dimension 3. Ce modèle est populaire, entre autres, parce qu'il est régulier, facilement extensible et qu'il peut simuler la plupart des autres topologies (JàJà (1992)).

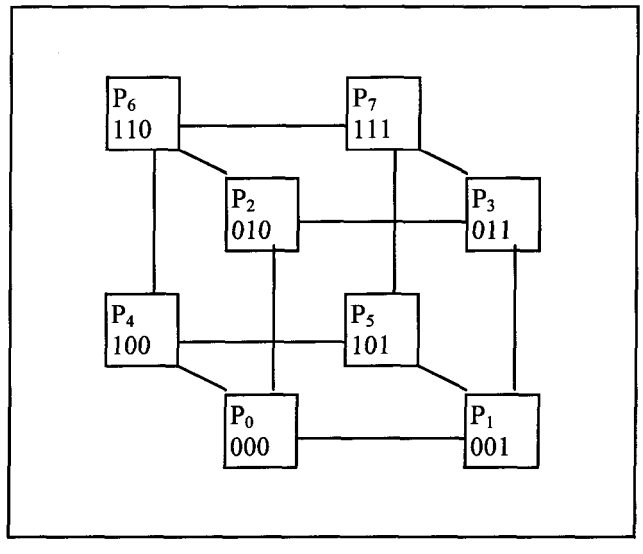

Figure 1.8 Un hypercube de dimension d = 3

Dans ce modèle, le tableau initial A de taille $n = 2^d$ est réparti entre les processeurs de sorte que le processeur P_i possède l'élément $A(i)$. Il faut noter, dans ce cas-ci, que l'indice i va de 0 à $(n - 1)$ et non de 1 à n comme dans l'exemple précédent. L'algorithme illustré à la Figure 1.9 effectue la somme des n éléments du Tableau A sur un hypercube. La

première itération calcule les sommes des paires d'éléments entre les processeurs dont les indices sont différents sur le bit le plus significatif. Ces sommes sont conservées dans le sous-hypercube de dimension $(d - 1)$ composé des processeurs dont le bit le plus significatif est égal à 0. Les itérations restantes s'effectuent de façon similaire.

```
POUR j = (d – 1) à 0 FAIRE
    POUR tous les i < 2^j FAIRE EN PARALLÈLE
        A(i) ← A(i) + A(i^(j))
```

Figure 1.9 Algorithme de calcul de la somme sur un hypercube

L'algorithme opère de façon synchrone et $i^{(j)}$ représente l'indice i pour lequel le bit j a été complémenté. L'instruction $A(i) \leftarrow A(i) + A(i^{(j)})$ implique deux sous-étapes. Dans la première sous-étape, P_i copie $A(i^{(j)})$ à partir du processeur $P_i(j)$ en passant par le lien de communication qui relie P_i et $P_i(j)$. Dans la deuxième sous-étape, P_i effectue l'addition $A(i) + A(i^{(j)})$ et conserve le résultat dans $A(i)$.

Si l'on considère l'exemple où $n = 8$ (donc où $d = 3$), à la première itération de la boucle parallèle, les sommes $A(0) = A(0) + A(4)$, $A(1) = A(1) + A(5)$, $A(2) = A(2) + A(6)$ et $A(3) = A(3) + A(7)$ sont calculées et conservées dans la mémoire des processeurs P_0, P_1, P_2 et P_3 respectivement. À la fin de la deuxième itération, on obtient $A(0) = A(0) + A(3)$ et $A(1) = A(1) + A(4)$. Finalement, à la dernière itération, la somme $A(0) = A(0) + A(1)$ est calculée et le processeur P_0 contient la somme S des n éléments. L'algorithme se termine donc après $d = \log n$ étapes parallèles.

Ces deux exemples avaient pour but de montrer comment la conception d'un algorithme parallèle peut-être influencée par le modèle d'architecture sous-jacent. Dans le modèle d'hypercube, la disposition des processeurs était prise en considération, ce qui n'était pas le cas dans le modèle PRAM. De plus, le fait que la mémoire soit distribuée dans le modèle d'hypercube a introduit la notion de communication entre les processeurs. Dans les cas d'algorithmes plus complexes, le coût des communications sur un modèle d'architecture donné est un élément important à considérer. La notion de communication entre les processeurs sera présentée plus loin.

À l'instar des algorithmes séquentiels, les algorithmes parallèles ne sont pas tous aussi efficaces les uns que les autres. De plus, la qualité d'un algorithme parallèle est influencée par différentes caractéristiques des systèmes parallèles, ce qui rend l'analyse plus complexe. L'analyse de la performance des algorithmes parallèles fera l'objet de la prochaine section.

2.3.3 L'analyse des algorithmes parallèles

Il existe plusieurs critères permettant d'évaluer la performance d'un algorithme parallèle. Cette section en présente quelques uns parmi les plus utilisés, comme l'accélération, l'efficacité, le nombre de processeurs et l'iso-efficacité.

2.3.3.1 L'accélération

L'efficacité des algorithmes séquentiels est généralement mesurée par leur temps d'exécution en fonction de la taille de leur entrée (Cormen *et al.* (1994)). Considérons P

comme étant un problème de calcul et n la taille de son entrée. On note $T^*(n)$ comme le temps d'exécution du meilleur algorithme séquentiel connu qui résout ce problème. La raison principale justifiant la conception d'algorithmes parallèles est en fait la réduction du temps d'exécution. Soit A, un algorithme parallèle qui résoud P dans un temps $T_p(n)$ sur un ordinateur parallèle avec p processeurs, l'accélération $S_p(n)$ obtenue par A est définie comme étant :

$$S_p(n) = \frac{T^*(n)}{T_p(n)} \qquad \textbf{(Équation 1)}$$

$S_p(n)$ mesure donc le facteur d'accélération obtenu par un algorithme A quand p processeurs sont disponibles (JàJà (1992)).

Il y a cependant une limite d'accélération qu'un algorithme peut atteindre. En effet, certaines parties d'un programme sont facilement parallélisables, alors que d'autres sont strictement séquentielles. Quand un grand nombre de processeurs est disponible, les parties parallélisables sont exécutés plus rapidement, mais les sections qui doivent demeurer séquentielles ne gagneront pas de vitesse. Cette observation est connue sous le nom de la loi d'Amdahl (1967) et s'exprime comme suit : si un programme consiste en deux sections, une qui est strictement séquentielle et une qui est pleinement parallélisable, et si la section strictement séquentielle utilise une fraction f du temps total de calcul, alors l'accélération est limitée par :

$$S_p(n) < \frac{1}{f + \frac{1-f}{p}} < \frac{1}{f} \quad , \quad \forall p \qquad \textbf{(Équation 2)}$$

2.3.3.2 L'efficacité

Une autre mesure de performance d'un algorithme parallèle est l'efficacité (*efficiency*) définie par :

$$E_p(n) = \frac{T_1(n)}{pT_p(n)} \qquad \textbf{(Équation 3)}$$

$T_1(n)$ représente le temps d'exécution de l'algorithme parallèle quand le nombre de processeurs p est égal à 1 (qui n'est pas nécessairement le même que le temps d'exécution séquentiel $T^*(n)$). Cette mesure donne une indication sur l'efficacité de l'utilisation des p processeurs dans l'algorithme parallèle. Une valeur de $E_p(n)$ approximativement égale à 1, pour un p donné, indique que l'algorithme A s'exécute p fois plus vite en utilisant p processeurs qu'il ne le fait avec 1 processeur. Chaque processeur effectue donc un « travail utile » durant chaque étape de l'algorithme (JàJà (1992)).

Cette mesure est souvent utilisée à des fins de comparaison avec les algorithmes séquentiels. Dans ce cas, $T_1(n)$ est remplacé par le temps d'exécution du meilleur algorithme séquentiel connu (ou possible). Plus grande est l'efficacité, meilleure est la solution parallèle (Akl (2000)).

Le nombre de processeurs est une mesure de performance étroitement liée à l'efficacité. De plus, pour des raisons économiques, le nombre de processeurs dont a besoin un algorithme parallèle est d'une importance considérable. Si deux algorithmes conçus pour résoudre un problème quelconque sur un modèle parallèle donné ont des temps d'exécution identiques, celui qui nécessite le moins de processeurs sera le moins coûteux à exécuter, donc sera le plus intéressant.

2.3.3.3 L'iso-efficacité

L'accélération et l'efficacité dépendent de la taille du problème et du nombre de processeurs. En fixant la taille du problème, il est possible d'étudier l'évolution de ces indicateurs en fonction du nombre de processeurs. L'inverse est également possible, c'est-à-dire qu'en fixant le nombre de processeurs, on peut étudier l'évolution de l'accélération et de l'efficacité en fonction de la taille du problème à résoudre. L'iso-efficacité représente la relation qu'il doit y avoir entre le nombre de processeurs p et la taille du problème n pour qu'une certaine efficacité donnée soit atteinte. Cette relation peut être utile pour déterminer le nombre idéal de processeurs pour une instance de problème donnée ou pour estimer la taille minimale que devrait avoir un problème pour être efficace avec un nombre déterminé de processeurs (Gentler *et al.* (1996)).

La performance des algorithmes parallèles est influencée par un ensemble de facteurs qui sont dépendants les uns des autres. La section suivante traitera de certains de ces éléments.

2.3.4 Facteurs de performance des algorithmes parallèles

L'approche la plus naturelle pour développer un algorithme parallèle est de prendre un algorithme séquentiel, d'identifier les étapes indépendantes et d'assigner ces étapes à processeurs. Ce faisant, nous obtenons une version parallèle de cet algorithme. Cependant, cette simple conversion donne souvent des algorithmes possédant un faible degré de parallélisme et qui ne sont pas efficaces (Codenotti et Leoncini (1993)). Il ne faut pas non plus oublier que l'extraction du parallélisme du meilleur algorithme séquentiel n'aboutit

pas forcément au meilleur algorithme parallèle, chaque algorithme présentant un profil plus ou moins favorable à une parallélisation (Gentler *et al.* (1996)).

Lorsque que l'on considère un algorithme parallèle ou lorsqu'on envisage la parallélisation d'un algorithme, il convient d'étudier certains paramètres importants. Cette section présente ceux qui sont les plus pertinents à ce travail.

2.3.4.1 Le grain de parallélisme

Le grain de parallélisme est la taille moyenne des tâches élémentaires qui ont guidé la parallélisation. On le mesure souvent en nombre d'instructions machine, en taille de la mémoire employée ou en durée d'exécution. Le choix de ce grain est fortement lié aux caractéristiques de la machine utilisée (architecture, nombre de processeurs, etc.). De manière sommaire, on parle souvent de gros grain (*coarse grain*) et de grain fin (*fine grain*) pour indiquer la taille des entités de parallélisation. Schématiquement, on considère la hiérarchie suivante (de gros grain à grain fin) : programmes, procédures, instructions, expressions, opérateurs et bits.

Mesure complémentaire très étroitement liée au grain de parallélisme, le degré de parallélisme correspond à une mesure du nombre d'opérations exécutées simultanément et reflète le nombre de processeurs que l'on pourra utiliser. Il peut être évalué systématiquement à partir de la description de l'algorithme parallèle ou évalué dynamiquement par des mesures pendant l'exécution. Le degré de parallélisme peut varier largement dans les différentes parties d'un programme. Ainsi, on est souvent amené à considérer les degrés de parallélisme maximal, minimal et moyen d'un algorithme.

Comme le degré maximal constitue une borne supérieure du nombre de processeurs, le choix d'utiliser ce maximum pour un algorithme donné permet d'obtenir une bonne rapidité d'exécution durant certaines parties de l'algorithme, mais peut donner une très mauvaise efficacité dans le cas où le degré maximal excède de loin le degré moyen. Dans ce cas, un grand nombre de processeurs sont laissés inactifs pendant une partie considérable de la durée d'exécution du programme. La détermination du nombre optimal de processeurs, du point de vue de l'efficacité obtenue, est souvent difficile.

2.3.4.2 Les communications

Une partie du temps d'exécution d'un algorithme est consacrée aux échanges d'information entre deux ou plusieurs processeurs. L'ensemble de ces échanges représente la charge de communication de cet algorithme, charge qui peut être plus ou moins prohibitive sur l'efficacité réalisable selon le cas.

Dans un modèle à mémoire partagée, l'échange d'information entre les processeurs se fait par l'entremise de la mémoire commune qui est lue et écrite par les processeurs. La charge de communication d'un algorithme dans ce modèle est donc représentée par la taille des données transférées entre la mémoire partagée et les mémoires locales (registres) de tous les processeurs (JàJà (1992)). Cette charge dépend entre autres des temps d'accès à la mémoire et des mécanismes de gestion des lectures/écritures simultanées.

Dans un modèle à mémoires distribuées, les communications sont effectuées par passage de messages, c'est-à-dire que les processeurs envoient/reçoivent explicitement des données et des instructions aux/des autres processeurs. Des algorithmes de routage sont

utilisés pour faire parvenir les messages d'un processeur à un autre. Le coût des communications d'un algorithme dans ce modèle dépend, entre autres, de la topologie du réseau de processeurs, de la bande passante disponible sur les canaux de communication reliant ces processeurs, de la taille des messages et de la quantité de messages envoyés durant l'exécution de l'algorithme. Les deux derniers éléments sont directement reliés à la structure de l'algorithme et au nombre de processeurs utilisés.

2.3.4.3 La synchronisation

Dans un environnement parallèle où des opérations sont effectuées simultanément, il est souvent nécessaire d'établir un certain ordre dans le déroulement des événements qui sont dépendants les uns des autres. Les opérations de synchronisation permettent, entre autres, de garantir que tous les processeurs ont terminé une partie de leurs calculs à un moment donné, ou qu'une opération est effectuée par un seul processeur à la fois si nécessaire. Cette opération est utile, par exemple, dans le cas où le processeur P_x a besoin d'une information qui est calculée par le processeur P_y. Il faut s'assurer que l'information a été effectivement calculée par P_y avant son utilisation par P_x pour que l'algorithme s'exécute correctement.

Dans un modèle à mémoire partagée, les synchronisations peuvent être effectuées, entre autres, par des barrières ou des sémaphores. Si une barrière est placée dans un programme, aucun processeur ne pourra continuer son exécution au-delà de ce point tant que tous les processeurs ne l'auront pas atteint. Les sémaphores permettent de gérer les entrées et les sorties dans les zones critiques, c'est-à-dire les parties de l'algorithme qui ne

peuvent être exécutées que par un seul processeur (ou un nombre maximal déterminé) à la fois.

Dans un modèle à passage de messages, les communications sont effectuées par des opérations d'envoi et de réception de messages. Une façon de réaliser une synchronisation dans ce modèle est en utilisant des envois et des réceptions bloquants. Par exemple, au moment où les processeurs ont fini leur calcul et sont prêts à être synchronisés, ils envoient un message indiquant ce fait à un processeur chargé de gérer la synchronisation. Ensuite, les processeurs attendent (n'effectuent aucun traitement) tant qu'ils n'ont pas reçu le message de fin de synchronisation.

Durant une synchronisation, les processeurs demeurent inactifs durant un certain délai qui peut être plus ou moins long selon l'algorithme et le modèle utilisé. Ce délai peut représenter un obstacle considérable à l'atteinte d'une efficacité parallèle satisfaisante, donc doit être minimisé autant que possible.

2.3.4.4 Le placement des tâches

La conception d'un algorithme parallèle implique la répartition de calculs et de données de tailles possiblement différentes sur l'ensemble des processeurs. Cette répartition des tâches doit être la plus équitable possible afin de réduire au minimum le temps où les processeurs sont improductifs et ainsi maximiser l'efficacité de l'algorithme. Il arrive cependant que la charge de calcul d'une tâche n'est pas connue à l'avance, que le nombre de tâches varie durant l'exécution ou que des contraintes système ou technologiques font en sorte de ralentir le temps d'exécution de certaines tâches.

Selon Calégari (1999), trois stratégies d'allocation des tâches sont possibles tout dépendant du moment où l'allocation et le nombre des tâches sont déterminés. S'ils sont tous les deux déterminés au moment de la compilation du programme, alors l'allocation est *statique*. Si le nombre de tâches est déterminé au moment de la compilation et l'allocation se fait durant l'exécution, alors l'allocation est *dynamique*. Finalement, s'ils peuvent tous les deux varier durant l'exécution du programme, alors l'allocation est *adaptative*. Dans les deux derniers cas, un algorithme d'*équilibre de charges* (*load balancing*) est nécessaire pour allouer les tâches aux processeurs au moment de l'exécution de l'algorithme. Si la charge de calcul des différentes tâches est plus ou moins la même, alors une stratégie d'allocation statique peut être suffisante si le nombre de tâches est connu. Si, à l'opposé, les tâches sont hétérogènes, alors une stratégie dynamique sera probablement nécessaire si l'on veut atteindre une bonne efficacité.

L'architecture utilisée peut influencer le choix de la stratégie de placement des tâches. Par exemple, si l'environnement parallèle utilisé est un réseau de stations dont la charge est variable, alors l'allocation dynamique des tâches aux processeurs pourrait être la plus appropriée même si leur hétérogénéité est relativement faible. Notons également que la granularité de la parallélisation et le placement des tâches sont considérablement reliées. Si l'allocation des tâches est coûteuse et qu'il y a plus de tâches concurrentes que de processeurs, il peut être intéressant de regrouper un certain nombre de tâches et, par le fait même, augmenter la granularité et la performance.

Afin d'utiliser de façon pratique, sur des ordinateurs réels, les algorithmes parallèles conçus, il existe divers environnements et langages de programmation parallèles. La

prochaine et dernière section de ce chapitre sur le parallélisme sera consacrée au survol des principaux modèles de programmation existants.

2.4 Les modèles de programmation

Les environnements et langages de programmation de haut niveau jouent un rôle important en informatique. Ils servent principalement à simplifier l'expression d'algorithmes complexes, à augmenter la portabilité du code, à faciliter la réutilisation du code et à réduire le risque d'erreurs de programmation en libérant le programmeur de la tâche de gérer les ressources de bas niveau. Un environnement de programmation inadéquat peut mener à un programme inefficace même si l'algorithme sous-jacent est performant. L'informatique parallèle complique d'autant plus la tâche de développer un programme correct et efficace. De plus, la diversité des plate-formes et les besoins spécifiques des utilisateurs ont fait en sorte qu'aucun langage ou environnement parallèle n'a pu s'imposer par rapport aux autres. Nous sommes plutôt en présence d'une variété de langages proposés, les uns recherchant la généralité et les autres la haute performance, mais aucun ne permettant l'exécution de haute performance pour un ensemble d'applications et sur une variété d'ordinateurs parallèles (Foster (2000)).

Dans cette section, un bref survol des modèles de programmation parallèle sera effectué en présentant une classification selon quatre façons d'aborder le développement de programmes parallèles : les paradigmes nouveaux ou non traditionnels, les extensions à des langages séquentiels existants, les langages à parallélisme de données et les approches basées sur les bibliothèques (Foster (2000)).

2.4.1 Les paradigmes nouveaux

Une vision de la programmation parallèle correspond à l'idée que les langages séquentiels, étant conçus pour représenter la manipulation de machines de Von Neumann (présentées au début de ce chapitre), sont inappropriés pour traiter le développement de programmes sur des machines parallèles, et ce particulièrement dans les cas où le but recherché est l'expression de parallélisme massif. Cette vision a été une source de motivation pour l'exploration d'un certain nombre de modèles de programmation comme la programmation fonctionnelle (Hilzer et Crowl (1995)), la programmation logique (Taylor (1989)) et les systèmes d'acteurs (Agha (1986)).

2.4.2 Les extensions à des langages séquentiels existants

Une autre façon de voir les choses est que, plutôt que de définir un langage complètement nouveau, on peut choisir d'ajouter des fonctionnalités de parallélisme dans un langage séquentiel existant par le biais d'extensions appropriées. Cette approche a été mieux acceptée que l'utilisation de nouveaux langages, mais elle comporte sa part de problèmes. La nécessité de travailler avec les contraintes d'un langage qui n'a pas été conçu à la base dans le but de traiter le parallélisme rend difficile, dans certains cas, l'exécution parallèle correcte et efficace.

Dans cette catégorie de langages, on retrouve, entre autres, OpenMP, qui est encore à ses débuts mais qui semble en voie de devenir largement adopté (Foster (2000)). C'est une collection de directives de compilation, de bibliothèques de fonctions et de variables d'environnement qui permettent d'exprimer le parallélisme sur des architectures à mémoire

partagée. Notons qu'OpenMP, étant une extension mais aussi une bibliothèque de fonctions, peut être vue comme faisant partie de la catégorie des approches basées sur les bibliothèques. Un exemple d'extension pour architectures à mémoires distribuées est Concert/C, qui permet au programmeur de diviser un programme en processus distincts, qui supporte l'appel de procédures distantes (RPC) et qui inclut des mécanismes d'envoi et de réception de messages. Les langages à parallélisme de données représentent un cas spécial de cette catégorie et font l'objet de la prochaine section.

2.4.3 Les langages à parallélisme de données

Le parallélisme de données représente la concurrence qui est obtenue quand la même opération est appliquée à certains éléments d'un ensemble de données. Dans ce contexte, un algorithme parallèle est obtenu en appliquant des techniques de décomposition de domaine aux structures de données sur lesquelles les opérations sont effectuées. Le modèle de programmation à parallélisme de données est un modèle de haut niveau dans lequel le programmeur n'a pas à spécifier les structures de communication de façon explicite, ces dernières étant établies par un compilateur à partir de la décomposition de domaine spécifiée par le programmeur. De plus, ce modèle est restrictif en ce sens que tous les algorithmes ne peuvent pas être exprimés en termes de parallélisme de données. Pour ces raisons, ce modèle de programmation est un paradigme important, mais non universel, du calcul parallèle. Les extensions à ce modèle dans le but de permettre son application à un éventail plus large de problèmes est un domaine de recherche actif.

Des exemples de langages à parallélisme de données sont High Performance Fortran (HPF), langage très utilisé pour des applications scientifiques et d'ingénierie, C* et pC++.

2.4.4 Les approches basées sur les bibliothèques

Les langages faisant partie de cette dernière approche ne sont pas vraiment des langages, mais plutôt des paradigmes du calcul parallèle qui sont implémentés par des appels à des bibliothèques (Foster (2000)). Dans la pratique, une majeure partie de la programmation parallèle est effectuée de cette façon, que ce soit pour les systèmes à mémoire partagée (principalement par les threads POSIX) que pour ceux à mémoire distribuée (PVM, MPI, etc.).

Deux approches très populaires, autant dans cette catégorie que dans la programmation parallèle en général, sont MPI (***Message Passing Interface***) et PVM (***Parallel Virtual Machine***). MPI est un standard qui définit un ensemble de fonctions implémentant une approche par passage de messages. Dans ce modèle, un programme comporte plusieurs processus (généralement un par processeur) qui communiquent les uns avec les autres en appelant des routines d'envoi et de réception de messages. Les processus/processeurs peuvent exécuter des programmes différents, raison pour laquelle ce modèle de programmation est parfois référé comme étant MPMD (***Multiple Program Multiple Data***) afin de le différencier du modèle SPMD (***Single Program Multiple Data***) dans lequel chaque processeur exécute le même programme. Les langages de programmation séquentielle les plus utilisés avec MPI sont C, C++ et Fortran.

PVM est également une bibliothèque implémentant un modèle à passage de messages, mais il est possible de l'utiliser sur un réseau de stations de travail de toutes sortes, ce qui n'est pas le cas pour MPI. C'est d'ailleurs dans ce but qu'il a été créé, c'est-à-dire pour supporter le développement de programmes parallèles et distribués sur un ensemble hétérogène de machines séquentielles (Cung *et al.* (2001)).

2.5 Conclusion

Les notions qui ont été présentées dans ce chapitre réfèrent au parallélisme de façon générale et aux algorithmes « conventionnels ». Dans le chapitre suivant, nous verrons comment ces concepts peuvent être appliquées à un type particulier d'algorithmes : les métaheuristiques.

CHAPITRE 3

LES MÉTAHEURISTIQUES PARALLÈLES

3.1 Introduction

De façon générale, le parallélisme est utilisé pour la résolution de problèmes complexes nécessitant des algorithmes coûteux en temps d'exécution. Dans le contexte d'algorithmes exacts produisant une solution déterminée et optimale à un problème donné, les principaux buts recherchés sont de réduire le temps de résolution, c'est-à-dire d'obtenir cette solution plus rapidement, et de s'attaquer à des instances de plus grande taille.

Il existe cependant certains problèmes qui ne peuvent être résolus en temps polynomial par des algorithmes exacts de nature séquentielle ou parallèle. La plupart des problèmes d'optimisation combinatoire font partie de classe de problèmes dits NP-Difficiles (Garey et Johnson (1979)). Une stratégie de résolution de plus en plus appliquée ce type de problème est l'utilisation de métaheuristiques. Ces algorithmes d'approximation ne garantissent pas l'obtention de solutions optimales mais trouvent généralement de bonnes solutions dans un temps de calcul raisonnable.

Plusieurs travaux réalisés au cours des dernières années ont démontré l'utilité et l'efficacité des métaheuristiques pour la résolution de problèmes d'optimisation combinatoire. Il demeure toutefois que ces algorithmes demandent un temps de calcul et une quantité de mémoire considérables qui sont étroitement reliés à la taille du problème et à l'obtention d'une certaine qualité de solution. De ce fait, ces algorithmes deviennent intéressants à paralléliser. Dans ce contexte, les objectifs de réduction du temps de calcul et de traitement de gros problèmes sont toujours pertinents, mais à cela s'ajoute l'intérêt

d'utiliser le parallélisme pour améliorer la performance des métaheuristiques en termes de qualité de solution obtenue et ce, sans augmenter la charge de calcul.

Dans ce chapitre, les principaux concepts sous-jacents aux métaheuristiques seront brièvement expliqués et les métaheuristiques les plus utilisées seront sommairement décrites. Ensuite, une revue des travaux de parallélisation de métaheuristiques sera présentée et les principaux enjeux à considérer dans ce contexte seront discutés. Finalement, les objectifs de ce travail de recherche seront décrits en relation avec la revue de littérature effectuée dans les chapitres 2 et 3.

3.2 Les métaheuristiques

Les métaheuristiques ont connu un essor considérable depuis leur apparition dans les années 1970. Elles sont présentées par Osman et Laporte (1996) comme étant des méthodes d'approximation conçues dans le but de s'attaquer à des problèmes complexes d'optimisation qui n'ont pu être résolus de façon efficace par les heuristiques et les méthodes d'optimisation classiques. Ces mêmes auteurs définissent formellement la notion de métaheuristique comme étant un processus itératif qui guide une heuristique subordonnée en combinant intelligemment différents concepts pour explorer et exploiter l'espace de recherche, et qui utilise des stratégies d'apprentissage pour structurer l'information dans le but de trouver efficacement des solutions les plus rapprochées possible de la solution optimale.

Le développement des métaheuristiques fait partie d'un effort soutenu investi dans le domaine de l'optimisation combinatoire, ce dernier étant défini comme étant l'étude

mathématique de la recherche d'un arrangement, d'un groupement, d'un ordonnancement ou d'une sélection d'objets discrets habituellement finis en nombre (Osman et Laporte (1996)). Malgré les progrès remarquables qu'ont connu les algorithmes exacts durant ces dernières années, l'optimisation combinatoire constitue toujours un défi de taille pour eux. Par conséquent, les algorithmes d'approximation sont devenus une sphère importante de recherche et d'applications dans ce domaine. De plus amples détails sur les problèmes d'optimisation combinatoire peuvent être trouvés dans (Garey et Johnson (1979), Nemhauser et Wolsey (1988), Papadimitriou et Steiglitz (1982), Wong (1995)).

Le domaine des métaheuristiques est considérablement vaste et ses applications sont nombreuses. Sans être détaillée ni exhaustive, la suite de cette section sera consacrée à la présentation sommaire des métaheuristiques les plus utilisées. Pour plus amples informations concernant les métaheuristiques, le lecteur peut consulter Colorni *et al.* (1996), Gagné (2001), Osman et Laporte (1996), Taillard *et al.* (1998), Widmer *et al.* (2001).

3.2.1 Les algorithmes génétiques

Les algorithmes génétiques, introduits par J. Holland (1975), sont des méthodes évolutives inspirées des principes de la sélection naturelle. Ils génèrent de façon habituellement aléatoire une population initiale de solutions qui sont aussi appelés individus ou chromosomes. Ensuite, à chaque itération de l'algorithme, on fait évoluer cette population en y appliquant des mécanismes de sélection, de recombinaison et de mutation dans le but d'obtenir, de génération en génération, une nouvelle population. Les

meilleurs individus ont de plus grandes chances d'être sélectionnés pour la reproduction, donc de transmettre leurs caractéristiques aux prochaines générations. Un mécanisme de mutation, par exemple une modification aléatoire effectuée sur certains individus, peut être utilisé afin de préserver une certaine diversité à l'intérieur de la population. Ce processus est répété jusqu'à ce qu'un état de convergence (tous les individus sont identiques) soit atteint ou jusqu'à ce qu'une condition d'arrêt définie soit remplie (par exemple un nombre d'itération maximum ou l'atteinte d'une qualité de solution satisfaisante). Un algorithme génétique standard est illustré à la Figure 3.1.

Générer aléatoirement une population
TANT QUE Nombre de générations ≤ Nombre de génération Maximum FAIRE
    Évaluation : Assigner une valeur d'adaptation (*fitness value*) à chaque individu
    Sélection : Établir de façon probabiliste les paires d'individus qui vont se reproduire en accordant une meilleure chance aux meilleurs individus
    Reproduction : Appliquer les opérateurs génétiques aux paires sélectionnées. Les nouveaux individus produits représentent la nouvelle population

Figure 3.1 Un algorithme génétique

Il existe plusieurs travaux portant sur les algorithmes génétiques et de nombreuses variantes portant, entre autres, sur la représentation de la population, sur les différents opérateurs et sur la définition des critères de terminaison de la méthode. Le lecteur peut consulter Davis (1991) pour plus amples informations.

3.2.2 La recherche avec tabous

La méthode de recherche avec tabous a été introduite par Glover (1986). Elle explore itérativement l'espace des solutions d'un problème en se déplaçant d'une solution courante

à une nouvelle solution située dans son voisinage. Le choix de cette nouvelle solution est déterminée par l'évaluation d'une fonction objectif (Soriano et Gendreau (1997)). Pour sa part, le voisinage d'une solution représente l'ensemble des solutions pouvant être obtenues en appliquant une transformation locale à la solution courante. Afin d'éviter la prise au piège dans un optimum local, une mémoire à court terme est utilisée pour conserver de l'information sur le cheminement récent effectué. Cette information permet d'interdire certaines transformations de la solution courante qui pourraient ramener la méthode à des solutions déjà visitées et qui pourraient la faire cycler indéfiniment dans la même région de l'espace de recherche. Les opérations interdites sont alors déclarées « *tabou* », d'où le nom de la méthode, ce qui a pour effet de restreindre le voisinage aux solutions les plus susceptibles de diriger l'exploration vers des régions inexplorées. Cependant, les restrictions engendrées par ces tabous peuvent parfois être trop contraignantes et empêcher des transformations menant à des solutions non encore visitées. Afin de contrer cette situation, la méthode de recherche avec tabous comporte une fonction d'aspiration qui peut, par exemple, permettre l'annulation du statut tabou d'une transformation locale si cette dernière permet d'obtenir une solution supérieure à la meilleure trouvée depuis le début de l'algorithme.

La méthode de recherche avec tabous, illustrée dans sa forme de base à la Figure 3.2 (Taillard *et al.* (1998)), comporte toutefois plusieurs composantes optionnelles supplémentaires, notamment une mémoire à long terme et des stratégies de diversification-intensification. Pour une description complète, le lecteur peut consulter les articles de référence de Glover (1989, 1990).

```
Générer aléatoirement une solution initiale s0 qui devient la meilleure solution s*
Initialiser k à 0 et initialiser les mémoires
TANT QU'un critère d'arrêt n'est pas satisfait FAIRE
    Choisir sk+1 dans le voisinage de la solution sk, en tenant compte des mémoires
        Si sk+1 est meilleur que s*, alors s* = sk+1
        Incrémenter k
        Mettre à jour les mémoires
```

Figure 3.2 Un algorithme de recherche avec tabous

3.2.3 Le recuit simulé

La métaheuristique du recuit simulé, qui prend ses origines dans la physique statistique, est le fruit des travaux de Metropolis *et al.* (1953). Elle a été introduite dans le domaine de l'optimisation combinatoire par Kirkpatrick *et al.* (1983). Cette méthode exploite le principe selon lequel un système physique qui est chauffé à une très haute température et ensuite graduellement refroidi atteindra en bout de ligne un niveau faible d'énergie correspondant à une structure moléculaire stable et forte. À chaque niveau décroissant de température, le système atteindra, après un certain temps, un état qui sera accepté automatiquement dans le cas où l'énergie du système est inférieure. Inversement, si l'énergie est supérieure, cet état sera accepté conditionnellement selon une certaine probabilité. On accepte donc une certaine dégradation de l'état du système selon certaines conditions et ces dernières deviennent de plus en plus restrictives à mesure que la température diminue. Ce processus est répété pour chaque niveau de température jusqu'à ce qu'un état solide soit atteint.

Une transposition de ces principes au domaine de l'optimisation combinatoire est illustrée par un algorithme à la Figure 3.3. La température initiale, le nombre d'itérations

effectuées à chaque température, la vitesse de refroidissement et le critère d'arrêt sont déterminés par le schéma de refroidissement (*cooling schedule*) (Osman et Laporte (1996)) et ce dernier joue un rôle déterminant sur la performance de l'algorithme.

```
Générer aléatoirement une solution initiale s0
Initialiser la température T
TANT QUE T > 0 FAIRE
    TANT QUE   l'état énergétique n'est pas accepté FAIRE
     Générer aléatoirement un état voisin et évaluer le changement du niveau d'énergie ΔE
        Si ΔE < 0 mettre à jour l'état courant avec le nouvel état
        Si ΔE ≥ 0 mettre à jour l'état courant avec le nouvel état selon une certaine probabilité
   Diminuer la température T selon le schéma de refroidissement
Retourner la solution qui possède le niveau d'énergie le plus bas
```

Figure 3.3 Un algorithme de recuit simulé

3.2.4 Le GRASP (*Greedy Randomized Adaptive Search Procedures*)

La méthode GRASP est une procédure itérative qui a été développée par Feo et Resende (1988). Celle-ci consiste en un certain nombre d'itérations constituées de deux phases : une phase de construction et une phase de recherche locale (Feo et Resende (1995)).

Dans la phase de construction, une solution admissible est construite, un élément à la fois, en choisissant aléatoirement un élément parmi une liste ordonnée de candidats. Cette dernière est établie à chaque construction par une méthode vorace qui mesure de façon « aveugle » le bénéfice qu'apporte la sélection de chaque élément. Ensuite, un certain nombre d'éléments correspondant aux meilleurs candidats est retenu pour constituer la liste de candidats. Le comportement adaptatif de la procédure provient du fait qu'à chaque fois qu'un élément est sélectionné par la fonction vorace, les bénéfices associés à la sélection de

chaque élément sont mis à jour en tenant compte des changements engendrés par la sélection qui a été précédemment effectuée.

La phase de recherche locale a pour but d'améliorer chaque solution générée dans la phase de construction. Cette phase s'effectue en remplaçant successivement la solution courante par une meilleure solution se trouvant dans son voisinage. Elle se termine au moment où aucune meilleure solution n'est trouvée dans le voisinage et cette solution représente alors l'optimum local (Feo et Resende (1995)).

Les phases de construction et de recherche locale sont répétées durant un certain nombre d'itérations maximal ou jusqu'à l'atteinte d'un critère d'arrêt défini. La meilleure solution trouvée durant le processus est conservée.

3.2.5 L'Optimisation par Colonies de Fourmis (OCF)

L'algorithme d'Optimisation par Colonies de Fourmis (OCF) prend son origine dans les travaux de Colorni *et al.* (1991) et de Dorigo (1992). La première version de l'OCF, nommée « *Ant System* », est basée sur des études du comportement collectif des fourmis dans la recherche de nourriture (Deneubourg et Goss (1989), Deneubourg *et al.* (1983), Goss *et al.* (1990)). Les fourmis communiquent entre elles par l'intermédiaire de la phéromone, une substance qu'elles déposent sur le sol en se déplaçant et qui est détectée par les autres fourmis. Si une source importante de nourriture est située près du nid, les premières fourmis qui la trouvent feront le trajet de cette source au nid plusieurs fois. Ce faisant, elles y déposeront une quantité croissante de phéromone qui attirera davantage de fourmis. Ce trajet deviendra donc, après un certain temps, privilégié par les fourmis. Une

fois la source de nourriture épuisée, les fourmis arrêteront de l'emprunter, la phéromone s'évaporera et d'autres trajets seront définis. S'il arrivait qu'un obstacle, par exemple une pierre ou une branche d'arbre, vienne bloquer le trajet emprunté par les fourmis, ces dernières essaieront alors de le contourner à gauche et à droite de façon à peu près aléatoire jusqu'à ce qu'elles aient retrouvé la trace de phéromone. Les fourmis ayant choisi la route la plus courte effectuant un plus grand nombre d'allers-retours, la quantité de phéromone déposée y sera plus importante et cette route deviendra éventuellement empruntée par toutes les fourmis. Les travaux de Colorni *et al.* (1991), Dorigo *et al.* (1991), Dorigo et Gambardella (1997a) apportent de l'information détaillée sur le fonctionnement de l'algorithme « *Ant System* ».

Durant les années qui ont suivi l'apparition de la version de base « *Ant System* », plusieurs travaux visant à améliorer et étendre cette méthode ont été effectuées. Par exemple, l'ACS (*Ant Colony System*) a été proposé par Dorigo et Gambardella (1997b) et le *MAX-MIN Ant System* (MMAS) a été proposé par Stützle et Hoos (1998). Dans un effort d'unification de toutes ces versions, Dorigo et Di Caro (1999) ont présenté une définition de la métaheuristique d'Optimisation par Colonies de Fourmis (OCF). Comme l'OCF est utilisée dans le cadre de ce travail, plus de détails sur son fonctionnement et son application au problème traité seront fournis au chapitre suivant.

À travers leur évolution, les métaheuristiques ont été appliquées à plusieurs problèmes d'optimisation combinatoire. Quelques exemples des problèmes les plus étudiés sont le problème du voyageur de commerce, le problème d'affectation quadratique, le problème d'ordonnancement de machines et le problème de tournées de véhicules.

Les métaheuristiques ont profondément changé la façon dont ces problèmes sont abordés et ont contribué significativement à l'atteinte de solutions efficaces (Crainic et Toulouse (1998)). Cependant, la charge importante de calcul associée à une bonne exploration de l'espace des solutions, particulièrement dans le cadre de problèmes de grande taille, demeure l'une des principales limitations de ces méthodes. Cette contrainte non négligeable justifie le recours au parallélisme (Cung *et al.* (2001)). Dans la prochaine section, l'intégration du parallélisme dans le monde des métaheuristiques en général sera abordée. Ensuite, des versions parallèles des métaheuristiques qui ont été décrites précédemment seront présentées.

3.3 Le parallélisme et les métaheuristiques

Même si les métaheuristiques offrent des stratégies efficaces pour la recherche de solutions à des problèmes d'optimisation combinatoire, les temps de calcul associés à l'exploration de l'espace de recherche peuvent être très importants (Cung *et al.* (2001)). Une façon d'accélérer cette exploration est l'utilisation du parallélisme. Une autre contribution du calcul parallèle aux métaheuristiques est de plus en plus constatée : en leur appliquant des paramètres appropriés, les métaheuristiques parallèles peuvent être beaucoup plus robustes que leurs équivalents séquentiels en termes de qualité de solutions trouvées (Crainic et Toulouse (1998)). Certains mécanismes parallèles permettent donc de trouver de meilleures solutions sans nécessiter un nombre total d'opérations plus grand.

Le parallélisme peut donc représenter un apport considérable aux métaheuristiques et un nombre grandissant de travaux sont effectués afin d'exploiter ce potentiel.

Actuellement, la plupart de ces travaux sont limités à une métaheuristique ou à une stratégie de parallélisation particulière et rares sont ceux qui apportent une vue globale et fondamentale aux métaheuristiques parallèles (Crainic et Toulouse (1998)). Constatant ce fait, certains auteurs ont proposé des classifications des métaheuristiques parallèles. La section suivante en présente deux.

3.3.1 Classification des métaheuristiques parallèles

Crainic et Toulouse (1998) ont tenté, en examinant les particularités communes aux différentes implémentations parallèles retrouvées dans le monde des métaheuristiques, d'en établir une classification.

3.3.1.1 La classification de Crainic et Toulouse

Cette classification, divisée en trois catégories, est basée sur le degré d'impact de la stratégie de parallélisation sur la structure algorithmique de la métaheuristique.

La première catégorie représente une stratégie de parallélisation à l'intérieur d'une itération de la métaheuristique. Cette stratégie, considérée comme étant une parallélisation de bas niveau ou une parallélisation interne, s'applique de façon relativement directe et a pour but d'accélérer les calculs sans chercher à effectuer une meilleure exploration. La méthode de solution parallèle est la même que son équivalent séquentiel dans le mesure où le nombre d'opérations total est identique pour les deux versions. À cet effet, il est possible de profiter de la puissance de calcul disponible et ainsi améliorer la recherche de solutions en utilisant cette stratégie sans changer fondamentalement l'algorithme. Il s'agirait alors

d'effectuer un plus grand nombre d'opérations total en parallèle tout en conservant un temps d'exécution global égal ou inférieur à la version séquentielle.

La deuxième catégorie est une stratégie de décomposition du domaine du problème ou de l'espace de recherche. Le principe de cette approche consiste à diviser le problème en sous-problèmes et à utiliser la puissance de calcul afin de résoudre ces sous-problèmes en parallèle. Le comportement de recherche est alors modifié par rapport à l'algorithme séquentiel. Un modèle maître-esclave est généralement utilisé pour appliquer cette stratégie. Le maître détermine séquentiellement les partitions initiales et les modifie durant l'exécution à des intervalles qui peuvent être prédéfinis ou déterminés dynamiquement durant l'exécution. Les esclaves explorent de façon concurrente et indépendante l'espace de recherche qui leur a été assigné par le maître et ce dernier assemble les solutions partielles trouvées par les esclaves en une solution complète au problème.

Dans la troisième catégorie, plusieurs processus de recherche (*multisearch threads*) sont appliqués avec différents degrés de synchronisation et de coopération. Par cette stratégie, on cherche à effectuer une recherche plus complète en mettant en œuvre plusieurs processus qui opèrent simultanément et qui sont guidés de différentes façons. La métaheuristique peut être calibrée différemment pour chacune des recherches, mais il est également possible d'utiliser de façon concurrente différentes métaheuristiques. Les processus peuvent communiquer entre eux durant leur recherche ou seulement à la toute fin afin d'identifier la meilleure solution.

On peut alors subdiviser les processus de recherche multiples de cette troisième catégorie en deux classes : approches indépendantes et approches coopératives. Dans la

première classe, plusieurs recherches sont initiées et le meilleur résultat est sélectionné parmi celles-ci à la fin. Cette approche est alors équivalente à une stratégie de redémarrage accéléré. Plutôt que de redémarrer l'algorithme plusieurs fois et selon différents paramètres, toutes les exécutions sont effectuées en parallèle. Afin de définir une recherche multiple coopérative, c'est-à-dire une approche de la deuxième classe, plusieurs paramètres importants doivent être déterminés. Ceux-ci sont :

- la topologie du réseau d'interconnexion qui définit comment les processus sont reliés entre eux ;
- la méthode de communication entre les processus (diffusion, point à point, mémoire centrale, etc.) ;
- le choix des processus participant aux échanges d'information ;
- le type de communication (synchrone ou asynchrone) ;
- les moments où de l'information sera échangée ;
- l'information à échanger.

L'ajustement de ces paramètres peut avoir un impact significatif sur le comportement et la performance de la recherche. En effet, selon les auteurs de cette classification, on commence à réaliser, dans la communauté des métaheuristiques parallèles, l'importance de la définition de mécanismes de coopération et de l'étude de leur impact sur la performance des métaheuristiques. Cet aspect comporte donc encore, selon eux, plusieurs enjeux à attaquer et représente un domaine de recherche prometteur.

Toujours dans l'optique d'en dégager les principales caractéristiques, Cung *et al.* (2001) ont proposé une classification des stratégies de parallélisation des métaheuristiques qui est indépendante de l'architecture utilisée. Cette classification, qui se rapproche beaucoup de

celle de Crainic et Toulouse tout en comportant ses propres particularités, se base sur les principes de recherche locale parallèle tels que présentés par Verhoeven et Aarts (1995).

3.3.1.2 La classification de Cung *et al.*

Les métaheuristiques basées sur la recherche locale peuvent être vues comme explorant un graphe de voisinage associé à une instance de problème. Dans ce graphe, les nœuds correspondent à des solutions et les arcs connectent des solutions voisines. Chaque itération consiste en l'évaluation des solutions présentes dans le voisinage de la solution courante et au déplacement vers une de celles-ci, tout en évitant le plus possible de rester pris au piège prématurément dans un optimum local. Ce processus d'évaluation-déplacement est répété jusqu'à l'atteinte d'un nombre maximal d'itérations ou jusqu'à ce qu'il ne soit plus possible de trouver de meilleures solutions. Les solutions visitées durant cette recherche définissent une *marche*, aussi appelée trajectoire, dans le graphe de voisinage. Les implémentations parallèles des métaheuristiques utilisent alors plusieurs processeurs afin de générer ou explorer ce graphe de façon concurrente. Comme ce dernier n'est pas connu d'avance, les parallélisations de métaheuristiques sont des *applications irrégulières* (Roch *et al.* (1995)) dont l'efficacité dépend fortement du choix de la granularité de l'algorithme parallèle et de l'utilisation de techniques d'équilibrage de charges.

On distingue alors deux approches pour la parallélisation de la recherche locale basées sur le nombre de marches qui sont effectuées dans le graphe de voisinage :

1. Marche simple : tâches de granularité fine à moyenne

2. Marches multiples : tâches de granularité moyenne à grosse
    a. Processus de recherche indépendants
    b. Processus de recherche coopératifs

Dans le cas de parallélisations à marche simple, une trajectoire unique est traversée dans le graphe de voisinage. La recherche du meilleur voisin est effectuée en parallèle à chaque itération par la parallélisation de la fonction d'évaluation des solutions ou par décomposition de domaine. Cette approche, dont le but est d'accélérer la traversée du graphe de voisinage, correspond sensiblement à la première catégorie définie plus tôt par Crainic et Toulouse (1998). Si la fonction d'évaluation des solutions voisines est parallélisée, alors une accélération substantielle peut être atteinte sans modifier la trajectoire parcourue par la méthode séquentielle. Dans l'autre cas, la décomposition du voisinage et sa répartition sur plusieurs processeurs permet d'examiner une part plus large du voisinage que par une approche séquentielle. Par conséquent, la trajectoire suivie peut être mieux guidée et ainsi conduire à de meilleures solutions.

La plupart des implémentations parallèles de métaheuristiques, mis à part le recuit simulé, sont basées sur des stratégies de marches multiples. L'idée principale est d'effectuer l'exploration de plusieurs trajectoires en parallèle sur des processeurs distincts. Les tâches exécutées en parallèle ont alors une granularité plus grosse. En plus d'accélérer les traitements, ces stratégies cherchent à améliorer la qualité des solutions trouvées. Cette catégorie, se subdivisant selon le caractère indépendant ou coopératif des processus de recherche, est semblable à la troisième catégorie définie par Crainic et Toulouse (1998).

Dans le cas de processus de recherche indépendants, on distingue deux approches de base. La première est l'exploration de plusieurs trajectoires provenant de différents endroits du graphe en débutant la recherche à partir de différentes solutions ou populations de départ. Les processus de recherche peuvent utiliser ou non le même algorithme avec des paramètres semblables ou différents. La deuxième correspond à l'exploration en parallèle de sous-graphes obtenus par décomposition de domaine sans qu'il n'y ait d'intersection entre les trajectoires correspondantes.

Si les processus de recherche sont coopératifs, ils échangent ou partagent l'information qu'ils ont accumulé durant leur parcours afin d'améliorer leur recherche respective. Ce partage d'information peut être implémenté par des variables globales dans une mémoire partagée ou par un bassin de variables locales à un processeur dédié accessible par tous les autres processeurs. Dans ce modèle, on cherche non seulement à accélérer la convergence vers de bonnes solutions mais aussi à trouver de meilleures solutions qu'en utilisant des stratégies de marches indépendantes tout en leur accordant le même temps d'exécution. Comme il a été mentionné plus tôt, la principale difficulté dans ce modèle de parallélisation est de déterminer la nature des partages ou des échanges d'information afin d'améliorer la recherche sans trop sacrifier de temps ou de mémoire. Le principe de coopération sera expliqué plus en détails dans une prochaine section.

Les classifications présentées soulignent l'importance de la stratégie de parallélisation sur la qualité de la métaheuristique résultante. Afin d'évaluer cette qualité, on peut, dans certains cas, utiliser les mesures standard de performance des algorithmes parallèles qui ont été présentées dans le deuxième chapitre. Ces mesures ne sont cependant pas suffisantes

pour la plupart des cas de parallélisation de métaheuristiques. La section suivante aborde cette question de façon plus détaillée.

3.3.2 Mesures de performance des métaheuristiques parallèles

Si la stratégie de parallélisation utilisée correspond à la première catégorie de la classification de Crainic et Toulouse (1998), c'est-à-dire à une parallélisation interne, l'algorithme parallèle suit la même voie d'exploration que son équivalent séquentiel et ne modifie pas son comportement. Le bénéfice apporté par la parallélisation consiste alors à réduire le temps d'exécution de la métaheuristique tout en conservant la même qualité de solution. On peut ainsi mesurer la performance de l'algorithme parallèle en lui appliquant directement les critères d'évaluation standard présentés dans le deuxième chapitre, notamment l'accélération et l'efficacité. Cependant, si la stratégie de parallélisation implique un nombre total d'opérations différent de la méthode séquentielle ou que des modifications ont été apportées dans le processus de recherche de solutions, il faut alors mettre la performance de l'algorithme parallèle en relation avec la qualité des solutions qu'il permet d'obtenir.

Cung *et al.* (2001) abordent l'efficacité des implémentations parallèles des métaheuristiques en utilisant une stratégie de marches multiples indépendantes basée sur l'exécution de plusieurs copies du même algorithme séquentiel. Une valeur cible τ pour la fonction à optimiser est diffusée à tous les processeurs qui exécutent ensuite l'algorithme séquentiel de façon indépendante. Tous les processeurs s'arrêtent aussitôt qu'un d'entre eux a trouvé une solution égale ou meilleure à τ. L'accélération est alors donnée par le

ratio entre le temps requis pour trouver une solution au moins aussi bonne que τ en utilisant l'algorithme séquentiel et celui requis en utilisant l'implémentation parallèle avec p processeurs. Cette approche est intéressante si l'algorithme est fortement basé sur un processus de recherche aléatoire (Stützle (1998)). Dans ce cas, les exécutions parallèles indépendantes ont de fortes chances d'explorer des régions différentes de l'espace de recherche. Certaines d'entre elles peuvent être plus intéressantes que d'autres et mener à de bonnes solutions plus rapidement qu'avec une seule exécution séquentielle. Cela revient à affirmer que l'algorithme séquentiel serait plus efficace si on le redémarrait k fois en lui donnant un temps t à chaque fois plutôt que de l'exécuter une fois seulement en lui donnant un temps $k*t$.

Comme il en a été discuté à quelques reprises, la contribution du parallélisme aux métaheuristiques peut être plus grande qu'une simple accélération du temps de calcul. Par des marches multiples indépendantes mais surtout coopératives, il est possible d'améliorer, dans plusieurs cas, la qualité des solutions trouvées par la métaheuristique séquentielle (Crainic et Toulouse (1998), Cung *et al.* (2001)) et ce, sans nécessiter un plus grand nombre de calculs. Afin de mesurer cette contribution, une démarche relativement simple et logique consiste à comparer les valeurs des meilleures solutions obtenues pour la résolution d'un même problème par les versions séquentielle et parallèle de la métaheuristique. Notons qu'il faut accorder un même nombre d'opérations total aux deux versions pour que la comparaison soit valide. Cette mesure peut relever davantage d'un jugement de valeur que de calculs mathématiques. En effet, il n'est pas tout de déterminer si les solutions obtenues par l'implémentation parallèle sont supérieures à l'algorithme

séquentiel car il faut également se demander si le gain obtenu justifie les moyens de calcul et l'effort algorithmique déployés. Cette justification est dépendante du problème abordé et des objectifs recherchés et nécessite l'apport d'un certain jugement.

Si la stratégie employée implique des marches multiples coopératives, une évaluation adéquate des bénéfices apportés par les mécanismes de coopération pourrait être souhaitable (Huberman (1990)). Il serait alors approprié de comparer les résultats d'une parallélisation à marches multiples indépendantes et d'une à marches multiples coopératives de la même métaheuristique afin de souligner les effets de la coopération.

Dans la prochaine section, le concept de coopération impliqué dans les stratégies de marches multiples coopératives sera présenté plus en détails.

3.3.3 La coopération

L'idée de base de la coopération est qu'un groupe d'agents coopérants engagés dans la résolution d'un problème peut résoudre ce dernier plus efficacement qu'un agent seul ou qu'un groupe d'agents opérant isolément les uns des autres. Plusieurs exemples tirés de la vie humaine et animale tendent à confirmer cette idée. D'un point de vue informatique, il peut être intéressant d'appliquer cette approche dans la conception d'algorithmes de résolution de problèmes complexes.

L'idée de résoudre un problème à l'aide de plusieurs agents coopérants ou isolés qui opèrent en même temps présente un parallélisme naturel. Une architecture parallèle semble donc appropriée pour transposer cette idée dans un contexte informatique.

De plus, un des principes fondamentaux des métaheuristiques est qu'elles mémorisent des solutions ou des caractéristiques des solutions visitées durant le processus de recherche passé et utilisent cette information pour les guider dans la recherche à venir (Taillard *et al.* (1998)). Un processus d'apprentissage est donc effectué durant l'exécution de l'algorithme et influence son comportement. Comme la qualité de l'algorithme est intimement liée à la pertinence de l'information conservée et utilisée, l'intégration de mécanismes de coopération peut créer de l'information nouvelle qui permettra à l'algorithme d'effectuer une meilleure recherche. La coopération permet donc une nouvelle forme d'apprentissage susceptible d'améliorer la performance des métaheuristiques.

Clearwater *et al.* (1992) définissent la coopération comme un processus impliquant un ensemble d'agents qui interagissent en se communiquant de l'information les uns aux autres durant la résolution d'un problème. La communication peut se faire, par exemple, par des diffusions entre les processeurs ou par l'accès à une source d'information centralisée. Selon ces mêmes auteurs, le comportement des agents impliqués dans un processus coopératif de recherche de solutions est déterminé par plusieurs éléments. Quelques uns de ceux-ci sont la nature du processus de recherche des agents (chaque agent peut avoir un processus semblable ou différent), la nature de l'information partagée, la façon dont elle est partagée et la structure organisationnelle du groupe d'agents.

Toulouse *et al.* (1996) ont cherché à identifier les principales questions qui devraient être posées lors de la conception de processus de recherche coopératifs et montrent que les questions reliées aux communications inter-agents sont de première importance. Selon ces

auteurs, il faut déterminer principalement la nature de l'information à partager, le moment où elle devrait être partagée et entre quels processus elle devrait être partagée.

3.3.3.1 La nature des informations à partager

Quand vient le temps de déterminer quelles informations doivent être partagées, un facteur important à considérer est la quantité d'information à partager parmi toute celle qui a été produite par les processus parallèles. Rendre disponible à tous les processus une grande quantité d'information peut engendrer des coûts de communication ou d'accès à la mémoire très prohibitifs. De plus, il ne sera pas nécessairement intéressant pour les autres processus d'avoir accès à toute l'information disponible. Il est possible que certaines informations produites par un processus n'apportent rien et même nuisent aux autres. L'étude de la pertinence des données à partager est donc importante. Cependant, cette pertinence est généralement difficile à représenter dans un modèle quantitatif. La connaissance du problème et le jugement sont bien souvent les meilleures façons d'évaluer l'intérêt de partager une information plutôt qu'une autre.

3.3.3.2 Le moment où l'information est partagée

Si les processeurs ont un accès sans restriction à l'information partagée, il peut se produire une détérioration de la performance de recherche de solutions. Par exemple, si l'information partagée fait trop souvent partie du processus de décision, il pourra alors se produire une certaine uniformisation des processus de recherche parallèles. Des comportements de recherche trop semblables pourraient impliquer un manque de diversité

et une convergence prématurée de l'algorithme. Par conséquent, il est important de trouver un équilibre entre l'influence de l'information partagée et celle des paramètres locaux. Cela revient non seulement à déterminer la nature de l'information à partager, mais également le moment où elle est accessible par les processus de recherche. Ces conditions d'accès peuvent être déterminées de différentes façons. Elles peuvent être prédéfinies et fixées au début de l'exécution ou varier dynamiquement selon l'évolution de l'algorithme. Un exemple simple de condition d'accès pourrait être de permettre l'accès à la mémoire partagée à toutes les t itérations.

3.3.3.3 Les processus entre lesquels l'information est partagée

Finalement, il faut déterminer entre quels processeurs se feront les échanges d'information. Autrement dit, il faut élaborer les liens logiques qui connectent les processus entre eux et qui contrôlent la propagation de l'information partagée. Cette structure de communication entre les processus doit être définie explicitement lors de la conception de l'algorithme. Par exemple, tous les processeurs pourraient participer à l'échange en diffusant certaines informations à un point de synchronisation donné et reprendre ensuite leur exécution. Une autre stratégie pourrait faire en sorte de permettre à tous les processeurs d'écrire certaines informations dans une mémoire partagée qui ne serait lue que par certains processus sélectionnés. Mentionnons que l'échange d'information peut être synchrone ou asynchrone. S'il est synchrone, les processus impliqués doivent tous avoir atteint un certain point de synchronisation avant que l'échange ne soit effectué et que

l'exécution ne soit reprise par ceux-ci. S'il est asynchrone, l'échange s'effectuera selon la logique interne de chaque processus.

Apporter des réponses aux trois questions présentées ci-haut permet de déterminer les principales règles d'interaction entre les processus et de définir le comportement coopératif de l'algorithme dans la résolution d'un problème.

3.3.3.4 La co-évolution

Le concept de processus coopératifs prend diverses formes dans la littérature. Dans la famille des algorithmes évolutifs, dont font partie les algorithmes génétiques, la coopération peut prendre la forme d'un phénomène de co-évolution. Ce dernier représente l'évolution simultanée de plusieurs entités (par exemple des individus ou des populations) qui cherchent à s'adapter à leur environnement tout en interagissant entre elles (Koza (1991)). Un exemple d'algorithme co-évolutif est l'algorithme génétique parallèle en îles où plusieurs populations indépendantes (une par île/processeur) coopèrent en échangeant des individus qui migrent périodiquement d'une île à une autre (Calégari (1999)). Le choix des individus appelés à migrer peut se faire de différentes façons. Par exemple, il peut être fait de façon probabiliste parmi les individus de la population ou en sélectionnant le meilleur individu de chaque population pour remplacer le pire d'une autre population (Cantú-Paz (1998)). La co-évolution peut également impliquer de la compétition plutôt de la coopération. Le principe sous-jacent est que les entités peuvent devoir lutter entre elles afin d'assurer leur subsistance dans l'environnement où elles évoluent. Juillé et Pollack (1996) montrent qu'un modèle de co-évolution compétitive entre les individus dans un

algorithme génétique peut apporter une certaine diversité dans une population, aider à prévenir la convergence prématurée de l'algorithme et permettre l'émergence de nouveaux comportements menant à de meilleures solutions.

Les concepts de coopération et de co-évolution font l'objet de nombreux travaux dans des domaines variés et une revue plus approfondie de ceux-ci dépasserait le cadre de ce mémoire. Le but recherché dans cette section était de présenter les éléments nécessaires à la compréhension du prochain chapitre et de montrer l'intérêt de la coopération dans la conception de métaheuristiques parallèles.

Afin de compléter cette section sur les métaheuristiques parallèles, quelques travaux de parallélisation des algorithmes présentées à la section 3.2 seront identifiés.

3.3.4 Quelques exemples de métaheuristiques parallèles

On réalise de plus en plus l'apport que peut apporter le parallélisme au domaine des métaheuristiques. En effet, des versions parallèles de métaheuristiques sont proposées en nombre grandissant dans la littérature. Cung *et al.* (2001), Eksioglu *et al.* (2001) ainsi que Crainic et Toulouse (1998) présentent des revues substantielles de parallélisation des principales métaheuristiques. Cette section en présente quelques unes afin de montrer la diversité des approches possibles.

3.3.4.1 Algorithmes génétiques parallèles

Les algorithmes génétiques sont ceux qui ont fait l'objet du plus grand nombre de travaux de parallélisation, notamment en raison de leur nature parallèle fondamentale

(Crainic et Toulouse (1998)). Cantú-Paz (1998) a présenté une revue des principales publications ayant trait aux algorithmes génétiques parallèle. Il distingue trois principales catégories d'algorithmes génétiques parallèles :

1. Parallélisation de forme maître-esclave sur une population unique
2. Parallélisation de grain fin sur une population unique
3. Parallélisation de gros grain sur des populations multiples

Dans le premier modèle, il existe une seule population résidant sur un seul processeur appelé le maître. Ce dernier effectue, de génération en génération, les différents opérateurs génétiques de l'algorithme sur la population et distribue ensuite l'évaluation des individus aux processeurs esclaves.

Dans le deuxième modèle, qui est adapté aux ordinateurs massivement parallèles, les individus de la population sont répartis sur les processeurs, préférablement à raison d'un individu par processeur. Les opérateurs de sélection et de reproduction des individus sont limités à leurs voisinages respectifs. Cependant, comme les voisinages se chevauchent (un individu peut faire partie du voisinage de plusieurs autres individus), un certain degré d'interaction entre tous les individus est possible.

La troisième catégorie, plus sophistiquée et plus populaire, consiste en plusieurs populations qui sont réparties sur les processeurs. Celles-ci peuvent évoluer indépendamment les unes des autres ou échanger occasionnellement des individus. Cet échange facultatif, appelé le phénomène de migration, est contrôlé par différents paramètres et permet généralement une meilleure performance de ce type d'algorithme. Cette classe d'algorithmes génétiques parallèles est cependant plus difficile à comprendre parce que les

effets de la migration sont complexes et difficiles à formaliser. Cette catégorie est aussi appelée entre autres « algorithmes génétiques distribués » ou « algorithmes génétiques parallèles en îles ».

3.3.4.2 Recherche avec tabous parallèle

Crainic *et al.* (1997) ont proposé une taxonomie des algorithmes parallèles de recherche avec tabous qui les regroupent selon trois dimensions : la cardinalité du contrôle de la recherche parallèle, le type de contrôle et de communication et la différenciation de la recherche.

La première dimension réfère au nombre de processeurs qui contrôlent la recherche. Cette dernière peut se faire par un ou plusieurs processeurs. Dans le premier cas, le processus de recherche est le même que celui de l'algorithme séquentiel. Un processeur maître exécute l'algorithme en déléguant certains calculs aux autres processeurs. La répartition des tâches peut impliquer entre autres la distribution des calculs coûteux en temps ou l'exploration parallèle du voisinage. Dans le deuxième cas, le contrôle de la recherche est partagé entre deux ou plusieurs processeurs, ce qui correspond à une recherche à processus multiples. Chaque processus est en charge de sa propre recherche et des communications avec les autres processus.

La deuxième dimension est basée sur le type et la flexibilité du contrôle de la recherche. Elle tient compte de l'organisation, de la synchronisation et de la hiérarchie de la communication ainsi que de la façon dont l'information est traitée et partagée entre les processeurs. Cette dimension est composée de quatre degrés : « *Rigid synchronization* »,

« *knowledge synchronization* », « *collegial* » et « *knowledge collegial* ». En les associant aux deux niveaux de cardinalité déjà mentionnés, ceux-ci définissent les stratégies de parallélisation relatives au traitement des processus et de l'information.

La troisième dimension correspond à la stratégie de différentiation de la recherche. Elle réfère au nombre de solutions initiales distinctes et au nombre de stratégies de recherche différentes (choix des paramètres, gestion de la liste tabou, etc.) utilisés par l'algorithme. On peut donc retrouver quatre cas de stratégies de recherche :

1. SPSS (***Single Point Single Strategy***) : c'est le cas le plus simple qui ne permet généralement que des parallélisations de bas niveau.
2. SPDS (***Single Point Different Strategy***) : plusieurs processeurs effectuent des recherches taboues différentes mais en partant du même point initial.
3. MPSS (***Multiple Points Single Strategy***) : chaque processeur démarre sa recherche à partir d'un point différent de l'espace de solutions, mais une seule stratégie de recherche commune est utilisée par tous les processeurs.
4. MPDS (***Multiple Points Different Strategies***) : c'est la classe la plus générale et elle considère les autres classes comme des cas particuliers.

Ces auteurs présentent, dans ce même travail, plusieurs implémentations parallèles d'algorithmes de recherche avec tabous retrouvés dans la littérature. En les mettant en relation avec cette taxonomie, ils montrent la diversité des approches possibles et l'utilité d'une vision globale des stratégies de parallélisation de cette méthode.

3.3.4.3 Optimisation par Colonies de Fourmis parallèle

L'Optimisation par Colonies de Fourmis étant une métaheuristiques relativement récente, peu de travaux relatifs à sa parallélisation sont retrouvés dans la littérature.

Bullnheimer *et al.* (1998) ont proposé deux versions parallèles de l'algorithme « *Ant System* » sur une architecture de type MIMD-DM : une version synchrone et une version partiellement asynchrone.

La version parallèle synchrone correspond à une parallélisation de bas niveau telle que définie à la section 3.3.1. Il s'agit ici d'accélérer l'algorithme en effectuant la génération des fourmis en parallèle selon une approche maître-esclave. À chaque cycle, le maître diffuse la matrice de trace (phéromone) aux esclaves, ceux-ci calculent leurs tournées et les résultats sont retournés au maître. Une synchronisation des processus est effectuée à chaque itération de l'algorithme. Cette approche sera discutée plus en détails au chapitre suivant.

Afin de réduire les coûts reliés aux communications de la version synchrone de cet algorithme, ces mêmes auteurs ont développé une version partiellement asynchrone où un certain nombre d'itérations de l'algorithme séquentiel sont effectués de façon indépendante sur différents processeurs. Après ces itérations « locales », une synchronisation globale est effectuée par le maître. Après une étude comparative des performances des deux versions parallèles, les auteurs concluent que l'approche partiellement synchrone est préférable en raison de la réduction considérable de la fréquence des communications.

Stützle (1998) présente deux stratégies de parallélisation sur une architecture de type MIMD : l'exécution parallèle de plusieurs copies du même algorithme et l'accélération

d'une seule exécution de cet algorithme. Dans le premier cas, on peut appliquer des paramètres de recherche identiques ou différents à chaque exécution parallèle. Même si l'auteur n'a pas obtenu de différences significatives entre les deux approches dans l'application du « *Max-Min Ant System* » au problème de voyageur de commerce, il mentionne qu'une amélioration de la performance pourrait être rencontrée dans le cadre d'autres problèmes. Le deuxième cas, c'est-à-dire l'accélération d'une seule exécution de l'algorithme, est semblable à la version synchrone de Bullnheimer *et al.* (1998) à la différence qu'il l'étend en considérant les algorithmes de fourmis qui utilisent une procédure de recherche locale. Une parallélisation efficace peut alors être obtenue en appliquant la recherche locale en parallèle sur les solutions préalablement générées.

Talbi *et al.* (1999) ont également proposé un modèle parallèle de colonies de fourmis semblable au modèle synchrone de Bullnheimer *et al.* (1998). De leur côté, ils y ajoutent une recherche taboue comme méthode de recherche locale. Cet algorithme est appliqué avec succès au problème d'affectation quadratique.

Michels *et al.* (1999) ont proposé une approche de colonies de fourmis en modèle d'îles où les processeurs contiennent une colonie de fourmis distincte et échangent leur meilleure solution locale à chaque nombre fixe d'itérations. Quand une colonie reçoit une solution meilleure que celle qu'il a trouvée jusqu'à maintenant, la solution reçue devient la meilleure solution trouvée. Cette nouvelle information influence la colonie parce que la mise à jour de la trace est toujours effectuée selon la meilleure solution trouvée.

Middendorf *et al.* (2000) ont étudié quatre stratégies d'échange d'information entre des colonies de fourmis multiples qui sont les suivantes :

1. L'échange de la meilleure solution globale : à chaque étape d'échange d'information, la meilleure solution globale est diffusée à toutes les colonies où elle devient la meilleure solution locale.
2. L'échange circulaire des meilleures solutions locales : un voisinage virtuel est établi entre les colonies de façon à ce qu'elles forment un anneau. À chaque étape d'échange d'information, chaque colonie envoie sa meilleure solution locale à la colonie qui le succède dans l'anneau.
3. L'échange circulaire de migrateurs : comme dans la deuxième stratégie, les processeurs forment un anneau virtuel. Dans un échange d'information, chaque colonie compare ses *m* meilleures fourmis avec les *m* meilleures de son successeur. Les *m* meilleures fourmis de ces deux groupes mettent alors à jour la matrice de trace.
4. L'échange circulaire des meilleures solutions locales et des migrateurs : combinaison des stratégies 2 et 3.

Ils ont montré qu'il peut être avantageux pour les colonies de ne pas échanger une trop grande quantité d'information et de ne le faire qu'à une fréquence limitée. Abandonnant alors l'idée d'échanger des matrices de trace complètes, ils ont basé leurs stratégies sur l'échange d'une seule solution à la fois. Ainsi, ils ont obtenu des implémentations parallèles efficaces.

3.3.4.4 Approches hybrides

Chaque métaheuristique possède ses propres caractéristiques et sa propre manière d'effectuer la recherche de solutions. Par conséquent, il peut être intéressant de faire coopérer plusieurs métaheuristiques différentes afin de créer de nouveaux comportements

de recherche. À ce sujet, Bachelet *et al.* (1998) ont déterminé trois principales formes d'algorithmes hybrides :

1. Hybride séquentiel où deux algorithmes s'exécutent l'un après l'autre, les résultats fournis par le premier étant les solutions initiales du deuxième.
2. Hybride parallèle synchrone où un algorithme de recherche est utilisé à la place d'un opérateur. Un exemple de ce type est de remplacer l'opérateur de mutation d'un algorithme génétique par une recherche taboue.
3. Hybride parallèle asynchrone où plusieurs algorithmes de recherche travaillent concurremment et s'échangent des informations

Ces auteurs, par l'utilisation avec succès d'un algorithme hybride parallèle asynchrone combinant un algorithme génétique et un algorithme de recherche avec tabou, montrent une autre facette de la puissance du parallélisme qui est la possibilité de collaboration de plusieurs méthodes de recherche en parallèle.

3.4 Objectifs de la recherche

Les chapitres 2 et 3 ont mis en relation différentes notions du parallélisme et des métaheuristiques. L'objectif général de ce travail de recherche est d'étudier certaines de ces relations afin de mieux comprendre les bénéfices que peuvent apporter les métaheuristiques parallèles dans la résolution de problèmes d'optimisation combinatoire. De cette façon, de nouvelles stratégies de résolution pourront être développées et appliquées à des problèmes réels. La démarche utilisée dans ce travail consiste à sélectionner une métaheuristique, d'y mettre en œuvre certaines stratégies de parallélisation et d'appliquer l'algorithme résultant à la résolution d'un problème d'ordonnancement

industriel. La parallélisation étant étudiée dans le cadre d'une architecture réelle à mémoire partagée et d'un environnement de programmation novateur, l'approche n'a pas été exploitée dans la littérature à notre connaissance et présente des éléments nouveaux.

Les objectifs spécifiques de cette recherche sont :

1. de développer une stratégie à marche simple (stratégie de parallélisation interne) pour paralléliser la métaheuristique d'Optimisation par Colonies de Fourmis sur une architecture parallèle à mémoire partagée et d'appliquer l'algorithme résultant à la résolution d'un problème d'optimisation combinatoire réel, c'est-à-dire un problème d'ordonnancement industriel rencontré dans une entreprise de fabrication d'aluminium. Pour évaluer l'efficacité de cette stratégie, les mesures standard de performance des algorithmes parallèles présentées à la section 2.3.3 seront utilisées.
2. d'introduire une stratégie de processus coopératifs à la métaheuristique d'Optimisation par Colonies de Fourmis afin d'améliorer la qualité des solutions trouvées au problème d'ordonnancement industriel. Pour en vérifier la performance, les résultats de cette stratégie seront évalués en relation avec ceux obtenus par une version séquentielle du même algorithme.

3.5 Conclusion

Dans les chapitres 2 et 3, les notions fondamentales du parallélisme et l'état de l'art des métaheuristiques parallèles ont été présentés. Les objectifs de recherche de ce travail ont également été définis. La suite de ce travail sera donc consacrée à la parallélisation d'un algorithme d'Optimisation par Colonies de Fourmis pour la résolution d'un problème d'ordonnancement industriel. La performance de certaines stratégies de parallélisation de métaheuristiques sera donc étudiée dans ce contexte réel.

CHAPITRE 4

PARALLÉLISATION D'UN ALGORITHME D'OPTIMISATION PAR COLONIES DE FOURMIS DANS UN CONTEXTE D'ORDONNANCEMENT INDUSTRIEL

4.1 Introduction

Les problèmes d'ordonnancement sont des problèmes d'optimisation combinatoire ayant fait l'objet de nombreux travaux dans le domaine de la recherche opérationnelle. Une des principales raisons justifiant ces efforts est l'intérêt d'appliquer les méthodes de résolution découlant de ces travaux à des problèmes réels d'ordonnancement rencontrés dans des contextes industriels variés. Comme les problèmes d'ordonnancement pratiques sont souvent plus complexes et plus contraignants que les problèmes théoriques classiques, l'intérêt de développer des méthodes de résolution à la fois rapides, efficaces et flexibles est d'autant plus justifié.

Les métaheuristiques se sont avérées des stratégies de résolution particulièrement performantes autant pour les problèmes d'optimisation combinatoires en général que pour les problèmes d'ordonnancement théoriques et pratiques. Par exemple, Gravel *et al.* (2002) ont présenté une méthode efficace, basée sur la métaheuristique d'Optimisation par Colonies de Fourmis, pour la résolution d'un problème réel d'ordonnancement rencontré dans l'industrie de production de l'aluminium. Cette méthode a été implantée dans un logiciel utilisé dans plusieurs usines de l'entreprise.

Toutefois, même si cet algorithme s'est avéré une solution efficace et viable à ce problème, la quantité de temps et de ressources de calcul qu'il nécessite a toujours intérêt à être réduite davantage dans un contexte d'application. De plus, comme l'algorithme s'avère une méthode approximative qui ne garantit pas l'obtention de la solution optimale, l'amélioration de la qualité de l'optimisation présente également un intérêt. Ces deux

constatations s'avèrent particulièrement vraies à mesure que la taille du problème augmente.

L'algorithme d'Optimisation par Colonies de Fourmis est un processus itératif de génération et d'évaluation de solutions. Ces deux procédures représentent à elles seules une très grosse proportion du temps de calcul et de la quantité de mémoire nécessaires à l'exécution de l'algorithme. De plus, pour certaines versions de l'OCF, elles présentent un faible degré de dépendance. En d'autres mots, le traitement d'une solution donnée n'est pas dépendant du résultat du traitement d'une autre solution durant une itération de l'algorithme. La structure de ce dernier le rend donc propice à une exécution parallèle efficace. De plus, comme il a été expliqué au chapitre précédent, le parallélisme permet le développement de mécanismes d'apprentissage pouvant améliorer de façon appréciable la qualité de l'optimisation offerte par l'OCF. Pour ces raisons, il s'avère intéressant d'étudier différentes possibilités de parallélisation de l'OCF pour la résolution du problème d'ordonnancement industriel décrit par Gravel *et al.* (2002).

Ce chapitre présente la démarche et les résultats de l'étude effectuée. En premier lieu, le problème d'ordonnancement industriel sera décrit sommairement pour assurer la compréhension du lecteur. Par la suite, les travaux pertinents pour résoudre ce problème, de façon séquentielle à l'aide de la métaheuristique OCF, seront présentés. Finalement, les deux dernières sections de ce chapitre seront consacrées au développement de deux stratégies de parallélisation de cette métaheuristique et à la présentation des résultats obtenus.

4.2 Le problème d'ordonnancement industriel

Le problème traité dans ce travail est un problème réel d'ordonnancement industriel rencontré dans une aluminerie. Pour une période de production donnée, un carnet de commandes de lingots possédant différentes caractéristiques doit être produit en usine sur une machine à coulée horizontale telle qu'illustrée à la Figure 4.1. Lors de la production d'une commande, la machine à coulée est alimentée par les fours de préparation. Ceux-ci sont utilisés pour établir le mélange désiré en ajoutant différents ingrédients d'alliage à de l'aluminium pur de manière à obtenir les caractéristiques spécifiques du produit. Le moule utilisé permet de donner la dimension voulue au lingot et ce dernier est découpé, à l'aide d'une scie, en pièces de la longueur désirée à la fin du processus de production. La machine à coulée traite les commandes en série, l'une à la suite de l'autre.

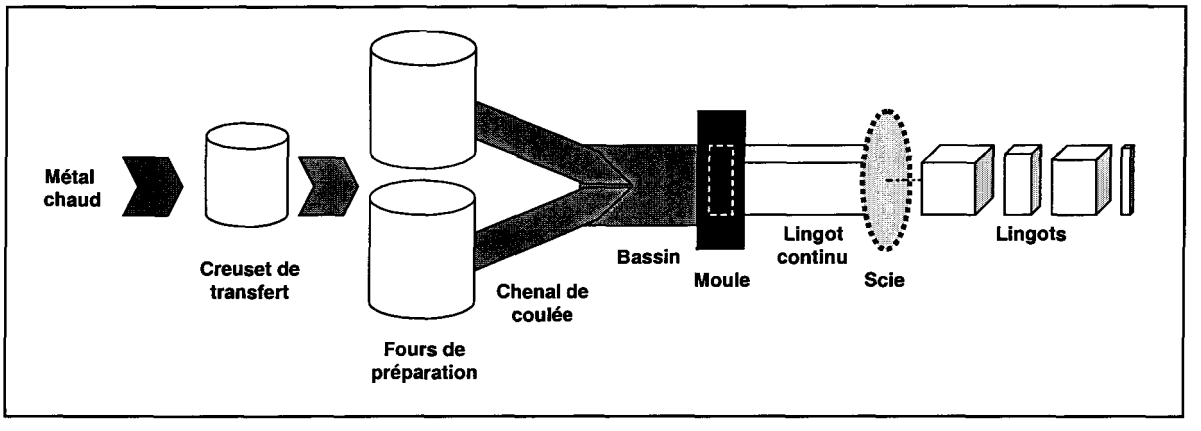

Figure 4.1 Le processus de coulée horizontale

La fabrication successive de deux commandes d'alliages différents peut, dans certains cas, entraîner des procédures de drainage et de nettoyage des fours de préparation. De plus,

la fabrication successive de deux commandes de lingots de dimensions différentes nécessite une procédure de changement de moule. Ces procédures, appelées mise-en-course ou réglages, représentent des pertes de capacité de production et doivent être autant que possible évitées. De plus, ces deux types de mises-en-course sont dépendants de la séquence de production.

Le processus de production des commandes doit être planifié de façon à réduire le plus possible non seulement les pertes de capacité du système de production, mais aussi le retard des commandes (chaque commande doit être livrée à son client respectif à une certaine date). Un troisième objectif est également visé et cherche à maximiser la capacité de livraison de l'entreprise.

Ce problème consiste donc à ordonnancer un carnet de n commandes, c'est-à-dire à déterminer l'ordre dans lequel les commandes seront traitées sur la machine à coulée de façon à minimiser les trois objectifs que sont la perte de capacité de la machine à couler, le retard total de l'ensemble des commandes et la perte de capacité de livraison. Cette situation industrielle a été associée à un problème théorique d'ordonnancement de n commandes sur une machine unique avec temps de réglages dépendants de la séquence. Du et Leung (1990) ont démontré que ce problème classique est NP-difficile lorsque les temps de réglages sont indépendants de la séquence. Le problème industriel s'avère donc davantage complexe en raison des temps de réglages dépendants de la séquence et des différentes contraintes technologiques qui lui sont spécifiques.

Plusieurs travaux (Gagné (2001), Gagné *et al.* (2001a), Gravel *et al.* (2002)), incluant une thèse de doctorat, ont mené à l'élaboration d'une méthode de résolution performante

pour ce problème multi-objectifs. Ce travail se limitera toutefois à l'optimisation uni-objectif et nous présentons, à la prochaine section, les versions spécifiques de l'Optimisation par Colonies de Fourmis retenues pour fins de parallélisation.

4.3 Description des versions séquentielles de l'OCF

La section 4.3.1 explique brièvement comment l'Optimisation par Colonies de Fourmis a été adaptée pour le traitement du problème d'ordonnancement industriel. Cet algorithme, noté OCF_{S1} dans la suite de ce travail, servira à tester la première stratégie de parallélisation. Plusieurs améliorations ont été apportées à OCF_{S1} et les plus importantes d'entre elles seront présentées à la section 4.3.2. Cette nouvelle version de l'OCF, notée OCF_{S2}, servira de base de comparaison pour tester la deuxième stratégie de parallélisation. L'utilisation, dans ce travail, de versions différentes de l'OCF séquentiel s'explique par le fait que les travaux sur la version améliorée de l'OCF séquentiel (OCF_{S2}) n'étaient pas complétés au moment où ce travail a débuté. De plus, nous avons voulu comparer la performance de l'algorithme parallèle développé dans la deuxième partie de ce travail avec l'algorithme séquentiel résultant des travaux les plus récents.

4.3.1 Algorithme de base (OCF_{S1})

Dans ce problème d'ordonnancement, nous devons déterminer la séquence de traitement d'un ensemble de commandes comportant des temps de réglages dépendants de la séquence. La formulation du problème est basée sur celle du problème du voyageur de commerce (VC). Chaque commande à être traitée est représentée par une « ville » dans un

réseau VC. Quand une fourmi se déplace de la ville *i* à la ville *j*, elle laisse une trace (dépôt d'une certaine quantité de phéromone) sur l'arc (*ij*). La trace enregistre l'information sur l'utilisation de l'arc (*ij*) de sorte que plus cette utilisation a été importante dans le passé, plus élevée est la probabilité que cet arc soit utilisé à nouveau dans le futur. Pour le problème d'ordonnancement, l'information contenue dans la trace de phéromone est basée sur la fréquence où les fourmis choisissent de traiter la commande *i* suivie de la commande *j*. La façon dont la trace est initialisée et modifiée en cours d'exécution sera expliquée plus loin.

Au temps *t*, à partir d'une séquence partielle de commandes déjà construite, chaque fourmi *k* choisit la prochaine commande à ajouter à la séquence en utilisant une règle probabiliste basée sur un compromis entre la *visibilité* (η_{ij}) et l'*intensité de la trace de phéromone* ($\tau_{ij}(t)$). Cette probabilité de choisir l'arc (*ij*), notée $p_{ij}^{k}(t)$, est calculée de la façon suivante :

$$p_{ij}^{k}(t) = \frac{\left[\tau_{ij}(t)\right]^{\alpha} \cdot \left[\eta_{ij}\right]^{\beta}}{\sum_{\ell \notin Tabou_k} \left[\tau_{i\ell}(t)\right]^{\alpha} \cdot \left[\eta_{i\ell}\right]^{\beta}} \qquad \textbf{(Équation 4)}$$

Dans le problème du VC, la visibilité est définie par ($1/d_{ij}$) qui prend en considération la distance entre les paires de villes. Pour le problème d'ordonnancement, la matrice *D* contiendra, pour chaque paire de commandes, de l'information particulière sur chacun des objectifs à minimiser de façon à guider la recherche. À l'initialisation de l'algorithme,

l'intensité de la trace pour toutes les paires de commandes (*ij*) est fixée à une petite valeur positive τ_0. Les paramètres α et β sont utilisés afin de déterminer l'importance relative de l'intensité de la trace et de la visibilité dans la construction d'une séquence. De plus, une liste taboue est maintenue pour garantir qu'une commande ayant déjà été affectée à la séquence en cours de construction ne soit pas sélectionnée une autre fois. Chaque fourmi *k* va donc posséder sa propre liste tabou, notée « $tabou_k$ » qui gardera en mémoire les commandes déjà sélectionnées.

Durant une itération de l'algorithme, plusieurs fourmis construisent à tour de rôle une séquence de commandes. Un cycle (*NC*) est complété au moment où la dernière des *m* fourmis a terminé sa construction. La version de l'algorithme présentée ici effectue une mise à jour de l'intensité de la trace à la fin de chaque cycle selon l'évaluation des solutions trouvées durant ce cycle. Soit $L_k^{h'}$ l'évaluation sur l'objectif à optimiser le plus important (*h*') de la solution trouvée par la $k^{\text{ième}}$ fourmi. La contribution sur la mise à jour de la trace de la fourmi *k* est alors calculée de la façon suivante : $\Delta\tau_{ij}^k(t) = Q/ L_k^{h'}(t)$, où Q est une constante. La mise à jour de la trace est aussi influencée par un facteur d'évaporation $(1-\rho)$ qui diminue la quantité de phéromone présente sur tous les arcs (*ij*) afin de prévenir une convergence prématurée de l'algorithme.

Le lecteur peut consulter Gravel *et al.* (2002) pour une description plus détaillée des modifications apportées à l'OCF original (Colorni *et al.* (1991), Dorigo (1992)) afin de l'adapter au problème industriel. Le traitement multi-objectifs est réalisé, entre autres, au moyen d'une structure de données arborescente proposée par Finkel & Bentley (1974) et appelée quadtree dans laquelle les solutions non dominées trouvées par la métaheuristique

sont stockées. Le traitement multi-objectifs étant ignoré dans ce travail, cet aspect ne sera pas détaillé davantage et le quadtree sera considéré simplement comme une structure de stockage de solutions.

Pour une meilleure compréhension de la section 4.4.1, la Figure 4.2 présente l'algorithme OCF_{S1} d'une façon moins formelle, plus simple et plus représentative de son implantation informatique.

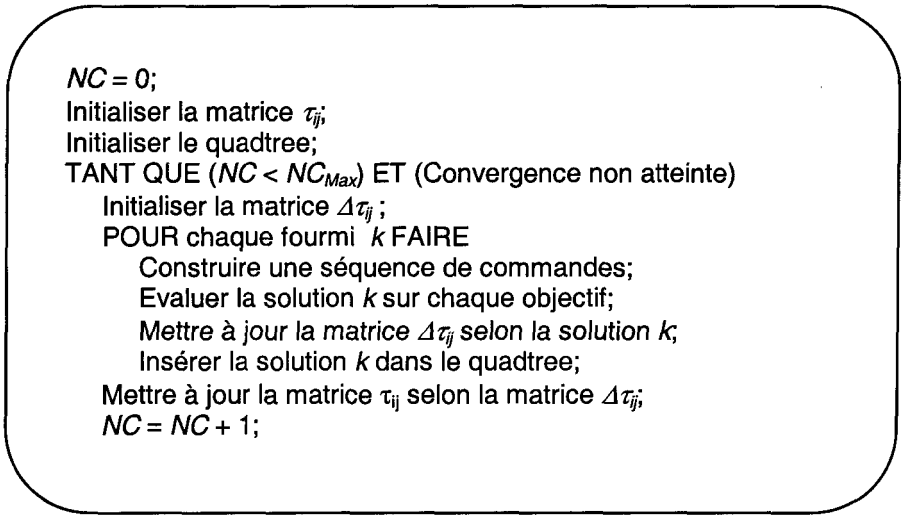

Figure 4.2 Version séquentielle de l'Optimisation par Colonies de Fourmis (OCF_{S1}).

4.3.2 Version améliorée de l'algorithme (OCF_{S2})

Suite à l'élaboration de la version OCF_{S1}, plusieurs améliorations lui ont été apportées afin d'en améliorer la performance. Ces améliorations comprennent notamment la modification de la règle de transition (qui permet de sélectionner la prochaine commande à être affectée à la séquence en construction), l'ajout d'une procédure de recherche locale, d'une liste de candidats et d'une fonction d'anticipation. Ces éléments n'ont aucune

incidence sur les stratégies de parallélisation étudiées dans ce travail et ne seront pas détaillées dans cette section. Pour de plus amples explications concernant ces modifications, le lecteur peut consulter Gagné *et al.* (2001a) et Gagné *et al.* (2001b).

Quelques éléments ajoutés à l'algorithme doivent cependant être mentionnés et précisés étant donné leur influence sur certaines parties à venir du présent travail. Deux de ceux-ci sont les mises à jour globale et locale de la trace. Ces deux procédures remplacent la mise à jour de la trace de l'algorithme de la Figure 4.1 et entraînent la disparition de la matrice $\Delta\tau_{ij}$. La mise à jour globale de la trace s'effectue toujours à la fin de chaque cycle, mais seulement à partir de la meilleure solution trouvée durant le cycle qui vient de se terminer. De son côté, la mise à jour locale de la trace s'effectue à chaque fois qu'une fourmi sélectionne une commande à être ajoutée à la séquence en cours de construction. Cette procédure diminue légèrement la quantité de trace de l'arc qui vient d'être sélectionné afin d'éviter que toutes les fourmis empruntent le même chemin durant le cycle. La Figure 4.3 illustre le fonctionnement de la nouvelle version de l'algorithme, notée OCF_{S2}, auquel ont été ajoutées les procédures de mise à jour locale et globale de la trace. Les variables m et n représentent respectivement le nombre de fourmis utilisé et le nombre de commandes à ordonnancer.

```
NC = 0;
Initialiser la matrice τij;
Initialiser le quadtree;
TANT QUE (NC < NCMax) ET (Convergence non atteinte)
   POUR i = 1 à n FAIRE
      POUR k = 1 à m FAIRE
         Sélectionner la commande j à être ajoutée à la séquence selon p_ij^k(t);
         Effectuer la mise à jour locale de la trace selon la paire de commandes (i, j) ;
   POUR chaque fourmi k FAIRE
      Evaluer la solution k sur chaque objectif;
      Insérer la solution k dans le quadtree;
   Effectuer la mise à jour globale de la trace selon la meilleure solution du cycle;
   NC = NC + 1;
```

Figure 4.3 Version séquentielle améliorée de l'Optimisation par Colonies de Fourmis (OCF_{S2}).

Un dernier élément important à mentionner est le remplacement de la matrice D, comportant des informations fusionnées sur l'ensemble des objectifs à optimiser, par plusieurs matrices tenant compte individuellement de chacun des objectifs. La matrice S inclut les informations pertinentes aux mises-en-course, la matrice M permet de tenir compte des dates d'échéance des commandes et la matrice C fournit de l'information sur les modes de livraison des différentes commandes. L'équation (5) remplace alors l'équation (4) pour la sélection des commandes dans la construction d'une solution par une fourmi.

$$p_{ij}^{k}(t) = \frac{\left[\tau_{ij}(t)\right]^{\alpha} \cdot \left[\frac{1}{s_{ij}}\right]^{\beta} \cdot \left[\frac{1}{m_{ij}}\right]^{\delta} \cdot \left[\frac{1}{c_{ij}}\right]^{\lambda} \cdot \left[\frac{1}{\overline{B}_{ij}}\right]^{\phi}}{\sum_{l \notin Tabou^{k}} \left[\tau_{il}(t)\right]^{\alpha} \cdot \left[\frac{1}{s_{il}}\right]^{\beta} \cdot \left[\frac{1}{m_{il}}\right]^{\delta} \cdot \left[\frac{1}{c_{il}}\right]^{\lambda} \cdot \left[\frac{1}{\overline{B}_{il}}\right]^{\phi}} \qquad \textbf{(Équation 5)}$$

Les paramètres β, δ et λ sont associés respectivement aux objectifs de capacité, de retard et de livraison et permettront de modifier l'importance relative accordée à ceux-ci dans la construction des solutions. Pour plus de détails, le lecteur peut consulter Gagné *et al.* (2001a).

4.4 Parallélisation de l'OCF

Ce travail représente une étude des stratégies de parallélisation des métaheuristiques sur une architecture à mémoire partagée, de même que l'application de ces stratégies à la résolution d'un problème d'ordonnancement industriel. Les deux versions de la métaheuristique OCF (OCF_{S1} et OCF_{S2}) présentées à la section précédente représentent les algorithmes séquentiels sur lesquels seront appliquées les stratégies de parallélisation.

La disponibilité d'OpenMP (1998), outil novateur permettant le développement de programmes sur des ordinateurs parallèles à mémoire partagée, a facilité l'implantation des stratégies de parallélisation développées dans ce travail. OpenMP se veut un regroupement de directives de compilation, de bibliothèques de fonctions et de variables d'environnement qui permet d'exprimer le parallélisme à mémoire partagée dans un programme écrit en langage C, C++ ou Fortran. Il étend ces langages séquentiels par des instructions permettant, entre autres, de répartir les opérations sur plusieurs processeurs en définissant des régions parallèles, de contrôler l'exécution concurrente par des barrières de synchronisation et des zones critiques, ainsi que de gérer le partage et la privatisation des données. Le logiciel d'ordonnancement, écrit en language C, a été parallélisé à l'aide de

directives OpenMP et exécuté sur un ordinateur parallèle de type Origin2000 fabriqué par Silicon Graphics.

Cette section se subdivise en deux sous-sections, chacune d'elles correspondant à l'étude d'une stratégie de parallélisation spécifique appliquée à une version particulière de l'OCF séquentiel. La section 4.4.1 sera consacrée au développement d'une stratégie de parallélisation interne, c'est-à-dire à la répartition sur plusieurs processeurs des calculs effectués par l'OCF séquentiel afin d'en accélérer l'exécution. Cette stratégie sera appliquée à la version séquentielle OCF_{S1}. Les résultats de ce travail ont déjà été publiés dans Delisle *et al.* (2001).

Ensuite, la section 4.4.2 présentera le développement d'une stratégie de parallélisation à processus coopératifs sur une architecture à mémoire partagée. Plus précisément, différentes stratégies de coopération seront adaptées au contexte présent et intégrées à la version OCF_{S2} dans le but d'en améliorer la qualité de recherche de solutions. Les résultats de ce travail ont été présentés dans un « workshop sur la résolution parallèle de problèmes NP-complets » dans le cadre du congrès RENPAR'14 tenu en Tunisie en avril 2002.

4.4.1 Parallélisation interne

Comme il a été mentionné au chapitre précédent, la métaheuristique OCF est relativement récente et peu de travaux relatifs à sa parallélisation sont présents dans la littérature. De plus, il semble que ces travaux n'aient été effectués que sur des architectures parallèles à mémoires distribuées de type MIMD. Ce phénomène pourrait s'expliquer par le fait que pendant plusieurs années, on a considéré que les architectures à mémoire

partagée n'étaient performantes que sur le plan théorique et que la construction de machines parallèles efficaces de ce type n'était pas envisageable à des coûts raisonnables. Ce travail est effectué sur une architecture à mémoire partagée, ce qui présente des enjeux différents au niveau de la conception de l'algorithme parallèle et de son développement informatique.

4.4.1.1 Parallélisation de l'OCF sur une architecture à mémoire partagée

Bullnheimer *et al.* (1998) ont proposé une version synchrone de l'algorithme « *Ant System* » pour une architecture à passage de messages. Les auteurs constatent le coût prohibitif des communications de cet algorithme et l'existence d'une procédure de synchronisation non négligeable, ces facteurs limitant le gain en temps de calcul pouvant être obtenu par l'exécution parallèle. Dans cette section, il est démontré que la parallélisation de ce même algorithme sur un modèle d'architecture à mémoire partagée permet une meilleure efficacité en réduisant les coûts de communication, mais que la procédure de synchronisation ne peut être évitée et ce, peu importe le modèle d'architecture sous-jacent.

Il est important de noter que le but recherché dans les travaux présentés dans cette section est de réduire le temps d'exécution de l'algorithme sans modifier son comportement. L'amélioration de la qualité des solutions trouvées par l'OCF en lui appliquant des mécanismes parallèles de recherche fera l'objet de la section 4.4.2.

4.4.1.1.1 Le parallélisme « naturel » de l'ACO

Comme il peut être constaté à la Figure 4.1, la boucle « pour » est la partie principale de l'algorithme et la source de sa complexité (l'algorithme OCF standard s'exécute en $O(n^3)$). En effet, la génération d'une solution s'effectue en $O(n^2)$ (où n est le nombre de commandes) et comme nous sommes dans un environnement d'ordonnancement réel, la fonction d'évaluation doit simuler le processus industriel pour chaque solution. Cette procédure nécessite un temps de calcul considérablement plus élevé que, par exemple, l'évaluation d'une tournée dans le VC. Notons également que ces deux opérations sont indépendantes pour chaque fourmi d'un cycle donné et peuvent donc être facilement parallélisées.

4.4.1.1.2 Modèle à passage de messages vs. modèle à mémoire partagée

La Figure 4.4 illustre le comportement d'un algorithme OCF parallèle basé sur le « *Ant System* » parallèle synchrone dans un modèle à passage de messages, tel que développé par Bullnheimer *et al.* (1998). Au début de l'algorithme, le processus maître initialise l'information, crée k processus (un pour chaque fourmi) et diffuse (envoie à tous les processeurs) l'information. Au début d'un cycle, la matrice τ_{ij} (la trace de phéromone) est envoyée à chaque processeur et les calculs (génération et évaluation des solutions) sont effectués en parallèle. Ensuite, les solutions et leurs évaluations sont retournées au maître, la matrice τ_{ij} est mise à jour et un nouveau cycle commence par la diffusion de la matrice τ_{ij} modifiée.

Comme l'ont montré les auteurs de cet algorithme, si les considérations relatives aux communications ne sont pas prises en compte et en assumant qu'il y ait un nombre suffisant

de processeurs pour affecter une fourmi à chaque élément de calcul (nombre de processeurs ≥ nombre de fourmis), cette stratégie de parallélisation implique une accélération optimale. Cependant, l'influence des communications ne peut être négligée étant donné la perte considérable d'efficacité entraînée par le grand nombre d'opérations de communication (empaquetage et dépaquetage des messages, envoi et réception des paquets, temps d'attentes, etc.) nécessaire à l'algorithme.

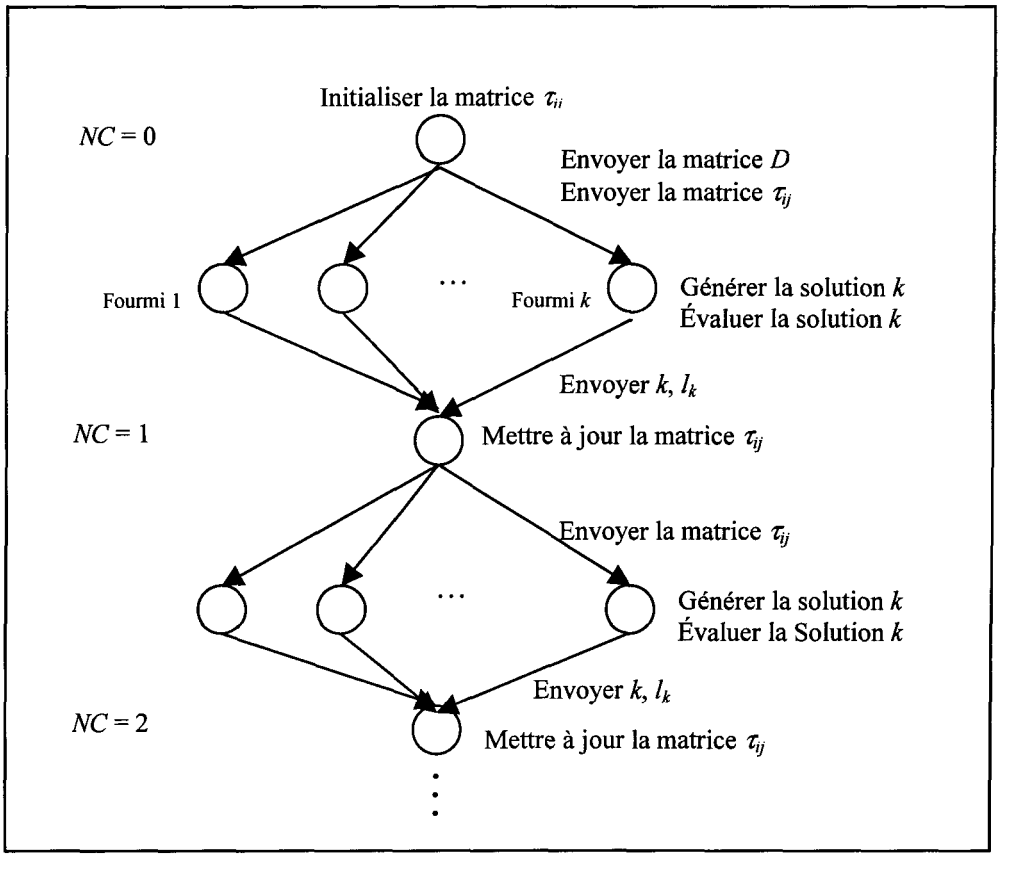

Figure 4.4 OCF parallèle dans un modèle à passage de messages

Sur une architecture à mémoire partagée, l'échange d'information entre processeurs se fait par l'intermédiaire de la mémoire et les procédures explicites de communication sont éliminées. Il est donc intéressant d'expérimenter une implémentation parallèle de l'OCF sur une machine parallèle à mémoire partagée pour résoudre le problème d'ordonnancement industriel et en étudier les performances obtenues.

Il est important de rappeler que dans un modèle théorique comme la PRAM, les accès par les processeurs à la mémoire partagée se font en temps constant et ne sont pas prises en considération dans l'étude de la performance des algorithmes. Il est cependant clair qu'en réalité, le système d'exploitation d'un ordinateur parallèle à mémoire partagée doit gérer les accès à la mémoire. De plus, la machine parallèle utilisée dans le cadre de ce travail, au même titre que la plupart des machines récentes de ce type, fonctionne selon un principe de mémoire virtuellement partagée et non réellement partagée. La gestion de la mémoire peut donc en réalité prendre un temps plus ou moins prohibitif qui doit être pris en compte.

Ce travail est basé sur le principe selon lequel un modèle théorique à mémoire partagée est plus performant que celui à mémoire distribuée, de même que sur l'hypothèse que la machine réelle gère la mémoire partagée de façon efficace (l'étude du fonctionnement réel des machines parallèles dépasse le cadre de ce travail). Dans les sections qui suivent, la transposition d'un OCF séquentiel en une version parallèle dans un modèle à mémoire partagée sera considérée d'un point de vue algorithmique, donc indépendant de la machine utilisée. Les considérations pratiques résultant de l'implantation de cet algorithme en un programme s'exécutant sur une machine parallèle réelle seront par la suite considérés.

4.4.1.1.3 Synchronisation pour la mise à jour de la matrice τ_{ij}

À la fin de chaque cycle, la matrice τ_{ij} doit être mise à jour pour les fourmis du prochain cycle. Même si cette version de l'OCF parallèle ne nécessite pas de communications explicites avant la procédure de mise à jour, le processus maître doit tout de même attendre que les processus subordonnés (les fourmis) aient calculé et évalué leurs séquences de commandes respectives. Cette barrière de synchronisation est donc indépendante du modèle utilisé et ne peut être évitée sans altérer le comportement original de l'algorithme. L'étude des modifications qui pourraient être faites à la fréquence de ces mises à jour pour atteindre une meilleure efficacité sans rien perdre sur la performance du processus de recherche de solutions présente un certain intérêt et sera envisagée dans des travaux futurs.

4.4.1.1.4 La répartition de la charge entre les processeurs

Une première étape dans le processus de parallélisation pourrait être d'affecter naïvement la génération et l'évaluation de chaque fourmi à un processeur différent. Les mises à jour de la matrice $\Delta\tau_{ij}$ et du quadtree demeureraient donc à l'extérieur de la région parallèle pour éviter les conflits d'accès en écriture à la mémoire partagée. Une autre façon de traiter ces mises à jour serait de les inclure dans la région parallèle, mais dans une section critique accessible par un processeur seulement à la fois. Cependant, les synchronisations nécessaires dans les deux cas entraîneraient une perte considérable d'efficacité.

Cette situation fait également en sorte que le nombre de fourmis choisi, qui est un paramètre de l'algorithme, est limité par le nombre de processeurs disponibles. Cette

limitation pourrait devenir problématique si nous avions besoin d'un très grand nombre de fourmis dans un problème de grande taille ou si nous voulions exécuter l'algorithme sur des machines parallèles possédant un nombre limité de processeurs. Pour toutes les raisons évoquées ci-haut, il est plus avantageux de répartir la charge de calcul entre les processeurs de sorte que plusieurs fourmis puissent être affectées à chaque processeur et ce, de façon dynamique. En procédant de cette façon, nous obtenons un algorithme plus efficace et des améliorations supplémentaires pourront être apportées en parallélisant les mises à jour de la matrice $\Delta\tau_{ij}$ et du quadtree, ce qui revient à paralléliser entièrement l'importante boucle « pour » de l'algorithme.

4.4.1.1.5 La mise à jour de la matrice $\Delta\tau_{ij}$ en parallèle

Les structures de données les plus importantes faisant partie de l'OCF_{S1} sont la matrice τ_{ij}, la matrice D, la matrice $\Delta\tau_{ij}$ et le quadtree. La matrice τ_{ij} est mise à jour à chaque cycle et ne peut être parallélisée sans changer le comportement de l'algorithme. Elle demeure donc en mémoire partagée durant l'exécution. Elle est accédée en lecture par la procédure de génération de solutions et sa mise à jour est effectuée par le processus maître à la fin de chaque cycle, une fois la région parallèle exécutée.

La matrice D est construite en début d'exécution et n'est jamais mise à jour. Elle demeure alors en mémoire partagée et est accédée en lecture seulement, au même titre que toutes les autres variables et structures de données servant de paramètres à l'algorithme.

Dans l'algorithme OCF_{S1}, la matrice $\Delta\tau_{ij}$ est mise à jour par chacune des fourmis une fois la génération et l'évaluation de leur solution terminée. Dans une version parallèle où

les fourmis sont réparties entre les processeurs, une première stratégie pourrait être d'effectuer la mise à jour à la sortie de la région parallèle, c'est-à-dire de façon séquentielle par le processus maître. Une deuxième façon de procéder pourrait être d'effectuer la mise à jour à l'intérieur de la région parallèle, mais dans une section critique qui n'est accessible que par une fourmi à la fois. La première stratégie implique une utilisation plus limitée des processeurs durant l'algorithme et la seconde, des temps d'attente importants pour les processeurs.

En introduisant un coût additionnel en mémoire, il est possible d'améliorer le temps d'exécution de l'algorithme en créant une matrice $\Delta\tau_{ij}$ pour chaque processeur, c'est-à-dire pour chaque groupe indépendant de fourmis. Cette modification peut se faire en créant, en mémoire partagée, un tableau de matrices $\Delta\tau_{ij}$ de dimension p (nombre de processeurs) dont chaque élément $\Delta\tau_{ij}[i]$ sera accessible par le processeur d'indice i. La seule utilisation de cette matrice est de mettre à jour la matrice τ_{ij} à la fin du cycle. La fusion de toutes les matrices ainsi créées peut donc se faire en parallèle après la région parallèle principale et avant la mise à jour de la matrice τ_{ij} et ce, sans modifier le comportement de l'algorithme.

4.4.1.1.6 La mise à jour du quadtree en parallèle

Un cas similaire est rencontré avec la procédure de mise à jour du quadtree. Originellement, une seule structure d'arbre existe et est mise à jour séquentiellement ou dans une section critique. Cependant, avec un autre coût additionnel en mémoire, il est possible de créer plusieurs arbres (un pour chaque processeur) et de les fusionner en dehors de la région parallèle. Un tableau de quadtrees de dimension p est donc créé en mémoire

partagée et chaque élément quadtree[i] du tableau est accessible par le processeur i. Comme le quadtree est une structure de stockage de données et non une structure d'information nécessaire à d'autres parties de l'algorithme comme l'est la matrice $\Delta\tau_{ij}$, la procédure de fusion des quadtrees peut se faire après l'exécution de tous les cycles.

En effectuant les modifications discutées précédemment à l'algorithme OCF_{S1}, nous obtenons la version parallèle présentée à la Figure 4.5 et notée OCF_P dans la suite de ce travail.

```
NC = 0;
Initialiser la matrice τij;
Initialiser les p quadtrees;
TANT QUE (NC < NCMax) ET (Convergence non atteinte)
    Initialiser les p matrices Δτij;
    RÉGION PARALLÈLE avec p processus
        POUR chaque fourmi k FAIRE
        Construire une séquence de commandes;
        Évaluer la solution k sur chaque objectif;
        Mettre à jour la matrice Δτij[i] selon la solution k;
        Insérer la solution k dans quadtree[i];
    Fusionner les p matrices Δτij en parallèle;
    Mettre à jour la matrice τij selon la matrice Δτij;
    NC = NC + 1;
Fusionner les p quadtrees;
```

Figure 4.5 Version parallèle de l'Optimisation par Colonies de Fourmis (OCF_P) dans un modèle à mémoire partagée

4.4.1.2 Essais numériques et résultats obtenus

La stratégie de parallélisation décrite à la section 4.4.1.1 a été implémentée à partir du programme séquentiel existant (écrit en langage C) en ajoutant les directives OpenMP appropriées et en apportant les changements discutés précédemment. L'expérimentation a

été réalisée sur des problèmes évalués précédemment par la version séquentielle et dont les résultats sont publiés dans Gravel *et al.* (2002). Nous avons pris soin de vérifier que la qualité des résultats obtenus par la version parallèle étaient identiques à la version séquentielle. Étant donné que la stratégie de parallélisation vise l'amélioration des performances relatives à la vitesse d'exécution, seul cet aspect sera considéré dans la discussion de cette section. De plus, comme les petits problèmes sont de peu d'intérêt pour cette étude, l'illustration des résultats obtenus se fera principalement à l'aide de problèmes de 50 et 80 commandes respectivement

Les Tableaux 4.1, 4.2, 4.3 et 4.4 montrent la performance de l'implémentation parallèle en faisant varier le nombre de processeurs (p) lorsque le nombre de fourmis (k) est fixé à 1000. Le temps d'exécution du programme, exprimé en secondes, est présenté pour chaque valeur de p. Les mesures d'accélération et d'efficacité permettent de mettre le temps d'exécution en relation avec le nombre de processeurs utilisés.

Les Tableaux 4.1 et 4.2 présentent les résultats d'un carnet de 50 commandes en utilisant respectivement 100 cycles et 200 cycles tout en conservant identiques les autres paramètres de l'algorithme. Ils permettent de constater la pénalité en efficacité causée par les synchronisations des différents processeurs lors de la mise à jour de la matrice τ_{ij}. Ces résultats montrent également qu'en augmentant le nombre de cycles de l'algorithme (NC) pour un même problème, ce qui implique également un plus grand nombre de mises à jour de la matrice τ_{ij} et plus de synchronisations, on observe une perte supplémentaire de l'efficacité.

Nombre de processeurs	Temps d'exécution (secondes)	Accélération	Efficacité
1	572	-	-
2	309	1.85	0.93
4	167	3.43	0.86
8	112	5.11	0.64
16	125	4.58	0.29

Tableau 4.1 Résultats obtenus par l'algorithme OCF_P pour un carnet de 50 commandes avec 100 cycles et 1000 fourmis

Nombre de processeurs	Temps d'exécution (secondes)	Accélération	Efficacité
1	1136	-	-
2	634	1.79	0.90
4	349	3.25	0.81
8	231	4.91	0.61
16	267	4.25	0.26

Tableau 4.2 Résultats obtenus par l'algorithme OCF_P pour un carnet de 50 commandes avec 200 cycles et 1000 fourmis

Les Tableaux 4.3 et 4.4 permettent de constater le même phénomène pour un problème de 80 commandes. Des pertes supplémentaires de l'efficacité sont également rencontrées lorsque le nombre de cycles passe de 100 à 200.

Nombre de processeurs	Temps d'exécution (secondes)	Accélération	Efficacité
1	1111	-	-
2	564	1.97	0.98
4	305	3.64	0.91
8	187	5.94	0.74
16	204	5.45	0.34

Tableau 4.3 Résultats obtenus par l'algorithme OCF_P pour un carnet de 80 commandes avec 100 cycles et 1000 fourmis

Nombre de processeurs	Temps d'exécution (secondes)	Accélération	Efficacité
1	2181	-	-
2	1152	1.89	0.95
4	613	3.56	0.89
8	381	5.72	0.72
16	429	5.08	0.32

Tableau 4.4 Résultats obtenus par l'algorithme OCF_P pour un carnet de 80 commandes avec 200 cycles et 1000 fourmis

La Figure 4.6 fournit une représentation graphique des temps d'exécution des Tableaux 4.1 et 4.3. On peut constater que le temps d'exécution diminue lorsque le nombre de processeurs utilisés augmente, mais que cette diminution devient plus limitée avec l'augmentation du nombre de processeurs.

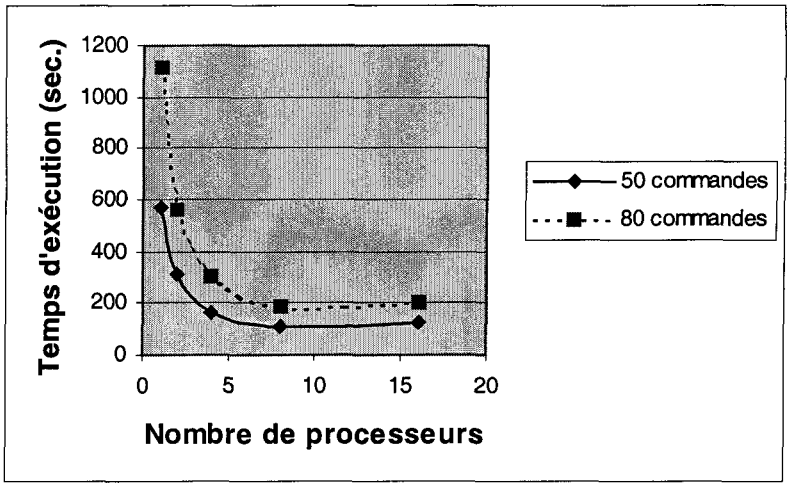

Figure 4.6 Temps d'exécution avec des carnets de 50 et 80 commandes (100 cycles et 1,000 fourmis)

De plus, en comparant les résultats des Tableaux 4.1 et 4.3, on peut voir l'influence de la taille du problème sur l'efficacité (iso-efficacité). En effet, une meilleure efficacité est

obtenue avec un carnet de 80 commandes comparativement avec un carnet de 50 lorsque que les mêmes paramètres sont appliqués. La Figure 4.7, en illustrant l'efficacité obtenue avec 8 problèmes de taille différente, montre bien le fait que la performance croît à mesure que la taille du problème augmente.

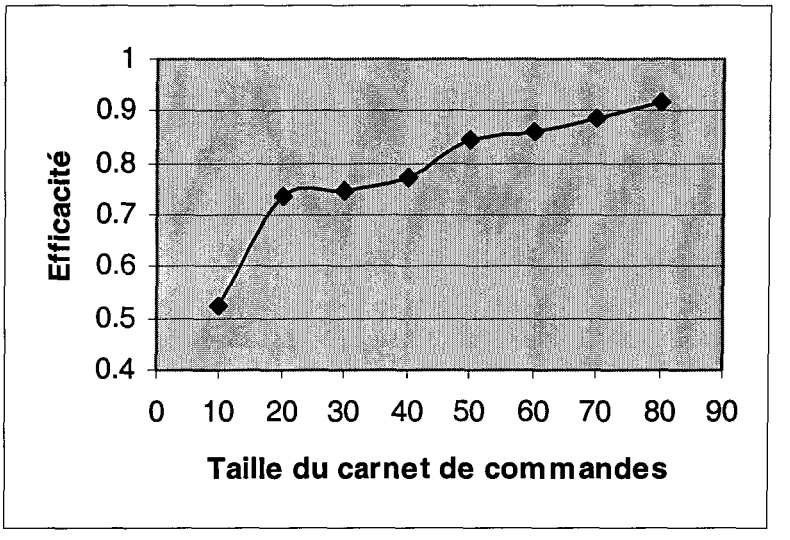

Figure 4.7 Efficacité avec 4 processeurs, 100 cycles, 1000 fourmis et des carnets de commandes variant entre 10 et 80

Des carnets de commandes de taille plus grande que 80 pourraient sans doute démontrer de façon encore plus convaincante les bénéfices qu'apportent le parallélisme, mais les limitations du programme original (particulièrement le processus complexe d'évaluation industrielle) empêchent l'utilisation de plus gros carnets pour fins de comparaisons. L'expansion de l'application pour traiter des problèmes de plus grande taille fera partie de travaux futurs.

De façon générale, les résultats montrent que la parallélisation interne de l'OCF_{SI} effectuée dans le cadre de ce travail mène à l'obtention de mesures d'efficacité

significatives. Cependant, les expérimentations ne sont pas aussi convaincantes qu'espérées, particulièrement avec 8 processeurs et plus. La dégradation de l'efficacité obtenue en augmentant le nombre de processeurs est plus rapide que prévue, et celle associée à l'augmentation du nombre de cycles est plus lente. De même, la perte associée à l'augmentation du nombre de fourmis est aussi plus lente. En effet, lorsque le nombre de fourmis est fixé à 10 000 (ce qui est un nombre considérablement plus élevé que celui habituel), l'efficacité augmente mais pas autant qu'anticipée. Par exemple, avec 2 processeurs, l'efficacité se situe 0.99, avec 4 processeurs, elle diminue à 0.92, avec 8 processeurs, elle diminue significativement à 0.76 et atteint 0.36 avec 16 processeurs.

Des expérimentations et études plus approfondies pourront permettre une meilleure compréhension des mécanismes de l'algorithme parallèle, des effets engendrés par la modification de ses paramètres et de l'influence des éléments logiciels et matériels utilisés pour le développement. Les principaux points à considérer dans les futurs travaux seraient :

- d'évaluer les effets de l'allocation dynamique de mémoire avec OpenMP;
- d'évaluer la performance globale du programme et la mettre en relation avec l'ordinateur parallèle utilisé et l'environnement OpenMP;
- d'effectuer des expérimentations extensives avec différents jeux de paramètres.

4.4.1.3 Sommaire

Dans la présente section de ce chapitre, une stratégie de parallélisation interne de la métaheuristique OCF_{SI} dans un modèle d'architecture à mémoire partagée a été présentée

et les principaux éléments du processus de parallélisation ont été expliqués. La transposition de l'algorithme séquentiel en version parallèle a été relativement simple à réaliser en raison de la nature même de la métaheuristique OCF qui s'y prête facilement. Certains changements importants ont tout de même dû être faits à la structure de l'algorithme et à son application informatique pour d'atteindre le degré d'efficacité décrit.

Le but de cette partie du travail était de réduire le temps d'exécution de l'algorithme en le parallélisant et de montrer certaines particularités relatives au modèle d'architecture sous-jacent. L'implémentation résultante a montré qu'il était possible de concevoir un OCF parallèle efficace sur une architecture à mémoire partagée. Certaines limites sur le nombre de processeurs utilisés ont également été établies et les causes de celles-ci devraient être analysées dans des travaux futurs.

Dans la prochaine section de ce travail, le potentiel de parallélisation de l'OCF sera exploité sous un autre angle. Le but recherché ne sera plus seulement de réduire son temps d'exécution, mais également d'améliorer ses capacités de recherche de solutions par des stratégies de processus coopératifs.

4.4.2 Parallélisation co-évolutive

Comme il a été présenté à la section 3.3, la contribution que peut apporter le parallélisme au monde des métaheuristiques n'est pas limitée à la réduction du temps de calcul. En effet, l'exécution concurrente de processus peut également permettre une meilleure qualité de recherche de solutions et ce, en un temps égal ou inférieur à la métaheuristique séquentielle correspondante.

Cette section présente comment la coopération d'agents engagés dans la résolution d'un problème d'ordonnancement industriel peut permettre de résoudre ce dernier plus efficacement qu'un agent seul ou qu'un groupe d'agents opérant isolément les uns des autres. Pour ce faire, des stratégies de parallélisation à processus coopératifs seront étudiées et les résultats obtenus seront mis en relation avec ceux donnés par la métaheuristique séquentielle.

Pour la suite de ce travail, la version améliorée de l'OCF (OCF_{S2}) sera utilisée. La performance des stratégies de coopération pourra donc être évaluée en relation avec les derniers résultats obtenus pour ce problème. L'optimisation multi-objectifs n'étant pas considérée dans ce travail, seul le retard total du carnet de commandes sera retenu comme objectif d'optimisation. Ce choix se justifie par le fait que pour les problèmes traités, le retard total semble être l'objectif le plus difficile à optimiser. Les résultats obtenus pour la minimisation de la perte de capacité et de la livraison sont déjà très satisfaisants et laissent peu de place à l'amélioration, ce qui n'est pas le cas pour le retard. La coopération entre des colonies de fourmis optimisant des objectifs différents est d'un intérêt certain et pourrait faire l'objet de travaux futurs.

Il est toutefois important de rappeler que la perte de capacité et le retard sont deux objectifs étroitement liés. Comme les opérations de réglage sont des pertes de temps non négligeables, une optimisation efficace du retard passe par la considération de ces pertes de capacité. Cette dépendance entre les deux objectifs doit être tenue en compte dans le paramétrage de l'algorithme.

Middendorf *et al.* (2000) ont proposé quatre stratégies de parallélisation de la métaheuristique OCF (voir la section 3.3.4.3) pour la résolution du problème du voyageur de commerce (VC). Dans ce travail, ces mêmes stratégies seront utilisées dans le contexte d'ordonnancement industriel présenté et adaptées pour s'exécuter sur une architecture à mémoire partagée. Cette étude permettra non seulement de constater s'il est possible d'améliorer la qualité des solutions obtenues par l'algorithme séquentiel de recherche en découvrant de meilleures solutions au problème, mais également d'analyser la performance comparative de ces stratégies pour la résolution d'un problème autre que le voyageur de commerce.

La première étape de cette étude est de développer une stratégie de parallélisation de l'OCF selon un modèle à processus multiples indépendants. Il sera ensuite possible d'introduire à cet algorithme les différentes stratégies d'échange d'information qui le transformeront en un algorithme parallèle à processus coopératifs.

4.4.2.1 Parallélisation à processus multiples indépendants

On peut définir l'algorithme OCF séquentiel comme étant un processus simple qui effectue de façon itérative la recherche et l'évaluation de solutions. À la section 4.4.1, les calculs à une itération donnée ont été répartis entre plusieurs processeurs et exécutés en parallèle pour obtenir une parallélisation à grain relativement fin. Une autre stratégie de parallélisation consiste à effectuer plusieurs exécutions de l'algorithme en parallèle. De cette façon, nous obtenons une parallélisation à plus gros grain où plusieurs processus de recherche sont mis en œuvre, chaque processeur effectuant son propre OCF séquentiel.

La Figure 4.8 présente la structure de l'algorithme parallèle à processus multiples indépendants sur une architecture à mémoire partagée. Cet algorithme, issu de la version OCF_{S2}, sera noté OCF_{P0} dans la suite de ce travail. Les paramètres de l'algorithme (*NC*, *k*, α, β, etc.), identiques pour chacun des *p* processeurs, sont initialisés de façon séquentielle par le processus maître au début de l'algorithme, stockés en mémoire partagée et accédés en lecture par chaque processus durant leur exécution.

```
Initialiser les p quadtrees;
RÉGION PARALLÈLE avec p processus
   NC = 0;
   Initialiser la matrice τij;
   TANT QUE (NC < NCMax) ET (Convergence non atteinte)
     POUR i = 1 à n FAIRE
        POUR k = 1 à m FAIRE
           Sélectionner la commande j à être ajoutée à la séquence selon p_ij^k(t);
           Effectuer la mise à jour locale de la trace selon la paire de commandes (i, j) ;
     POUR chaque fourmi k FAIRE
        Évaluer la solution k sur chaque objectif;
        Insérer la solution k dans quadtree[l];
     Effectuer la mise à jour globale de la trace selon la meilleure solution du cycle;
     NC = NC + 1;
Fusionner les p quadtrees;
```

Figure 4.8 Parallélisation de l'OCF_{S2} en processus multiples indépendants (OCF_{P0})

Chaque processus effectue son propre OCF séquentiel et produit un certain nombre de solutions qu'il évalue et insère dans un quadtree. À la fin de la région parallèle, les seules informations pertinentes à conserver de l'exécution de chaque processus sont les solutions générées et évaluées par chacun d'entre eux, informations présentes dans le quadtree. Afin

de conserver cette information, un tableau de p (le nombre de processeurs) quadtrees est initialisé en mémoire partagée avant de débuter la région parallèle. Chaque processus i insère les solutions générées dans le quadtree[i] et une fusion des quadtrees est effectuée à la sortie de la région parallèle. De cette façon, le résultat de l'algorithme est un arbre contenant toutes les solutions non dominées trouvées par l'ensemble des processeurs. La Figure 4.9 présente une illustration de l'utilisation du quadtree en mémoire partagée.

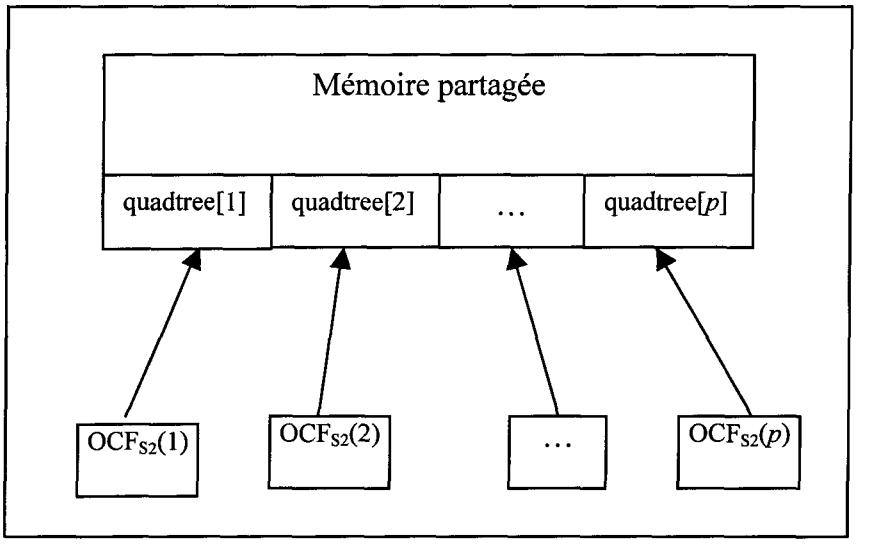

Figure 4.9 Utilisation du quadtree en mémoire partagée

Chaque processus effectue sa recherche en utilisant sa propre matrice τ_{ij}. La création, l'initialisation et la mise à jour de cette matrice s'effectue de façon privée par chaque processeur et n'entraîne aucun problème de synchronisation.

De façon générale, on peut constater que contrairement au cas de parallélisation interne, la parallélisation en processus multiples indépendants n'entraîne pas de limitations

relatives aux synchronisations et aux communications et ce, autant sur une architecture à passage de messages que sur une architecture à mémoire partagée. Le seul moment où de telles opérations seront effectuées est en fin d'exécution. Dans un modèle à passage de messages, chaque processeur possèderait un quadtree privé et devrait l'envoyer, à la fin de son exécution, à un processeur assigné qui s'occuperait de les fusionner. Dans le modèle à mémoire partagée de ce présent travail, la barrière de synchronisation implicite présente à la sortie de la région parallèle garantit que tous les processeurs ont terminé leur exécution au moment où la fusion des quadtrees est réalisée. Cet algorithme permet donc une efficacité parallèle théorique presque optimale.

Dans le cas où l'on compare le résultat d'une exécution parallèle sur n processeurs avec celui de n exécutions de l'algorithme séquentiel, la forme de parallélisation présentée ci-haut ne modifie pas le processus de recherche de solutions de l'algorithme. Les bénéfices engendrés ne sont encore que des améliorations du temps de calcul. Cependant, ce modèle de parallélisation à processus multiples présente un potentiel d'un tout autre ordre. L'élaboration de stratégies d'échange d'information entre les processus de recherche permet d'introduire à l'algorithme une nouvelle forme d'apprentissage qui lui permettra de modifier son comportement de recherche de solutions et, possiblement, de l'améliorer. Il s'agit ici de transformer la parallélisation en processus multiples indépendants en une parallélisation en processus multiples coopératifs afin de permettre à l'algorithme de trouver des solutions de meilleure qualité. Les bénéfices engendrés par la coopération sont intimement liés à la qualité des stratégies d'échange d'information sous-jacentes. Une étude de ces stratégies dans le cadre de l'OCF a été effectuée par Middendorf *et al.* (2000).

Dans la suite de ce travail, cette étude est reprise et adaptée au contexte présent où l'algorithme, le problème traité et le modèle d'architecture parallèle sont différents et présentent leurs propres particularités.

4.4.2.2 Stratégies d'échange d'information

Les quatre stratégies de parallélisation proposées par Middendorf *et al.* (2000) présentent diverses variations dans la structure des canaux virtuels de communication entre les processeurs et dans la nature de l'information échangée. Ces stratégies sont les suivantes :

P1. Échange de la meilleure solution globale ;
P2. Échange circulaire des meilleures solutions locales ;
P3. Échange circulaire de migrateurs ;
P4. Échange circulaire des meilleures solutions locales et de migrateurs.

Décrivons brièvement le fonctionnement de chacune de ces stratégies et les implications dans le cadre de notre application.

4.4.2.2.1 Échange de la meilleure solution globale (P1)

Cette stratégie implique qu'à chaque étape d'échange d'information, la meilleure solution trouvée par l'ensemble des processeurs sera diffusée à toutes les colonies où elle devient la meilleure solution locale. Il est important de spécifier que l'algorithme utilisé par Middendorf *et al.* (2000) effectue une mise à jour de la trace supplémentaire selon le principe d'élitisme, c'est-à-dire que la meilleure solution trouvée par l'algorithme depuis le début de son exécution est utilisée pour renforcer davantage la quantité de phéromone. La

diffusion de la meilleure solution globale vient donc modifier les paramètres de cette mise à jour élitiste de la trace.

Dans notre cas, l'architecture à mémoire partagée fait en sorte que la procédure de diffusion n'est pas pertinente et doit être mise en œuvre différemment. Pour ce faire, un emplacement dédié à contenir la meilleure solution trouvée par l'ensemble des processeurs est réservé en mémoire partagée et accessible en écriture par tous les processeurs. Avant chaque étape d'échange d'information, chaque processeur comparera la meilleure solution qu'il a trouvée avec celle présente à l'emplacement partagé. Si cette meilleure solution locale est supérieure à la meilleure solution globale, alors elle deviendra la nouvelle meilleure solution globale. L'instauration d'une zone critique permettra à tous les processeurs d'effectuer cette vérification tout en évitant les problèmes d'écriture concurrente. Ensuite, tous les processeurs utiliseront la meilleure solution globale pour effectuer une mise à jour additionnelle de la trace. La Figure 4.10 illustre le fonctionnement de cette stratégie et la Figure 4.11 en présente l'algorithme qui sera noté OCF_{PI} dans la suite de ce travail.

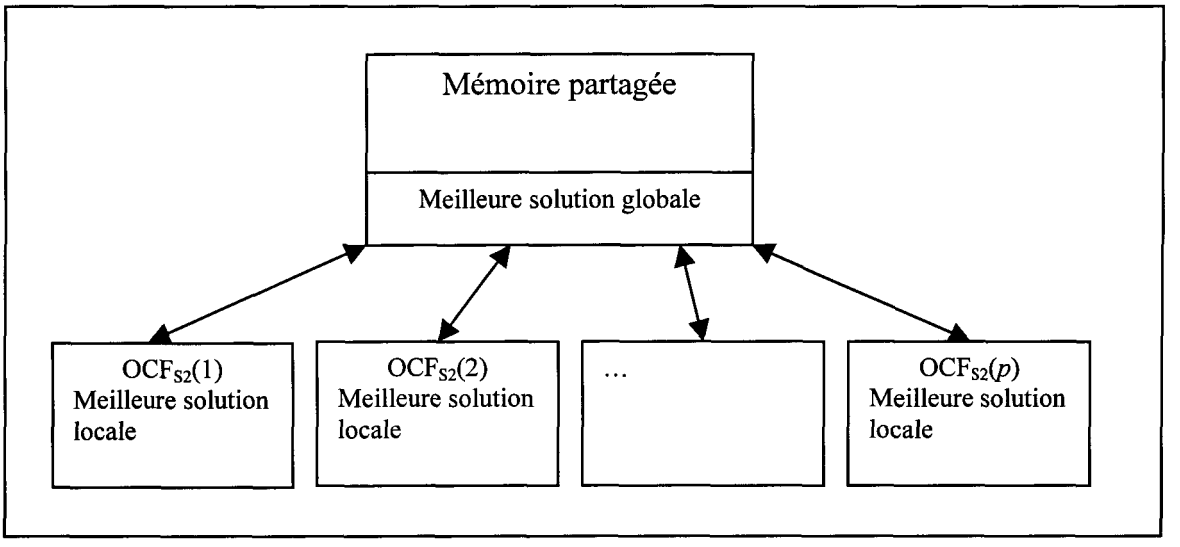

Figure 4.10 Diffusion de la meilleure solution globale dans un modèle à mémoire partagée

```
Initialiser les p quadtrees;
Initialiser meilleure solution globale;
RÉGION PARALLÈLE avec p processus
   NC = 0;
   Initialiser la matrice τij;
   TANT QUE (NC < NCMax) ET (Convergence non atteinte)
      POUR i = 1 à n FAIRE
         POUR k = 1 à m FAIRE
            Sélectionner la commande j à être ajoutée à la séquence selon p_ij^k(t);
            Effectuer la mise à jour locale de la trace selon la paire de commandes (i, j) ;
      POUR chaque fourmi k FAIRE
         Évaluer la solution k sur chaque objectif;
         Insérer la solution k dans quadtree[i];
      Effectuer la mise à jour globale de la trace selon la meilleure solution du cycle
      SI étape d'échange d'information
            Si Meilleure solution locale < Meilleure solution globale
                  Meilleure solution globale = meilleure solution locale
            Effectuer la mise à jour de la trace selon la meilleure solution globale
      NC = NC + 1;
Fusionner les p quadtrees;
```

Figure 4.11 OCF$_{S2}$ parallèle avec stratégie d'échange de la meilleure solution globale (OCF$_{P1}$)

4.4.2.2.2 Échange circulaire des meilleures solutions locales (P2)

L'élaboration d'une stratégie d'échange circulaire des meilleures solutions locales nécessite qu'un voisinage virtuel soit établi entre les colonies de façon à ce qu'elles forment un anneau. À chaque étape d'échange d'information, chaque colonie envoie sa meilleure solution locale à la colonie qui la succède dans l'anneau. L'information partagée est la même que dans le cas de la stratégie précédente, mais l'échange se fait entre des couples de processeurs plutôt que globalement. Un processeur fournit de l'information à son successeur et utilise celle de son prédécesseur.

Cette stratégie est mise en œuvre en réservant, en mémoire partagée, un tableau de solutions T de dimension égale au nombre de processeurs p. À chaque étape d'échange d'information, le processeur i écrit à la case $T[i+1]$ la meilleure solution qu'il a trouvée localement. Comme chaque processeur écrit à un emplacement différent du tableau, cette procédure ne nécessite pas l'instauration d'une zone critique. Ensuite, après l'atteinte d'une barrière de synchronisation assurant que tous les processeurs ont écrit leur solution respective, chaque processeur i compare sa meilleure solution locale à celle présente à la case $T[i]$ du tableau (celle provenant de son prédécesseur). La mise à jour élitiste de la trace est finalement effectuée selon la meilleure solution d'entre les deux. Les Figures 4.12 et 4.13 présentent respectivement une représentation graphique du fonctionnement de cette stratégie et de l'algorithme OCF_{P2} qui en résulte.

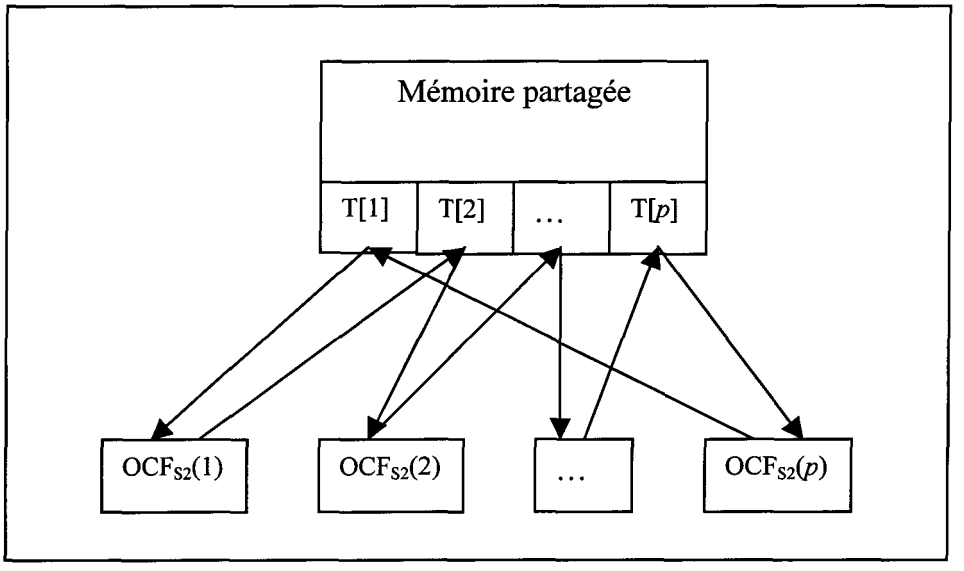

Figure 4.12 Échange circulaire des meilleures solutions locales dans un modèle à mémoire partagée

Initialiser les *p* quadtrees;
Initialiser tableau *T* des meilleures solutions locales;
RÉGION PARALLÈLE avec *p* processus
  NC = 0;
  Initialiser la matrice τ_{ij};
  TANT QUE (*NC* < NC_{Max}) ET (Convergence non atteinte)
    POUR *i* = 1 à *n* FAIRE
      POUR *k* = 1 à *m* FAIRE
        Sélectionner la commande *j* à être ajoutée à la séquence selon $p_{ij}^{k}(t)$;
        Effectuer la mise à jour locale de la trace selon la paire de commandes (*i*, *j*) ;
    POUR chaque fourmi *k* FAIRE
      Évaluer la solution *k* sur chaque objectif;
      Insérer la solution *k* dans quadtree[*i*];
    Effectuer la mise à jour globale de la trace selon la meilleure solution du cycle
    SI étape d'échange d'information
      T[*i*+1] = Meilleure solution locale ;
      Si *T*[*i*] < Meilleure solution locale
        Meilleure solution locale = *T*[*i*]
      Effectuer la mise à jour de la trace selon la meilleure solution locale
    NC = *NC* + 1;
Fusionner les *p* quadtrees;

Figure 4.13 OCF_{S2} parallèle avec stratégie d'échange circulaire des meilleures solutions locales (OCF_{P2})

4.4.2.2.3 Échange circulaire de migrateurs (P3)

Comme dans le cas de la deuxième stratégie, les processeurs forment un anneau virtuel. À chaque étape d'échange, chaque colonie compare ses m meilleures fourmis avec les m meilleures de son prédécesseur. La circulation des données se fait donc de la même façon que pour l'OCF_{P2} (Figure 4.12). Cependant, les meilleures solutions trouvées durant le cycle en cours sont échangées plutôt que les meilleures solutions locales. Les m meilleures fourmis de ces deux groupes sont alors utilisées pour la mise à jour globale de la trace. Il est important de souligner ici l'importante différence entre l'échange de meilleures solutions et l'échange de migrateurs. Dans les stratégies précédentes, les meilleures solutions étaient échangées et influençaient la mise à jour élitiste de la trace. La procédure de mise à jour globale demeurait inchangée et ne concernait que les fourmis propres au processeur. Dans le cas présent, l'échange de migrateurs concerne la mise à jour globale de la trace et la mise à jour élitiste n'est pas effectuée.

L'intégration de cette stratégie se fait également par la création d'un tableau de solutions en mémoire partagée. Ce tableau est noté U. Comme l'algorithme utilisé dans ce travail effectue la mise à jour globale de la trace à partir de la meilleure solution trouvée pendant le cycle présent, une seule solution ($m = 1$) sera échangée dans l'anneau, c'est-à-dire la meilleure solution trouvée par chaque processeur durant le cycle d'échange. La meilleure solution entre celle obtenue par l'échange et celle trouvée localement durant le cycle sera retenue pour effectuer la mise à jour globale. Cette version de l'algorithme, illustrée à la Figure 4.14, sera notée OCF_{P3} dans la suite de ce travail.

```
Initialiser les p quadtrees;
Initialiser le tableau U des meilleures solutions trouvées durant un cycle;
RÉGION PARALLÈLE avec p processus
   NC = 0;
   Initialiser la matrice τij;
   TANT QUE (NC < NCMax) ET (Convergence non atteinte)
      POUR i = 1 à n FAIRE
         POUR k = 1 à m FAIRE
            Sélectionner la commande j à être ajoutée à la séquence selon p_ij^k(t);
            Effectuer la mise à jour locale de la trace selon la paire de commandes (i, j) ;
      POUR chaque fourmi k FAIRE
         Évaluer la solution k sur chaque objectif;
         Insérer la solution k dans quadtree[i];
      SI étape d'échange d'information
         U[i+1] = Meilleure solution trouvée durant le cycle ;
         Si U[i] < Meilleure solution trouvée durant le cycle
               Meilleure solution trouvée durant le cycle = U[i]
      Effectuer la mise à jour globale de la trace selon la meilleure solution du cycle
      NC = NC + 1;
Fusionner les p quadtrees;
```

Figure 4.14 OCF_{S2} parallèle avec stratégie d'échange circulaire des meilleures solutions locales (OCF_{P3})

4.4.2.2.4 Échange circulaire des meilleures solutions locales et des migrateurs (P4)

La stratégie P4 est tout simplement la combinaison des stratégies P2 et P3 présentées dans les sections précédentes. Elle nécessite la création en mémoire partagée de deux tableaux de solutions de dimension p. Le tableau T sert à stocker la meilleure solution locale de chaque processeur et le tableau U contient la meilleure solution trouvée durant le cycle. Chaque processeur écrit, à l'emplacement approprié de chaque tableau, chacune de ces deux solutions. Ensuite, chacun utilise l'information qu'il a reçue de son prédécesseur pour effectuer la mise à jour globale de sa trace, suivie de sa mise à jour élitiste. La Figure 4.15 présente cette version de l'algorithme qui sera notée OCF_{P4} dans la suite de ce travail.

```
Initialiser les p quadtrees;
Initialiser T;
Initialiser U;
RÉGION PARALLÈLE avec p processus
   NC = 0;
   Initialiser la matrice τij;
   TANT QUE (NC < NCMax) ET (Convergence non atteinte)
      POUR i = 1 à n FAIRE
         POUR k = 1 à m FAIRE
            Sélectionner la commande j à être ajoutée à la séquence selon p_ij^k(t);
            Effectuer la mise à jour locale de la trace selon la paire de commandes (i, j) ;
      POUR chaque fourmi k FAIRE
         Évaluer la solution k sur chaque objectif;
         Insérer la solution k dans quadtree[i];
      SI étape d'échange d'information
         U[i+1] = Meilleure solution trouvée durant le cycle ;
         Si U[i] < Meilleure solution trouvée durant le cycle
            Meilleure solution trouvée durant le cycle = U[i]
         T[i+1] = Meilleure solution locale ;
         Si T[i] < Meilleure solution locale
            Meilleure solution locale = T[i]
         Effectuer la mise à jour de la trace selon la meilleure solution locale
      Effectuer la mise à jour globale de la trace selon la meilleure solution du cycle
      NC = NC + 1;
Fusionner les p quadtrees;
```

Figure 4.15 OCF_{S2} parallèle avec stratégie d'échange circulaire des meilleures solutions locales et de migrateurs (OCF_{P4})

4.4.2.3 Les paramètres de l'algorithme parallèle

Le paramétrage de l'algorithme séquentiel a fait l'objet de plusieurs travaux et expérimentations. Les derniers résultats publiés se retrouvent dans Gagné *et al.* (2001a) mais des essais numériques additionnels ont permis d'améliorer encore davantage ces résultats et c'est avec ces derniers que se réalisera la comparaison numérique dans cette étude. Toutefois, on peut trouver dans Gagné *et al.* (2001a) toutes les informations concernant les paramètres utilisés par les versions parallèles. Les principaux paramètres

concernés sont le nombre de cycles (*NC*), le nombre de fourmis (*k*), les paramètres d'évaporation et de mise à jour de la trace (ρ local et ρ global), l'application de la règle de transition, ainsi que l'importance accordée à la fonction d'anticipation et à la recherche locale. Bien entendu, les stratégies d'échange d'information viennent modifier le comportement de recherche de l'algorithme et il est possible qu'une nouvelle étude du paramétrage permette l'obtention de meilleures performances. Une telle étude dépasse cependant le cadre de ce travail et sera considérée ultérieurement.

Les paramètres étudiés seront plutôt ceux qui sont pertinents à la parallélisation et aux échanges d'information. Entre autres, le nombre de processeurs utilisés (qui définit le nombre de colonies de fourmis opérant en parallèle) et la fréquence des étapes d'échange sont les principaux paramètres étudiés dans ce travail. La diversification de la recherche de solutions par les processeurs représente également un facteur important pour la performance de l'algorithme parallèle et sera présentée en détails à la section suivante.

4.4.2.4 Diversification de la recherche de solutions

Dans un problème de voyageur de commerce, la ville de départ est générée aléatoirement, ce qui permet d'assurer une bonne couverture de l'espace des solutions. Par contre, dans le problème d'ordonnancement industriel, la commande de départ correspond à la dernière commande de la période précédente de planification et est, par le fait même, identique pour chacune des fourmis. En conséquence, si les paramètres de l'OCF étaient identiques pour chaque processeur, les gains liés à la coopération seraient alors minimes étant donné le peu de diversité de l'information échangée entre les processeurs. De ce fait,

l'utilisation de valeurs différentes pour les paramètres β et δ, associés respectivement aux matrices de réglages et de retard (voir Équation 5), devrait permettre de mieux diversifier la recherche et l'information échangée et ce, sans augmenter le nombre total de solutions explorées.

4.4.2.5 Essais numériques et résultats obtenus

Afin de vérifier leur performance, les quatre stratégies de parallélisation mentionnées précédemment ont été implantées dans le logiciel d'ordonnancement existant (OCF_{S2}) en utilisant une fois de plus l'environnement de programmation parallèle OpenMP. Les expérimentations ont été effectuées à l'aide des huit problèmes utilisés pour tester les versions séquentielles (Gagné *et al.* (2001a)).

L'ajustement des paramètres propres à l'algorithme parallèle a été effectué à l'aide de tests empiriques. Ces derniers ont permis d'identifier des configurations intéressantes, certaines correspondant aux conclusions formulées par Middendorf *et al.* (2000) et d'autres y étant contradictoires. Les principaux paramètres à déterminer sont le nombre de processeurs, la fréquence des échanges d'information et la quantité d'information échangée.

Dans un premier temps, la concordance entre la qualité des solutions obtenues par l'OCF_{S2} et la version parallèle à processus indépendants OCF_{P0} sera vérifiée. Ensuite, la configuration retenue pour ces paramètres sera discutée et justifiée par la présentation de certaines données empiriques ayant guidé le processus de décision. Finalement, cette configuration sera utilisée pour montrer l'influence de la diversification de la recherche de

chaque processeur et pour tester la performance des algorithmes parallèles en comparaison avec l'OCF_{P0}.

4.4.2.5.1 Parallélisation en processus multiples indépendants

La stratégie de parallélisation en processus multiples indépendants (OCF_{P0}), décrite à la section 4.4.2.1, ne modifie pas le processus de recherche de solutions. L'algorithme résultant doit donc produire des résultats similaires à la version séquentielle originale. Toutefois, les résultats publiés avec la version séquentielle OCF_{S2} ont été obtenus avec dix exécutions successives de l'algorithme. Afin de reproduire cette même configuration avec la version parallèle, dix processeurs ont été utilisés et chacun a effectué une exécution de l'algorithme. Cette validation de performance entre les deux algorithmes permettra de garantir ultérieurement que les stratégies de coopération sont réellement responsables de l'amélioration de la qualité des solutions et que celle-ci n'est pas attribuable à des facteurs non inhérents à la parallélisation. Le Tableau 4.5 permet de constater la similarité des résultats obtenus pour chacun des 8 problèmes tests sur l'objectif de retard. Dans ce tableau, nous présentons, pour chaque problème, le retard moyen et l'écart-type obtenus pour le même nombre d'évaluation de solutions par les deux versions de l'algorithme, chacune d'entre elles ayant été exécutée dix fois.

	OCF_{S2}		OCF_{P0}	
Problème	**Retard**		**Retard**	
10	**6.86**	(0.12)	**6.76**	(0.08)
20	**16.28**	(0.08)	**16.36**	(0.12)
30	**27.44**	(0.89)	**26.71**	(0.78)
40	**49.63**	(2.97)	**50.37**	(2.24)
50	**18.59**	(0.69)	**18.87**	(0.50)
60	**31.31**	(1.66)	**29.67**	(1.57)
70	**126.77**	(6.29)	**127.73**	(3.77)
80	**183.34**	(6.00)	**189.17**	(6.51)

Tableau 4.5 Résultats comparatifs des versions de OCF_{S2} et OCF_{P0} lorsque le "retard" est l'objectif à minimiser. Le premier résultat de chacune des colonnes correspond à la moyenne de dix essais et le second résultat est l'écart-type.

Une analyse statistique a été réalisée pour comparer ces résultats et s'assurer de leurs égalités. À cet effet, le résultat de Conover et Iman (1981) permet d'utiliser un test d'hypothèse paramétrique sur la moyenne des rangs. Les rangs ont été établis à partir des 20 résultats obtenus par les deux algorithmes et le Tableau 4.6 présente la compilation de ces tests statistiques. Ceux-ci permettent de conclure, pour chaque problème au seuil de 0.01, à l'égalité des résultats. Dans les sections qui suivront, la version OCF_{P0}, équivalente à l'OCF_{S2} en termes de solutions obtenues, servira donc de base de comparaison pour valider la performance des stratégies d'échange d'information.

	Problème 10		**Problème 20**		**Problème 30**		**Problème 40**	
	OCF_{S2}	OCF_{P0}	OCF_{S2}	OCF_{P0}	OCF_{S2}	OCF_{P0}	OCF_{S2}	OCF_{P0}
Retard moyen	*6.86*	*6.76*	*16.28*	*16.36*	*27.44*	*26.71*	*49.63*	*50.37*
Variance	*0.12*	*0.08*	*0.08*	*0.12*	*0.89*	*0.78*	*2.97*	*2.24*
Moyenne des rangs	*8.25*	*12.75*	*12.3*	*8.7*	*13.65*	*7.35*	*11.70*	*9.30*
Variance des rangs	*14.74*	*30.07*	*24.96*	*16.23*	*33.45*	*18.34*	*24.9*	*45.79*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*-2.126*		*1.774*		*-2.77*		*0.903*	
P(T<=t) bilatéral	*0.048**		*0.093**		*0.013**		*0.379**	

	Problème 50		Problème 60		Problème 70		Problème 80	
	OCF_{S2}	OCF_{P0}	OCF_{S2}	OCF_{P0}	OCF_{S2}	OCF_{P0}	OCF_{S2}	OCF_{P0}
Retard moyen	*18.59*	*18.87*	*31.31*	*29.67*	*126.77*	*127.73*	*183.34*	*189.17*
Variance	*0.69*	*0.50*	*1.66*	*1.57*	*6.29*	*3.77*	*6.00*	*6.51*
Moyenne des rangs	*11.85*	*9.15*	*7.90*	*13.1*	*11.10*	*9.90*	*13.00*	*8.00*
Variance des rangs	*32.11*	*37.61*	*34.77*	*24.10*	*30.32*	*42.77*	*34.89*	*25.11*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*1.022*		*-2.143*		*0.444*		*2.041*	
P(T<=t) bilatéral	*0.320**		*0.046**		*0.662**		*0.056**	

**t-test effectué avec variances égales*

Tableau 4.6 Tests paramétriques t d'égalité sur la moyenne des rangs entre la version OCF_{S2} et OCF_{P0} pour les huit problèmes tests. Le seuil utilisé pour les tests est de 1%.

4.4.2.5.2 La fréquence des échanges d'information

Une des conclusions formulées par Middendorf *et al.* (2000) suite à l'étude des quatre stratégies d'échange d'information était qu'une fréquence d'échange trop élevée empêchait les colonies de fourmis d'évoluer dans des directions différentes. Par conséquent, il était préférable de limiter cette fréquence pour obtenir les meilleurs résultats. Dans le cadre de ce travail, des échanges à tous les cycles, à tous les cinq cycles et à tous les dix cycles ont été expérimentés et les résultats se sont avérés statistiquement plus intéressants lorsque les échanges étaient effectués à tous les cycles. À titre d'exemple, le Tableau 4.7 présente les résultats obtenus avec le problème test 70 pour différentes fréquences d'échange d'information. Les résultats des tests statistiques présentés au Tableau 4.8 confirment, au seuil de 0.01, que la fréquence d'échange à tous les cycles donne généralement de meilleurs résultats et qu'au pire, dans certains cas, les résultats s'avèrent équivalents. Plus spécifiquement, pour les stratégies P1 et P3, il est confirmé statistiquement que l'échange à tous les cycles donne de meilleurs résultats que ceux à tous les 5 cycles ou 10 cycles. Pour la stratégie P2, les échanges à tous les cycles et à tous les 5 cycles sont équivalents

statistiquement et préférables à l'échange à tous les 10 cycles. Pour la stratégie P4, toutes les fréquences d'échanges sont jugées équivalentes.

Fréquence de l'échange (cycle)	OCF_{P1}	OCF_{P2}	OCF_{P3}	OCF_{P4}
1	**115.85 (3.89)**	**118.08 (3.76)**	**117.39 (6.51)**	117.51 (7.96)
5	123.62 (4.84)	**121.90 (5.49)**	122.31 (4.75)	122.58 (5.04)
10	120.83 (2.80)	123.84 (4.22)	122.87 (4.04)	121.14 (4.70)

Tableau 4.7 Retard moyen et écart type obtenus pour 10 essais à l'aide des 4 stratégies d'échange d'information sur le problème de taille 70 lorsque la fréquence d'échange varie.

Des essais réalisés sur d'autres problèmes tests vont dans la même direction et confirment de meilleurs résultats lorsque l'échange est réalisée à tous les cycles. Cette configuration a donc été retenue pour la suite des expérimentations. Divers facteurs pourraient expliquer ce paradoxe, notamment la nature différente du problème, la taille des problèmes testés et les versions différentes de l'OCF. Des études plus approfondies pourraient permettre de mieux comprendre ce phénomène.

	OCF_{P1}				OCF_{P2}			
	1 cycle	5 cycles	1 cycle	10 cycles	1 cycle	5 cycles	1 cycle	10 cycles
Retard moyen	*115.85*	*123.62*	*115.85*	*120.83*	*118.08*	*121.90*	*118.08*	*123.84*
Variance	*3.89*	*4.84*	*3.89*	*2.80*	*3.76*	*5.49*	*3.76*	*4.22*
Moyenne des rangs	*6.5*	*14.5*	*6.9*	*14.1*	*8.4*	*12.6*	*7.0*	*14.0*
Variance des rangs	*22.5*	*15.83*	*29.43*	*15.66*	*23.38*	*41.71*	*23.33*	*23.33*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*-4.086*		*-3.391*		*-1.65*		*-3.24*	
P(T<=t) bilatéral	*0.001**		*0.003**		*0.114*		*0.005**	

	OCF_{P3}				OCF_{P4}			
	1 cycle	5 cycles	1 cycle	10 cycles	1 cycle	5 cycles	1 cycle	10 cycles
Retard moyen	*117.39*	*122.31*	*117.39*	*122.87*	*117.51*	*122.58*	*117.51*	*121.14*
Variance	*6.51*	*4.75*	*6.51*	*4.04*	*7.96*	*5.04*	*7.96*	*4.70*
Moyenne des rangs	*7.4*	*13.6*	*7.2*	*13.8*	*7.9*	*13.1*	*8.6*	*12.4*
Variance des rangs	*40.27*	*12.27*	*36.4*	*13.29*	*38.99*	*19.88*	*43.82*	*22.04*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*-2.705***		*-2.96*		*-2.14*		*-1.48*	
P(T<=t) bilatéral	*0.017**		*0.008**		*0.046*		*0.156*	

**significatif* ***t-test effectué avec variances inégales*

Tableau 4.8 Tests paramétriques t d'égalité sur la moyenne des rangs pour chacune des quatre stratégies lorsque la fréquence d'échange diffère. Le seuil utilisé pour les tests est de 1%.

4.4.2.5.3 La quantité d'information échangée

Une deuxième constatation mentionnée par Middendorf *et al.* (2000) était qu'il était préférable de limiter la quantité d'information échangée entre les processeurs. Ils soutenaient qu'il était préférable de n'échanger qu'un petit nombre de solutions plutôt que des matrices de trace complètes. À la suite d'une série d'expérimentations, la même conclusion est tirée dans le cas présent. L'échange de matrices complètes de trace n'apporte généralement aucun bénéfice et, dans plusieurs cas, des performances inférieures à la version séquentielle ont même été obtenues.

4.4.2.5.4 Visibilité fixe et visibilité variable

Nous avons également voulu vérifier la performance des échanges d'information lorsque toutes les colonies de fourmis opèrent selon des paramètres de visibilité identiques. À titre d'exemple, la première partie du Tableau 4.9 présente les résultats obtenus par les versions parallèles utilisant des échanges d'information et la version parallèle sans échange d'information (OCF_{P0}) pour le problème test de 70 commandes lorsque les paramètres de visibilité sont identiques. Les résultats des tests statistiques présentés dans la seconde partie du Tableau 4.9 montrent, qu'avec des paramètres de visibilité identiques, les versions incluant des stratégies d'échanges d'information permettent toujours d'obtenir une meilleure qualité de solutions au seuil de 0.01. Le retard moyen est réduit entre 8 et 11 jours environ pour ce problème de 70 commandes.

	Problème 70							
	OCF_{P0}	**OCF_{P1}**	**OCF_{P0}**	**OCF_{P2}**	**OCF_{P0}**	**OCF_{P3}**	**OCF_{P0}**	**OCF_{P4}**
Retard moyen	*127.73*	*115.85*	*127.73*	*118.08*	*127.73*	*117.39*	*127.73*	*117.51*
Variance	*3.77*	*3.89*	*3.77*	*3.76*	*3.77*	*6.51*	*3.77*	*7.96*
Moyenne des rangs	*15.5*	*5.5*	*15.5*	*5.5*	*15.5*	*5.5*	*15.5*	*5.5*
Variance des rangs	*9.17*	*9.17*	*9.17*	*9.17*	*9.17*	*9.17*	*9.17*	*9.17*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*7.39*		*7.39*		*7.39*		*7.39*	
P(T<=t) bilatéral	*0.000*		*0.000*		*0.000*		*0.000*	

Tableau 4.9 Tests paramétriques t d'égalité sur la moyenne des rangs entre la version parallèle de base (OCF_{P0}) et les quatre versions basées sur des stratégies d'échange d'information sur le problème de taille 70 lorsque les paramètres de visibilité sont identiques pour tous les processeurs. Le seuil utilisé pour les tests est de 1%.

Cependant, comme il a été mentionné à la section 4.4.2.4, la nature du problème et l'OCF_{S2} entraîne une exploration plus limitée de l'espace des solutions comparativement à la résolution du VC effectuée par Middendorf *et al.* (2000). Il était donc possible que cette

situation limite la performance des stratégies d'échange d'information. Dans le but de diversifier la recherche de solutions par les processeurs et ainsi enrichir l'information échangée, des jeux de valeurs différents pour les paramètres (β,δ) associés aux matrices de visibilité (de réglages et de retard) ont été utilisés pour chaque processeur. Ces paramètres déterminent l'importance relative accordée aux matrices de réglages et de retard dans la sélection des commandes lors de la construction des solutions. La différentiation des paramètres fait donc en sorte que chaque processeur guidera sa recherche de façon différente. Le processeur p_0, en utilisant les valeurs $\beta = 1$ et $\delta = 1$, est davantage guidé par la trace de phéromone et le processeur p_9, en utilisant les valeurs $\beta = 5$ et $\delta = 20$, est davantage dirigé par la matrice de retard dans la recherche de solutions. Les autres processeurs sont distribués uniformément entre ces deux extrêmes.

Comme on peut le constater dans la première partie du Tableau 4.10, la diversification des paramètres de visibilité a effectivement permis d'obtenir de meilleurs résultats pour chacune des quatre stratégies d'échange d'information. Cette fois-ci, les stratégies d'échange ont permis, en comparaison avec la version OCF_{P0}, une réduction du retard moyen variant de 17 à 21 jours environ. Les tests statistiques présentés dans la seconde partie du Tableau 4.10 confirment une amélioration de qualité de solutions lorsqu'on utilise une visibilité variable.

	Problème 70							
	OCF_{P1}		OCF_{P2}		OCF_{P3}		OCF_{P4}	
	Fixe	**Var.**	**Fixe**	**Var.**	**Fixe**	**Var.**	**Fixe**	**Var.**
Retard moyen	*115.85*	*107.43*	*118.08*	*108.92*	*117.39*	*107.66*	*117.51*	*105.11*
Variance	*3.89*	*5.44*	*3.76*	*6.05*	*6.51*	*7.41*	*7.96*	*6.04*
Moyenne des rangs	*14.7*	*6.3*	*14.6*	*6.4*	*13.6*	*7.4*	*14.3*	*6.7*
Variance des rangs	*17.79*	*16.90*	*14.93*	*21.60*	*20.93*	*31.60*	*20.46*	*21.34*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*4.51*		*4.290*		*2.705*		*3.717*	
P(T<=t) bilatéral	*0.000*		*0.000*		*0.014*		*0.002*	

Tableau 4.10 ***Tests paramétriques t d'égalité sur la moyenne des rangs pour deux stratégies de visibilité pour chacune des quatre versions utilisant des stratégies d'échange d'information sur le problème de taille 70. Le seuil utilisé pour les tests est de 1%.***

On observe des résultats similaires pour d'autres problèmes tests. L'utilisation d'une visibilité variable est un autre élément inclus dans la configuration servant à réaliser des essais numériques sur l'ensemble des 8 problèmes tests.

À cet effet, le Tableau 4.11 présente les résultats moyens obtenus sur l'objectif de minimisation du retard pour les 8 problèmes tests en comparaison avec l'algorithme OCF_{P0}. La configuration des versions parallèles avec échange d'information inclut l'échange d'une seule solution (OCF_{P1}, OCF_{P2}, OCF_{P3}) ou de deux solutions (OCF_{P4}) selon le cas, un échange à tous les cycles et une visibilité variable selon les caractéristiques précisées à la section précédente.

Taille du problème	OCF_{P0}	OCF_{P1}	OCF_{P2}	OCF_{P3}	OCF_{P4}
10	6.76 (0.08)	6.85 (0.13)	6.78 (0.10)	6.78 (0.10)	6.80 (0.12)
20	16.36 (0.12)	15.99 (0.68)	15.91 (0.58)	16.14 (0.30)	**15.54** (0.56)
30	26.54 (0.31)	**24.34** (0.59)	**24.70** (0.90)	**24.60** (0.70)	**24.65** (0.54)
40	50.37 (2.24)	**41.93** (4.56)	**45.23** (2.89)	**44.15** (2.71)	**41.45** (3.97)
50	18.87 (0.50)	18.85 (1.44)	**17.90** (1.18)	18.54 (1.30)	18.45 (1.19)
60	29.67 (1.57)	28.41 (2.96)	27.62 (1.92)	29.29 (1.55)	**26.89** (1.36)
70	126.77 (6.29)	**107.43** (5.44)	**108.92** (6.05)	**107.66** (7.41)	**105.11** (6.04)
80	189.17 (6.51)	**160.45** (7.59)	**167.34** (8.29)	**162.63** (8.57)	**155.94** (6.68)

Tableau 4.11 Retard moyen et écart type obtenus à l'aide des 4 stratégies d'échange d'information sur les problème de taille 10 à 80.

Ces résultats permettent de constater que les stratégies parallèles produisent de meilleurs résultats moyens que ceux de l'algorithme parallèle de base OCF_{P0} (ou séquentiel) pour tous les problèmes à l'exception de celui de 10 commandes. Plus spécifiquement, les tests statistiques présentés au Tableau 4.12 démontrent que les 4 stratégies permettent une amélioration significative de l'objectif de retard total pour les problèmes de 30 commandes et plus à l'exception du problème de 50 commandes où seule la stratégie P2 améliore significativement l'objectif et pour le problèmes de 60 commandes où seule la stratégie P4 améliore significativement le retard. La quatrième stratégie permet également une amélioration significative du problème de 20 commandes. Les résultats significatifs sont indiqués en caractères gras dans le Tableau 4.11. On peut également

constater que la quatrième stratégie donne généralement les meilleurs résultats. En ce qui concerne les problèmes de 50 et 60 commandes, des essais numériques ont montré qu'il est possible de trouver des paramètres (β,δ) associés à chaque processeur qui font en sorte que les quatre stratégies donnent des résultats significativement meilleurs que la version séquentielle. Toutefois, cette étude a été réalisée en utilisant des paramètres identiques pour résoudre les 8 problèmes sans chercher à calibrer les problèmes de façon individuelle. Cette manière de procéder est guidée par la possibilité d'intégrer certains éléments de cette étude au logiciel utilisé par l'entreprise et ne peut donc pas être basée sur du cas par cas. D'autres études pourront permettre une meilleure compréhension de ces phénomènes et possiblement l'amélioration de l'algorithme.

	Problème 10							
	OCF_{P0}	OCF_{P1}	OCF_{P0}	OCF_{P2}	OCF_{P0}	OCF_{P3}	OCF_{P0}	OCF_{P4}
Retard moyen	*6.76*	*6.85*	*6.76*	*6.78*	*6.76*	*6.78*	*6.76*	*6.80*
Variance	*0.08*	*0.13*	*0.08*	*0.10*	*0.08*	*0.10*	*0.08*	*0.12*
Moyenne des rangs	*8.75*	*12.25*	*10.4*	*10.6*	*10.4*	*10.6*	*9.85*	*11.15*
Variance des rangs	*14.29*	*30.63*	*16.27*	*19.60*	*16.27*	*19.60*	*15.56*	*25.73*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*-1.65*		*-0.11*		*-0.11*		*-0.64*	
P(T<=t) bilatéral	*0.116*		*0.917*		*0.917*		*0.530*	

	Problème 20							
	OCF_{P0}	OCF_{P1}	OCF_{P0}	OCF_{P2}	OCF_{P0}	OCF_{P3}	OCF_{P0}	OCF_{P4}
Retard moyen	*16.36*	*15.99*	*16.36*	*15.91*	*16.36*	*16.14*	*16.36*	*15.54*
Variance	*0.12*	*0.68*	*0.12*	*0.58*	*0.12*	*0.30*	*0.12*	*0.56*
Moyenne des rangs	*12.3*	*8.7*	*12.8*	*8.2*	*13.2*	*7.8*	*14.4*	*6.6*
Variance des rangs	*13.57*	*43.63*	*13.96*	*41.23*	*17.96*	*17.23*	*10.49*	*25.66*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*1.51***		*1.96*		*2.54*		*4.10*	
P(T<=t) bilatéral	*0.154*		*0.066*		*0.021*		*0.001**	

	Problème 30							
	OCF_{P0}	OCF_{P1}	OCF_{P0}	OCF_{P2}	OCF_{P0}	OCF_{P3}	OCF_{P0}	OCF_{P4}
Retard moyen	*26.54*	*24.34*	*26.54*	*24.70*	*26.54*	*24.60*	*26.54*	*24.65*
Variance	*0.31*	*0.59*	*0.31*	*0.90*	*0.31*	*0.70*	*0.31*	*0.54*
Moyenne des rangs	*15.5*	*5.5*	*14.9*	*6.1*	*15.3*	*5.7*	*15.3*	*5.7*
Variance des rangs	*9.17*	*9.17*	*16.54*	*14.27*	*11.12*	*11.34*	*11.57*	*11.12*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*7.39*		*5.01*		*6.40*		*6.37*	
P(T<=t) bilatéral	*0.000**		*0.000**		*0.000**		*0.000**	

	Problème 40							
	OCF_{P0}	OCF_{P1}	OCF_{P0}	OCF_{P2}	OCF_{P0}	OCF_{P3}	OCF_{P0}	OCF_{P4}
Retard moyen	*50.37*	*41.93*	*50.37*	*45.23*	*50.37*	*44.15*	*50.37*	*41.45*
Variance	*2.24*	*4.56*	*2.24*	*2.89*	*2.24*	*2.71*	*2.24*	*3.97*
Moyenne des rangs	*15.5*	*5.5*	*15.0*	*6.0*	*15.4*	*5.6*	*15.0*	*6.0*
Variance des rangs	*9.17*	*9.17*	*15.56*	*13.33*	*10.27*	*10.27*	*12.22*	*16.67*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*7.39*		*5.30*		*6.84*		*5.30*	
P(T<=t) bilatéral	*0.000**		*0.000**		*0.000**		*0.000**	

	Problème 50							
	OCF_{P0}	OCF_{P1}	OCF_{P0}	OCF_{P2}	OCF_{P0}	OCF_{P3}	OCF_{P0}	OCF_{P4}
Retard moyen	*18.87*	*18.85*	*18.87*	*17.90*	*18.87*	*18.54*	*18.87*	*18.45*
Variance	*0.50*	*1.44*	*0.50*	*1.18*	*0.50*	*1.30*	*0.50*	*1.19*
Moyenne des rangs	*11.5*	*9.5*	*14.0*	*7.0*	*10.7*	*10.3*	*12.3*	*8.7*
Variance des rangs	*18.06*	*53.61*	*14.89*	*31.67*	*22.68*	*50.90*	*25.34*	*41.29*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*0.75*		*3.24*		*0.15*		*1.39*	
P(T<=t) bilatéral	*0.464*		*0.005**		*0.884*		*0.180*	
	Problème 60							
	OCF_{P0}	OCF_{P1}	OCF_{P0}	OCF_{P2}	OCF_{P0}	OCF_{P3}	OCF_{P0}	OCF_{P4}
Retard moyen	*29.67*	*28.41*	*29.67*	*27.62*	*29.67*	*29.29*	*29.67*	*26.89*
Variance	*1.57*	*2.96*	*1.57*	*1.92*	*1.57*	*1.55*	*1.57*	*1.36*
Moyenne des rangs	*11.7*	*9.3*	*13.7*	*7.3*	*11.1*	*9.9*	*14.9*	*6.1*
Variance des rangs	*24.90*	*45.79*	*22.9*	*28.23*	*37.88*	*35.21*	*16.54*	*14.32*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*0.90*		*2.83*		*0.44*		*5.01*	
P(T<=t) bilatéral	*0.379*		*0.011*		*0.662*		*0.000**	

	Problème 70							
	OCF_{P0}	OCF_{P1}	OCF_{P0}	OCF_{P2}	OCF_{P0}	OCF_{P3}	OCF_{P0}	OCF_{P4}
Retard moyen	*127.73*	*107.43*	*127.73*	*108.92*	*127.73*	*107.66*	*127.73*	*105.11*
Variance	*3.77*	*5.44*	*3.77*	*6.05*	*3.77*	*7.41*	*3.77*	*6.04*
Moyenne des rangs	*15.5*	*5.5*	*15.5*	*5.5*	*15.5*	*5.5*	*15.5*	*5.5*
Variance des rangs	*9.17*	*9.17*	*9.17*	*9.17*	*9.17*	*9.17*	*9.17*	*9.17*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*7.39*		*7.39*		*7.39*		*7.39*	
P(T<=t) bilatéral	*0.000**		*0.000**		*0.000**		*0.000**	

	Problème 80							
	OCF_{P0}	OCF_{P1}	OCF_{P0}	OCF_{P2}	OCF_{P0}	OCF_{P3}	OCF_{P0}	OCF_{P4}
Retard moyen	*189.17*	*160.45*	*189.17*	*167.34*	*189.17*	*162.63*	*189.17*	*155.94*
Variance	*6.51*	*7.59*	*6.51*	*8.29*	*6.51*	*8.57*	*6.51*	*6.68*
Moyenne des rangs	*15.5*	*5.5*	*15.4*	*5.6*	*15.5*	*5.5*	*15.5*	*5.5*
Variance des rangs	*9.17*	*9.17*	*10.27*	*10.27*	*9.17*	*9.17*	*9.17*	*9.17*
Observations	*10*	*10*	*10*	*10*	*10*	*10*	*10*	*10*
Statistique t	*7.39*		*6.84*		*7.39*		*7.39*	
P(T<=t) bilatéral	*0.000**		*0.000**		*0.000**		*0.000**	

**significatif* ***t-test effectué avec variances inégales*

Tableau 4.12 ***Tests paramétriques t d'égalité sur la moyenne des rangs entre la version parallèle de base (OCF_{P0}) et les quatre versions basées sur des stratégies d'échange d'information sur les 8 problèmes tests. Le seuil utilisé pour les tests est de 1%.***

4.4.2.5.5 La contribution des processeurs lors de la recherche

Le Tableau 4.11 présenté à la section précédente fournit des résultats moyens obtenus avec l'utilisation de 10 processeurs pour chaque version de l'algorithme. Pendant ces essais numériques, il a été permis de constater l'obtention occasionnelle de très bonnes solutions se démarquant des moyennes obtenues. Ce phénomène laisse entrevoir la possibilité d'améliorations supplémentaires significatives. À cet effet, une analyse de la contribution des processeurs lors de la recherche a été effectuée dans le but de découvrir des pistes pouvant mener à l'obtention régulière de solutions de cette qualité.

Cette analyse a permis de découvrir un phénomène intéressant : en début d'exécution, les processeurs dont les paramètres de visibilité sont les plus élevés contribuent davantage à l'optimisation. Cependant, en fin d'exécution, la contribution provient des processeurs dont les paramètres de visibilité sont les plus bas. La Figure 4.16 illustre ce phénomène. À chaque cycle où la meilleure solution connue est améliorée, le processeur ayant obtenu la meilleure solution est représenté par un point dans le graphique. Les processeurs sont ordonnés selon le degré d'influence de leurs paramètres de visibilité, le processeur p_0 étant le moins influencé et le processeur p_9 étant le plus influencé.

Ces résultats permettent également de constater une durée de contribution relativement limitée de la part de plusieurs processeurs. Les processeurs 4 à 9 sont très utiles pour découvrir des solutions de départ et ainsi alimenter la recherche des processeurs 0 à 3, mais ils deviennent rapidement inutiles à la recherche. À l'inverse, les processeurs 0 à 3 apportent peu de contribution en début d'exécution, mais effectuent tout le travail efficace une fois alimentés par les processeurs 4 à 9.

D'autres tests effectués après la découverte de ce phénomène ont révélé que ni le groupe de processeurs numérotés de 4 à 9, ni le groupe de processeurs numérotés de 0 à 3 n'était en mesure d'atteindre à lui seul la qualité de solution obtenue dans la configuration actuelle de l'algorithme. Il semble donc que la présence des deux groupes soit nécessaire.

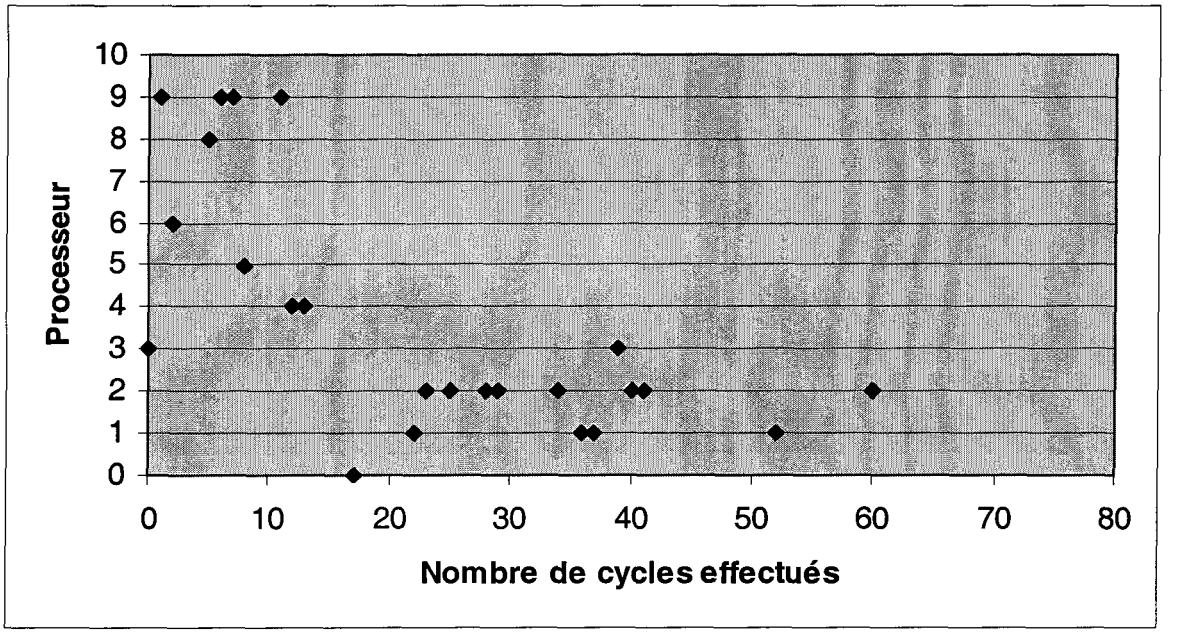

Figure 4.16 Contribution des processeurs lors de l'exécution de l'algorithme

Dans le but d'augmenter le rendement et l'utilité des processeurs aux divers stades de l'algorithme, il semblait approprié de modifier dynamiquement leurs paramètres de visibilité en leur affectant des paramètres élevés en début d'exécution, puis de les abaisser après un certain nombre de cycles. Il était espéré que, de cette façon, la recherche serait améliorée à chacun des stades d'exécution. Ces modifications ont apporté des résultats intéressants dans certains cas. Par exemple, en utilisant la stratégie P4 pour le problème de 70 commandes, il a été possible d'obtenir un retard moyen de 101.37 sur dix exécutions en attribuant des paramètres élevés à tous les processeurs durant les dix premiers cycles et en les abaissant ensuite jusqu'à la fin de l'exécution. Les cas semblables sont cependant limités à des configurations spécifiques, ce qui laisse supposer que la dynamique de coopération soit plus complexe qu'elle ne le semblait à première vue. Certaines avenues

présentant un potentiel non négligeable ont tout de même été identifiées. Une étude plus approfondie pourrait permettre d'exploiter davantage ces avenues, de mieux comprendre la dynamique générale de coopération et d'en arriver à un meilleur algorithme de résolution.

4.4.2.6 Sommaire

Dans cette section, différentes stratégies d'échange d'information relatives à la parallélisation en processus coopératifs de l'OCF_{S2} ont été présentées et appliquées à la résolution d'un problème d'ordonnancement industriel réel. Ces stratégies ont fait l'objet d'adaptations afin d'être implantées sur une architecture à mémoire partagée à l'aide de l'environnement de programmation OpenMP.

La qualité des solutions obtenues comparativement à l'algorithme séquentiel et la découverte de nouveaux minimums pour plusieurs problèmes ont montré l'efficacité de telles stratégies de parallélisation. Les résultats obtenus laissent croire que la performance pourrait être encore plus convaincante dans la résolution de problèmes de plus grande taille et justifient le déploiement de travaux ultérieurs en ce sens.

De plus, des travaux exploratoires ont permis d'identifier plusieurs éléments jouant un rôle important dans la performance de l'algorithme. Des études plus approfondies principalement axées sur le nombre de processeurs utilisés, les paramètres de l'OCF séquentiel et la configuration des échanges d'information sont d'un intérêt certain lorsque considérés individuellement. Toutefois, il semble que les interrelations entre ces éléments jouent un rôle encore plus déterminant sur la performance de l'algorithme parallèle. Ces relations représentent alors des avenues de recherche d'autant plus intéressantes.

Seulement quelques stratégies d'échange d'information ont été considérées dans ce travail, mais il est clair que la grande diversité des méthodes de coopération possibles reflète un potentiel considérable pour de futurs travaux.

4.5 Conclusion

Le présent chapitre, en présentant une résolution efficace d'un problème d'ordonnancement industriel à l'aide de deux versions parallèles de l'OCF, a montré que l'utilisation du parallélisme à l'intérieur d'une métaheuristique peut améliorer de façon appréciable la performance de l'optimisation et ce, autant au niveau du temps d'exécution qu'à celui de la qualité des solutions trouvées.

Dans un contexte où la méthode de résolution séquentielle originale ne tenait aucunement compte d'une future exécution parallèle, le parallélisme a pu être intégré sans obstacles majeurs. Le potentiel de parallélisation présenté par l'OCF et la convivialité de l'environnement de programmation parallèle OpenMP ont d'autant plus facilité cette intégration.

Une stratégie de parallélisation interne de l'OCF sur une architecture à mémoire partagée a été développée. L'OCF parallèle résultant a présenté des mesures d'accélération et d'efficacité significatives et ce, dans un contexte d'optimisation d'un problème réel présentant des contraintes spécifiques.

De plus, l'introduction de stratégies de coopération à l'OCF a permis d'améliorer la qualité des solutions trouvées à un niveau n'ayant pu être atteint précédemment par la métaheuristique séquentielle. Aucun effort d'adaptation et de paramétrage n'a été

nécessaire pour constater une amélioration de l'optimisation, et des efforts limités ont permis d'obtenir une certaine régularité dans l'obtention de solutions de qualité.

CHAPITRE 5

CONCLUSION

Le travail réalisé dans ce mémoire a permis de mieux comprendre comment l'informatique, par le biais du parallélisme, peut apporter une contribution importante au domaine des métaheuristiques et à celui de l'optimisation combinatoire. Les métaheuristiques et le parallélisme représentent toutefois des domaines encore très jeunes et leur coexistence à l'intérieur d'une même méthode de résolution en est encore à ses premiers balbutiements. Chose certaine, ils présentent un potentiel fort prometteur qui justifie le déploiement de travaux de recherche futurs et ce, autant d'un point de vue informatique que du côté de la recherche opérationnelle.

Ce mémoire a permis d'apporter une certaine contribution à ces domaines à travers les réponses apportées aux deux objectifs fixés au départ de cette recherche. Le premier objectif de la recherche consistait à développer une stratégie de bas niveau pour paralléliser la métaheuristique d'Optimisation par Colonies de Fourmis sur une architecture parallèle à mémoire partagée et d'appliquer l'algorithme résultant à la résolution d'un problème d'optimisation combinatoire réel. L'élaboration et l'implantation d'un algorithme parallèle (OCF_P) dans le logiciel d'ordonnancement industriel a montré que des mesures d'accélération et d'efficacité significatives pouvaient être obtenues sans apporter de modifications fondamentales à l'algorithme séquentiel et au programme initial. En effet, par une manipulation judicieuse des principales procédures de calcul et des structures de données de l'algorithme, il a été possible de transposer efficacement le programme séquentiel dans un modèle d'architecture à mémoire partagée. Les fonctionnalités offertes par OpenMP ont permis de refléter cette transposition dans un programme d'application qui n'avait pas été conçu initialement à des fins de parallélisation et qui présentait certaines

contraintes pratiques. Les résultats obtenus justifient la pertinence de l'approche et les limites identifiées suscitent un intérêt certain pour de futurs travaux de recherche et de développement.

Le deuxième objectif de ce travail visait à introduire une stratégie de processus coopératifs à la métaheuristique d'Optimisation par Colonies de Fourmis afin d'améliorer la qualité des solutions trouvées au problème d'ordonnancement industriel. Par l'adaptation de stratégies d'échange d'information retrouvées dans la littérature au contexte spécifique d'ordonnancement, il a été possible de surpasser significativement la qualité de l'optimisation d'une méthode séquentielle ayant déjà fait l'objet de nombreux travaux et expérimentations. Des gains allant jusqu'à 18% ont été constatés et de nouveaux minimums ont été obtenus pour plusieurs problèmes tests. Le gain de performance obtenu en ajoutant une forme d'apprentissage additionnelle à un algorithme séquentiel déjà performant montre la puissance et le potentiel de la coopération entre plusieurs agents.

Ce mémoire représentait une première exploration du potentiel des métaheuristiques parallèles pour la résolution d'un problème spécifique. De futurs projets permettront de mieux comprendre la complexité de leur fonctionnement, d'améliorer leur rendement et de les appliquer à des problèmes diversifiés. Ce travail de recherche a donc non seulement atteint ses objectifs, mais a également ouvert la voie à plusieurs avenues de recherche futures en raison des nombreuses interrogations soulevées et de certaines limites constatées.

Les expérimentations au niveau de la stratégie de parallélisation interne ont donné de bons résultats en utilisant un nombre de processeurs variant de 1 à 8, mais une dégradation majeure a été constatée avec un plus grand nombre de processeurs. Ce phénomène pourrait

être dû notamment à certaines limitations techniques de nature matérielle et logicielle. Une évaluation de la performance globale du programme pourrait être de mise à ce stade du développement, de même qu'une analyse plus détaillée de son exécution sur l'ordinateur parallèle utilisé. De plus, l'exploitation de certaines fonctionnalités avancées de l'environnement OpenMP, permettant un plus grand contrôle sur la gestion de la mémoire virtuellement partagée, pourrait également mener à un programme plus performant. Notons également que la technologie OpenMP en est encore à ses tous débuts et qu'elle n'a atteint qu'une partie de son potentiel, ce qui peut influencer de façon plus ou moins directe la performance des programmes basés sur celle-ci. Somme toute, il serait intéressant d'étudier plus en profondeur l'ensemble de ces aspects techniques.

Les stratégies de parallélisation développées ont apporté des résultats significatifs dans la résolution des problèmes tests actuels, mais tout laisse croire que leur performance pourrait être encore plus convaincante dans la résolution de problèmes de plus grande taille. Cette idée n'ayant pu être vérifiée en raison des contraintes logicielles actuelles, l'extension de l'application pourrait être envisagée à court terme. Le traitement de problèmes de plus grande taille pourrait également permettre de vérifier s'il est possible d'obtenir de meilleures performances en utilisant un nombre de processeurs plus grand que 8 pour la parallélisation interne.

Un des aspects les plus intéressants issus de ce travail est le phénomène de coopération, autant par sa puissance que par sa complexité. Une meilleure exploitation de ce concept, notamment par l'élaboration de stratégies plus intelligentes, dynamiques et adaptatives, s'avère d'un intérêt certain. La formalisation et la définition de modèles de coopération

indépendants des méthodes de résolution et l'élaboration d'approches dynamiques s'adaptant au processus de recherche de solutions pourraient permettre de tirer davantage profit du potentiel offert par la coopération.

Dans ce travail, l'optimisation multi-objectifs a été mise de côté et les efforts ont été concentrés sur l'optimisation d'un seul objectif. Il demeure toutefois que cet aspect d'un problème d'ordonnancement et de plusieurs autres problèmes d'optimisation combinatoire présente un grand intérêt au niveau de la parallélisation et surtout de la coopération. Par exemple, l'utilisation de groupes de processeurs optimisant des objectifs différents, associée à la définition de règles de coopération adéquates, pourrait être une façon intéressante de mieux couvrir l'espace de solution d'un problème multi-objectifs.

L'atteinte des objectifs de recherche de ce mémoire a passé par la parallélisation de versions différentes de l'OCF selon des paradigmes distincts. L'intégration de ces deux formes de parallélisation à l'intérieur d'un seul algorithme permettrait de tirer avantage des deux approches en même temps. Il reste que les améliorations apportées à la version séquentielle OCF_{SI} au niveau des mises à jour locale et globale de la trace viennent modifier le comportement de l'algorithme et la nature des dépendances entre ses procédures de calcul. Une étude sur cet aspect, de même que l'étude plus générale de stratégies de parallélisation à multiples niveaux à l'intérieur des métaheuristiques, s'avèrent des sujets très intéressants.

Finalement, l'étude du comportement de l'OCF parallèle et les gains obtenus en utilisant des paramètres variables de visibilité a ouvert la voie à une meilleure compréhension du comportement de l'algorithme séquentiel. Dans l'immédiat, des travaux

visant à ajuster dynamiquement les paramètres de visibilité pourraient permettre d'obtenir un algorithme séquentiel plus performant. De cette façon, ce travail aura alors permis des retombées immédiates dans le monde industriel dans l'attente de la disponibilité d'une technologie parallèle à coûts abordables.

BIBLIOGRAPHIE

OpenMP (1998). OPENMP ARCHITECTURE REVIEW BOARD. OpenMP C and C++ Application Program Interface Version 1.0.

Agha, G. (1986). ACTORS: A Model of Concurrent Computation in Distributed Systems. Cambridge, Massachusetts, MIT Press.

Akl, S. G. (2000). The Design of Efficient Parallel Algorithms. Handbook on Parallel and Distributed Processing. J. Blazewicz, Ecker, K, Plateau, B., Trystram, D., Springer.

Amdahl, G. M. (1967). Validity of the single processor approach to achieving large scale computer capabilities. Proceedings AFIPS Spring Joint Computer Conference, Atlantic City, N. J.

Bachelet, V., Hafidi, Z., Preux, P. et Talbi, E.-G. (1998). "Vers la coopération des métaheuristiques." Calculateurs parallèles, réseaux et systèmes répartis 10(2).

Bullnheimer, B., Kotsis, G. et Strauss, C. (1998). Parallelization strategies for an ant system. High performance algorithms and software in nonlinear optimization. A. M. R. De Leone, P. Pardalos, and G. Toraldo, Kluwer, Dordrecht. 24 of Applied optimization: 87-100.

Calégari, P. (1999). Parallelization of population-based evolutionary algorithms for combinatorial optimization problems. Lausanne, Suisse, Swiss Federal Institute of Technology (EPFL).

Cantú-Paz, E. (1998). "A survey of parallel genetic algorithms." Calculateurs parallèles, réseaux et systèmes répartis 10(2): 141-171.

Clearwater, S. H., Hogg, T. et Huberman, B. A. (1992). Cooperative Problem Solving. Computation: The Micro and the Macro view. B. A. Huberman., World Scientific: 33-70.

Codenotti, B. et Leoncini, M. (1993). Introduction to Parallel Processing, Addison-Wesley.

Colorni, A., Dorigo, M., Maffioli, F., Maniezzo, V., Righini, G. et Trubian, M. (1996). "Heuristics from nature for hard combinatorial optimization problems." International Transactions in Operational Research 3(1): 1-21.

Colorni, A., Dorigo, M. et Maniezzo, V. (1991). Distributed optimization by ant-colonies. Proceedings of the first European Conference on Artificial Life (ECAL'91), edited by F. Verela and P. Bourgine, 134-142. Cambridge, Mass, USA, MIT Press.

Conover, W. J. et Iman, R. L. (1981). "Rank transformations as a bridge between parametric and nonparametric statistics." The American Statistician 35: 124-129.

Cormen, T., Leiserson, C. et Rivest, R. (1994). Introduction à l'algorithmique, Dunod.

Cosnard, M. et Trystram, D. (1995). Parallel Algorithms and Architectures, International Thomson Computer Press.

Crainic, T. G. et Toulouse, M. (1998). Parallel Metaheuristics. Fleet Management and Logistics. T. G. C. a. G. Laporte. Norwell, MA., Kluwer Academic: 205-251.

Crainic, T. G., Toulouse, M. et Gendreau, M. (1997). "Towards a taxonomy of parallel tabu search." INFORMS Journal on Computing 9(1): 61-72.

Cung, V.-D., Martins, S. L., Ribeiro, C. C. et Roucairol, C. (2001). Strategies for the parallel implementation of metaheuristics. Essays and Surveys in Metaheuristics. C. C. R. a. P. Hansen, Kluwer: 263-308.

Davis, L. (1991). Hanbook of Genetic Algorithms. New York, Van Nostrand Reinhold.

Delisle, P., Krajecki, M., Gravel, M. et Gagné, C. (2001). Parallel implementation of an ant colony optimization metaheuristic with OpenMP. International Conference on Parallel Architectures and Compilation Techniques, Proceedings of the 3rd European Workshop on OpenMP (EWOMP'01), Barcelone, Espagne.

Deneubourg, J. L. et Goss, S. (1989). "Collective patterns and decision-making." Ethology & Evolution 1: 295-311.

Deneubourg, J. L., Pasteels, J. M. et Verhaeghe, J. C. (1983). "Probabilistic behaviour in ants: A strategy of errors?" Journal of Theorical Biology 105: 259-271.

Dorigo, M. (1992). Optimization, learning and natural algorithms. Italy, Ph.D. Thesis, Politecnico di Milano.

Dorigo, M. et Di Caro, G. (1999). The Ant Colony Optimization Meta-Heuristic. New Ideas in Optimization. D. Corne, Dorigo, M. and Glover, F., McGraw-Hill.

Dorigo, M. et Gambardella, L. M. (1997a). "Ant colonies for the traveling salesman problem." BioSystems 43: 73-81.

Dorigo, M. et Gambardella, L. M. (1997b). "Ant Colony System: A Cooperative Learning Approach to the Traveling Salesman Problem." IEEE Transactions on Evolutionary Computation 1: 53-66.

Dorigo, M., Maniezzo, V. et Colorni, A. (1991). Positive feedback as a search strategy, Technical Report No 91-016, Politecnico di Milano, Italy: 20 pages.

Du, J. et Leung, J. Y. (1990). "Minimizing total tardiness on one machine is NP-hard." Mathematics of Operations Research 15: 483-494.

Eksioglu, S. D., Pardalos, P. M. et Resende, M. G. C. (2001). Parallel metaheuristics for combinatorial optimization. Advanced Algorithmic Techniques for Parallel Computation with Applications. R. C. e. al., Kluwer Academic Publishers.

Feo, T. A. et Resende, M. G. C. (1988). "A probabilistic heuristic for a computationally difficult set covering problem." Operations Research Letters 8: 767-771.

Feo, T. A. et Resende, M. G. C. (1995). "Greedy Randomized Adaptive Search Procedures." Journal Of Global Optimization 6: 109-133.

Finkel, R. A. et Bentley, J. L. (1974). "Quad trees. A data structure for retrieval on composite keys." Acta Informatica 4: 1-9.

Flynn, M. J. (1966). Very High Speed Computing Systems. Proceedings IEEE.

Foster, I. (2000). Languages for Parallel Processing. Handbook on Parallel and Distributed Processing. J. Blazewicz, Ecker, K, Plateau, B., Trystram, D., Springer.

Gagné, C. (2001). L'ordonnancement industriel : stratégies de résolution métaheuristiques et objectifs multiples. Faculté des sciences de l'administration, Université Laval.

Gagné, C., Gravel, M. et Price, W. L. (2001a). Extension de l'algorithme d'optimisation par colonie de fourmis pour la résolution d'un problème d'ordonnancement industriel, Document de travail 2001-007, Faculté des Sciences de l'Administration, Université Laval, Canada. (soumis pour publication).

Gagné, C., Price, W. L. et Gravel, M. (2001b). "Comparing an ACO algorithm with other heuristics for the single machine scheduling problem with sequence dependent setup times." Journal of the Operational Research Society (à paraître).

Garey, M. S. et Johnson, D. S. (1979). Computer and Intractability : A Guide to the Theory of NP-Completeness. New York, W.H. Freeman and Co.

Gentler, M., Ubéda, S. et Desprez, F. (1996). Initiation au parallélisme. Concepts, architectures et algorithmes, Masson.

Germain-Renaud, C. et Sansonnet, J.-P. (1992). Les ordinateurs massivement parallèles, Armand Colin.

Glover, F. (1986). "Futur paths for integer programming and links to artificial intelligence." Computers and Operations Research 5: 533-549.

Glover, F. (1989). "Tabu search – Part I." ORSA Jounal on Computing 1: 190-206.

Glover, F. (1990). "Tabu search – Part II." ORSA Jounal on Computing 2: 4-32.

Goss, S., Beckers, R., Deneubourg, J. L., Aron, S. et Pasteels, J. M. (1990). How trail laying and trail following can solve foraging problems for ant colonies. Behavioural Mechanisms of Food Selection. R. N. Hughes. Berlin, Springler-Verlag. Vol. G20.

Gravel, M., Price, W. L. et Gagné, C. (2000). "Scheduling jobs in a Alcan aluminium factory using a genetic algorithm." International Journal of Production Research 38(13): 3031-3041.

Gravel, M., Price, W. L. et Gagné, C. (2002). "Scheduling continuous casting of aluminum using a multiple-objective ant colony optimization metaheuristic." European Journal of Operational Research(to appear).

Hilzer, R. C. et Crowl, L. A. (1995). A Survey of Sequential and Parallel Implementation Techniques for Functional Programming Languages, Department of Computer Science, Oregon state University.

Holland, J. H. (1975). Adaptation in natural and artificial systems. Ann Arbor, Michigan, The University of Michigan Press.

Huberman, B. A. (1990). "The performance of Cooperative Processes." Physica D 42: 38-47.

JàJà, J. (1992). An Introduction to Parallel algorithms, Addison-Wesley.

Juillé, H. et Pollack, J. B. (1996). Dynamics of Co-evolutionary Learning. Proceedings of the Fourth International Conference on Simulation of Adaptive Behavior, Cape Cod, MA, MIT Press.

Kirkpatrick, S., Gelatt, J. C. D. et Vecchi, M. P. (1983). "Optimization by simulated annealing." Science 220: 671-680.

Koza, J. R. (1991). Genetic evolution and co-evolution of computer programs. Artificial Life II, SFI Studies in the Sciences of Complexity. C. Langton, Taylor, Charles, Farmer, J. Doyne, and Rasmussen, Steen. Redwood city, CA, Addison-Wesley. X: 603-629.

Leopold, C. (2001). Parallel and Distributed Computing: A Survey of Models, Paradigms and Approaches, Wiley-Interscience.

Metropolis, N., Rosenbluth, A., Rosenbluth, M., Teller, A. et Teller, E. (1953). "Equation of state calculations by fast computing machines." Journal of Chemical Physics 21: 1087-1092.

Michels, R. et Middendorf, M. (1999). An ant system for the shortest common supersequence problem. New Ideas in optimization. M. D. D. Corne, F. Glover, McGraw-Hill: 51-61.

Middendorf, M., Reischle, F. et Schmeck, H. (2000). Information Exchange in Multi Colony Ant Algorithms. Parallel and Distributed Computing, Proceedings of the 15 IPDPS 2000 Workshops, Third Workshop on Biologically Inspired Solutions to Parallel Processing Problems (BioSP3), Cancun, Mexico, Springer-Verlag.

Nemhauser, G. L. et Wolsey, L. A. (1988). Integer and Combinatorial Optimization. Chichester, Wiley.

Osman, I. H. et Laporte, G. (1996). "Metaheuristics: A bibliography." Annals of Operations Research 63: 513-623.

Papadimitriou, C. H. et Steiglitz, K. (1982). Combinatorial Optimization, Algorithms and Complexity. Englewood Cliffs, Prentice-Hall.

Plateau, B. et Trystram, D. (2000). Parallel and Distributed Computing: State-of-the-Art and Emerging Trends. Handbook on Parallel and Distributed Processing. J. Blazewicz, Ecker, K, Plateau, B., Trystram, D., Springer.

Roch, J. L., Villard, G. et Roucairol, C. (1995). Parallélisme et Applications Irrégulières. Algorithmes Irréguliers et Ordonnancement, Hermès: 73-88.

Soriano, P. et Gendreau, M. (1997). "Fondements et application des methodes de recherche avec tabous." RAIRO 31(2): 133-159.

Stützle, T. (1998). Parallelization strategies for ant colony optimization. Proceedings of parallel problem solving from nature -- PPSN-V, Amsterdam, Springler Verlag.

Stützle, T. et Hoos, H. (1998). Improvements on the Ant System: Introducing the MAX - MIN Ant System. Artificial Neural Networks and Genetic Algorithms. N. C. S. R.F. Albrecht G.D. Smith. New York, Springer Verlag: 245-249.

Taillard, E., Gambardella, L. M., Gendreau, M. et Potvin, J.-Y. (1998). Adaptive memory programming: a unified view of metaheuristics, Technical Report IDSIA-19-98, IDSIA, Lugano: 1-22.

Talbi, E.-G., Roux, O., Fonlupt, C. et Robillard, D. (1999). Parallel ant colonies for combinatorial optimization problems. BioSP3 Workshop on Biologically Inspired

Solutions to Parallel Processing Systems, in IEEE IPPS/SPDP'99 (Int. Parallel Processing Symposium / Symposium on Parallel and Distributed Processing), San Juan, Puerto Rico, USA, Springer-Verlag.

Taylor, s. (1989). Parallel Logic Programming Techniques. Englewood Cliff, New Jersey, Prentice-Hall.

Toulouse, M., Crainic, T. G. et Gendreau, M. (1996). Communication Issues in Designing Cooperative Multi-Thread Parallel Searches. Meta-Heuristics: Theory and Applications. Norwell MA, Kluwer Academic Publishers: 501-522.

Verhoeven, M. G. A. et Aarts, E. H. L. (1995). "Parallel Local Search." Journal of Heuristics 1: 43-65.

Widmer, M., Hertz, A. et Costa, D. (2001). Les métaheuristiques. In: Ordonnancement de la production. Paris, Hermès science publications: 55-93.

Wong, W. S. (1995). "Matrix representation and gradient flows for NP-hard problems." Journal of Optimization Theory and Applications 87(1): 197-220.