N° d'ordre : 4078

THESE

pour obtenir le grade de

DOCTEUR DE L'USTL
DISCIPLINE : INFORMATIQUE

présentée et soutenue publiquement,
le 27 Novembre 2007, par

Mohand Mezmaz

Une approche efficace pour le passage sur grilles de calcul de méthodes d'optimisation combinatoire

Jury

Président :	Frédéric Semet	Professeur, UVHC, Valenciennes
Rapporteurs :	Pierre Manneback	Professeur, FPMs, Belgique
	Stéphane Vialle	Professeur, SUPELEC, Metz
Examinateur :	Emmanuel Jeannot	Chargé de Recherche, INRIA, Nancy
Directeurs :	El-Ghazali Talbi	Professeur, USTL, Lille
	Nouredine Melab	Professeur, USTL, Lille

Université des Sciences et Technologies de Lille
Laboratoire LIFL/INRIA
Bâtiment A, Parc Scientifique de la Haute Borne, 40 avenue Halley
59655 Villeneuve d'Ascq Cedex

Table des matières

Introduction

La thèse, présentée dans ce document, s'inscrit dans les domaines de l'optimisation combinatoire et des grilles de calcul. Elle a été réalisée au sein de l'équipe d'Optimisation PArallèle Coopérative (OPAC) du Laboratoire d'Informatique Fondamentale de Lille (CNRS/LIFL) à l'Université des Sciences et Technologies de Lille (USTL). Les travaux de cette thèse entrent également dans le cadre du projet INRIA DOLPHIN.

Les problèmes d'optimisation combinatoire sont souvent NP-difficiles. Selon la qualité exigée des solutions, les méthodes de résolution de ces problèmes se déclinent en deux grandes catégories : les méthodes *approchées*, appelées aussi *heuristiques*, et les méthodes *exactes*. Il existe également des méthodes *hybrides* combinant différentes méthodes aux comportements complémentaires. Parmi les heuristiques, on distingue les *méta-heuristiques* pouvant s'appliquer à différents problèmes. Ces méta-heuristiques peuvent être également à population de solutions ou à solution unique. Les méta-heuristiques, souvent employées pour résoudre des problèmes de grande taille, produisent en temps raisonnable des solutions de bonnes qualités mais pas forcément optimales. Par contre, les méthodes exactes permettent de trouver des solutions optimales avec preuve d'optimalité. Cependant, leur coût exorbitant en temps CPU, sur des problèmes de grande taille, les rend souvent inexploitables faute de ressources de calcul suffisantes.

Le parallélisme à grande échelle, basé sur les grilles informatiques, s'avère un moyen indispensable pour supporter le coût des méthodes d'optimisation, notamment des méthodes exactes et des méthodes hybrides. Une grille peut être vue comme un ensemble de ressources de calcul hétérogènes, volatiles, réparties sur plusieurs domaines d'administration autonomes, et inter-connectés par un réseau à grande échelle. L'exploitation de cet environnement nécessite la gridification des modèles parallèles et des mécanismes d'hybridation en vue de leur adaptation aux caractéristiques des grilles de calcul. La prise en compte de telles caractéristiques se traduit notamment par la résolution des problèmes liés à la sécurité (traversée de pare-feux), à la volatilité (tolérance aux pannes), aux délais de communication (optimisation des coûts de communication) et à l'hétérogénéité (régulation de charge).

Dans le domaine de l'optimisation combinatoire parallèle, plusieurs travaux ont été consacrés à l'identification et à l'étude de différents modèles de parallélisation des méthodes exactes [GC94, CDC+94] et des méta-heuristiques [CP98, CMRR02, AT02,

TAML05]. Ces études sont riches en enseignements, y compris sur la parallélisation des méthodes hybrides. Cependant, elles ne sont pas totalement adaptées aux grilles informatiques. En effet, les caractéristiques des grilles ne sont souvent pas prises en compte dans ces études. Ceci s'explique essentiellement par la jeunesse du domaine des grilles informatiques. Dans cette thèse, nous proposons une étude des modèles parallèles et des mécanismes d'hybridation dans le contexte des grilles de calcul. Cette étude met notamment en évidence l'importance des modèles asynchrone et de vol de cycles dans un environnement d'exécution volatile, hétérogène et multi-domaine d'administration. Notre étude identifie également pour chaque modèle l'information à stocker et à restaurer pour mettre en œuvre des stratégies de tolérance aux pannes. Ceci permet de prendre en compte la volatilité liée à la disponibilité variable et aux pannes des machines. Nous nous sommes également intéressés aux mécanismes de régulation de charge dans le contexte des grilles informatiques, notamment pour les méthodes exactes.

Pendant une résolution, l'irrégularité de l'arbre exploré par les méthodes exactes, l'hétérogénéité des grilles informatiques et leur volatilité impliquent un nombre considérable d'opérations de régulation de charge et de sauvegarde/restauration. Toutes ces opérations se traduisent par des coûts de communication exorbitants pour le transfert, le stockage, et la restauration des unités de travail. Ces coûts sont accentués par l'échelle d'une grille et l'importance des délais de communication dans cet environnement. Il est donc primordial de proposer un codage de ces unités afin de réduire leur taille, et ainsi minimiser les coûts de communication. Dans cette thèse, nous proposons une gridification de l'algorithme *Branch-and-Bound* (*B&B*) basée sur un codage d'une unité de travail, souvent composée d'une liste de sous-problèmes, par un intervalle, défini avec deux entiers naturels. Dans cette nouvelle approche, appelée B&B@Grid, l'information transférée, stockée et restaurée est un intervalle au lieu d'une liste de sous-problèmes. Par conséquent, ce codage permet de minimiser le coût de communication, et rend les stratégies de régulation de charge, de tolérance aux pannes, et de détection de terminaison plus efficaces. De plus, à l'inverse des autres approches de régulation de charge publiées dans la littérature [ABGL02, AF02, CR95], la granularité des unités de travail dans B&B@Grid n'est pas fixe, mais varie selon les caractéristiques de la grille. Afin de faciliter l'utilisation de B&B@Grid, nous avons implémenté cette approche au travers d'une plate-forme logicielle. Contrairement aux autres plates-formes, la plate-forme B&B@Grid permet le déploiement de méthodes exactes de type B&B avec tout paradigme parallèle (fermier-travailleur, fermier-travailleur hiérarchique, pair-à-pair, etc.). Cette plate-forme permet de résoudre aussi bien les problèmes mono-critères que les problèmes multi-critères.

Pour sélectionner le prochain sous-problème à traiter, l'approche B&B@Grid est basée, à l'instar de la plupart des approches de parallélisation du B&B, sur la stratégie *profondeur d'abord*. L'intérêt de cette stratégie est de réduire le nombre de sous-problèmes se trouvant simultanément sur les différents processus parallèles. Par contre, comparée à la stratégie *meilleur d'abord*, le nombre de sous-problèmes traités, avec la stratégie *profondeur d'abord*, est considérable. L'initialisation des algorithmes B&B par

une (ou plusieurs) bonnes solution(s) permet de remédier à cet inconvénient. Cependant, de telles solutions ne sont pas disponibles pour certains problèmes. Par conséquent, il est nécessaire d'hybrider tout B&B parallèle, dont la stratégie est de type *profondeur d'abord*, par une méthode approchée. Le rôle de la méthode approchée est de fournir au B&B des solutions de bonne qualité afin de lui permettre d'élaguer le plus grand nombre de sous-problèmes. Dans cette thèse, nous avons proposé une approche pour l'hybridation de B&B@Grid avec une méta-heuristique parallèle sur les grilles de calcul.

Les intergiciels de grille permettent de résoudre, de manière transparente, une partie des problèmes rencontrés lors de déploiement d'applications sur cet environnement. Cependant, ils ne sont pas totalement adaptés au parallélisme coopératif, tel que celui supporté par l'approche hybride que nous avons proposée. Par exemple, les intergiciels de type *coordinateur-travailleur* (*dispatcher-worker*), tel que *XtremWeb* [Fed03], ne prennent souvent pas en compte la génération dynamique de tâches et leur coopération à l'exécution. Nous avons donc proposé pour ce type d'intergiciels un modèle de coopération inspiré de Linda [Gel85] et adapté à l'optimisation multi-critère sur grilles. Ce modèle peut être intégré comme couche de coordination dans ces intergiciels, et une première intégration de ce modèle a été réalisée sur *XtremWeb*.

Ce document est organisé en deux parties présentant le bilan de nos travaux et les contributions apportées. La première partie comporte trois chapitres consacrés aux méthodes exactes dans le contexte des grilles de calcul. Le **chapitre 1** explique brièvement les problèmes d'optimisation combinatoire ainsi que leurs méthodes de résolution, présente les grilles de calcul notamment leurs caractéristiques, et résume les principaux travaux effectués sur la gridification des algorithmes B&B afin de les adapter aux caractéristiques des grilles. Ce chapitre décrit également la grille d'expérimentation utilisée pour valider nos travaux. Le **chapitre 2** est consacré à la présentation de B&B@Grid. Ce chapitre décrit les concepts ainsi que les opérateurs définissant cette nouvelle approche, et détaille ses stratégies de tolérance aux pannes, d'équilibrage de charge, de détection de terminaison et de partage de solutions. Ce chapitre présente également le problème du Flow-Shop, problème utilisé pour valider nos travaux au cours de cette thèse, ainsi que les expérimentations effectuées pour montrer l'efficacité de l'approche B&B@Grid et la comparer aux principales approches de la littérature. Le **chapitre 3** porte sur la présentation de la plate-forme B&B@Grid supportant cette nouvelle approche. Dans ce chapitre, nous identifions les trois principales approches dédiées à la réutilisation du code de méthodes exactes parallèles tout en justifiant le choix de présenter l'approche à travers une plate-forme. Ensuite, ce chapitre présente brièvement les plates-formes les plus connues pour la parallélisation des méthodes exactes, décrit l'architecture de la plate-forme B&B@Grid, et présente ses mécanismes d'évaluation de performances. Le chapitre est terminé par la présentation des expérimentations effectuées pour valider B&B@Grid sur des problèmes mono-critères et bi-critères.

La deuxième partie, organisée en deux chapitres, porte sur la coopération des méthodes exactes avec les méta-heuristiques. Le **chapitre 4** présente les différents modèles

parallèles dédiés aux méta-heuristiques à population de solutions ou à solution unique, et justifie l'intérêt de combiner les différentes méthodes d'optimisation. Dans ce chapitre, nous présentons l'approche hybride que nous avons proposée, ses deux modes d'hybridation possibles, et son déploiement selon deux modèles parallèles. Le **chapitre 5** décrit notre modèle de coordination pour les méthodes parallèles hybrides déployées sur les grilles de calcul. Ce nouveau modèle est une extension de Linda. Ce chapitre présente le modèle de coordination Linda, justifie le choix de ce modèle, identifie ses limites quant à son utilisation dans les grilles, et présente le nouveau modèle de coordination Linda étendu ainsi que son implémentation sur XtremWeb. Dans ce chapitre, nous expliquons la mise en œuvre, avec le nouveau modèle de coordination, de trois modèles parallèles ainsi que de deux schémas d'hybridation. Ce chapitre est terminé par la présentation des expérimentations effectuées pour valider le nouveau modèle de coordination, et pour démontrer l'intérêt de notre approches hybrides.

Ce document comprend deux annexes. L'**annexe A** présente les problèmes d'ordonnancement en général, catégorie de problèmes à laquelle appartient le Flow-Shop, et les bornes de l'algorithme B&B utilisées dans nos expérimentations. L'**annexe B** décrit la façon d'utiliser la plate-forme B&B@Grid.

Première partie

Méthodes exactes sur grilles de calcul

Chapitre 1

Méthodes exactes parallèles sur grilles de calcul

Introduction

De nombreux problèmes, rencontrés dans différents secteurs économiques, sont de nature combinatoire. Selon le nombre de critères pris en compte, ces problèmes peuvent être mono ou multi-critères. Les méthodes de résolution de ces problèmes se déclinent en deux grandes familles : les méthodes approchées et les méthodes exactes. Contrairement aux méthodes approchées, les méthodes exactes, tels que les algorithmes B&B, permettent de trouver la ou les solution(s) optimale(s). Relativement à une énumération exhaustive de toutes les solutions, ces algorithmes réduisent considérablement la puissance de calcul nécessaire pour résoudre un problème d'optimisation combinatoire. Néanmoins, cette puissance reste considérable lorsqu'il s'agit de résoudre des instances de très grande taille. C'est pourquoi un recours aux grilles de calcul est nécessaire pour la résolution de telles instances. Les ressources d'une grille sont en général hétérogènes, volatiles, distribuées sur plusieurs domaines d'administration, et interconnectées avec un réseau de grande taille. L'adaptation des algorithmes B&B aux grilles de calcul nécessite la prise en compte de ces caractéristiques. Dans ce chapitre, nous présentons une étude sur la parallélisation des méthodes exactes sur les grilles informatiques ainsi que les principaux travaux réalisés dans ce domaine.

Ce chapitre est organisé en quatre sections. La **section 1.1** est une introduction à l'optimisation combinatoire mono et multi-critère. Nous y expliquons les notions de base de cette discipline. La **section 1.2** donne un aperçu des grilles de calcul ainsi que leurs caractéristiques, et décrit la grille d'expérimentation utilisée pour valider nos travaux. Dans la **section 1.3**, nous nous intéressons aux méthodes exactes parallèles. Cette section introduit notamment la taxinomie proposée par [Mel05]. La **section 1.4** décrit les travaux les plus connus pour la conception d'algorithmes B&B pour les grilles de calcul. Cette section présente les différentes approches proposées pour réduire les délais de communication, partitionner la charge entre les processus B&B, gérer la tolérance

aux pannes, prendre en charge le passage à l'échelle et la traversée de pare-feux.

1.1 Méthodes exactes

En *optimisation combinatoire*, appelée également optimisation discrète, on s'intéresse aux problèmes requérant de trouver la ou les meilleure(s) configuration(s) parmi un ensemble fini mais très grand de configurations possibles. Chacune des configurations constitue une *solution* au problème considéré et appartient à un espace appelé *espace des solutions*. Le choix de la solution est fonction de son coût, lui-même basé sur un ou plusieurs *critère(s)*. Le coût d'une solution est défini dans un autre espace appelé *espace des coûts*. Résoudre un *problème d'optimisation combinatoire* avec une méthode exacte revient à trouver la ou les solution(s) de coût *optimal*. En optimisation combinatoire, un problème peut être *mono-critère* ou *multi-critère*, selon qu'on s'intéresse respectivement à un ou plusieurs critère(s). Selon le nombre de critères du problème, le coût d'une solution peut être une simple valeur ou un vecteur de valeurs.

Si la notion d'optimalité est simple à définir dans le cas d'un problème mono-critère, elle n'est pas aussi évidente pour un problème multi-critère. Généralement, l'optimalité pour les problèmes multi-critères est définie à l'aide de la relation de *dominance* entre vecteurs [Par12].

Définition 1
Soient $X = (X_1, .., X_N)$ *et* $Y = (Y_1, .., Y_N)$ *deux vecteurs et deux entiers i,j* $\in$ *[1,N].*
Dans un problème de minimisation, X *domine* $Y \Longleftrightarrow (\forall i / X_i \leq Y_i) et (\exists j / X_j < Y_j)$.
Dans un problème de maximisation, X *domine* $Y \Longleftrightarrow (\forall i / X_i \geq Y_i) et (\exists j / X_j > Y_j)$.

La relation de dominance constitue un ordre partiel, ce qui implique que plusieurs solutions optimales peuvent exister. Cet ensemble est appelé *front Pareto*.

Définition 2
Soit E un ensemble de vecteurs quelconque, et F le front Pareto de E
$X \in F \Longleftrightarrow \nexists\ Y \in E\ /\ Y$ *domine* X

Selon la qualité exigée des solutions, deux familles de méthodes de résolution peuvent être utilisées : les *méthodes exactes* et les *méthodes approchées* ou *heuristiques* (FIG. 1.1). Les heuristiques produisent de bonnes solutions, et sont applicables à des instances de taille importante. Les heuristiques peuvent être *spécifiques* à un problème donné, mais elles peuvent également être génériques i.e. applicables à différents types de problèmes dans quel cas elles sont appelées *méta-heuristiques*. Ces dernières peuvent être *à solution unique* (méthodes de descente, recherche tabou, récuit simulé, ...) ou *à population de solutions* (algorithmes évolutionnaires, colonies de fourmis, ...). Les méthodes exactes (B&B, programmation dynamique, programmation par contraintes, etc.) permettent de trouver des solutions exactes avec preuve d'optimalité. Cependant, elles restent inutili-

sables, en pratique, sur des instances de grande taille. En effet, la résolution de ce type d'instances nécessite une puissance de calcul considérable. Les grilles de calcul sont un moyen pour réunir une telle puissance de calcul.

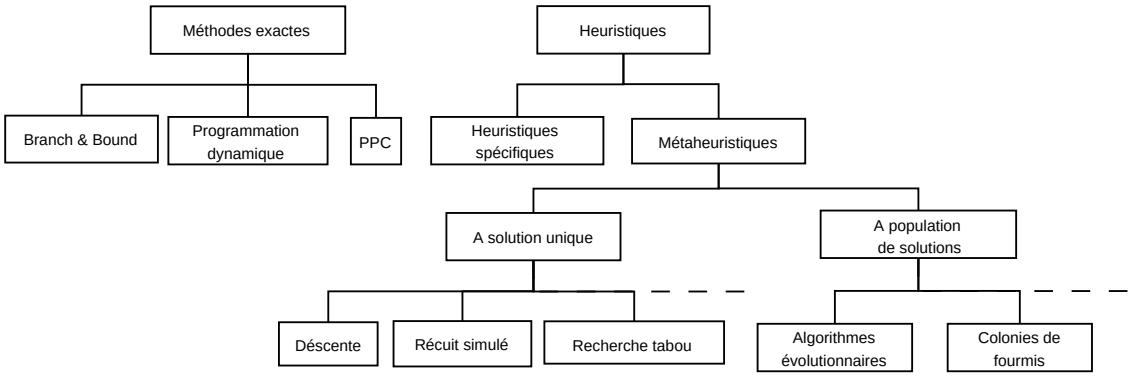

FIG. 1.1 – Une taxinomie des méthodes de résolution en optimisation combinatoire.

1.2 Grilles de calcul

Au cours de ces deux dernières décennies, les systèmes distribués et parallèles ont connu une évolution significative sur les plans matériel et logiciel. En peu de temps, ce domaine est passé des supercalculateurs aux architectures variées (vectorielles, à mémoire partagée ou distribuée) aux grilles informatiques, en passant par les réseaux (*Networks of Workstations ou NOW*) et grappes (*Clusters of Workstations ou COW*) de stations de travail et les méta-systèmes. L'engouement suscité aujourd'hui par les grilles est dû à plusieurs facteurs, notamment à : l'émergence et la popularité grandissante d'Internet et des technologies associées, et à l'évolution technologique et économique des ressources informatiques en termes de puissance de calcul, de capacité de stockage, de bande passante, et de prix de plus en plus abordables. Cet engouement a déjà conduit à la prolifération de projets sur les grilles avec différentes incarnations traduisant différentes approches : méta-calcul ou *metacomputing*, super-calcul virtuel ou *virtual supercomputing*, calcul global ou *global/Internet computing*, systèmes pair-à-pair ou *peer-to-peer computing*, etc.

Historiquement, l'idée des grilles est apparue en 1995 avec le projet I-WAY [DFP+96] visant à construire un méta-système multi-site et une infrastructure logicielle permettant son exploitation pour le traitement d'applications haute-performance. Le méta-système inter-connecte des périphériques de visualisation (caves de réalité virtuelle immersive et instruments d'observation) et des super-calculateurs répartis sur 17 sites. L'idée a été étendue à tous types de ressources informatiques incluant les applications, le stockage et les masses de données. Cette extension se retrouve dans la définition donnée dans [FKT01], selon laquelle la grille vise le partage coordonné de ressources et la résolution

de problèmes dans des organisations virtuelles dynamiques et multi-institutionnelles. Suivant les objectifs visés par l'exploitation de ressources réparties, plusieurs autres définitions sont proposées (e.g. [FKT01, KBM02]), ce qui justifie d'ailleurs la publication d'articles tels que [Fos02, Gri02, BLDGS03] dédiés à l'analyse de ces définitions dans le but de clarifier le concept de grille. Il nous semble donc important de préciser le contexte dans lequel on se place lorsqu'on aborde des travaux sur grilles informatiques.

1.2.1 Caractéristiques et applications des grilles

Dans le cadre de nos travaux sur l'optimisation combinatoire parallèle sur grilles, une grille est considérée comme un ensemble de ressources de calcul réparties ayant les caractéristiques présentées dans [BBL02] : multi-domaine d'administration, hétérogène, à grande échelle et dynamique (à disponibilité spatio-temporelle variable).

La grille est multi-domaine d'administration : les ressources sont réparties sur plusieurs domaines d'administration et gérées par différentes organisations. Les utilisateurs et fournisseurs de ressources sont clairement identifiés atténuant ainsi les problèmes de sécurité. Néanmoins, la traversée de pare-feu (ou *firewalls*) demeure un problème crucial qu'il convient de résoudre. Dans les systèmes de calcul global tels que XtremWeb [Fed03] basés sur le vol de cycles étendu à Internet, ce problème est résolu de manière naturelle puisque la communication est initiée par les machines volontaires "de l'intérieur d'un domaine d'administration". La résolution du problème n'est pas toujours aussi simple, et cela constitue souvent le défi des systèmes de *metacomputing*. Par exemple, pour Condor [LLM88], différentes approches ont été envisagées pour traiter le problème : *flocking*, *Condor-G* et *Condor Glide-in* [TTL02]. La première solution consiste à mettre en place un gestionnaire de ressources par domaine d'administration et de permettre à chacun de soumettre à d'autres gestionnaires les travaux qu'il ne peut traiter. Les deux autres approches sont basées sur le couplage avec Globus [FK97].

La grille est hétérogène : l'hétérogénéité des ressources matérielles et logicielles est accentuée par le nombre important de machines dans la grille et appartenant à différentes organisations. La résolution de ce problème en utilisant les standards d'échange récents tels que XML et SOAP constitue, entre autres, un facteur ayant fortement encouragé l'émergence des grilles. Néanmoins, l'hétérogénéité rend particulièrement plus difficile l'évaluation de performances des applications déployées, et constitue un thème de recherche à part entière [GWB+04, NGB04]. Plusieurs colloques sont dédiés à ce thème, en particulier *Automatic Performance Analysis Tools-and-Performance Tools for the GRID* (*APART Workshop*).

La grille a une large échelle : la grille a une échelle importante en termes de nombre de machines potentiellement disponibles et de la taille du réseau d'interconnexion de ces machines. Suivant l'échelle visée, on distingue souvent les systèmes de calcul global et pair-à-pair des grilles de calcul. Les premiers supportent pour le mo-

ment bien plus de machines que les grilles de calcul. Ils sont destinés en particulier aux applications basées sur le volontariat de milliers voire de millions de machines d'Internet. Le plus représentatif et populaire des intergiciels gérant ce type de systèmes *de calcul* est SETI@home [ACK+02]. XtremWeb [Fed03] en fait également partie mais, à la différence de SETI@home, il n'est pas dédié à un seul projet. Les applications gourmandes en ressources doivent être conçues de manière à supporter le passage à l'échelle. Il faut prendre en compte notamment les délais de communication importants.

La grille est dynamique : la disponibilité variable des ressources due à leurs pannes ou aux départs/retours de leurs propriétaires n'est pas une exception mais une règle dans la grille. En effet, avec un nombre si important de ressources, la probabilité de disparition des ressources est plus importante. Cette volatilité des ressources pose des problématiques de découverte dynamique de ressources, de tolérance aux pannes, de sauvegarde/restauration des données, de synchronisation, etc. Ces problématiques sont souvent difficiles à gérer de manière transparente et efficace dans les intergiciels. C'est pourquoi, leur prise en compte est parfois nécessaire au niveau applicatif.

Différents types d'utilisation de la grille sont identifiés dans [FK99, KBM02, BLDGS03]. Il s'agit principalement du super-calcul virtuel distribué à grande échelle (ou *distributed virtual supercomputing*), du calcul haut débit (ou *high-throughput computing*), du traitement intensif de masses de données (e.g. *data mining*), du traitement à la demande (ou *on-demand computing*) permettant l'accès à des ressources distantes, du travail coopératif, et du multimédia (*QoS*, Vidéo, etc.). Les deux premiers types d'utilisation permettent de réduire le temps d'exécution respectivement d'une seule application et d'un ensemble (ou *pool*) d'applications. Le traitement intensif de données a pour objectif de faire la synthèse ou la découverte de connaissances à partir de référentiels et bases de données et de librairies numériques. Le projet européen *DataGrid* [HJMS+00] est le plus représentatif des grilles dédiées à ce type d'applications. Les trois derniers types d'utilisations permettent l'accès à des services sur la grille tels que l'exploitation d'une ressource distante matérielle ou logicielle, l'interaction entre utilisateurs présents sur la grille, ou l'utilisation d'un support multimédia tel que la vidéo-conférence.

1.2.2 Grille d'expérimentation de nos travaux

Les expérimentations présentées dans ce document ont été réalisées sur la grille détaillée dans le tableau TAB. 1.1. Comme l'indique la figure FIG. 1.2, notre grille expérimentale est essentiellement composée de grappes de *Grid'5000*[1], mais aussi de grappes d'établissements de l'Université des Sciences et Technologies de Lille (USTL[2]). Le projet *Grid'5000* a été lancé en 2003 par le Ministère de la Recherche dans le cadre de l'ACI GRID "Programme national *Grid'5000*". Le projet est également supporté par d'autres organismes, notamment l'INRIA, le CNRS, les universités, RENATER,

[1] https ://www.*Grid'5000*.fr/
[2] http ://www.univ-lille1.fr/

et quelques conseils généraux et régionaux. Le but du projet est d'acquérir une grille expérimentale de 5000 cœurs de calcul répartis en grappes hébergées par 9 sites en France : Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Rennes, Sophia-Antipolis et Toulouse. *Grid'5000* compte actuellement (octobre 2007) environ 3000 cœeurs de calcul. Les grappes universitaires se trouvent à l'IEEA-FIL[3], Polytech'Lille[4] et l'IUT-A[5]. La grille expérimentale utilisée dans nos travaux est conforme aux caractéristiques des grilles énumérées dans la section précédente.

CPU (GHz)	**Domaine(Grappe)**	**Nombre**
P4 1.70	IEEA-FIL (USTL)	24
P4 2,40		48
P4 2,80		59
P4 3,00		27
AMD 1,30	Polytech'Lille (USTL)	14
Celeron 2,40		35
Celeron 0,80		14
Celeron 2,00		13
Celeron 2,20		28
P3 1,20		12
P4 3,20		12
P4 1,60	IUT-A (USTL)	22
P4 2,00		18
P4 2,80		45
P4 2,66		57
P4 3,00		41
AMD 2,2	Bordeaux (Grid'5000)	2x47
AMD 2,2	Lille (Grid'5000)	2x54
Xeon 2,4	Rennes (Grid'5000)	2x64
AMD 2,2		2x64
AMD 2,0		2x100
AMD 2,0	Sophia (Grid'5000)	2x107
AMD 2,2	Toulouse (Grid'5000)	2x58
AMD 2	Orsay (Grid'5000)	2x216
Total		**1889**

TAB. 1.1 – Détails de la grille utilisée.

En effet, cette grille est d'une échelle importante en termes du nombre de processeurs et de la taille du réseau d'interconnexion de ces machines. Notre grille d'expérimentation est composée d'environ 2000 processeurs répartis sur neuf grappes. Six grappes

[3] http ://www.fil.univ-lille1.fr/
[4] http ://www.polytech-lille.fr/
[5] http ://www-iut.univ-lille1.fr/

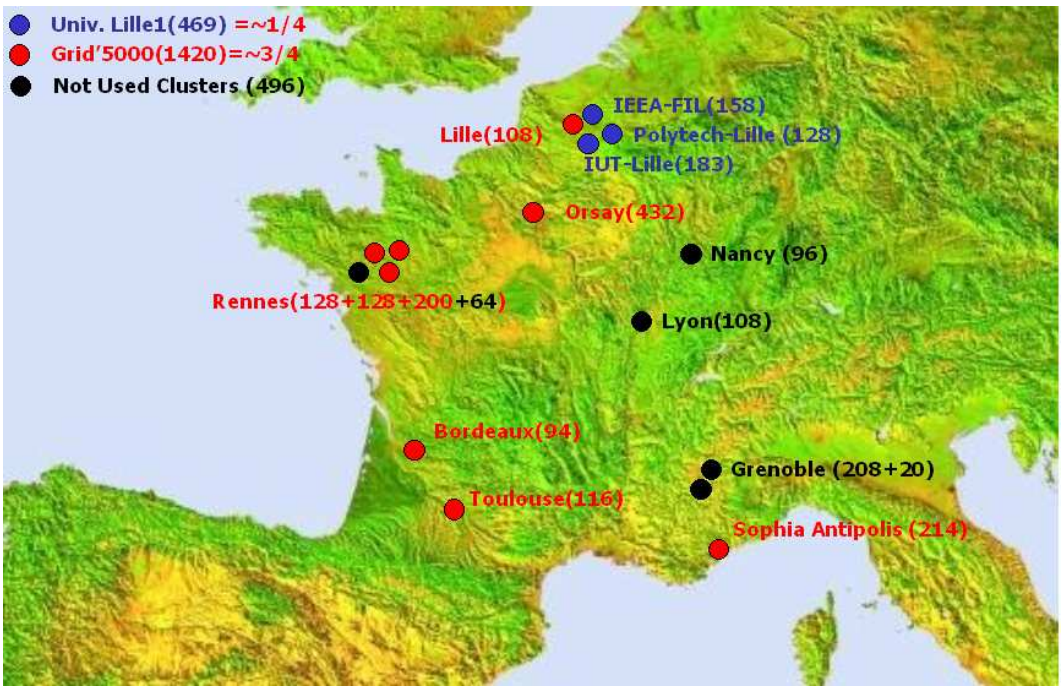

FIG. 1.2 – Une grille géographiquement répartie.

appartiennent à la grille expérimentale Grid'5000, et trois grappes à trois établissements de l'USTL. A la différence des machines universitaires qui sont mono-processeur, tous les nœuds de calcul de Grid'5000 utilisés sont des bi-processeurs. Les six grappes de Grid'5000 exploitées sont celles des sites de Bordeaux, Lille, Nice, Orsay, Rennes et Toulouse. Trois grappes de *Grid'5000* ne sont donc pas utilisées. Il s'agit de Grenoble, grappe surexploitée par les autres utilisateurs, Nancy et Lyon, grappes non encore opérationnelles au moment des tests. Grid'5000 apporte les trois quarts du nombre de processeurs de la grille expérimentale utilisée dans nos travaux. Comme l'indique la figure FIG. 1.2, les processeurs de cette grille sont géographiquement répartis.

Les ressources de calcul de cette grille expérimentale sont gérées par différentes organisations. Chacun des sites de *Grid'5000* est administré par un ingénieur différent. Cependant, les sites de cette grille ne sont pas protégés entre eux par des pare-feux. Autrement dit, tout processeur de *Grid'5000* peut joindre tout autre processeur de cette grille. Par contre, les machines des grappes universitaires se trouvent derrière des pare-feux. Les grappes universitaires sont administrées par des équipes réseau distinctes. Chaque grappe est protégée par un pare-feux, et appartient à un domaine d'administration distinct. Les machines universitaires ne sont donc pas joignables en dehors des machines de la même grappe.

La grille expérimentale utilisée est un environnement dynamique. En effet, les pro-

cesseurs de Grid'5000 doivent être réservés avant leur utilisation. Une réservation peut se faire selon trois modes. Il s'agit d'une réservation normale, d'une réservation avec déploiement du système d'exploitation, et d'une réservation en *besteffort*. Les expérimentations présentées dans nos travaux sont faites en *besteffort*. Une réservation d'un processeur en *besteffort* serait annulée si un autre utilisateur demande ce processeur. Le système de réservation n'informe pas les processus lancés de la fin d'une réservation *besteffort*. Un processeur perdu, à cause d'une réservation d'un autre utilisateur, peut donc être considéré comme un processeur tombé en panne. Pendant nos expérimentations, tout processeur libre de *Grid'5000* est réservé en *besteffort*. De plus, les machines des grappes universitaires sont principalement dédiées à l'enseignement et ne peuvent donc être exploitées que lorsqu'elles sont oisives. Par conséquent, notre grille expérimentale constitue un environnement hautement dynamique où les ressources de calcul sont exploitées en vol de cycles. Ceci permet de n'exploiter que la puissance de calcul perdue, et de ne pas pénaliser les autres utilisateurs de *Grid'5000* et les étudiants de l'USTL.

L'hétérogénéité des ressources matérielles et logicielles caractérisent également notre grille expérimentale. Comme l'indique le tableau TAB. 1.1, les processeurs des différentes grappes ne sont pas similaires. De plus, certaines grappes contiennent même des processeurs différents. Par ailleurs, les grappes sont interconnectées différemment. En effet, les grappes universitaires sont reliées en 1 GigaBit, tandis que les grappes de Grid'5000 sont interconnectées en 2,5 GigaBits par le réseau national RENATER3. En plus de cette hétérogénéité matérielle, la grille expérimentale se caractérise également par une hétérogénéité dans les distributions Linux installées sur les grappes. Ceci accentue l'hétérogénéité des outils logiciels se trouvant sur chaque grappe.

1.3 Méthodes exactes parallèles

Les méthodes de résolution exactes utilisées en optimisation combinatoire sont, pour un bon nombre d'entre elles, de type B&B. Ces méthodes se déclinent essentiellement en trois variantes : B&B simple, le *Branch-and-Cut* (B&C), et le *Branch-and-Price* (B&P). Il existe d'autres variantes du B&B moins connues : *Branch-and-Peg* [GGS04], *Branch-and-Win* [PC04], et *Branch-and-Cut-and-Solve* [CZ06]. Cette liste est certainement non exhaustive. Il est également possible de considérer un simple algorithme de type *Diviser pour Régner* comme une base pour l'algorithme B&B. Il suffit de supprimer, du *Branch-and-Bound*, l'opérateur d'élimination, expliqué plus loin, pour obtenir un algorithme de type *Diviser pour Régner*. Certains auteurs considèrent les algorithmes B&C et B&P, ainsi que les autres variantes, comme des algorithmes B&B distincts. Ces auteurs parlent alors de B&X pour désigner les algorithmes B&B, B&C, B&P, etc. Dans la suite du document, l'algorithme B&B désigne B&B simple lui-même ou toute autre variante de cet algorithme.

Les algorithmes B&B reposent sur une exploration implicite de toutes les solutions

du problème considéré. L'espace de ces solutions est exploré en construisant dynamiquement un arbre dont le sous-problème racine représente la totalité de cet espace, les sous-problèmes feuilles sont les différentes solutions possibles, et les sous-problèmes internes sont des sous-espaces de l'espace global des solutions. La taille de ces sous-espaces est de plus en plus réduite à mesure qu'on s'approche des feuilles. La construction d'un tel arbre et son exploration se font à l'aide de quatre opérateurs. Ce sont les opérateurs de *décomposition*, d'*évaluation*, de *sélection* et d'*élimination*. L'algorithme procède en plusieurs itérations, durant lesquelles la meilleure solution trouvée est conservée et améliorée au fur et à mesure de l'exploration. Les sous-problèmes générés et non encore traités sont conservés dans une liste dont le contenu initial est le sous-problème racine.

A chaque itération de l'algorithme, l'*opérateur de sélection* choisit, selon une certaine stratégie, un sous-problème, autrement dit un sous-espace, de cette liste. L'*opérateur de décomposition* le divise en plusieurs sous-espaces plus petits et deux à deux disjoints. L'*opérateur d'évaluation* calcule une borne des solutions de chaque sous-espace généré, et l'*opérateur d'élimination* élimine de la liste tout sous-espace dont l'évaluation a démontré qu'il ne contient pas de solution pouvant améliorer la meilleure solution déjà trouvée.

Pour les problèmes multi-critères, la meilleure solution trouvée peut être constituée de plusieurs solutions Pareto. Par conséquent, en plus de l'ensemble des sous-espaces, l'algorithme garde dans une autre liste toutes les solutions Pareto obtenues. A l'inverse des opérateurs de sélection et de décomposition qui restent inchangés, les opérateurs d'évaluation et d'élimination, utilisés dans le contexte mono-critère, doivent nécessairement être adaptés aux problèmes multi-critères. Utiliser la règle de dominance entre vecteurs au lieu d'une simple comparaison entre valeurs, et évaluer un sous-espace selon plusieurs critères au lieu d'un seul, sont principalement les deux modifications à faire pour adapter l'algorithme B&B mono-critère aux problèmes multi-critères. Les algorithmes B&B réduisent considérablement la puissance de calcul nécessaire pour explorer tout l'espace des solutions. Toutefois, une telle puissance peut s'avérer toujours considérable, notamment lorsqu'il s'agit d'algorithmes B&B multi-critères. Recourir à plusieurs processeurs en parallèle est un des moyens efficaces utilisés pour réduire le temps d'exploration. De nombreuses approches de parallélisation des algorithmes B&B sont ainsi proposées dans la littérature. La section suivante donne une brève classification de ces méthodes.

Une taxinomie des méthodes de parallélisation du B&B est présentée dans [Mel05]. Cette taxinomie est basée sur les classifications proposées dans [GC94, CDC+94]. Quatre modèles sont identifiés : le modèle multi-paramétrique parallèle, le modèle d'exploration arborescente parallèle, le modèle d'évaluation parallèle des bornes, et le modèle d'évaluation parallèle d'une borne. Les quatre sous-sections qui suivent présentent ces modèles.

1.3.1 Modèle multi-paramétrique parallèle

Le modèle multi-paramétrique parallèle (FIG. 1.3), très peu étudié dans la littérature, consiste à considérer plusieurs algorithmes $B\&B$. Il s'agit d'un modèle à gros grain. Plusieurs variantes de ce modèle peuvent être envisagées en fixant différemment un ou plusieurs paramètre(s) des algorithmes. Les algorithmes diffèrent seulement par l'opérateur de décomposition dans [MP93]. Ils sont différents uniquement par l'opérateur de sélection dans [JAM88] où une variante de la stratégie *profondeur d'abord* est utilisée. Chaque algorithme sélectionne de manière aléatoire le prochain sous-problème à traiter parmi les derniers sous-problèmes générés. Dans [KK84], les algorithmes utilisent chacun une borne supérieure différente dans leurs tests. L'idée est qu'un seul algorithme utilise la meilleure borne supérieure trouvée tandis que les autres (ϵ-approchés) font usage de cette borne diminuée d'une valeur $\epsilon > 0$. Une autre variante de ce modèle parallèle consiste à décomposer l'intervalle défini par la borne inférieure et la borne supérieure du problème à traiter en sous-intervalles. Chaque sous-intervalle est affecté à un des algorithmes. Dans le contexte multi-critère, le front Pareto (composé de solutions appelées *solutions supportées*) peut être calculé par un algorithme $B\&B$. Ensuite, plusieurs algorithmes $B\&B$ peuvent coopérer pour calculer les solutions non supportées situées dans le voisinage du front Pareto. Une telle variante a été proposée dans [LDT04].

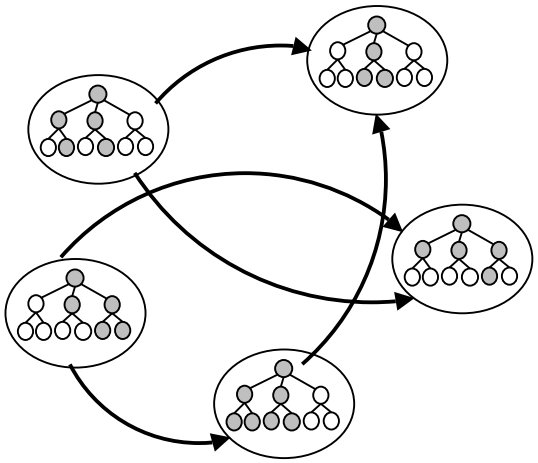

FIG. 1.3 – Illustration du modèle multi-paramétrique.

Le principal avantage du modèle multi-paramétrique parallèle est sa généricité permettant son exploitation de manière transparente à l'utilisateur. Son inconvénient est le surcoût d'exploration qu'il engendre puisque certains sous-problèmes de l'arborescence sont explorés de manière redondante. Toutefois, ce surcoût a une moindre conséquence lorsque le modèle est déployé sur une grille de calcul puisque l'on dispose de suffisamment de ressources. Par ailleurs, le nombre d'algorithmes mis en concurrence n'étant pas important, son exploitation sur une grille ne peut être justifiée que s'il est combiné avec d'autres modèles parallèles.

1.3.2 Modèle d'exploration arborescente parallèle

Le modèle d'exploration arborescente parallèle consiste à parcourir en parallèle différents sous-problèmes racines de sous-arbres définissant des sous-espaces de recherche du problème (FIG. 1.4). Cela signifie que les opérateurs de décomposition, sélection, évaluation et élimination sont exécutés en parallèle de manière (a)synchrone par différents processus explorant ces sous-espaces. Dans le mode synchrone, un algorithme de $B\&B$ comporte différentes phases. Durant chaque phase, les processus de l'algorithme effectuent leur exploration de manière indépendante. Entre les phases, les processus se synchronisent pour s'échanger de l'information, par exemple la meilleure solution trouvée. Dans le mode asynchrone, les processus de l'algorithme communiquent de manière imprévisible.

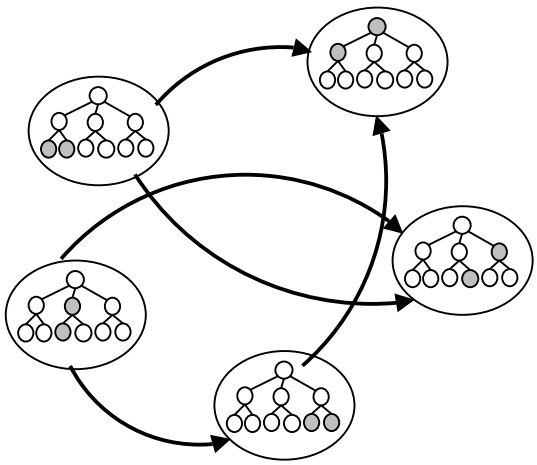

FIG. 1.4 – Illustration du modèle d'exploration arborescente parallèle.

Comparé aux autres modèles, le modèle d'exploration arborescente parallèle a suscité beaucoup d'intérêt et a fait l'objet de beaucoup de travaux de recherche pour deux raisons majeures. D'une part, le degré de parallélisme de ce modèle peut être très important sur les instances de grande taille justifiant à lui tout seul l'intérêt d'une grille de calcul. D'autre part, l'implémentation du modèle pose plusieurs problèmes constituant des défis de recherche dans le domaine de l'algorithmique parallèle. Parmi ces problèmes, on peut citer le placement et la gestion de la liste des sous-problèmes à résoudre, la répartition et le partage de la charge (des sous-problèmes générés), la communication de la meilleure solution trouvée, la détection de la terminaison de l'algorithme, et la tolérance aux pannes.

1.3.3 Modèle d'évaluation parallèle des bornes

Le modèle d'évaluation parallèle des bornes permet la parallélisation de la phase d'évaluation des sous-problèmes générés par l'opérateur de décomposition (FIG. 1.5). Ce modèle est exploitable dans le cas où la phase d'évaluation des bornes est entièrement exécutée à la suite de l'opérateur de décomposition. Il ne change pas la conception de l'algorithme i.e. il est identique à la version séquentielle sauf que la phase d'évaluation est plus rapide. Le principal avantage de ce modèle est sa généricité.

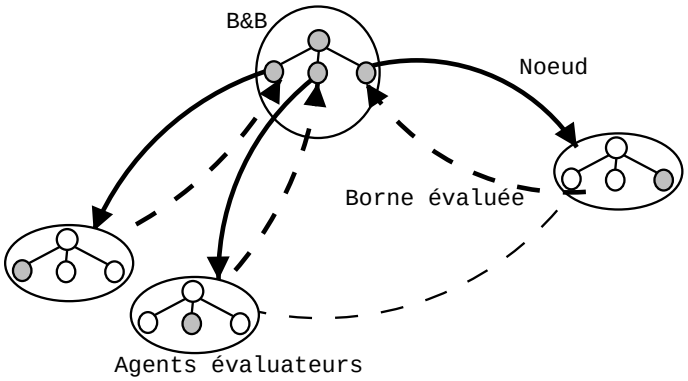

FIG. 1.5 – Illustration du modèle d'évaluation parallèle des bornes.

Par contre, dans un environnement de type grille de calcul il peut s'avérer inefficace pour les raisons suivantes : (1) le modèle est de nature synchrone, ce qui le rend coûteux en temps CPU dans un contexte hétérogène et volatile, (2) sa granularité (coût de la fonction de calcul de la borne) peut s'avérer fine et donc pénalisante dans un environnement large échelle. Par exemple, dans le cas du problème du flow-shop le coût CPU de l'évaluation d'une borne n'est pas suffisant pour justifier son exploitation, et (3) le degré de parallélisme de ce modèle est dépendant du problème traité. Il est souvent limité et décroît au fur et à mesure de l'exploration car le nombre de sous-problèmes générés diminue au fur et mesure qu'on rajoute des contraintes à l'opérateur de décomposition. Sa combinaison avec le modèle d'exploration arborescente parallèle peut générer un parallélisme massif ne pouvant être exploité de manière efficace que sur grille de calcul.

1.3.4 Modèle d'évaluation parallèle d'une borne

Ce modèle ne change pas la conception de l'algorithme car il est identique à la version séquentielle sauf que la phase d'évaluation est plus rapide. En outre, le modèle est dépendant du problème traité, synchrone et de type centralisé. Son extensibilité étant limitée, son exploitation sur une grille est plus efficace lorsqu'il est combiné à d'autres modèles.

Le modèle d'évaluation parallèle d'une borne est particulièrement intéressant lors du calcul de la borne du problème original. En effet, l'opérateur de borne est souvent beaucoup plus coûteux pour le problème original que pour les sous-problèmes. Ceci se produit notamment lors de la résolution des problèmes de programmation linéaire en nombres entiers mixte, *Mixed Integer Linear Programming* (MILP). Dans ce type de problèmes, le calcul de la borne d'un sous-problème est basé sur le calcul fait pour le sous-problème père. Typiquement, l'opérateur de décomposition rajoute une seule nouvelle contrainte pour définir les sous-problèmes. Par conséquent, le calcul de la borne se fait souvent relativement très vite. Par contre, le calcul de la borne du problème original démarre en général avec aucune contrainte. PICO [EPH00] est un des exemples de plate-forme de parallélisation de B&B supportant le modèle d'évaluation parallèle d'une borne. Dans l'approche de PICO, ce modèle est conseillé pour borner le problème original. Une fois la borne du problème original calculée, PICO permet de continuer la résolution selon le modèle d'exploration arborescente parallèle.

1.4 Gridification des méthodes exactes

Dans la section 1.2, nous avons présenté les caractéristiques des grilles de calcul. Dans la section 1.3, nous avons décrit les modèles parallèles des méthodes exactes. Dans cette section, nous nous attachons à présenter l'adaptation de ces modèles parallèles, autrement dit leur gridification, pour prendre en compte les caractéristiques des grilles. Cette section décrit des approches proposées pour la résolution des problèmes posés par les différentes caractéristiques de la grille. Il s'agit notamment de l'optimisation du coût de communication pour pallier au problème des délais de communication, du passage à l'échelle pour exploiter toutes les ressources de calcul de la grille, du partitionnement et de la régulation de la charge pour remédier au problème de l'hétérogénéité de ces ressources, du passage de pare-feux pour s'adapter à un environnement géré par différentes organisations, et de la tolérance aux pannes pour pallier au problème de la volatilité des ressources de la grille.

1.4.1 Optimisation du coût de communication

Les délais de communication dans une grille de calcul nationale ou internationale sont plus grands que dans une grappe de machines. Les délais de communication sont l'une des sources majeures de perte d'efficacité dans toute approche parallèle du B&B. L'article [MS05] présente des résultats expérimentaux illustrant les coûts d'une communication dans un réseau à grande échelle. Ces résultats sont comparés avec leurs équivalents dans une grappe de machines locale. Les auteurs ont déployé un B&B parallèle avec un seul fermier et un seul processus travailleur. Dans les deux cas, le fermier se trouve à l'Université de Bergen en Norvège. L'utilisation d'un seul processus travailleur garantit que les résolutions parallèles et séquentielles se comportent toujours de la même façon. Dans le premier test, le processus travailleur se trouve à l'Université de

Crête en Grèce. Dans le deuxième test, le processus travailleur se trouve l'Université de Bergen. Les tests sont faits à l'aide de trois instances standards du problème d'affectation quadratique, *Quadratic Assignment Problem* (QAP), et du problème du voyageur de commerce, *Travelling Salesman Problem* (TSP). Le QAP et le TSP sont connus pour être NP-difficiles. Ces tests consistent à résoudre les instances standards nug15 et nug17 du QAP (QAPLIB [BKR97]) et l'instance gr21 du TSP (TSPLIB [Rei90]).

Le tableau TAB. 1.2 résume les résultats obtenus. Les colonnes Bergen et Crête donnent le temps moyen de chaque communication en millisecondes lorsque le travailleur se trouve respectivement à l'Université de Bergen et l'Université de Crête. La dernière colonne donne le rapport entre les temps de communication des colonnes Bergen et Crête. Les tests montrent que les temps de communication sont beaucoup plus importants dans un réseau étendu que dans un réseau local. Or la communication est l'une de sources de perte d'efficacité dans un algorithme parallèle. Ceci est encore plus vrai dans les système de calcul sur grille.

Instance	Bergen	Crête	Bergen/Crête
gr21	16 ms	18914 ms	1182
nug15	14 ms	27729 ms	1980
nug17	37 ms	47525 ms	1284

TAB. 1.2 – Comparaison des délais de communication d'une grille avec celle d'une grappe locale.

Trois stratégies possibles peuvent être utilisées pour réduire le coût de communication. Il s'agit de diminuer (1) le temps d'une communication, (2) le nombre de messages communiqués et (3) la taille de ces messages. L'amélioration du temps d'une communication dépend des avancées technologiques réalisées dans le matériel utilisé. La réduction du nombre de messages échangés dépend en grande partie de la stratégie d'équilibrage de charge adoptée. En effet, un manque de sous-problèmes au niveau d'un processus B&B se traduit toujours par au moins une communication pour trouver du travail. Enfin, la réduction de la taille des messages échangés est une autre piste à explorer pour améliorer l'efficacité parallèle. Les messages communiqués dans un B&B parallèle sont constitués en grande partie de sous problèmes. Par conséquent, certaines approches, comme celles explorées dans PICO et [IF00], sont basées sur une représentation des sous-problèmes permettant de réduire leur taille.

Dans PICO, chaque travailleur gère un pool de sous-problèmes à traiter, appelé *pool travailleur*, et un autre pool de sous-problèmes sauvegardés, appelé *pool serveur*. Un travailleur prend toujours le prochain sous-problème à traiter à partir de son pool travailleur. Une fois qu'un nouveau sous-problème est généré, un travailleur décide soit de le garder ou de le transférer ailleurs. Un sous-problème gardé est mis dans le pool travailleur pour être traité plus tard. Par contre, un sous-problème transféré est mis dans le pool serveur et un jeton de ce sous-problème est communiqué à un des sous-fermiers.

Un jeton est une information permettant d'identifier et de localiser un sous-problème. La taille d'un jeton est celle d'un entier, codé en général sur 48 bits, nettement inférieure à la taille complète d'un sous-problème. Un travailleur perd le contrôle de tous les sous-problème transférés. Le sous-fermier récepteur peut ainsi prendre la décision de donner ce sous-problème à un des processus travailleurs de la grille. Ce travailleur peut être le même que le générateur de ce sous-problème. Un sous-fermier peut également prendre la décision de donner le sous-problème à un autre sous-fermier. Chaque sous-fermier gère donc un pool de jetons. Pour donner un sous-problème à un autre travailleur, le sous-fermier demande au travailleur détenteur de ce sous-problème d'effectuer lui-même le transfert du sous-problème au nouveau travailleur récepteur. L'approche de PICO a été expérimentée avec une grille de 128 processeurs. Le nombre de sous-fermiers utilisés est de 32, et chaque sous-fermier contrôle 4 processus travailleurs. Les expérimentations sont faites sur des instances du problème de programmation en nombres entiers mixte, Mixed-integer programming (MIP), se trouvant dans la bibliothèque MIPLIB. Le tableau TAB. 1.3 résume les résultats obtenus.

Instance	bell3a	lseu	misc07	mod008	qiu	stein46
Nr. Proc.	128	128	128	128	128	128
Eff. paral.	56,8	21,5	15,3	36,7	37,3	45,7

TAB. 1.3 – Efficacités parallèles enregistrées par PICO.

L'approche à jetons de PICO permet de réduire la taille de l'information échangée entre le fermier global, les différents sous-fermiers et les processus travailleurs. En effet, seulement les jetons sont communiqués. En outre, cette approche permet de réduire la quantité d'information manipulée au niveau d'un fermier puisqu'un fermier ne manipule que des jetons. Ceci atténue le risque d'apparition du goulot d'étranglement, permet d'augmenter la capacité du passage à l'échelle de cette approche, et améliore ainsi son efficacité parallèle. Cependant, le jeton permet juste de localiser un sous-problème pour le récupérer ensuite. Il est impossible de déduire un sous-problème à partir de son jeton.

Les auteurs de [IF00] proposent une autre approche de codage qui se base sur une représentation de chaque sous-problème en fonction de sa position dans l'arbre de B&B. Un sous-problème est représenté par une séquence de couples $<X_i,V_i>$. La figure FIG. 1.6 donne un exemple d'une telle représentation. Dans la figure FIG. 1.6, le sous-problème A est représenté par la séquence ($<X_2,0>$,$<X_1,1>$,$<X_3,0>$). Dans ce codage, X_i est la variable prise en compte pour faire la décomposition d'un sous-problème. Le sous-problème A, par exemple, est généré avec une décomposition selon la variable X_3. V_i est l'ordre de génération d'un sous-problème lors de la décomposition de son sous-problème père. Le sous-problème A, par exemple, est le premier sous-problème généré lors de la décomposition selon la variable X_3.

Comme indiqué auparavant, l'intérêt de cette représentation est de permettre de retrouver un sous-problème à partir de son codage. Cependant, le codage utilisé dans

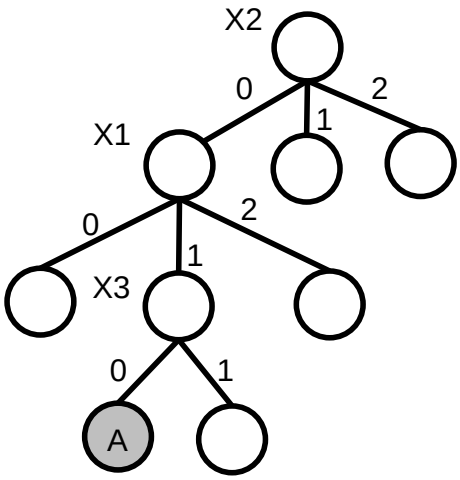

FIG. 1.6 – Codage d'un sous-problème dans l'approche proposée dans [IF00].

[IF00] ne permet pas forcément de réduire la taille d'un sous-problème. En effet, la taille du codage peut être supérieure à celle du sous-problème. Soit l'exemple d'une instance du problème du Flow-Shop avec 10 tâches. Dans le Flow-Shop, un sous-problème est toujours une suite de tâches ordonnées et de tâches non ordonnées. Le sous-problème (4,0,2,1,{3,5,6,7,8,9}) signifie que les tâches $4, 0, 2$ et 1 sont ordonnées dans cet ordre, tandis que les tâches $3, 5, 6, 7, 8$ et 9 ne sont pas encore ordonnées. Le codage du sous-problème (5,{0,1,2,3,4,6,7,8,9}) donne (<X_1,5>). Pour ce sous-problème, la taille du codage est bien inférieure à celle du sous-problème. Par contre, le codage de (8,5,0,1,4,6,7,2, 3,{9}) donne (<X_1,8>,<X_2,5>,<X_3,0>,<X_4,1>,<X_5,4>,<X_6,6>,<X_7,7>,<X_8,2>, <X_9,3>). Pour cet exemple, la taille du codage devient nettement plus grande que celle du sous-problème. Le gain dans la taille du codage dépend donc du nombre de tâches ordonnées. Le codage n'est plus intéressant lorsque plus de la moitié des tâches sont ordonnées. Néanmoins, le B&B manipule souvent des sous-problèmes dont le nombre de tâches ordonnées est petit. Ce sont des sous-problèmes qui se situent en haut de l'arbre du B&B. Par conséquent, ce codage réduit toujours en moyenne la tailles des sous-problèmes échangés.

Les résultats expérimentaux présentés dans le chapitre suivant montrent que notre approche B&B@Grid réduit davantage la taille des sous-problèmes que ces deux approches. B&B@Grid propose donc un codage plus intéressant, en terme de la taille en bits du codage obtenu, que les approches de [IF00] et de PICO. Contrairement à PICO, B&B@Grid peut réduire tout un ensemble de sous-problèmes. En effet, le codage de PICO concerne un sous-problème à la fois. PICO attribue autant de codes que de sous-problèmes, tandis que B&B@Grid peut codifier un ensemble de sous-problèmes avec le même code. De plus, à l'inverse de [IF00], cet ensemble peut ne pas constituer un même sous-arbre. L'approche [IF00] exige que tous les sous-problèmes appartiennent au même sous-arbre pour être codifiés avec le même code. Par contre, B&B@Grid peut représen-

ter, avec un même code, un ensemble de sous-problèmes même s'ils n'appartiennent pas au même sous-arbre.

1.4.2 Partitionnement et régulation de charge

La nature hétérogène et dynamique des grilles de calcul rend la définition de la répartition de la charge entre les processus travailleurs plus difficile. Les stratégies d'équilibrage de charge utilisées dans la parallélisation du B&B sur grilles de calcul sont souvent basées sur le paradigme fermier-travailleur. Ces stratégies sont appliquées à différents problèmes d'optimisation combinatoire. Cette section présente trois approches particulièrement référencées dans la littérature. Il s'agit des approches proposées dans [ABGL02, AF02, CR95].

L'approche publiée dans [ABGL02] est certainement l'une des plus abouties pour la parallélisation du B&B selon le paradigme fermier-travailleur. Souvent dans une approche fermier-travailleur, un processus travailleur reçoit un certain nombre de sous-problèmes du fermier, les traite, arrête le traitement lorsqu'une certaine condition est vérifiée, et renvoie au fermier tous les sous-problèmes non encore résolus. Les sous-problèmes renvoyés au fermier doivent être de bonne qualité et d'une quantité suffisante. Autrement dit, le fermier ne doit pas manquer de sous-problèmes et ces sous-problèmes doivent être suffisamment difficiles. Un sous-problème est difficile lorsque son temps de résolution est important. Les sous-problèmes faciles augmentent la fréquence de sollicitation du fermier. En effet, un processus travailleur traite en peu de temps un sous-problème facile, et contacte donc rapidement le fermier pour lui demander un autre sous-problème. Un fermier, distribuant beaucoup de sous-problèmes faciles, risque donc de passer son temps à répondre aux sollicitations des travailleurs, et même de manquer de sous-problèmes. Par conséquent, la condition de renvoi de sous-problèmes détermine en grande partie la qualité de la politique de régulation de charge. Dans [ABGL02], les processus travailleurs renvoient au fermier tous les sous-problèmes non traités après une certaine période de temps. Cette période n'est pas fixe, mais dépend de la qualité et de la quantité de sous-problèmes d'un processus travailleur. Cependant, un processus travailleur doit renvoyer tous ses sous-problèmes, quelque soit leur qualité et leur quantité, au bout d'une certaine période maximale.

L'approche proposée dans [ABGL02] a été expérimentée sur le problème du QAP. L'un des objectifs de l'expérimentation est la résolution d'instances du QAP dont certaines sont restées non résolues pendant une trentaine d'années. Il s'agit notamment des instances nug27, nug28, nug30, kra30b, kra32 et tho30. Comme l'indique le tableau TAB. 1.4, les efficacités parallèles enregistrées pour ces instances sont nettement meilleures que celles enregistrées par beaucoup d'autres approches de la littérature.

Une autre approche avec le paradigme fermier-travailleur est publiée dans [AF02]. Dans cette approche, le fermier envoie à chaque processus travailleur un seul sous-

Instance	nug27	nug28	nug30	kra30b	kra32	tho30
Nr. Proc.	185	224	653	462	576	661
Eff. parall.	91%	90%	92%	92%	87%	89%

TAB. 1.4 – Efficacités parallèles enregistrées sur différentes instances du QAP.

problème. Un processus travailleur opère alors au plus N décompositions pour générer un certain nombre de sous-problèmes, élimine tout sous-problème dont la borne indique qu'il ne peut améliorer la meilleure solution connue, et renvoie au fermier les sous-problèmes non éliminés. Un processus travailleur contacte le fermier après traitement d'un certain nombre de sous-problèmes. Autrement dit, la condition de renvoi est donc de nature événementielle. L'événement est la fin de la résolution d'un certain nombre de sous problèmes. Cette approche a été expérimentée sur le problème des valeurs propres de l'inégalité bilinéaire matricielle, *Bilinear Matrix Inequality (BMI) Eigenvalue Problem*. Ce problème est connu pour être NP-difficile [TO95]. Dans les tests effectués, l'approche est paramétrée avec N égal à 1. Le tableau TAB. 1.5 donne les efficacités parallèles obtenues sur deux instances. Il s'agit d'une instance pour le contrôle d'un hélicoptère [KBH88] et d'une autre instance synthétique [AF02].

Instance	Contrôle d'un hélicoptère				Synthétique			
Nr. proc.	16	32	64	128	16	32	64	128
Eff. parall.	91%	83%	33%	21%	93%	93%	87%	71%

TAB. 1.5 – Efficacités parallèles enregistrées sur un problème de type BMI.

BoB++ [CR95] peut également être utilisé avec le paradigme fermier-travailleur. Dans BoB++, un processus travailleur opère une seule décomposition du sous-problème reçu et renvoie les sous-problèmes obtenus au processus fermier. L'approche de BoB++ a été expérimentée sur le problème du QAP. Le tableau TAB. 1.6 résume les efficacités parallèles obtenues sur les instances standards nug20, nug21 et nug22.

Instance	nug20			nug21			nug22		
Nr. proc.	40	50	80	40	50	80	40	50	80
Efff. parall.	81%	88%	41%	72%	79%	25%	64%	94%	44%

TAB. 1.6 – Efficacités parallèles enregistrées par BOB++.

Contrairement aux autres approches, la stratégie de régulation de charge de B&B@Grid ne s'applique pas à un seul paradigme parallèle, mais peut être utilisée avec différents paradigmes. De plus, l'inconvénient des approches proposées dans la littérature est la granularité fixe des unités de travail allouées. Or, dans une grille, la taille d'une unité de travail doit dépendre de la taille de la grille, de la nature hétérogène de ses ressources, et de leurs disponibilités variables. En effet, à l'inverse des autres stratégies,

la taille d'une unité de travail dans B&B@Grid dépend du nombre de processeurs de la grille, de leur puissance et de leur disponibilité. La granularité du travail attribué à un processeur puissant est plus grande que celle d'un processeur moins puissant. Cette stratégie tient également compte de la disponibilité d'un processeur puisque B&B@Grid peut être utilisé avec le modèle de vol de cycles. Dans un tel modèle, les processeurs ne sont exploités que pendant la période où ils sont oisifs. Dans B&B@Grid, le travail est également réparti entre tous les processeurs quelque soit la taille de la grille utilisée. Les expérimentations montrent d'ailleurs que B&B@Grid exploite à plein régime les processeurs d'une grille.

1.4.3 Tolérance aux pannes

La dynamicité et volatilité des grilles imposent, à toute approche de parallélisation du B&B, la définition d'une stratégie de tolérance aux pannes. Le paradigme fermier-travailleur convient parfaitement à la nature volatile des grilles de calcul. Il suffit de redonner un sous-problème, dont le processus travailleur est en panne, à un autre processus travailleur. Une panne du fermier peut être gérée avec des sauvegardes régulières. En cas de panne du fermier, il suffit de le relancer en l'initialisant avec la dernière sauvegarde. La tolérance aux pannes des processus travailleurs est plus facile à gérer lorsque la condition de renvoi est de type périodique, en d'autres termes, lorsque les processeurs travailleurs communiquent leurs sous-problèmes non résolus après une certaine période de temps. En effet, après cette période, un sous-problème distribué, dont la réponse n'est pas reçue, est considéré comme perdu. Il doit donc être redonné à un autre processus travailleur. Par contre, il est impossible d'utiliser cette technique dans une approche où la condition de renvoi est événementielle, autrement dit, une approche où les processeurs travailleurs communiquent leurs sous-problèmes non résolus après le traitement d'un certain nombre d'entre eux. En effet, le temps de traitement d'un certain nombre de sous-problèmes peut être quelconque. Pour ce type d'approches, il est nécessaire au fermier de recevoir régulièrement un signal de vie envoyé par les processus travailleurs.

La problématique de la tolérance aux pannes se pose d'une façon plus accrue lors de la parallélisation d'un B&B selon le paradigme pair-à-pair. Pour ce paradigme, l'approche [IF00] associe un état à chaque sous-problème. Cet état peut être *résolu*, *traité* ou *non traité*. Un problème est résolu s'il est une feuille de l'arbre ou si tous ses sous-problèmes fils sont résolus. Un sous-problème est traité s'il est déjà décomposé et non encore résolu. Tandis qu'un sous-problème non décomposé est considéré comme non traité. Dans un B&B classique, une seule liste de sous-problèmes est gérée. Il s'agit de la liste des sous-problèmes non traités. Par contre, un B&B dans [IF00] gère deux listes de sous-problèmes : la liste des sous-problèmes non traités et celle des sous-problèmes résolus. Pendant une résolution, la taille de la liste des sous-problèmes résolus varie constamment. Elle augmente lorsqu'un nouveau sous-problème est résolu, et diminue lorsque tous les sous-problèmes enfants d'un même sous-problème sont résolus. En effet, tous les sous-problèmes enfants sont équivalents à leur sous-problème père. Par

conséquent, il est possible de les remplacer dans cette liste par le sous-problème père. La résolution se termine quand la liste des sous-problèmes résolus ne contient que le problème racine de l'arbre du B&B. L'approche de [IF00] suppose le lancement de plusieurs processus B&B selon le modèle d'exploration arborescente parallèle. Ces processus s'échangent leur liste de sous-problèmes résolus. La liste de sous-problèmes résolus permet à chaque processus B&B d'avoir une certaine connaissance du travail fait, et par conséquent de ce qui reste à faire. Lorsqu'un processus tombe en panne, une partie seulement des sous-problèmes résolus est perdue. Ce sont les sous-problèmes résolus, mais non communiqués à au moins un autre processus. Par contre, les sous-problèmes communiqués ne sont pas forcément résolus une deuxième fois. L'approche est donc tolérante aux pannes.

Comme indiqué auparavant, B&B@Grid réduit la taille des sous-problèmes. Cette approche peut coder une liste de sous-problèmes par un intervalle d'entiers. Le coût de sauvegarde d'un intervalle, défini par deux entiers, est beaucoup plus faible que la sauvegarde d'une liste de sous-problèmes. Par conséquent, les stratégies de tolérance aux pannes, présentées dans cette section, deviennent plus efficaces lorsqu'elles utilisent le codage de B&B@Grid. Dans le paradigme fermier-travailleur, le coût d'une sauvegarde de l'ensemble des sous-problèmes en cours de traitement est plus faible avec B&B@Grid. Comme l'explique le chapitre suivant, B&B@Grid peut également coder l'ensemble des sous-problèmes résolus. Par conséquent, B&B@Grid s'applique à la stratégie de tolérance aux pannes décrite dans [IF00]. D'ailleurs, les expérimentations, présentées dans le prochain chapitre, montrent le gain que procure notre approche B&B@Grid.

1.4.4 Passage à l'échelle

L'échelle d'une grille est considérable en terme du nombre de ses ressources de calcul. Un des plus grands inconvénients de l'utilisation du paradigme fermier-travailleur dans une grille de calcul est le goulot d'étranglement que peut constituer le fermier lorsque le nombre de processus travailleurs devient important. Dans une grille, ce risque est d'autant plus accentué que cet environnement peut contenir un grand nombre de processeurs. Ceci constitue donc une limite quant au passage à l'échelle de ce paradigme. Toutefois, une hiérarchisation des fermiers permet de pousser les limites de ce paradigme pour supporter davantage de processeurs travailleurs. C'est ce que propose les approches ALPS, PICO, BOB++ et celle décrite dans [ANF03].

L'article [ANF03] introduit une approche qui hiérarchise le B&B parallèle présenté dans [AF02]. Dans la nouvelle approche, plusieurs fermiers, appelés dans cette section sous-fermiers, sont utilisés. Chaque sous-fermier contrôle ses propres processus travailleurs selon l'approche décrite dans [AF02]. En plus des sous-fermiers, l'approche emploie un fermier global dont le rôle est de contrôler les sous-fermiers selon le paradigme fermier-travailleur. Les approches ALPS et PICO sont davantage référencées dans la littérature que celle présentée dans [ANF03]. Ces deux approches se ressemblent

beaucoup et considèrent une architecture organisée en grappes. Une grappe contient un seul fermier local et un ou plusieurs processus travailleurs. Dans ces deux approches, un seul fermier global est utilisé. Comme dans [ANF03], son rôle est de contrôler tous les fermiers locaux selon le paradigme fermier-travailleur. Les fermiers locaux contrôlent également les processus travailleurs de leur grappe selon le même paradigme fermier-travailleur. Contrairement à [ANF03], un sous-fermier, dans ces deux approches, peut également se comporter comme un processus travailleur. Autrement dit, il peut se charger de traiter lui-même quelques sous-problèmes.

La hiérarchisation des fermiers permet effectivement d'avoir des approches qui passent davantage à l'échelle sans perte significative d'efficacité parallèle. Cependant, il existe un autre paradigme parallèle qui peut supporter beaucoup plus de processeurs. Il s'agit du paradigme pair-à-pair. DIB [FM87] et [IF00] font partie des travaux basés sur la parallélisation du B&B selon le paradigme pair-à-pair. L'intérêt de ce modèle est de pallier au goulot d'étranglement des approches centralisées.

B&B@Grid peut être utilisée avec les paradigmes hiérarchiques et pair-à-pair. Grâce à la réduction des sous-problèmes, B&B@Grid permet de repousser davantage les limites de ces paradigmes. En effet, les expérimentations décrites dans le deuxième chapitre montrent que l'utilisation d'un simple paradigme centralisé, sans aucune hiérarchisation, permet l'exploitation de plusieurs milliers de processeurs.

1.4.5 Passage de pare-feux

Les processeurs d'une grille peuvent appartenir à des domaines d'administration différents, et les processeurs d'un même domaine sont souvent protégés par un pare-feu. Par conséquent, ils ne peuvent pas être contactés par des processeurs d'un autre domaine. Dans le paradigme fermier-travailleur, deux modes de communication peuvent être employés : le mode *push* et le mode *pull*. Dans le mode *push*, il revient au fermier de contacter les processus travailleurs pour leur donner des sous-problèmes à résoudre. Par contre, dans le mode *pull*, ce sont les processus travailleurs qui prennent l'initiative de contacter le fermier pour récupérer des sous problèmes. Dans le mode *push*, le processus fermier doit donc être capable de contacter tous les processus travailleurs. Cependant, il est rare de trouver une machine pouvant jouer un tel rôle. Par conséquent, le mode *pull* peut être une solution à la présence de pare-feux. GRIBB [MS05] est l'exemple d'une approche qui parallélise le B&B avec le modèle fermier-travailleur en utilisant le mode *pull*. Dans GRIBB, un travailleur demande au fermier un sous-problème à la fois, le traite, et renvoie l'ensemble des sous-problèmes obtenus une fois la décomposition faite. B&B@Grid peut être utilisé avec le mode *pull*. Ceci permet d'exploiter les processeurs qui se trouvent derrière des pare-feux.

Conclusion

Dans ce chapitre, nous avons présenté une étude sur les méthodes exactes, et leurs différents modèles de parallélisation, en vue de leur gridification. La gridification d'un algorithme signifie la prise en compte des caractéristiques des grilles pour l'adaptation de cet algorithme à de tels environnements, notamment la nature dynamique et hétérogène de leurs ressources de calcul. En plus de leur nombre considérable, ces ressources sont distribuées sur plusieurs domaines d'administration, et sont donc protégées par des pare-feux. Ces ressources sont également interconnectées par un réseau de grande taille, ce qui augmente les délais de communication.

Dans ce chapitre, nous avons également présenté différents travaux faits sur la gridification des méthodes exactes, notamment les algorithmes B&B. Ces travaux concernent essentiellement l'optimisation des coûts de communication, le partitionnement et la régulation de la charge, la tolérance aux pannes, le passage à l'échelle, et le passage de pare-feux. Deux conclusions essentielles ressortent de ces travaux. Premièrement, dans les grilles, la stratégie de répartition de la charge entre les processus devient encore plus problématique à cause de la nature hétérogène et dynamique de leurs ressources. L'inconvénient des approches publiées dans la littérature est la granularité fixe des unités de travail. Or, ces unités doivent s'adapter à la taille des grilles, à la nature hétérogène de leurs ressources, et à leur disponibilité variable. Deuxièmement, certaines approches permettent de réduire le coût des communications en réduisant la taille de l'information échangée entre les processus. Comme cette information est constituée essentiellement de sous-problèmes, ces approches proposent de représenter les sous-problèmes avec un codage de taille plus petite. Toutefois, les codages, que nous avons trouvés dans la littérature, ne permettent pas forcement de réduire la taille des sous-problèmes, ou ne permettent pas de régénérer les sous-problèmes correspondants, mais juste de les localiser sur la grille. Par contre, le codage, que nous proposons dans le prochain chapitre, permet à la fois de réduire la taille des sous-problèmes et de les régénérer à partir du codage obtenu.

Chapitre 2

B&B@Grid : une approche pour la gridification des méthodes exactes

L'approche B&B@Grid, présentée dans ce chapitre, a été publiée et présentée à la conférence internationale *IEEE IPDPS'2007* **[MMT07c]. La stratégie d'équilibrage de charge de B&B@Grid est publiée dans la revue internationale** *Parallel Computing* **[MMT07b].**

Introduction

Dans ce chapitre, nous proposons une nouvelle approche, appelée B&B@Grid, pour la gridification des algorithmes B&B. L'approche B&B@Grid vise à prendre en compte les caractéristiques des grilles de calcul. Dans B&B@Grid, nous proposons de représenter un ensemble de sous-problèmes par un intervalle de deux entiers naturels. L'objectif est de réduire la taille des sous-problèmes communiqués pour diminuer le coût des communications. Ceci permet également à la stratégie de tolérance aux pannes de réduire le coût en mémoire de la sauvegarde de l'état d'une résolution. En outre, B&B@Grid définit une stratégie d'équilibrage de charge où les unités de travail sont distribuées en tenant compte de l'hétérogénéité des ressources de la grille, de leur nombre et de leur disponibilité. Le chapitre décrit également les stratégies de détection de terminaison et de partage de la ou des solution(s) trouvée(s) pendant une résolution.

Ce chapitre est organisé en six sections. La **section 2.1** décrit les concepts et les opérateurs définissant cette nouvelle approche. Cette section présente des expérimentations qui comparent B&B@Grid à PICO et à l'approche publiée dans [IF00]. Dans la **section 2.2**, nous décrivons le déploiement fermier-travailleur et les expérimentations réalisées avec ce paradigme. La **section 2.3** explique la stratégie de tolérance aux pannes adoptée dans cette nouvelle approche. Nous y présentons également des expérimentations pour comparer B&B@Grid avec les autres approches de la littérature. La **section 2.4** détaille la stratégie d'équilibrage de charge de B&B@Grid. Cette section se termine par

des expérimentations permettant d'évaluer les performances de cette stratégie. Dans les **sections 2.5** et **2.6**, nous expliquons les deux stratégies employées dans B&B@Grid pour gérer la détection de la terminaison et le partage de la ou des solution(s) trouvée(s).

2.1 Nouvelle approche pour la compression de sous-problèmes

Comme indiqué dans le chapitre précédent, la plupart des approches dédiées à la parallélisation de l'algorithme B&B sont basées sur le modèle d'exploration arborescente parallèle avec une stratégie de parcours d'arbre de type *profondeur d'abord*. B&B@Grid considère également le même modèle parallèle et la même stratégie de parcours. Cette nouvelle approche utilise la notion de liste de sous-problèmes actifs. Les sous-problèmes actifs d'un B&B sont ceux générés mais non encore traités. Au cours d'une résolution, cette liste évolue constamment et l'algorithme s'arrête lorsqu'elle devient vide. Comme l'indique la figure FIG. 2.1, une liste de sous-problèmes actifs couvre un certain ensemble de sous-problèmes de l'arbre. Cet ensemble est constitué de tous les sous-problèmes pouvant être explorés à partir des sous-problèmes de cette liste. Le nombre de sous-problèmes couverts par les sous-problèmes actifs diminue au fil d'une résolution, tandis que le nombre de sous-problèmes implicitement ou explicitement explorés augmente. Un sous-problème est implicitement exploré lorsque un algorithme B&B conclut, sans le visiter, qu'il ne peut pas contenir de solutions optimales. Par contre, un sous-problème visité est dit explicitement exploré. Au début d'une résolution, l'ensemble des sous-problèmes actifs ne contient que le problème racine, tous les autres sous-problèmes de l'arbre sont couverts par le problème racine. A la fin d'un calcul, tous les sous-problèmes de l'arbre du B&B appartiennent à l'ensemble des sous-problèmes implicitement ou explicitement visités.

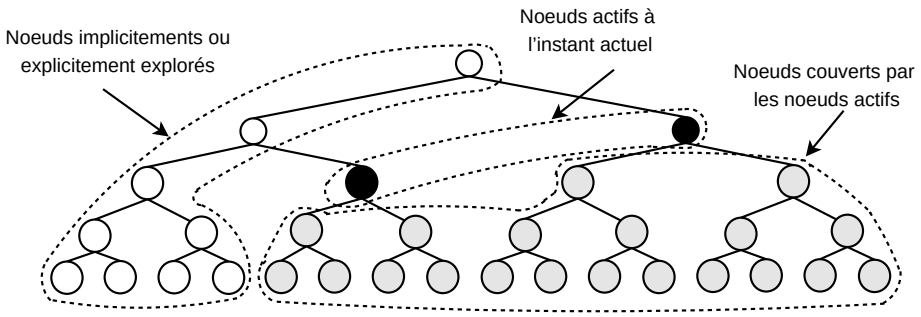

FIG. 2.1 – Sous-problèmes couverts par un ensemble de sous-problèmes actifs.

Le principe de l'approche est basé sur l'assignation d'un numéro à chaque sous-problème de l'arbre. Cette assignation est telle que tous les numéros de toute liste de sous-problèmes actifs constitue un intervalle d'entiers. L'approche définit ainsi une relation d'équivalence entre le concept de liste de sous-problèmes actifs et celui d'intervalle

de numéros de sous-problèmes. La connaissance de l'un doit permettre de déduire de façon unique l'autre. Etant donnée sa petite taille, l'intervalle est utilisé pour les communications et pour les sauvegardes, tandis que la liste des sous-problèmes actifs est utilisée pour l'exploration. Afin de passer de l'un à l'autre des deux concepts, l'approche définit deux opérateurs supplémentaires de **pliage** et **dépliage** en complément des quatre opérateurs de base des algorithmes B&B. L'opérateur de pliage déduit un intervalle de numéros à partir d'une liste de sous-problèmes actifs. L'opérateur de dépliage retrouve une liste de sous-problèmes actifs à partir d'un intervalle. Pour définir ces deux opérateurs, trois nouveaux concepts sont introduits. Ce sont les concepts de **numéro**, de **poids** et de **portée** d'un sous-problème.

Les opérateurs de *pliage* et *dépliage* sont définis à l'aide du concept de *portée* d'un sous-problème. Les concepts de *numéro* et de *poids* d'un sous-problème sont utilisés pour définir la *portée* d'un sous-problème. Le concept de *poids* est employé pour définir le *numéro* d'un sous-problème. Dans cette section, nous présentons successivement donc le *poids* d'un sous-problème, son *numéro*, sa *portée*, l'opérateur de *pliage* et l'opérateur de *dépliage*. Chaque concept est présenté indépendamment de la structure de l'arbre du B&B, et donc du problème combinatoire sous-jacent. Toutefois, une définition plus concise est donnée pour les arbres de certains problèmes d'optimisation combinatoire, tels que le QAP, le TSP ou le Flow-Shop. Ces problèmes sont très étudiés dans la littérature de l'optimisation combinatoire. En plus des deux opérateurs de pliage et dépliage, B&B@Grid définit également deux types de processus. Il s'agit du **processus coordinateur** et du **processus B&B**. Ces deux processus permettent de paralléliser un B&B avec tout paradigme. Cette section se termine donc par la présentation de ces deux processus.

2.1.1 Poids d'un sous-problème

Le poids d'un sous-problème n de l'arbre, noté *poids(n)*, est le nombre de feuilles du sous-arbre dont il est la racine. La formule (2.1) définit le poids d'un sous-problème d'une façon récursive. Comme l'indique la formule (2.1), le poids d'une feuille est égal à 1, et le poids d'un sous-problème interne est égal à la somme des poids de ses sous-problèmes fils. Cette définition est à la fois générale et inapplicable directement. En effet, elle est générale puisque elle définit le poids d'un sous-problème pour tout arbre, et inapplicable puisque la taille de l'arbre est exponentielle.

$$\text{poids}(n) = \begin{cases} 1 \ \ si \ \ \text{fils}(n) = \phi \\ \sum_{i \in \text{fils}(n)} \text{poids}(i) \quad \text{sinon} \end{cases} \tag{2.1}$$

Cependant, la connaissance de la structure d'un arbre permet de simplifier cette définition. Les formules (2.2) et (2.3) définissent, d'une façon plus simple, le poids d'un sous-problème n pour respectivement un arbre binaire et un arbre de permutation. Dans ces deux définitions, la profondeur d'un sous-problème n, notée *profondeur(n)*,

est le nombre de sous-problèmes se trouvant sur le même chemin qui le séparent de la racine de l'arbre. P est la profondeur associée aux feuilles de l'arbre. L'arbre binaire est celui où, hormis les feuilles, chaque sous-problème a deux sous-problèmes fils. L'arbre de permutation est celui associé aux problèmes où il est question de retrouver une permutation parmi un ensemble fini d'éléments.

$$\text{poids}(n) = 2^{(P-\text{profondeur}(n))} \tag{2.2}$$

$$\text{poids}(n) = (P - \text{profondeur}(n))! \tag{2.3}$$

Plusieurs problèmes d'optimisation combinatoire peuvent être représentés par un arbre de permutation. Dans ce type d'arbre, tout sous-problème n, hormis le sous-problème racine, respecte la condition (2.4).

$$|\text{fils}(n)| = |\text{fils}(\text{père}(n))| - 1 \tag{2.4}$$

A l'instar de l'arbre binaire, de l'arbre de permutation, et de tout autre arbre de structure régulière, les sous-problèmes de même profondeur ont le même poids. Par conséquent, au lieu d'associer les poids aux sous-problèmes, il est plus simple de leur associer leurs profondeurs, et de déduire le poids d'un sous-problème à partir de sa profondeur. Ainsi au début de l'algorithme B&B, un vecteur qui donne le poids associé à chaque profondeur est calculé. A l'aide de ce vecteur, il est possible de retrouver le poids d'un sous-problème en connaissant sa profondeur. La figure FIG. 2.2 montre un exemple des poids associés aux profondeurs dans un arbre de permutation.

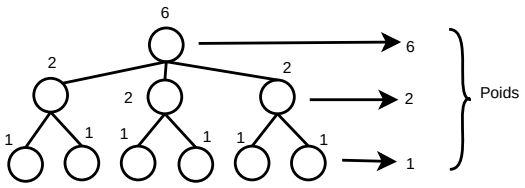

FIG. 2.2 – Poids d'un sous-problème.

2.1.2 Numéro d'un sous-problème

A chaque sous-problème n de l'arbre, on associe un numéro noté *numéro(n)*. Comme l'indique la formule (2.5), le numéro d'un sous-problème n peut être obtenu à l'aide des sous-problèmes de son chemin. Le chemin d'un sous-problème n, noté $chemin(n)$, est l'ensemble des sous-problèmes rencontrés en allant de la racine de l'arbre au sous-problème n. Le sous-problème n et la racine de l'arbre sont donc toujours inclus dans $chemin(n)$. Pour retrouver le numéro d'un sous-problème n, il suffit alors de connaître

les précédents de chaque sous-problème du chemin de n. Les précédents d'un sous-problème n, noté *précédents(n),* est l'ensemble des sous-problèmes frères de n qui sont générés avant n.

$$\text{numéro}(n) = \sum_{i \in \text{chemin}(n)} \sum_{j \in \text{précédents}(i)} \text{poids}(j) \tag{2.5}$$

La définition (2.5) s'applique à tout arbre indépendamment de sa structure. La formule (2.6) donne une définition plus simple pour les arbres de structure régulière tels que les arbres binaires ou les arbres de permutation. Cette définition repose sur le constat que, dans ce type d'arbre, les sous-problèmes de même profondeur ont un poids similaire. Pour avoir le numéro d'un sous-problème, il suffit donc de connaître le chemin d'un sous-problème et le rang de chaque sous-problème de ce chemin. Le rang d'un sous-problème n, noté $rang(n)$, est la position de n parmi ses sous-problèmes-frères. Ainsi, lors de la génération des sous-problèmes-fils d'un sous-problème quelconque, le rang du premier sous-problème généré est égal à 0, le rang du deuxième sous-problème généré est égal à 1, et ainsi de suite. La figure FIG. 2.4 donne un exemple illustrant les numéros obtenus dans un arbre de permutation.

$$\text{numéro}(n) = \sum_{i \in \text{chemin}(n)} \text{rang}(i) * \text{poids}(i) \tag{2.6}$$

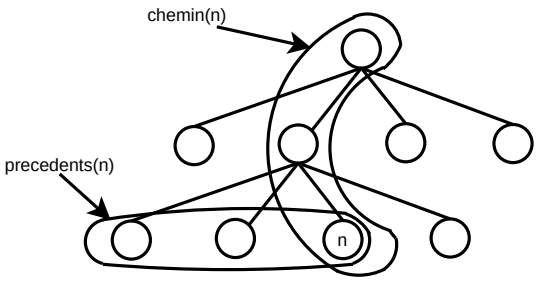

FIG. 2.3 – Chemin et précédents d'un sous-problème.

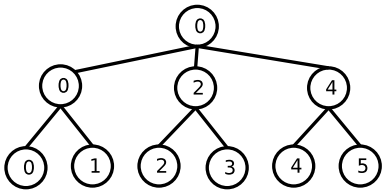

FIG. 2.4 – Numéro d'un sous-problème.

2.1.3 Portée d'un sous-problème

La portée d'un sous-problème n, noté portée(n), est l'intervalle auquel appartiennent les numéros des sous-problèmes du sous-arbre dont n est la racine. La figure FIG. 2.5 donne un exemple de la portée associée à chaque sous-problème d'un arbre de permutation. Comme l'indique la formule (2.7), le début de la portée d'un sous-problème est égal à son numéro, et sa fin est égale à la somme de son numéro et de son poids.

$$\text{portée}(n) = \quad [\text{numéro}(n), \text{numéro}(n) + \text{poids}(n)[\tag{2.7}$$

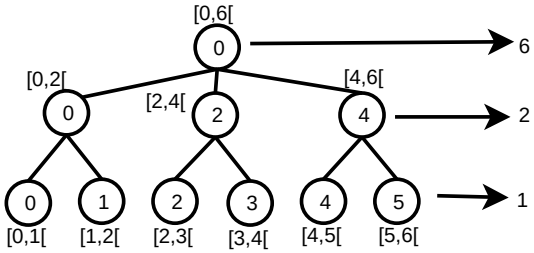

FIG. 2.5 – Portée d'un sous-problème.

2.1.4 Opérateur de pliage

Le rôle de cet opérateur est de déduire, à partir d'une liste N de sous-problèmes actifs, l'intervalle auquel appartiennent les numéros de sous-problèmes pouvant être explorés en partant des sous-problèmes de N. Cet intervalle est noté $Fold(N)$. Comme l'indique la formule (2.8), ceci revient à calculer l'union de toutes les portées des sous-problèmes de N.

$$\text{Fold}(N) = \cup_{i \in N} \ \text{portée}(i) \tag{2.8}$$

Dans un B&B, la disposition des sous-problèmes actifs dans une liste N dépend de la stratégie de parcours adoptée par l'opérateur de sélection. Soient N_1, N_2 ... N_k les rangs par lesquels ces sous-problèmes sont ordonnés, et $[A_1,B_1[$, $[A_2,B_2[$ ... $[A_k,B_k[$ leurs portées respectives. La condition (2.9) est toujours vérifiée lorsque la stratégie de parcours est de type **profondeur d'abord**. Ainsi, $intervalle(N)$ peut être retrouvé sans calculer toutes les portées des sous-problèmes actifs. Comme l'indique la formule (2.10), il suffit de connaître les portées de A_1 et A_k, ou plus simplement, il suffit de connaître les numéros de ces deux sous-problèmes et le poids de A_k. La figure FIG. 2.6 donne un exemple illustrant le passage d'une liste de sous-problèmes actifs vers un intervalle en utilisant l'opérateur de pliage.

$$\forall\ i < k\ \ B_i = A_{i+1} \tag{2.9}$$

$$\text{Fold}(N) = \ [\text{numéro}(A_1), \text{numéro}(A_k) + \text{poids}(A_k)[\tag{2.10}$$

2.1.5 Opérateur de dépliage

Cet opérateur est chargé de déduire, à partir d'un intervalle [A,B[, une liste de sous-problèmes actifs notée $Unfold([A,B[)$. Comme l'indique la formule (2.11), *Unfold*([A,B[) est constitué des sous-problèmes de l'arbre dont la portée est incluse dans [A,B[, et dont la portée de leur père n'est pas incluse dans [A,B[. Ces deux conditions garantissent que $Unfold([A,B[)$ est une liste unique et minimale. En effet, il est impossible de trouver une autre liste dont la cardinalité est inférieure à $Unfold([A,B[)$, et qui permet d'explorer les sous-problèmes dont le numéro appartient à [A,B[. La cardinalité d'une liste est le nombre d'éléments qu'elle contient.

$$\text{Unfold}([A,B[) = \left\{ \begin{array}{c} n/ \\ \text{portée}(n) \subseteq [A,B[\\ \text{portée}(p) \not\subset [A,B[\\ p = \text{père}(n) \end{array} \right\} \tag{2.11}$$

$$\text{élimination}(n) = \left(\begin{array}{c} \text{portée}(n) \subseteq [A,B[\\ ou \\ (\text{portée}(n) \cap [A,B[) = \phi \end{array} \right) \tag{2.12}$$

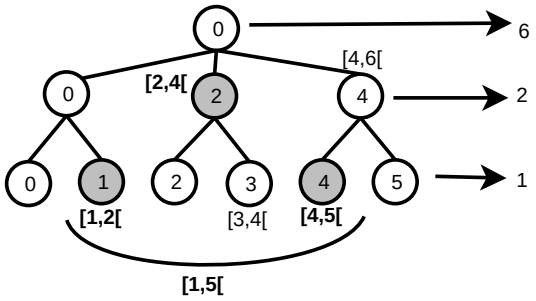

FIG. 2.6 – Correspondance entre ensemble de sous-problèmes actifs et intervalle.

Le calcul de $Unfold([A,B[)$ peut être fait à l'aide d'un algorithme B&B dont les opérateurs, hormis l'opérateur d'élagage, sont les mêmes que ceux du B&B. Cet algorithme est basé sur la portée d'un sous-problème pour choisir entre un élagage ou une décomposition. Comme l'indique la formule (2.12), un sous-problème n est élagué lorsque sa portée est incluse dans [A,B[, ou lorsque sa portée et [A,B[sont totalement disjoints. Dans le cas contraire, le sous-problème n est décomposé. Dans un arbre de profondeur maximale P, cet algorithme B&B opère au plus P décompositions. Ceci garantit le faible coût de l'opérateur de dépliage. Comme l'indique la formule (2.13), la

liste de Unfold([A,B[) est constituée alors par tous les sous-problèmes élagués et dont l'intervalle est inclus dans [A,B[. La figure FIG. 2.6 donne un exemple illustrant le passage d'un intervalle vers une liste de sous-problèmes actifs.

$$\text{Unfold}([A,B[) = \left\{ \begin{array}{c} n/ \\ \text{élagage}(n) \\ \text{portée}(n) \subseteq [A,B[\end{array} \right\} \quad (2.13)$$

2.1.6 Processus coordinateur et B&B

En plus des deux opérateurs de pliage et de dépliage, B&B@Grid utilise deux types de processus. Il s'agit du processus coordinateur et du processus B&B. Un processus coordinateur s'occupe de la gestion d'un ensemble d'intervalles. Le rôle d'un processus B&B est de traiter un intervalle. L'intervalle est donc l'unité de travail allouée aux processus B&B. Le coordinateur gère principalement un ensemble d'intervalles. Ces intervalles sont stockés dans un ensemble appelé INTERVALLES. Dans la suite de la section, INTERVALLES est supposé être comme un ensemble d'intervalles non encore traités, même si, en fait, il peut être également un ensemble d'intervalles traités. Par contre, INTERVALLES ne peut pas contenir à la fois des intervalles traités et non traités.

Il est facile de passer de l'ensemble des intervalles traités à l'ensemble des intervalles non encore traités et vice versa. Il suffit de connaître l'ensemble de départ et de faire une opération de soustraction. Soit {[0,20[} l'ensemble initial des intervalles à explorer, et {[3,6[, [9,12[, [15,20[} l'ensemble des intervalles qui restent à explorer au bout d'un certain temps. L'ensemble des intervalles explorés depuis le début est donc égal à {[0,3[, [6,9[, [12,15[}. L'utilisation de B&B@Grid avec les approches fermier-travailleur proposées dans la littérature requiert la manipulation d'un ensemble d'intervalles non encore traités. Par contre, l'utilisation de B&B@Grid avec l'approche pair-à-pair proposée dans [IF00] requiert la manipulation d'un ensemble d'intervalles traités.

Le processus B&B ne gère qu'un seul intervalle à la fois. Il reçoit cet intervalle d'un processus coordinateur, utilise l'opérateur de dépliage pour déduire l'ensemble des sous-problèmes à résoudre, traite une partie ou la totalité de ces sous-problèmes à l'aide de l'algorithme B&B, utilise l'opérateur de pliage pour déduire l'intervalle qui correspond aux sous-problèmes qui restent, et communique l'intervalle déduit à un processus coordinateur.

2.1.7 Application au problème du Flow-Shop Scheduling

Afin de valider l'approche B&B@Grid, les expérimentations décrites dans cette section ainsi que celles présentées dans le reste de cette thèse sont faites sur un problème de Flow-Shop. Ce problème fait partie des problèmes d'ordonnancement. Un problème d'ordonnancement est défini par un ensemble de tâches et de ressources. Le Flow-Shop

est un problème multi-opération où chaque opération est le traitement d'une tâche par une machine. Dans le Flow-Shop, les ressources sont considérées comme des machines se trouvant dans un atelier de production. Ces machines traitent les tâches selon le principe de la chaîne de production. Les machines sont donc disposées dans un certain ordre. Ainsi, une machine ne peut commencer le traitement d'une tâche que lorsque toutes les machines, qui se trouvent en amont, ont fini leur traitement. Une durée est associée à chaque opération, celle-ci correspond au temps nécessaire à la machine pour finir son traitement. Une opération ne peut pas être interrompue, et les machines sont des ressources critiques, car une machine ne peut traiter qu'une tâche à la fois. La figure FIG. 2.7 montre un exemple d'une solution d'une instance d'un problème de Flow-Shop définie par 6 tâches et 3 machines.

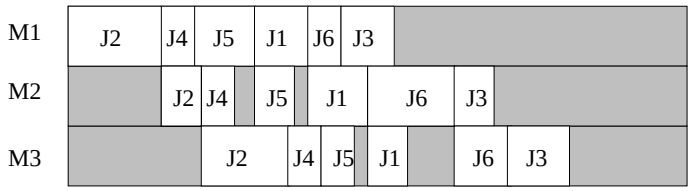

FIG. 2.7 – Instance du problème du Flow-Shop avec 6 tâches et 3 machines.

Il existe plusieurs variantes du Flow-Shop. Dans cette thèse, l'intérêt est porté aux variantes notées $(F/permut/C_{max})$ et $(F/d_i, permut/C_{max}, \bar{T})$. Cette façon de noter les différentes variantes du Flow-Shop, et les problèmes d'ordonnancement en général, est présentée dans l'annexe A. Les notations $(F/permut/C_{max})$ et $(F/d_i, permut/C_{max}, \bar{T})$ correspondent respectivement à des problèmes mono-critère et bi-critère, où les critères à optimiser sont d'une part le *makespan* (C_{max}) seul, et d'autre part le *makespan* et le *retard total* $(\bar{T})$ simultanément. Le *makespan* d'une solution correspond à la date de fin de sa dernière tâche sur la dernière machine. L'objectif est donc de trouver la solution qui minimise le *makespan*. Dans $(F/d_i, permut/C_{max}, \bar{T})$, une contrainte de type date due, notée d_i, est ajoutée au problème. Cette contrainte signifie que pour chaque tâche une date, à laquelle celle-ci devrait finir, est donnée. Au-delà de ce délai, la tâche est dite en retard. Le *retard total* d'une solution correspond à la somme des retards de toutes ses tâches. Dans $(F/d_i, permut/C_{max}, \bar{T})$, l'objectif est de trouver les solutions qui minimisent le *makespan* et le *retard total* simultanément.

Dans [GJS76], il a été montré que la minimisation du C_{max} est NP-difficile à partir de trois machines, et $\bar{T}$ est, quant à lui, NP-difficile même sur une seule machine, sauf dans le cas d'une machine où toutes les dates de fin sont dépassées. Ceci a été démontré dans [DL90b]. Dans toutes nos expérimentations, la borne inférieure utilisée pour le *makespan* est celle décrite dans [LLK78], et les bornes utilisées pour le *retard total* sont décrites dans l'annexe A. Pour le *retard total*, on sélectionne toujours la borne qui donne le meilleur résultat.

Dix expérimentations avec B&B@Grid ont été effectuées pour évaluer la capacité de cette nouvelle approche à optimiser les communication des sous-problèmes, et la comparer avec les autres approches connues dans la littérature. Chaque expérimentation utilise une des dix instances définies par 50 tâches et 20 machines et publiées dans [Tai93]. Ce sont des instances du problème ($F/permut/C_{max}$). Leurs solutions optimales ne sont pas encore connues, mais le site [6] publie les meilleures solutions trouvées pour chaque instance. Ces instances sont notées *Ta051*, *Ta052*, *Ta053*, *Ta054*, *Ta055*, *Ta056*, *Ta057*, *Ta058*, *Ta059* et *Ta060*. La grille utilisée pendant nos expérimentations est celle présentée dans le premier chapitre.

Un seul processus coordinateur est utilisé et un processeur B&B est lancé sur chaque processeur libre de notre grille. Les dix expérimentations sont faites selon l'approche fermier-travailleur, décrite dans la section 2.2, et utilisent les stratégies de tolérance aux pannes et d'équilibrage de charge présentées respectivement dans les sections 2.3 et 2.4. Afin de sauvegarder leurs intervalles, les processus B&B communiquent leurs intervalles au coordinateur toutes les 3 minutes. Cette valeur est prise au hasard puisque le système ne semble pas fortement sensible au choix de la valeur de ce paramètre. Chaque expérimentation a été initialisée par la meilleure solution connue de l'instance utilisée. Les expérimentations ont été arrêtées après une heure de calcul puisque l'objectif n'était pas de les résoudre.

Le tableau TAB. 2.1 permet de comparer les taux de compression des sous-problèmes communiqués avec les approches à jeton de PICO [EPH00], à variables de [IF00], et B&B@Grid. Le taux de compression d'une approche, généralement exprimé en pourcentage, est défini comme le rapport de la taille des données après compression sur leur taille initial avant compression. Pour obtenir le taux de compression de chaque approche, notre algorithme calcule la taille de chaque intervalle communiqué, détermine le codage associé à cet intervalle avec les approches à jetons et à variables, calcule la taille de chaque codage, et compare les trois tailles ainsi obtenues avec la taille des sous-problèmes associés à l'intervalle communiqué. Les deux premières colonnes du tableau désignent respectivement le nom de l'instance expérimentée et le nombre moyen de processeurs exploités pendant l'expérimentation. Les autres colonnes donnent le taux de compression de chaque approche. La dernière ligne donne le nombre moyen de processeurs, exploités durant les 10 expérimentations, et le taux de compression moyen de chaque approche.

Le tableau TAB. 2.1 montre que, dans les dix expérimentations, le taux de compression de B&B@Grid est toujours le meilleur. Le taux de compression moyen obtenu par B&B@Grid est d'environ $0,13\%$. Il signifie que cette approche réduit, en moyenne, de 766 fois la taille des sous-problèmes communiqués. Les rapports entre les taux de compression moyens permettent de comparer les différentes approches entre elles. Ainsi les rapports du taux de compression moyen de B&B@Grid avec ceux des autres approches

[6] http ://ina2.eivd.ch/Collaborateurs/etd/problemes.dir/ordonnancement.dir/ordonnancement.html

montrent que le codage de B&B@Grid est en moyenne 92 fois plus petit que celui de l'approche à jetons, et 465 fois plus petit que celui de l'approche à variables.

Inst.	Nbr. Moy. Proc.	App. Var.(%)	App. Jet.(%)	B&B@Grid(%)
Ta051	0692,0	62,4133	11,9278	00,1386
Ta052	1276,3	60,2331	11,9170	00,1295
Ta053	1055,9	60,8091	11,7572	00,1264
Ta054	1236,7	61,6834	11,9346	00,1256
Ta055	1312,3	60,1542	11,9537	00,1285
Ta056	1213,0	61,2483	12,0936	00,1300
Ta057	1442,6	60,7235	12,0155	00,1291
Ta058	1432,2	61,6729	11,9850	00,1248
Ta059	1448,2	60,0258	11,9016	00,1293
Ta060	1398.7	61,0899	12,0405	00,1267
Moy.	**1111**	**61,0053**	**11,9527**	**00,1289**

TAB. 2.1 – Taux de compression des sous-problèmes communiqués.

2.2 Déploiement fermier-travailleur à grande échelle

Les deux processus coordinateur et B&B peuvent être utilisés pour paralléliser un B&B avec n'importe quel paradigme parallèle. Dans une approche fermier-travailleur, par exemple, le fermier héberge un seul coordinateur et les processus travailleurs hébergent des processus B&B. Dans une approche pair-à-pair, un pair héberge un coordinateur et un processus B&B. L'approche pair-à-pair est celle qui permet d'exploiter le plus grand nombre de processeurs. Toutefois, afin de tester et de valider B&B@Grid, le paradigme fermier-travailleur est choisi dans cette thèse. Dans ce paradigme, un seul hôte joue le rôle de fermier, et tous les autres celui de travailleur. Ce paradigme est relativement simple à mettre en œuvre. Son inconvénient majeur est que le fermier peut constituer un goulot d'étranglement. Cependant, communiquer et manipuler des intervalles au lieu de listes de sous-problèmes permet de réduire le coût des communications et la charge du fermier. Ce paradigme est donc choisi pour expérimenter cette approche. Le but est de montrer que B&B@Grid arrive à pousser les limites d'un paradigme aussi centralisé que celui du fermier-travailleur. Dans le paradigme fermier-travailleur adopté, les hôtes travailleurs hébergent autant de processus B&B qu'ils ont de processeurs, et le hôte fermier héberge le coordinateur. La figure FIG. 2.8 donne un exemple avec quatre processus B&B et un coordinateur. Dans cet exemple, quatre intervalles restent à explorer. Trois de ces intervalles sont en cours d'exploration, [1,5[, [9,19[, et [20,22[, tandis que le quatrième, [25,40[, est en attente d'un processus B&B.

Le tableau TAB. 2.2 indique les taux d'utilisation CPU du coordinateur pendant les expérimentations décrites dans la section 2.1.7. La première colonne donne le nom

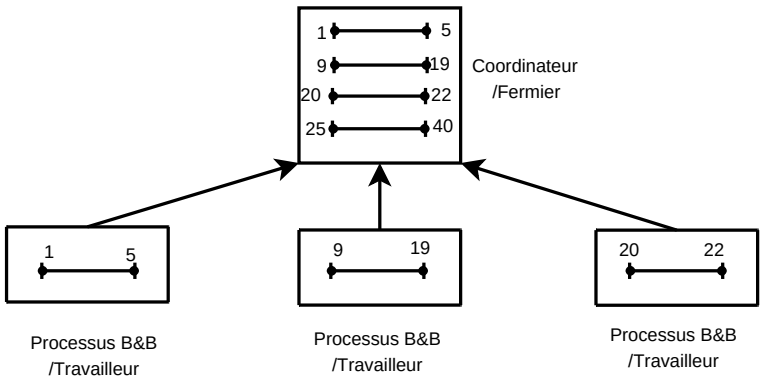

FIG. 2.8 – Exemple avec quatre processus B&B et un coordinateur.

de l'instance utilisée, la deuxième colonne le nombre moyen de processeurs exploités pendant chaque expérimentation, et la troisième colonne le taux d'utilisation CPU du processeur qui héberge le coordinateur. Comme l'indique ce tableau, malgré le nombre considérable de processeurs exploités, le taux d'utilisation du processeur par le coordinateur reste faible. Environ 1111 processus B&B sont lancés en moyenne et le taux d'utilisation CPU du processeur par le coordinateur est égal en moyenne à $1,29\%$. Ce taux est un bon indicateur de la capacité de passage à l'échelle de B&B@Grid. Dans un paradigme centralisé, tel que celui du fermier-travailleur, le goulot d'étranglement survient lorsque le processeur du fermier utilise le processeur à 100%. Par ailleurs, il faut noter que le passage à l'échelle dépend également de la taille de l'information reçue par le coordinateur. En effet, le réseau ne doit pas être saturé. Comme indiqué auparavant, B&B@Grid réduit la taille de l'information communiquée. Par conséquent, le taux de réduction de la taille des sous-problèmes et le taux d'utilisation CPU du coordinateur permettent à B&B@Grid de passer à l'échelle.

Inst.	**Nbr. Moy. Proc.**	**Taux. Util. CPU Coord.(%)**
Ta051	0692,0	00,39
Ta052	1276.3	01,29
Ta053	1055.9	01,52
Ta054	1236.7	01,43
Ta055	1312.3	01,27
Ta056	1213.0	01,26
Ta057	1442.6	01,44
Ta058	1432.2	01,33
Ta059	1448.2	01,59
Ta060	1398.7	01,47
Moy.	**1111**	**01,29**

TAB. 2.2 – Taux d'utilisation CPU par le processus coordinateur.

2.3 Tolérance aux pannes

La tolérance aux pannes est l'aptitude d'un système à fonctionner malgré ses défaillances éventuelles. Dans les méthodes heuristiques, une perte de traitement est autorisée puisque ce type de méthodes cherche une (ou plusieurs) solution(s) approchée(s). Par contre, la tolérance aux pannes est plus cruciale pour les méthodes exactes. Une perte d'un (ou plusieurs) sous-problème(s) risque d'entraîner une perte définitive d'une (ou plusieurs) solution(s) optimale(s). Par conséquent, il est indispensable de définir une stratégie de tolérance aux pannes pour toute méthode exacte destinée à exploiter un environnement de calcul parallèle volatile. Sur une grille, la tolérance aux pannes peut être faite au niveau de l'intergiciel ou au niveau applicatif. Au niveau intergiciel, les stratégies proposées sont souvent indépendantes des applications. A l'instar de $XtremWeb$, le processus en panne est souvent relancé depuis le début. Une telle stratégie peut s'avérer inefficace pour les processus à longue durée de vie. Au niveau applicatif, les stratégies proposées sont souvent basées sur la sauvegarde et la restauration de l'état d'un processus (check-pointing). Le processus est alors relancé à partir de son dernier point de sauvegarde.

Il est donc important d'identifier les données à sauvegarder lors de la définition d'une stratégie de tolérance aux pannes. Dans le B&B, l'information sauvegardée par chaque processus contient principalement la (ou les) meilleure(s) solution(s) trouvée(s) et le ou le(s) sous-problème(s) non encore traité(s). Dans B&B@Grid, le coordinateur gère une éventuelle panne de sa machine hôte en sauvegardant périodiquement dans un fichier le contenu de l'ensemble INTERVALLES et l'ensemble des solutions trouvées. En cas de panne du fermier, le coordinateur est lancé en initialisant INTERVALLES par le contenu de ce fichier. Le coordinateur récupère également les solutions sauvegardées.

Un processus B&B utilise l'opérateur de pliage pour déduire l'intervalle auquel appartiennent leurs sous problèmes non encore explorés. Il gère les pannes du travailleur en actualisant régulièrement la copie de son intervalle dans INTERVALLES, et en informant le coordinateur de toute nouvelle solution trouvée. En cas de panne d'un travailleur, la dernière copie de son intervalle est soit affectée en totalité à un autre processus B&B, soit partagée entre plusieurs processus B&B. Un processus B&B actualise la copie de son intervalle à l'aide d'une simple opération d'intersection d'intervalles. Cette opération actualise à la fois l'intervalle en cours de traitement et sa copie dans INTERVALLES. Soient [A,B[un intervalle en cours de traitement dans un processus B&B, et [A',B'[sa copie dans INTERVALLES au niveau du coordinateur. Comme l'indique la formule (2.14), l'intersection entre deux intervalles se fait en considérant le maximum de leurs débuts et le minimum de leurs fins.

$$[A, B[\cap[A', B'[= [max(A, A'), min(B, B')[\tag{2.14}$$

Au cours d'une résolution, les deux intervalles [A,B[et [A',B'[évoluent constamment. En effet, un processus B&B, en traitant [A,B[, incrémente la valeur de A et laisse

la valeur de B inchangée. Par contre, le mécanisme d'équilibrage de charge, expliqué dans la section suivante, décrémente la valeur de B' et laisse la valeur de A' inchangée. Le début d'un intervalle est susceptible également d'être incrémenté par plusieurs processus B&B. Ceci survient lorsque la stratégie d'équilibrage de charge attribut le même intervalle à plusieurs processus.

La figure FIG. 2.9 illustre la stratégie de tolérance aux pannes de B&B@Grid à l'aide d'un exemple. Au début, le coordinateur contient l'intervalle [0,6[. Cet intervalle peut correspondre, par exemple, à une instance du Flow-Shop de 6 tâches. L'intervalle [0,6[est attribué en entier au premier processus qui rejoint la grille. Lorsqu'un deuxième processus rejoint le calcul, le coordinateur lui accorde l'intervalle [3,6[. Chacun des deux processus traite l'intervalle reçu. Après une certaine période, le premier processus contacte le coordinateur pour lui communiquer son intervalle. Cet intervalle représente l'ensemble des numéros des sous-problèmes non traitées. Dans l'exemple illustré par la figure FIG. 2.9, l'intervalle restant au premier processus est égal à [1,6[, car les sous-problèmes portant le numéro 0 sont explorés. Le coordinateur opère alors une intersection entre l'intervalle reçu et sa copie au niveau du coordinateur.

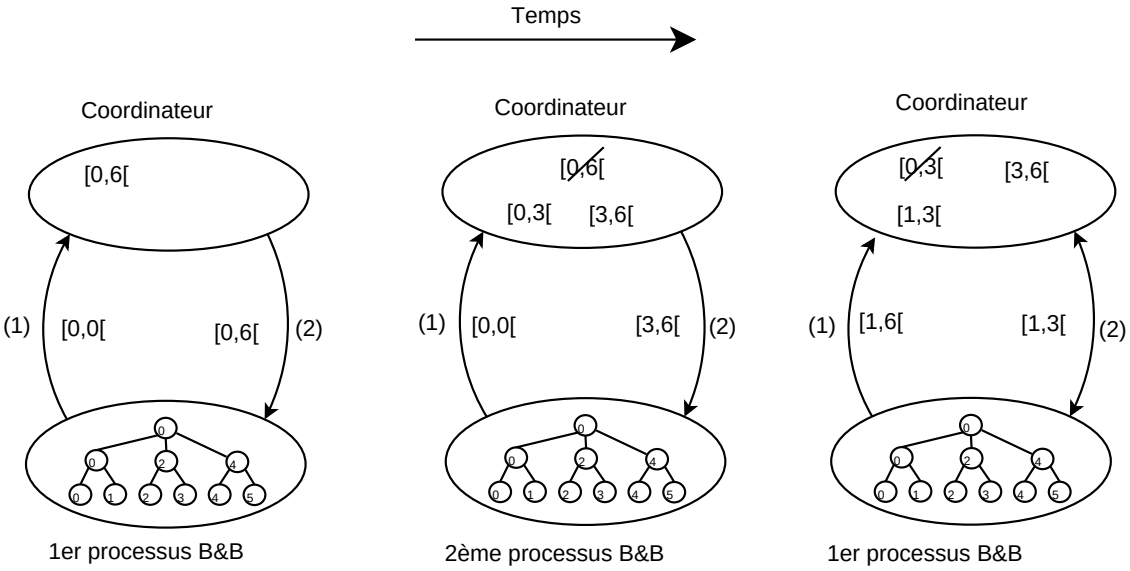

FIG. 2.9 – Stratégie de tolérance aux pannes de B&B@Grid.

Afin d'être efficace dans la gestion de la tolérance aux pannes, les intervalles de l'ensemble INTERVALLES doivent être ordonnés. Deux listes de références sont utilisées pour ordonner ces intervalles. Il s'agit des listes AJOUR et IDENTITE. Ces deux listes permettent respectivement de retrouver un intervalle dans INTERVALLES d'une façon plus efficace, et de détecter les processus en panne. Il est important de noter que INTERVALLES contient les copies des intervalles en cours de traitement, tandis

que AJOUR et IDENTITE ne contiennent que des références vers les intervalles de INTERVALLES. La cardinalité des ensembles INTERVALLES, AJOUR et IDENTITE est toujours identique.

IDENTITE ordonne les intervalles selon leur identifiant. L'identifiant d'un intervalle est juste un numéro. Un compteur est utilisé par le coordinateur pour générer un nouvel identifiant à chaque fois qu'un nouvel intervalle est créé. Cet identifiant est attribué alors au nouvel intervalle. IDENTITE est utilisée lorsque un processus B&B contacte le coordinateur pour actualiser la copie de son intervalle. Le but est de retrouver rapidement la bonne copie de INTERVALLES à actualiser. Il suffit de lancer une recherche dans IDENTITE selon l'identifiant de l'intervalle reçu. Cette recherche peut être de type dichotomique, et peut donc être faite en $O(log(N))$, où N est le nombre d'intervalles de IDENTITE.

Comme indiqué auparavant, la liste AJOUR est utilisée pour détecter les processus en panne. Cette liste ordonne les intervalles selon leur dernier instant de mise à jour. A chaque fois qu'un intervalle est mis à jour, sa référence est retirée de la liste AJOUR, et remise au début de cette liste. Lorsqu'un intervalle est créé, sa référence est toujours placée au début de AJOUR. La référence du dernier intervalle mis à jour se trouve donc toujours à la fin de la liste. Il est alors facile de savoir si son processus est en panne. Il suffit de comparer sa dernière date de mise à jour avec le temps actuel. Si la période écoulée dépasse la période de mise à jour alors le processus chargé d'explorer cet intervalle est en panne. Par conséquent, cet intervalle peut être assigné à un autre processus.

Dans cette section, nous expliquons la stratégie de tolérance aux pannes de B&B@Grid à l'aide du paradigme fermier-travailleur. Cependant, le principe de cette stratégie reste valable pour les autres paradigmes. Très peu de modifications sont nécessaires pour utiliser cette stratégie dans l'approche de tolérance aux pannes pair-à-pair proposée dans [IF00]. Il suffit à chaque pair d'utiliser une liste d'intervalles à la place de la liste de sous-problèmes résolus. Un pair intègre alors le fonctionnement du coordinateur décrit dans cette section. Cependant, l'ensemble INTERVALLES ne contient pas la liste des intervalles non traités, mais ceux traités. Les listes IDENTITE et AJOUR doivent être également employées pour manipuler plus efficacement l'ensemble INTERVALLES. Néanmoins, le pair doit utiliser l'opération d'union d'intervalles au lieu d'utiliser l'opération d'intersection. Comme l'indique la formule (2.15), l'union de deux intervalles se fait en considérant le minimum de leurs débuts et le maximum de leurs fins. La formule (2.15) impose bien évidemment que les deux intervalles [A,B[et [A',B'[ne soient pas disjoints.

$$[A, B[\cup [A', B'[= [min(A, A'), max(B, B')[\tag{2.15}$$

Le tableau TAB. 2.3 permet de comparer B&B@Grid et les autres approches en terme de la taille global des sous-problèmes stockés par le coordinateur. Ces sous-

problèmes représentent l'état global d'une résolution. En les sauvegardant, il est toujours possible de réinitialiser une résolution et de la relancer à partir de son point de sauvegarde. Les dix expérimentations présentées dans le tableau TAB. 2.3 sont celles décrites dans la section 2.1.7. Pour chaque expérimentation, la première colonne de ce tableau indique le nom de l'instance utilisée, et la deuxième colonne le nombre moyen de processeurs exploités. Les troisième, quatrième et cinquième colonnes donnent le taux de compression de chaque approche. Le tableau TAB. 2.3 indique principalement que B&B@Grid réduit la taille des sous-problèmes du coordinateur, et améliore les taux de compressions des autres approches, dans les mêmes proportions qu'elle le fait pour les sous-problèmes communiqués.

Inst.	**Nbr. Moy. Proc.**	**App. Var.(%)**	**App. Jet.(%)**	**B&B@Grid(%)**
Ta051	0692,0	62,2347	11,8811	00,1414
Ta052	1276,3	60,2902	12,0052	00,1319
Ta053	1055,9	60,6217	11,9170	00,1295
Ta054	1236,7	61,6560	11,9745	00,1273
Ta055	1312,3	60,1562	11,9791	00,1302
Ta056	1213,0	60,2080	11,9635	00,1300
Ta057	1442,6	60,1562	11,9791	00,1302
Ta058	1432,2	61,5869	11,9647	00,1259
Ta059	1448,2	59,8684	11,9736	00,1315
Ta060	1398.7	60,9254	11,9537	00,1285
Moy.	**1111**	**60,7704**	**11,9592**	**00,1306**

TAB. 2.3 – Taux de compression de sous-problèmes du coordinateur.

2.4 Equilibrage de charge

Lors de la parallélisation de B&B, il est nécessaire de définir une politique de régulation de charge. L'objectif de cette politique est de minimiser l'occurrence de situations dans lesquelles des processus ont beaucoup de sous-problèmes à traiter pendant que d'autres ont des listes vides. Cette politique initie, aux moments opportuns, des transferts de sous-problèmes des machines en surcharge vers les machines en sous-charge.

Une stratégie d'équilibrage de charge peut être statique ou dynamique. Dans une stratégie d'équilibrage de charge statique [BL02, BdKT95], l'allocation de toutes les tâches est décidée avant le début d'un calcul. Par contre, dans une stratégie d'équilibrage de charge dynamique [Sin98], les tâches sont allouées et réallouées durant le calcul. Ceci est fait en fonction de l'état du système. L'équilibrage de charge statique est intéressant lorsque le coût de calcul des tâches est connu à l'avance. Dans un algorithme B&B, il est impossible de connaître le coût des tâches avant leurs traitements. Par conséquent, une stratégie d'équilibrage de charge pour un tel algorithme ne peut

être que dynamique.

A notre connaissance, aucune stratégie, décrite dans la littérature pour équilibrer la charge entre processus B&B, ne prend en compte la nature hétérogène et volatile des systèmes de calcul distribués. Or, les processeurs d'une grille de calcul ont souvent des puissances variées, ne sont pas forcément dédiés au calcul, et leur nombre varie constamment. En plus d'être dynamique, une stratégie d'équilibrage de charge d'un B&B sur grilles de calcul doit donc être également adaptative. Autrement dit, la granularité du travail alloué aux processeurs doit s'adapter à leur puissance, leur disponibilité et leur nombre. B&B@Grid adapte la taille des intervalles à la puissance et la disponibilité du processeur demandeur de travail, ainsi qu'au nombre de processeurs participants au calcul. En effet, dans B&B@Grid, la granularité des unités de travail, autrement dit les tailles des intervalles, accordées par le(s) coordinateur(s) aux processus B&B dépendent toujours des puissances, des disponibilités et du nombre de processeurs intervenants dans le calcul.

Dans B&B@Grid, le travail d'un processus B&B consiste à explorer un intervalle. A la réception d'un intervalle, un processus B&B utilise l'opérateur de dépliage pour déduire la liste de sous-problèmes à traiter. Un processus B&B sollicite un intervalle lorsqu'il rejoint le calcul pour la première fois, ou lorsqu'il termine l'exploration de son intervalle. Le rôle du coordinateur est alors de lui assigner un intervalle. Le coordinateur tente d'abord de sélectionner un intervalle libre. Un intervalle est considéré comme libre lorsqu'il n'est alloué à aucun processus B&B.

Durant une résolution, certains intervalles de l'ensemble INTERVALLES peuvent être ou devenir libres. Ceci survient notamment lorsque le processus qui traite un intervalle tombe en panne. Comme expliqué dans la section précédente, la consultation de l'ensemble AJOUR permet au coordinateur de détecter ce type d'intervalles libres. L'autre situation où des intervalles peuvent devenir libres est celle où le coordinateur tombe en panne. Après ce type de panne, le coordinateur initialise, à son démarrage, l'ensemble INTERVALLES avec la dernière copie du fichier de sauvegarde. Le coordinateur attribue alors, à tous ces intervalles, la date de dernière mise à jour la moins récente, i.e. 1er janvier 1970 minuit, et initialise l'ensemble AJOUR à l'aide de cette date. La dernière situation où un intervalle est libre se présente au début d'une résolution. En effet, l'intervalle original est toujours libre au départ. L'intervalle original est celui qui contient l'ensemble des numéros de tous les sous-problèmes de l'arbre de l'algorithme B&B. Cet intervalle représente la portée du problème original à résoudre. Au début, l'ensemble INTERVALLES ne contient que l'intervalle original. La référence de cet intervalle est également rajoutée aux ensembles IDENTITE et AJOUR. Au départ, la date de mise à jour de l'intervalle original est considérée comme étant le 1 janvier 1970 minuit. Le coordinateur le traite donc comme un intervalle en panne. Dans ces trois situations, la consultation de l'ensemble AJOUR permet toujours au coordinateur de déterminer les intervalles libres.

Le coordinateur tente donc d'abord de satisfaire une demande de travail par un intervalle libre. En absence d'intervalle libre, le coordinateur sélectionne le plus grand intervalle de INTERVALLES. Une fois sélectionné, l'intervalle est découpé et sa première partie est attribuée au processeur demandeur. Afin de retrouver efficacement un tel intervalle, le coordinateur gère une autre liste appelée TAILLE. Cette liste est également une liste de références vers les intervalles de INTERVALLES. TAILLE ordonne les intervalles de INTERVALLES selon leur taille. La référence du plus grand intervalle se trouve toujours à la tête de la liste, et la référence du plus petit intervalle se trouvant à la queue de la liste. Il suffit donc au coordinateur de prendre l'intervalle dont la référence est au début de TAILLE et de le partitionner.

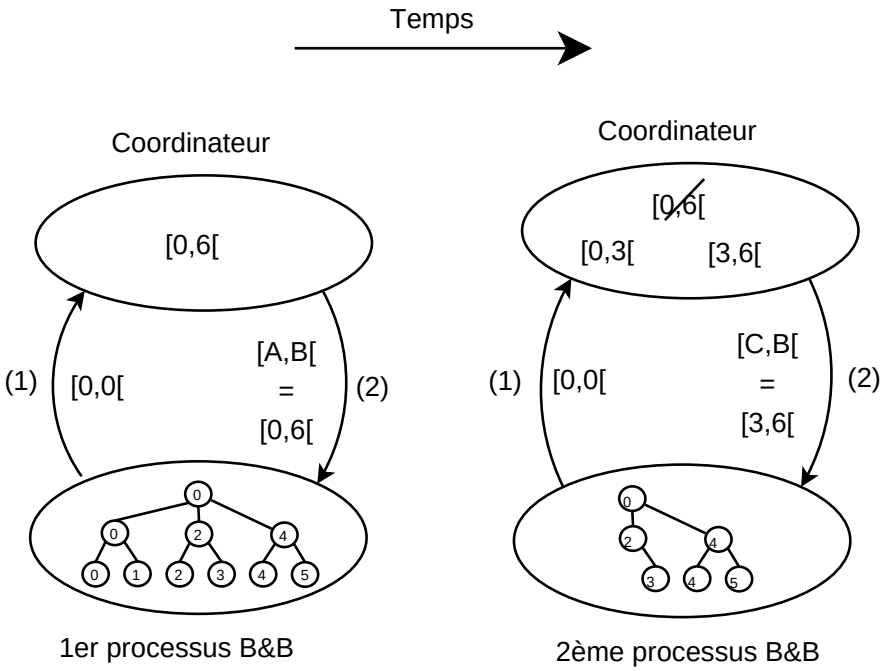

FIG. 2.10 – Stratégie d'équilibrage de charge de B&B@Grid.

Soit [A,B[l'intervalle sélectionné pour être partitionné. Le coordinateur découpe un intervalle [A,B[en deux intervalles [A,C[et [C,B[. Le processus B&B détenteur, celui à qui appartient [A,B[, garde [A,C[puisque il l'explore déjà en partant de A. Le processus B&B demandeur, celui qui sollicite un nouvel intervalle, obtient [C,B[. Au bout d'un certain temps, le processus B&B détenteur est également informé de son nouvel intervalle [A,C[. En effet, comme indiqué dans la section précédente, les processus B&B contactent régulièrement le coordinateur pour actualiser leurs intervalles ainsi que leurs copies dans INTERVALLES. Les deux intervalles [A,C[et [C,B[n'ont pas forcément la même longueur. En effet, les processus B&B demandeur et détenteur sont déployés dans un environnement où les hôtes sont souvent hétérogènes et non dédiés. Par consé-

quent, les longueurs des deux intervalles doivent être proportionnelles à la participation des deux processus au calcul. Comme l'indique la formule (2.16), le choix du point de partitionnement C dépend de la puissance et de la disponibilité des hôtes qui hébergent les deux processus détenteur et demandeur.

$$\frac{C-A}{B-C} = \frac{Puissance(pH) * Disponibilite(pH)}{Puissance(pR) * Disponibilite(pR)} \tag{2.16}$$

Où :

- pH et pR : sont les processeurs qui hébergent les deux processus B&B respectivement détenteur et demandeur.
- $[A, B[$: l'intervalle à diviser.
- C : le point de division de l'intervalle [A,B[.
- $Puissance(p)$: la puissance d'un processeur p, pouvant s'exprimer en Million d'Instructions Par Seconde (MIPS).
- $Disponibilite(p)$: la disponibilité d'un processeur p, i. e. le pourcentage de cycles CPU accordés par le système d'exploitation au processus B&B durant une période donnée. Dans B&B@Grid, cette période est égale à la période de mise à jour des intervalles définie par la stratégie de tolérance au pannes.

Après un découpage, les listes INTERVALLES, IDENTITE, AJOUR et TAILLE doivent être mises à jour.

- INTERVALLES est mis à jour en remplacent [A,B[par [A,C[, et en y insérant l'intervalle [C,B[,
- TAILLE est actualisée en supprimant la référence de l'intervalle [A,B[de la tête la liste, et en insérant les références de [A,C[et [C,B[à leurs bonnes positions. Ces deux insertions se font en $2 * log(N)$, où N est le nombre d'intervalles dans INTERVALLES.
- IDENTITE est mis à jour en insérant la référence de [C,B[à la tête de la liste IDENTITE. En effet, l'identifiant de [C,B[est le plus grand après ce découpage, tandis que [A,C[garde l'identifiant de [A,B[.
- Et AJOUR est actualisée en supprimant la référence de [A,B[et en insérant les deux références de [A,C[et [C,B[à la tête de la liste.

Dans la phase terminale de la résolution d'une instance d'un problème, le plus grand intervalle de INTERVALLES est tellement petit qu'il est préférable de ne pas le partitionner. En effet, la granularité d'une tâche sur une grille de calcul doit dépasser un certain seuil puisque la latence de ce type d'environnement n'est pas négligeable. Par conséquent, l'approche B&B@Grid est paramétrable avec un seuil ϵ. Un intervalle est dupliqué au lieu d'être découpé lorsque sa taille est inférieure à ϵ. Le coordinateur ne garde qu'une seule copie d'un intervalle dupliqué, même s'il est accordé à plusieurs processus.

La figure FIG. 2.10 illustre le fonctionnement de la stratégie d'équilibrage du B&B@Grid sur un exemple. Dans cet exemple, deux processus B&B interviennent dans le calcul. Le but est d'explorer l'intervalle [0,6[qui est la portée du problème original. Au départ, l'ensemble INTERVALLES contient l'intervalle libre [0,6[. Il l'attribue donc complètement au premier processus B&B. Au bout d'un certain temps, le deuxième processus B&B rejoint le calcul et demande à son tour du travail. Comme INTERVALLES ne contient plus d'intervalles libres, un intervalle non libre est sélectionné pour être découpé. Le plus grand et seul intervalle de la liste est [0,6[. Dans cet exemple, la participation des deux processus au calcul est supposée similaire. Par conséquent, le point de découpage de [0,6[est égal 3. Le processeur B&B détenteur garde l'intervalle [0,3[, tandis que le processeur B&B demandeur obtient l'intervalle [3,6[. Comme indiqué dans la section précédente, le premier processeur limite son exploration à l'intervalle [0,3[après une première opération de sauvegarde.

Pendant une résolution, un processus B&B passe son temps à déplier les intervalles reçus du coordinateur, résoudre des sous-problèmes, plier les sous-problèmes non résolus, et communiquer au coordinateur les intervalles obtenus après pliage. Le tableau TAB.2.4 détaille la répartition du temps entre ces quatre différentes fonctions. Les dix expérimentations de ce tableau sont celles décrites dans la section 2.1.7. La première colonne donne le nom de l'instance expérimentée et la deuxième colonne le nombre moyen de processeurs utilisés pendant une expérimentation. Les troisième, quatrième, cinquième et sixième colonnes indiquent respectivement le temps consacré au dépliage, à la résolution de sous-problèmes, au pliage, et à la communication. Comme le temps de pliage est le plus faible et pour plus de lisibilité, les temps de chaque expérimentation sont normalisés par rapport au temps du pliage. Par exemple, pendant l'expérimentation faite avec l'instance *Ta051*, le temps du dépliage est 77 fois plus grand que celui du pliage. La dernière colonne est la plus importante du tableau, elle donne le taux d'utilisation des processeurs par les processus B&B pour faire de la résolution de sous-problèmes. Ce taux est calculé en faisant le rapport entre le temps de résolution, d'un côté, et la somme des temps de pliage, de dépliage, de résolution et de communication, de l'autre. Comme l'indique cette colonne, les processus B&B passent plus de 99% de leur temps à faire de la résolution de sous-problèmes. Ceci montre que la répartition de la charge entre les processus B&B assure un bon recouvrement des communications par le traitement utile.

2.5 Détection de la terminaison

La gestion de la fin des traitements est une autre problématique qui se pose dans tout environnement parallèle, notamment les grilles de calcul. Dans notre approche, la résolution s'arrête lorsque l'ensemble INTERVALLES devient vide. Au départ, INTERVALLES contient un seul intervalle qui correspond à la totalité de l'arbre. Le début de cet intervalle est égal à zéro, le plus petit numéro de l'arbre, et sa fin est égale au plus

Inst.	Nbr. Moy. Proc.	Unfold	Résol.	Fold	Com.	Taux Résol.(%)
Ta051	0692,0	77	92 249	1	20	99,91
Ta052	1276.3	31	93 207	1	20	99,94
Ta053	1055.9	124	78 819	1	21	99,99
Ta054	1236.7	47	98 281	1	18	99,93
Ta055	1312.3	31	103 612	1	15	99,95
Ta056	1213.0	67	96 592	1	18	99,91
Ta057	1442.6	34	101 406	1	15	99,95
Ta058	1432.2	43	99 692	1	15	99,94
Ta059	1448.2	32	102 144	1	14	99,95
Ta060	1398.7	58	95 209	1	15	99,92
Moy.	**1111**	**54**	**96 121**	**1**	**17**	**99,94**

TAB. 2.4 – Taux d'utilisation du processeur par les processus B&B.

grand numéro de l'arbre. Autrement dit, INTERVALLES est initialisé par la portée du problème racine de l'arbre.

La cardinalité de INTERVALLES est le nombre d'intervalles qu'il contient, et la taille de INTERVALLES est la somme des longueurs de ses intervalles. Au cours d'une résolution, la cardinalité de INTERVALLES est plus ou moins égale au nombre de processus B&B impliqués dans le calcul, tandis que sa taille diminue constamment. Cette taille correspond au nombre de solutions non encore explorées de l'espace de recherche. Dans B&B@Grid, le coordinateur supprime automatiquement tout intervalle vide de l'ensemble INTERVALLES. La résolution s'arrête lorsque INTERVALLES devient vide, et donc lorsque sa taille devient nulle. Lorsque INTERVALLES est vide, un processus B&B qui contacte le coordinateur pour actualiser ou demander un intervalle reçoit l'ordre par le coordinateur de s'arrêter. Par conséquent, la détection de la terminaison est implicite et ne nécessite aucune communication explicite additionnelle.

La gestion de la fin des traitements est plus difficile dans un paradigme complètement décentralisé tel que le paradigme pair-à-pair. Comme indiqué auparavant, B&B@Grid peut être utilisée avec l'approche présentée dans [IF00]. Dans ce cas, le coordinateur garde, dans l'ensemble INTERVALLES, les intervalles déjà traités. Au départ, INTERVALLES est vide. Au cours d'une résolution, la cardinalité de INTERVALLES est constamment plus ou moins égale au nombre de processus B&B impliqués dans le calcul. Cependant, la taille de INTERVALLES augmente constamment. Cette taille correspond au nombre de solutions explorées de l'espace de recherche. Le calcul s'arrête une fois que INTERVALLES contient l'intervalle global.

2.6 Partage de la solution globale

En plus d'une liste d'intervalles, le coordinateur garde également une liste globale des solutions trouvées. Ces solutions sont gardées dans une liste appelée SOLUTIONS. Lors de la résolution d'un problème mono-critère, l'ensemble SOLUTIONS ne contient qu'une seule solution. Pendant la résolution d'un problème multi-critère avec une approche de type Pareto, cet ensemble contient l'ensemble des solutions Pareto optimales trouvées.

Comme le coordinateur, un processus B&B gère également une liste locale des solutions trouvées. Cette liste doit être régulièrement mise à jour avec les meilleures solutions trouvées par les autres processus B&B. Ceci permet à ce processus B&B d'éliminer le plus grand nombre de sous-problèmes avant de les traiter.

L'ensemble SOLUTIONS est utilisé notamment pour permettre aux processus B&B d'échanger leurs solutions optimales. L'objectif de la gestion des solutions est de répercuter toute amélioration d'un des fronts Pareto locaux sur tous les autres fronts. Ceci se fait à l'aide de trois règles :

- Un processus B&B initialise son ensemble local de solutions par SOLUTIONS. L'initialisation est faite au moment où ce processus rejoint le calcul.
- Un processus B&B informe le coordinateur de toute solution qui améliore son ensemble local. Il le fait immédiatement après avoir trouvé cette solution.
- Un processus B&B lit régulièrement SOLUTIONS pour actualiser son ensemble local de solutions.

Afin de minimiser le coût des communications, les messages échangés pour la gestion des solutions sont insérés dans les messages échangés pour la gestion de la tolérance aux pannes et d'équilibrage de charge. Lorsqu'un processus B&B récupère son premier intervalle, il récupère également le contenu de SOLUTIONS. Quand le processus B&B envoie une nouvelle solution au coordinateur, il insère également dans ce message l'intervalle en cours de traitement. Ceci lui permet d'opérer une sauvegarde de son état actuel. Lorsqu'un processus B&B reçoit un nouvel intervalle, il reçoit également le contenu de SOLUTIONS.

Conclusion

Dans ce chapitre, nous avons présenté une nouvelle approche, appelée B&B@Grid, pour la gridification des algorithmes B&B mono et multi-critères. L'approche B&B@Grid est basée sur le modèle d'exploration parallèle arborescente, modèle présenté dans le premier chapitre, et sur une stratégie de parcours de type *profondeur d'abord*. Le principe de

B&B@Grid consiste à assigner un numéro à chaque sous-problème de l'arbre, et à définir une relation d'équivalence entre le concept d'intervalle de numéros de sous-problème, d'une part, et celui d'ensemble de sous-problèmes, d'autre part. Afin de passer d'un concept à l'autre, B&B@Grid étend les algorithmes B&B par deux opérateurs de pliage et dépliage. L'opérateur de pliage déduit un intervalle de numéros à partir d'une liste de sous-problèmes, et l'opérateur de dépliage retrouve une liste de sous-problèmes à partir d'un intervalle. L'approche B&B@Grid peut être utilisée pour la gridification des méthodes exactes de type B&B avec différents paradigmes. L'approche se contente de la définition de deux types de processus : un processus coordinateur et un processus B&B. Dans une paradigme *fermier-travailleur*, par exemple, le fermier héberge un coordinateur et un travailleur héberge un processus B&B. Dans un paradigme pair-à-pair, chaque pair héberge un coordinateur et un processus B&B.

Dans ce chapitre, nous avons également proposé des stratégies de tolérance aux pannes, d'équilibrage de charge, de détection de terminaison, et de partage des solutions trouvées. Toutefois, l'apport le plus significatif de B&B@Grid concerne la stratégie de répartition de la charge entre les processus B&B. Contrairement aux autres approches, cette stratégie est la seule, à notre connaissance, à prendre en compte la nature hétérogène et volatile des grilles de calcul, ainsi que leur échelle. En effet, la granularité d'une unité de travail, autrement dit la taille d'un intervalle, s'adapte dynamiquement aux puissances des processeurs, à leurs disponibilités et à leur nombre.

Dix expérimentations ont été effectuées pour évaluer l'intérêt de B&B@Grid. Chaque expérimentation est faite selon le paradigme *fermier-travailleur*, et a duré une heure. Les dix expérimentations ont exploité en moyenne 1111 processeurs. Les résultats montrent que B&B@Grid a un taux de compression des sous-problèmes communiqués égal en moyenne à $0,1289\%$. Autrement dit, dans B&B@Grid, la taille des sous-problèmes communiqués a été réduite de 766 fois en moyenne. Ces expérimentations ont également montré que le codage de B&B@Grid est en moyenne 92 fois plus petit que celui de l'approche à jetons de PICO [EPH00], et 465 fois plus petit que celui de l'approche à variables publiée dans [IF00]. Les dix expérimentations démontrent donc l'intérêt d'utiliser B&B@Grid pour réduire les délais de communication dans les grilles.

L'expérimentation de B&B@Grid avec le paradigme *fermier-travailleur* permet d'évaluer la capacité de cette approche à passer à l'échelle dans des conditions d'utilisation extrêmes. Dans un tel paradigme, le goulot d'étranglement survient lorsque le fermier utilise son processeur à 100%. Malgré le nombre considérable de travailleurs, i. e. 1111 en moyenne, les expérimentations ont montré que le fermier exploite son processeur à $1,29\%$. Ce taux ainsi que le taux de réduction de l'information communiquée prouvent que B&B@Grid peut être déployée facilement sur une grille composée de plusieurs milliers de processeurs. De plus, les processus B&B passent en moyenne $99,94\%$ de leur temps à résoudre des sous-problèmes. Ce taux montre que la stratégie d'équilibrage de charge de B&B@Grid assure un bon recouvrement des communications par le traitement utile.

Les expérimentations permettent également de comparer B&B@Grid avec les autres approches en terme de la taille globale des sous-problèmes stockés par le coordinateur. Ces sous-problèmes permettent à une stratégie de tolérance aux pannes de relancer une résolution en cas de panne. Les résultats montrent principalement que B&B@Grid améliore les taux de compression des autres approches, et réduit la taille des sous-problèmes du coordinateur, dans les mêmes proportions qu'elle le fait pour les sous-problèmes communiqués. Ceci prouve l'efficacité de la stratégie de tolérance aux pannes de B&B@Grid.

Chapitre 3

B&B@Grid : une plate-forme Branch-and-Bound sur grilles de calcul

Introduction

A l'inverse d'une librairie, une plate-forme inverse le contrôle de l'interaction entre son code et celui de l'utilisateur. En effet, c'est le code de la plate-forme qui appelle celui de l'utilisateur. Ce chapitre décrit la plate-forme, appelée également B&B@Grid, qui implémente l'approche que nous avons proposée dans le chapitre précédent. L'objectif de la plate-forme B&B@Grid est de faciliter l'utilisation de l'approche proposée dans cette thèse. Par conséquent, B&B@Grid vise principalement à prendre en compte les caractéristiques des grilles de calcul, à permettre le déploiement de méthodes exactes de type B&B avec tout paradigme parallèle, et à résoudre à la fois les problèmes d'optimisation combinatoire mono et multi-critères.

Ce chapitre est organisé en cinq sections. La **section 3.1** identifie les trois approches principales dédiées à la réutilisation du code de méthodes exactes parallèles, et justifie le choix de présenter l'approche B&B@Grid à l'aide d'une plate-forme. La **section 3.2** survole très brièvement les plates-formes les plus connues dédiées à la parallélisation des méthodes exactes. Cette section identifie les points communs essentiels de ces plates-formes. Ceci permet de positionner et de comparer B&B@Grid par rapport aux autres plates-formes. La **section 3.3** présente l'architecture et les classes de B&B@Grid. Elle scinde notamment la quarantaine de classes de B&B@Grid en six catégories. La **section 3.4** s'intéresse à la façon d'observer une résolution dans B&B@Grid. Offrir un mécanisme d'observation de résolutions est indispensable pour une plate-forme dédiée à des calculs de grande échelle. La **section 3.5** décrit les expérimentations faites pour valider B&B@Grid sur des problèmes mono-critères et bi-critères. Cette section présente notamment une expérimentation qui a permis de résoudre, après plusieurs semaines de calcul, une instance dont la solution optimale est restée inconnue durant plusieurs an-

nées.

3.1 Réutilisation logicielle et intérêt des plates-formes

La réutilisation de composants logiciels, en particulier de ceux implémentant des méthodes exactes parallèles, se définit par leur capacité à être exploités au sein de plusieurs applications différentes sans effort significatif de la part du développeur [FVW99]. Cette définition a amené [Mel05] à identifier trois approches principales dédiées à la réutilisation de méthodes exactes parallèles : *"Aucune réutilisation"* ou *"From scratch"*, *"Seule réutilisation du code"* et enfin *"Réutilisation du code et de la conception"*.

La motivation de l'approche *"Aucune réutilisation"* est l'apparente simplicité des algorithmes mis en œuvre incitant le développeur à les implémenter lui-même. Néanmoins, outre la difficulté de maintenance du code qu'elle implique, une telle approche requiert du temps et de l'énergie en étant sujette à erreurs.

L'approche *"Seule réutilisation du code"* consiste à réutiliser un code dit de "tierce-partie", souvent disponible sur le Web sous la forme de *programmes indépendants* et libres, ou de *librairies*. La réutilisation de programmes indépendants nécessite un examen détaillé du code et la réécriture de certaines sections spécifiques au problème traité. Cette tâche s'avère longue, fastidieuse et sujette à erreurs. C'est pourquoi, les librairies constituent une solution plus efficace et plus fiable pour la réutilisation de code existant [Wal99]. De plus, étant souvent testées et documentées, leur maintenance est ainsi grandement facilitée. Par ailleurs, l'avènement de nouveaux langages orientés objet tels que Java a conduit au développement de librairies, voire la réécriture de certaines librairies existantes, facilitant et encourageant ainsi leur réutilisation. Cependant, les librairies permettent seulement la réutilisation de code, mais ne sont pas dédiées à faciliter la réutilisation de conception.

L'objectif de l'approche *"Réutilisation du code et de la conception"* est de pallier aux limites de l'approche précédente en permettant à la fois la réutilisation du code et celle de la conception. En d'autres termes, elle s'attache à minimiser aux utilisateurs la quantité de code développé (et donc l'effort de développement) chaque fois qu'un nouveau problème d'optimisation est considéré. Dans le contexte de l'optimisation combinatoire, cette approche est fondée sur la séparation de la *partie invariante* des méthodes de résolution de la *partie spécifique* aux problèmes à traiter. La partie invariante, indépendante des problèmes, constitue un ensemble de composants pouvant être des modèles de conception (ou *design patterns* [GHJV94]). Ces composants sont implémentés et encapsulés au sein de plates-formes [JF88].

Ainsi, une plate-forme peut être définie comme un ensemble de classes intégrant des modèles génériques de solutions logicielles dédiées à une famille de problèmes apparen-

tés [FVW99]. Contrairement aux librairies, les plates-formes sont caractérisées par un mécanisme inverse de contrôle de l'interaction vis à vis du code de l'utilisateur. Dans une plate-forme, le code encapsulé fait appel à celui développé par l'utilisateur selon le principe de Hollywood : *Ne nous appelez pas, nous vous appelons* (*Do not call us, we call you*). Ainsi, les plates-formes garantissent le contrôle total de la partie invariante des algorithmes. L'utilisateur fournit seulement la partie spécifique à son problème. Par conséquent, l'approche plate-forme lui permet de passer plus de temps sur la modélisation de son problème que sur l'implémentation de la méthode de résolution associée.

Une analyse du domaine d'application doit permettre une séparation entre la partie invariante (ou constante) et la partie variable (ou adaptable). La partie constante est implémentée dans la plate-forme notamment sous forme d'une collection de classes génériques concrètes. La partie variable est spécifiée dans la plate-forme, à travers notamment des classes abstraites, et son implémentation est laissée à la charge de l'utilisateur. Dans la plate-forme B&B@Grid, l'approche B&B@Grid est considérée comme la partie invariante et les opérateurs d'un algorithme B&B sont la partie variable.

L'objectif principal d'une plate-forme est d'être exploitée par la plus large communauté d'utilisateurs possible. Celle-ci doit donc offrir aux programmeurs une grande variété de méthodes exactes mono et multi-critères, différents types de modèles parallèles distribués en particulier gridifiés. En outre, le succès d'une plate-forme dépend aussi des différents supports qu'elle offre : une interface graphique, un outil d'instrumentation et de suivi de l'exécution, des métriques pour la mesure de performance, etc. Notre objectif est de proposer une plate-forme à large utilité permettant de résoudre des problèmes de type mono-critère et multi-critère, avec tous les algorithmes de type B&B, et selon différents paradigmes parallèles.

La portabilité constitue un critère déterminant pour une large exploitation de la plate-forme. Elle doit être déployable sur des plates-formes matérielles variées (réseaux et grappes de PC, machines massivement parallèles, machines parallèles à mémoire partagée, etc.) utilisant différents systèmes d'exploitation (différentes distributions de Linux, Windows, etc.). Par conséquent, il est important de développer la plate-forme en utilisant un langage portable, des librairies standards pour la concurrence et la synchronisation (*e.g.* Pthreads) et pour la communication (*e.g.* MPI, PVM, etc.). Pour cela, nous avons choisi d'écrire la plate-forme B&B@Grid avec la libraire standard STL de C++. B&B@Grid utilise des sockets pour établir les communications entre les processus.

3.2 Plates-formes parallèles pour le B&B

Il existe plusieurs plates-formes dédiées à la parallélisation des méthodes exactes. Le tableau TAB. 3.1 présente les plus connues et les plus représentatives d'entre elles. Toutes ces plates-formes sont dédiées à la réutilisation de code et de conception.

	Méth. exact.	Paradigmes (mode de comm.)
Symphony	Branch & Cut	Fermier-travailleur (push)
BCP	B&B	Fermier-travailleur (push)
GRIBB	B&B	Fermier-travailleur (pull)
PPBB	B&B	Fermier-travailleur (push)
PUBB	Div. pour Régn. et B&B	Fermier-travailleur (push)
PICO	B&B	Fermier-travailleur (push)
Bob	B&B	Fermier-travailleur (push)
ZRAM	Div. pour Régn. et B&B	Fermier-travailleur (push)
MallBa	Div. pour Régn. et B&B	Fermier-travailleur (push)
BOB++	Div. pour Régn. et B&B	Fermier-travailleur hiérar. (push)
ALPS	Div. pour Régn., B&B, Branch &Cut et Branch & Price	Fermier-travailleur hiérar. (push)
B&B@Grid	Toutes méth. exact. de type B&B	Tous paradigmes (pull)

TAB. 3.1 – B&B@Grid et principales plates-formes dédiées à la parallélisation des méthodes exactes.

Ces plates-formes diffèrent essentiellement par les méthodes exactes prises en charge, le paradigme parallèle utilisé et le mode de communication. Toutes ces plates-formes s'intéressent aux méthodes de type B&B. Les algorithmes de type B&B se déclinent essentiellement sous trois variantes : B&B simple, Branch-and-Cut (B&C) et Branch-and-Price (B&P). Comme indiqué au premier chapitre, il est possible de considérer un simple algorithme *Diviser pour Régner* comme une base pour l'algorithme B&B. Il suffit de rajouter, à l'algorithme Diviser pour Régner (DR), un opérateur d'élimination pour obtenir un algorithme B&B. Toute plate-forme offrant un B&B simple peut donc servir à l'implémentation des algorithmes Branch-and-Cut et Branch-and-Price. Il suffit essentiellement d'écrire l'opérateur de borne à l'aide d'un solveur linéaire. Cependant, il est indispensable de mémoriser les plans de coupe pour rendre une plate-forme efficace.

Dans ZRAM [BMF+99] et MallBa [AAB+02], les algorithmes Diviser pour Régner et B&B sont distincts et implémentés séparément. BOB++[7], ALPS[8] [XRLS05] et PUBB [SHH95] proposent une solution plus intelligente. Dans ces trois plates-formes, l'algorithme *Diviser pour Régner* est une classe de base pour l'algorithme B&B. En plus

[7] http ://www.prism.uvsq.fr/blec/Research/BOBO/
[8] http ://www.coinor.org/

de ZRAM, MallBa, *Bob++*, ALPS et PUBB, quatre autres plates-formes proposent une interface pour l'implémentation d'un B&B simple. Ce sont Bob [BCD+95], PICO [EPH00], BCP[9]. Par contre, la plate-forme Symphony[10] [RG05] est dédiée seulement à l'implémentation d'un *Branch-and-Cut*. *ALPS* propose une interface pour l'implémentation à la fois du *Branch-and-Cut* et du *Branch-and-Price*.

Une bonne partie des plates-formes citées est basée sur le paradigme fermier-travailleur. Pour passer à l'échelle et supporter davantage de processus travailleurs, certaines plates-formes hiérarchisent les fermiers. Des fermiers locaux sont ainsi insérés entre le fermier global et les processus travailleurs. Chaque fermier local se charge alors de la gestion d'un certain nombre de processus travailleurs. BOB++, ALPS et PICO, par exemple, proposent une architecture avec fermier global, fermiers locaux et travailleurs.

La version actuelle de B&B@Grid fournit l'implémentation d'un algorithme B&B simple. Il suffit d'ignorer l'opérateur d'élimination pour obtenir un algorithme de *Diviser pour Régner* et d'utiliser un solveur linéaire pour avoir un *Branch and Cut* ou un *Branch and Price*. Contrairement à la plupart des plates-formes, B&B@Grid fournit le paradigme fermier-travailleur avec le mode de communication *pull*. La plate-forme peut également être étendue pour être utilisée avec les autres paradigmes. Le mode *pull* convient davantage aux environnements comme les grilles où des pare-feux sont présents. Le modèle fermier-travailleur avec mode *pull* suppose donc que seul le fermier n'est pas protégé par un pare-feux, contrairement au mode *push* où toutes les processus travailleurs doivent être joignables par le fermier.

Contrairement aux autres plates-formes, B&B@Grid est pensée dès le départ pour une exploitation sur grilles. Par conséquent, elle a été conçue avec l'objectif de gérer les problématiques spécifiques à ce type d'environnement. Toutes ces plates-formes s'intéressent seulement à l'optimisation avec un seul critère. Par conséquent, elles ne sont utilisables que pour résoudre les problèmes mono-critères. B&B@Grid peut être également utilisée pour l'optimisation multi-critère.

3.3 La plate-forme B&B@Grid

B&B@Grid est une plate-forme conçue et réalisée selon les principes de la programmation orientée objet. Il s'agit d'une hiérarchie d'une quarantaine de classes représentant 6000 lignes de code. Comme l'indique la figure FIG. 3.1, les classes de B&B@Grid sont classées en six catégories. Il s'agit de classes de représentation, de travail, d'opérateurs spécifiques, de solveurs et d'utilitaires. Les sections suivantes présentent brièvement les classes de chaque catégorie.

[9] http ://www.coinor.org/
[10] http ://www.branchandcut.org/

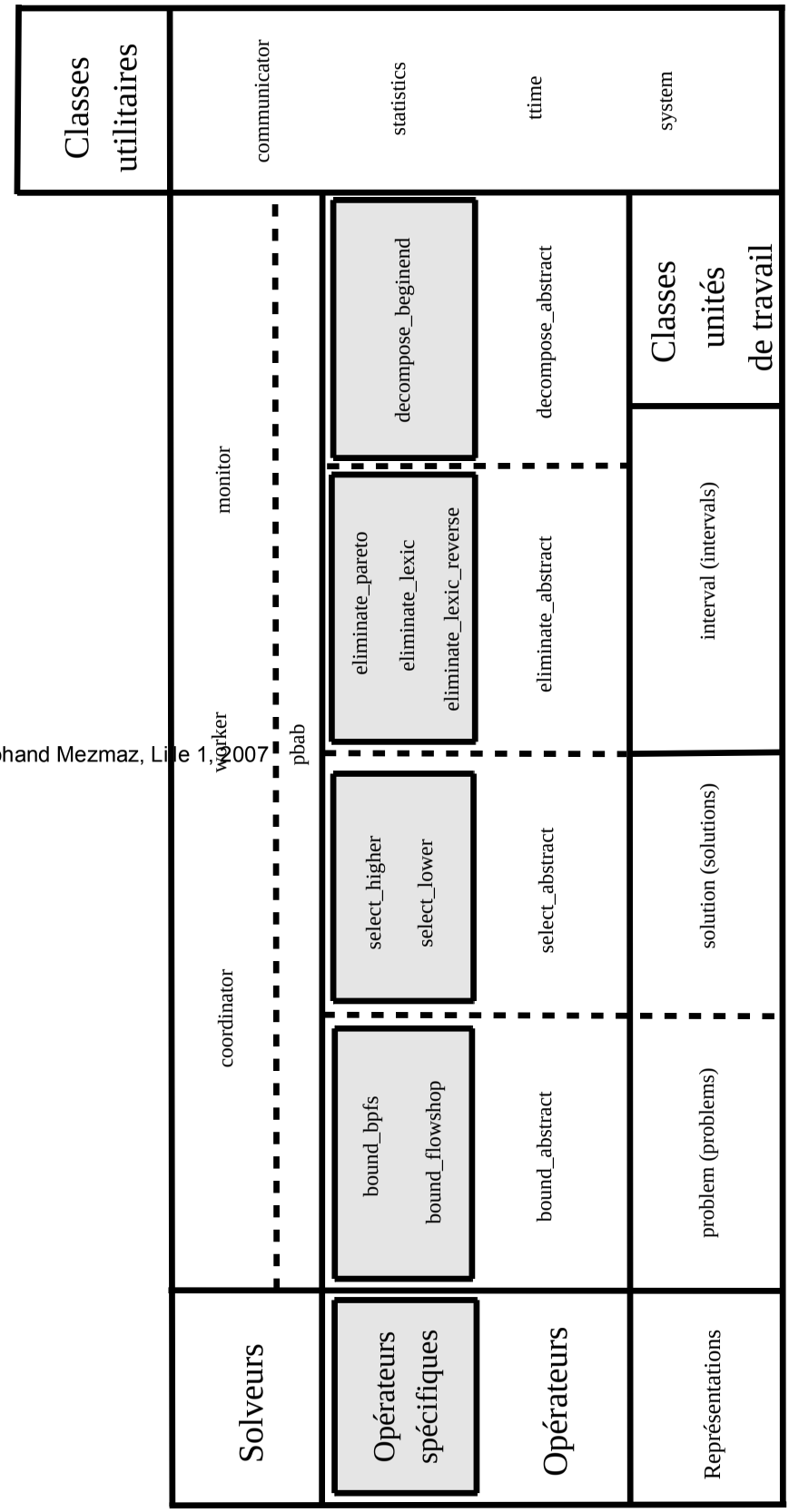

FIG. 3.1 – L'architecture de B&B@Grid.

3.3.1 Classes de représentation

Cette catégorie comprend six classes fournies et liées à l'arbre exploré par un B&B. Comme le montre la figure FIG. 3.2, il s'agit des classes *node*, *tree*, *problem*, *problems*, *solution* et *solutions*. La classe *problem* est le nœud interne d'un arbre du *B&B* et la classe *solution* est un nœud feuille de l'arbre. Par conséquent, la classe *node* est la classe de base des classes *problem* et *solution*. Un objet de la classe *problems* (resp. *solutions*), contient un ensemble d'objets de type *problem* (resp. *solution*). L'opérateur de décomposition d'un algorithme B&B, par exemple, reçoit toujours comme argument un objet *problem* et renvoie un objet de type *problems*. Un objet *problems* est donc constitué d'un ensemble de nœuds de l'arbre ayant le même nœud père. Un objet de type *solutions* peut contenir l'ensemble des solutions Pareto d'un problème multi-critère. De la même façon que les classes *problems* et *solutions* sont respectivement des agrégats des classes *problem* et *solution*, la classe *tree* est aussi un agrégat de la classe *node*. A tout instant, un objet de la classe *tree* contient l'ensemble des objets *node* générés et non encore traités.

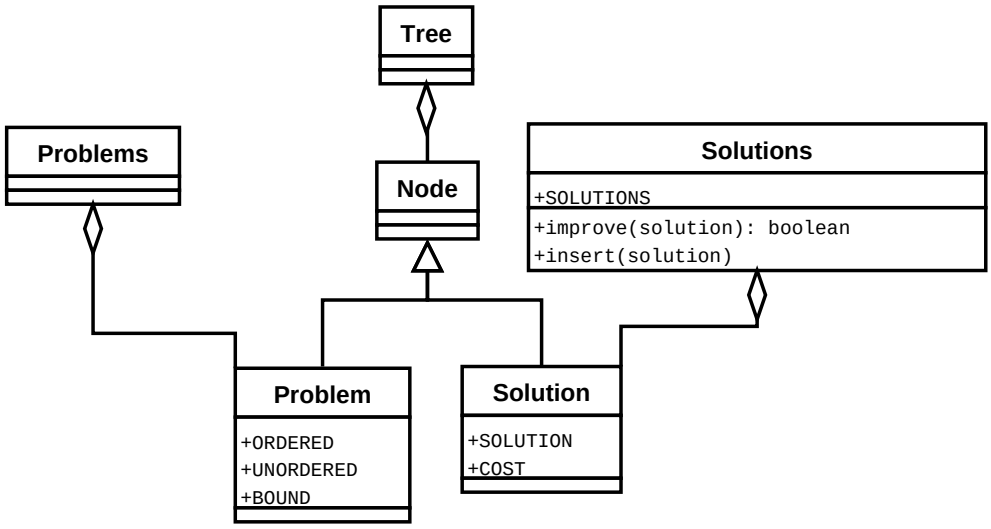

FIG. 3.2 – Les classes de représentation.

La classe *solution* contient deux attributs : la solution elle-même et son coût. Un problème est défini par ses éléments ordonnés, ses éléments non encore ordonnés et le coût de sa borne. La classe *solutions* est définie à l'aide d'un attribut et deux méthodes : $SOLUTIONS$, *improve* et *insert*. L'attribut $SOLUTIONS$ contient l'ensemble des solutions optimales, celles-ci forment un front Parto. La méthode *improve* vérifie si une solution améliore les solutions de $SOLUTIONS$. La méthode *insert* rajoute une solution dans $SOLUTIONS$ et en supprime toutes les solutions dominées.

3.3.2 Classes opérateurs

Quatre classes abstraites principales sont fournies pour implémenter un algorithme avec B&B@Grid. Ce sont les classes *bound_abstract*, *select_abstract*, *eliminate_abstract* et *decompose_abstract*. Elles constituent la partie variable de la plate-forme, et doivent être fournies par l'utilisateur pour exploiter B&B@Grid. Chacune de ces classes abstraites définit un des quatre opérateurs d'un algorithme $B\&B$ et contient une seule méthode virtuelle. Il est donc nécessaire de définir les quatre méthodes virtuelles, décrites dans le tableau TAB. 3.2.

Classes	Méthodes virtuelles
decompose_abstract	problems* operator()(problem* p) ;
bound_abstract	cost* operator()(problem* p) ;
eliminate_abstract	bool operator()(node* n1, node* n2) ;
select_abstract	bool operator()(problem* p1, problem* p2) ;

TAB. 3.2 – Les méthodes virtuelles des classes requises.

Comme l'indique ce tableau, toutes ces classes sont des classes fonctions et chargent l'opérateur de fonction *operator*(). En *C++*, le concept de classe fonction permet d'utiliser un objet comme s'il était une fonction. La méthode virtuelle de *decompose_abstract* prend comme argument un problème p, le décompose et renvoie comme résultat l'ensemble des sous-problèmes de p. La méthode virtuelle de *bound_abstract* reçoit un problème p, calcule sa borne et renvoie une borne du coût de p. La méthode virtuelle de *eliminate_abstract* renvoie *vrai* si le nœud $n1$ domine le nœud $n2$, et $faux$ dans le cas contraire. La méthode virtuelle de *select_abstract* permet de sélectionner le prochain problème à traiter parmi les problèmes ayant le même problème père. Pour exploiter B&B@Grid, un programmeur doit également définir une autre classe virtuelle. Il s'agit de la classe *instance_abstract*, qui contient un constructeur et un objet de type *stringstream*. Le rôle du constructeur est de mettre, dans l'objet *stringstream*, les données de l'instance à résoudre. Le constructeur de la classe *instance_abstract* accepte un seul argument. Il s'agit d'un entier définissant le numéro de l'instance à résoudre.

3.3.3 Classes opérateurs spécifiques

B&B@Grid est fournie avec des instances de chacune des classes abstraites. Ces instances sont spécifiques à l'optimisation multi-critère et au problème du Flow-Shop. On distingue :

- *eliminate_pareto*, *eliminate_lexic* et *eliminate_lexic_reverse* dérivent de *eliminate_abstract*. La classe *eliminate_pareto* utilise la relation de dominance Pareto pour comparer les solutions du front Pareto optimal retrouvé. Les classes *eliminate_lexic*

et *eliminate_lexic_reverse* définissent, respectivement, la dominance à l'aide de l'ordre lexicographique normal et de l'ordre lexicographique inversé.

- *bound_flowshop* et *bound_bpfs* dérivent de *bound_abstract* et permettent de calculer la borne d'un problème dans le Flow-Shop ($F/permut/C_{max}$) et calculer la borne pour le problème du Flow-Shop bi-critère.

- *decompose_beginend* dérive de *decompose_abstract*. Dans un opérateur de décomposition simple, les éléments sont toujours ordonnés au début d'une permutation. Par contre, *decompose_beginend* ordonne les éléments soit au début ou à la fin de la permutation.

- *select_lower* et *select_higher* dérivent de *select_abstract*. La classe *select_lower* sélectionne le problème dont la borne est la plus petite. Par contre, *select_higher* sélectionne le problème dont la borne est la plus grande.

- *instance_flowshop* et *instance_bpfs* dérivent de *instance_abstract*, et implémentent respectivement un algorithme pour générer les instances publiées dans [Tai93] et celui qui génère les instances publiées dans *http ://www.lifl.fr/OPAC/*.

3.3.4 Classes solveurs

La classe *pbab* est la classe principale de cette catégorie. Cette classe encapsule et fournit l'implémentation d'un algorithme B&B placé dans le contexte distribué. Le tableau TAB. 3.3 résume les principales méthodes de *pbab*. Le constructeur *pbab* permet de définir un certain nombre d'adresses IP et de ports nécessaires à un processus B&B :

- *pname* : le nom, ou l'adresse IP, de la machine qui accueille le processus B&B.
- *pport* : le port sur lequel le processus B&B se met à l'écoute. Le constructeur de *pbab* lance toujours un thread à l'écoute du port *pport*.
- *cname* : le nom, ou l'adresse IP, du processus coordinateur.
- *cport* : le port du processus coordinateur.

Les méthodes de *pbab* servent à définir les objets *ttime*, *statistic*, *instance_abstract*, *bound_abstract*, *eliminate_abstract*, *decompose_abstract*, *select_abstract*, ainsi que le paradigme parallèle choisi. Actuellement, ce paradigme ne peut être que fermier-travailleur. La méthode *explore* lance l'exploration de l'arbre du B&B. Cette méthode n'est pas lancée quand un processus joue le rôle de *coordinateur*.

L'algorithme 1 donne un exemple de création et de lancement d'un processus B&B travailleur. La plate-forme offre également la possibilité de créer et de lancer un pro-

constructor(pname, pport, rname, rport, cname, cport)
define(ttime)
define(statistic)
define(instance_abstract)
define(bound_abstract)
define(eliminate_abstract)
define(select_abstract)
define(decompose_abstact)
define(paradigm)
explore()

TAB. 3.3 – Les principaux membres de la classe *pbab*.

Algorithm 1 L'algorithme B&B dans un contexte distribué.

```
pbab *pbb=new pbab(pname,pport,rname,rport,cname,cport);
ttime*ttm=new ttime();
ttm->period_set(COORIDINATOR_CHECKPOINT, 30*60);
ttm->period_set(WORKERS_CHECKPOINT, 3*60);
pbb->define(ttm);
pbb->define(new statistic("/tmp/pbab/"));
pbb->define(new eliminate_pareto());
pbb->define(new decompose_beginend());
pbb->define(new instance_flowshop(25));
pbb->define(new bound_flowshop());
pbb->define(FARMER_WORKER);
pbb->explore();
```

cessus travailleur, ou tout autre processus, d'une façon beaucoup plus simple. Il suffit d'utiliser les classes *worker*, *coordinator* et *monitor*. Ces trois classes et la classe *pbab* constituent l'ensemble des classes solveurs. Selon le rôle qui lui est assigné, un processus B&B@Grid se contente de créer un seul objet de l'une de ces trois classes et d'appeler sa méthode *run*. En plus du constructeur et de la méthode *run*, chacune des classes contient un certain nombre de méthodes. Pour la plupart ces méthodes sont privées, les seules méthodes publiques sont celles qui correspondent aux requêtes auxquelles l'objet d'une classe doit répondre.

Un objet de la classe *coordinator* répond à deux requêtes. Ces requêtes proviennent toujours des objets de la classe *worker*. Les requêtes portent sur un intervalle ou sur les solutions provisoirement optimales trouvées. Les deux services sont assurés respectivement à l'aide des deux méthodes *exchange_solutions* et *exchange_interval*. Le tableau TAB. 3.4 résume les principaux membres de la classe *coordinator*.

intervals INTERVALS
solutions SOLUTIONS
constructor(monitor_host, solutions_file, intervals_file, port, period, fault)
exchange_interval(interval)
exchange_solutions(solutions)
run()

TAB. 3.4 – Les principaux membres de la classe *coordinator*.

La méthode *exchange_solutions* peut recevoir comme argument un certain nombre de solutions et rend comme résultat les solutions obtenues après fusion des solutions reçues et celles déjà présentes dans *SOLUTIONS*. L'attribut *SOLUTIONS* de la classe contient les meilleures solutions trouvées. Elle est mise à jour à chaque invocation de la méthode *exchange_solutions*. La méthode *exchange_interval* reçoit un intervalle, le fusionne éventuellement avec sa copie dans *INTERVALS* et rend comme résultat un autre intervalle. *INTERVALS* contient l'ensemble des intervalles non encore explorés. L'intervalle, envoyé en réponse à la requête d'un processus B&B demandeur de travail, est nul lorsque *INTERVALS* est vide. L'intervalle reçu peut être également vide lorsque un processus B&B rejoint le calcul pour la première fois, ou un pair termine l'exploration de son intervalle. La méthode *exchange_interval* actualise à la fois l'intervalle passé comme argument ainsi que sa copie dans *INTERVALS*. Cette méthode fait appel notamment aux opérateurs d'intersection, de division et de sélection d'intervalles.

Comme le montre le tableau TAB. 3.5, la classe *worker* contient deux attributs : *PROBLEMS* et *SOLUTIONS*. L'attribut *SOLUTIONS* contient une copie des solutions qui se trouvent au niveau du processus coordinateur.

string COORDINATOR
problems PROBLEMS
solutions SOLUTIONS
constructor(coordinartor_name, monitor_name, period)
run()

TAB. 3.5 – Les principaux membres de la classe *worker*.

Le rôle de la classe *monitor* est de collecter certaines statistiques nécessaires au suivi d'une résolution. Ces statistiques concernent tous les processus de B&B@Grid et l'état d'avancement de l'exploration de l'arbre du B&B.

Un objet de la classe *monitor* fournit quatre services. Ils consistent à récupérer les statistiques relatives aux travailleurs, au(x) coordinateur(s), aux sous-problèmes traités et aux solutions retrouvées. Ces quatre services sont assurés par les méthodes *worker_statistics*, *coordinator_statistics*, *problems_statistic* et *solutions_statistic*.

Le tableau TAB. 3.6 présente les principaux membres de la classe *monitor*.

constructor(port, period)
worker_statistics(EXPLOITATION, POWER)
coordinator_statistics(EXPLOITATION)
problems_statistic(TOTAL, REDUNDANT)
solutions_statistic(TOTAL)

TAB. 3.6 – Les principaux membres de la classe monitor.

3.3.5 Classes unités de travail

Deux classes principales sont définies dans cette catégorie et fournies dans la plate-forme B&B@Grid : *interval* et *INTERVALS*. Un objet de la classe *INTERVALS* contient un ensemble d'objets de type *interval*. La classe *interval*, décrite dans le tableau TAB.3.7, fournit notamment quatre méthodes. Ces méthodes concernent les opérateurs d'intervalles déjà décrits au chapitre précédent. Il s'agit des opérateurs d'intersection, de division, de soustraction, de pliage et de dépliage. En plus de ces méthodes, *interval* utilise principalement six attributs afin de stocker l'identité, le début, la fin, la taille, la date de mise à jour de l'intervalle.

IDENTITY : integer
BEGIN, END, SIZE : BigInteger
UPDATE : time
intersect(interval)
devide() : interval
substract(interval)
fold(problems) : interval
unfold() : problems

TAB. 3.7 – Les membres de la classe *interval*.

La classe *INTERVALS*, décrite dans le tableau TAB. 3.8, est une classe définie seulement par trois attributs et deux méthodes. Les attributs sont utilisés seulement pour accélérer la recherche d'un intervalle quelconque. En effet, ils contiennent la liste des références aux intervalles triée selon leur identité, leur taille et leur date de dernière mise à jour. Par conséquent, la recherche d'un intervalle se fait toujours en $O(log(N))$, où N est le nombre d'intervalles. L'attribut $INTERVALS\_UPDATE$ est utilisée principalement comme une pile pour retrouver l'intervalle le plus anciennement mis à jour. En comparant sa date à la date actuelle, il est possible de déterminer si la machine travailleur qui détient un intervalle est en panne. L'attribut $INTERVALS\_SIZES$ sert à sélectionner le plus grand intervalle. Selon l'ordre de tri, il suffit de prendre l'intervalle qui se trouve au début ou à la fin de $INTERVALS\_SIZES$. Les deux méthodes

select et *update* définissent l'opérateur de sélection et la procédure de mise à jour d'un intervalle.

INTERVALS_IDENTITIES
INTERVALS_SIZES
INTERVALS_UPDATE
select() : interval
update(interval)

TAB. 3.8 – Les membres de la classe *INTERVALS*.

3.3.6 Classes utilitaires

Quatre classes utilitaires sont définies dans B&B@Grid : *statistics*, *ttime*, *system* et *communicator*. La classe *statistics* permet d'observer une résolution et l'ensemble des machines exploitées. Cette classe est indispensable pour un environnement dédié à l'optimisation exacte de problèmes de taille réelle nécessitant plusieurs semaines voire plusieurs mois de calcul. Comme son nom l'indique, la classe *ttime* permet de gérer le temps. Cette classe est fortement sollicitée dans une résolution. Elle indique, par exemple, à un processus coordinateur le moment de sauvegarde du travail non fait. La classe *system* permet d'interagir directement avec la machine. Elle permet, par exemple, de scruter l'activité d'une machine et de mettre un processus travailleur en veille. En effet, la plate-forme peut être lancée selon le modèle de vol de cycles et peut tourner sur des machines non dédiées. Si un utilisateur recommence à exploiter sa machine, la classe *system* met en veille le processus B&B. Dès qu'un utilisateur cesse d'exploiter sa machine, la classe *system* relance le processus B&B. La classe *communicator* est une souche de communication permettant aux processus de B&B@Grid de communiquer entre eux. Dans B&B@Grid, toutes les communications se font par l'intermédiaire d'objets *communicator*. Tout processus doit donc posséder un objet de ce type. La classe *communicator* fonctionne selon le modèle d'appel de méthodes distantes. Elle fournit un ensemble de méthodes nécessaires pour emballer les arguments d'un appel de procédure distante, de déballer ces arguments et d'appeler la bonne méthode.

3.4 Observation d'une résolution et évaluation de performances

L'évaluation des performances des méthodes d'optimisation parallèles exactes sur grilles de calcul est complexe. Cette complexité est liée à la fois à la nature particulière de ces méthodes et aux caractéristiques des grilles de calcul. En effet, les méthodes exactes ont un comportement non déterministe à l'exécution, et les caractéristiques des grilles rendant l'évaluation des performances complexe sont essentiellement l'hétérogé-

néité (notamment des machines, du réseau, et des systèmes d'exploitation) et la disponibilité variable des machines et/ou du réseau. Les mesures traditionnellement utilisées ne peuvent pas convenir ici puisqu'elles ne s'appliquent qu'à un ensemble de ressources homogènes et dédiées. Une nouvelle approche de l'évaluation des performances est proposée dans [GKYL00].

Dans B&B@Grid, l'évaluation de performances et l'observation d'une résolution exacte d'un problème sur la grille se fait à l'aide des objets de la classe *statistics*. Chaque processus B&B crée un objet de cette classe. Pendant la résolution, ces objets se chargent de recueillir des informations relatives au calcul lui-même et aux machines hébergeant les différents processus. Les informations relatives à une machine portent sur sa disponibilité, le taux de l'utilisation de sa CPU et la puissance de son processeur exprimée en MIPS. Les informations relatives à l'exploration de l'arbre du B&B concernent le nombre total de problèmes traités, de problèmes redondants et de solutions restantes. Le nombre de problèmes traités est égal au nombre de nœuds visités de l'arbre du B&B. Un problème est dit redondant s'il est traité par plus d'un processus-travailleur. Le nombre de solutions restantes est le nombre de solutions de l'espace de recherche non encore implicitement ou explicitement visitées. A la fin d'une résolution, ce nombre devient nul. Les principales statistiques utilisées dans B&B@Grid sont les suivantes :

- W est l'ensemble des processeurs de la grille qui hébergent un processus travailleur, et c le processeur qui héberge le processus coordinateur.

- T est un ensemble de dates choisies durant l'expérimentation. L'ensemble T dépend de la période de mise à jour notée $Period$.
  $T = \{t_1, t_2, ..., t_n\}$, avec $\forall\ 1 < i \leq n$, $t_i - t_{i-1} = Period$,
  et t_n est la date de fin de l'expérimentation.

- $Power(p)$ est la puissance d'un processeur p de la grille.

- $Rate(p, t_i)$ est le taux d'utilisation d'un processeur p durant la période $[t_i, t_{i+1}[$.

- $Availability(p, t_i)$ est la disponibilité d'un processeur p durant la période $[t_i, t_{i+1}[$. Elle vaut 1 si le processeur p est disponible durant cette période, et à 0 dans le cas contraire.

- $Rate(p, T)$ est le taux d'utilisation d'un processeur p durant l'expérimentation. $Rate(c, T)$ est donc le taux d'utilisation du coordinateur durant l'expérimentation.

$$Rate(p, T) = \frac{\sum_{t \in T} Availability(p, t) * Rate(p, t)}{\sum_{t \in T} Availability(p, t)}$$

- $Rate(P, T)$ est le taux d'utilisation moyen de l'ensemble des processeurs P durant

l'expérimentation. $Rate(W,T)$ est donc le taux d'utilisation moyen de l'ensemble des processeurs travailleurs durant l'expérimentation.

$$Rate(P,T) = \frac{\sum_{p \in P} Power(p) * Rate(p,T)}{\sum_{p \in P} Power(p)}$$

- $Availability(P,t_i)$ est le nombre de processeurs de P disponibles durant la période $[t_i, t_{i+1}[$.
$$Availability(P,t_i) = \sum_{p \in P} Availability(p,t_i)$$
- $Average(P,T)$ est le nombre moyen de processeurs de P disponibles durant l'expérimentation. $Average(W,T)$ est donc le nombre moyen de processeurs travailleurs disponibles durant l'expérimentation.
$$Average(P,T) = \frac{\sum_{t \in T} Availability(P,t)}{|T| - 1}$$
- $Maximum(P,T)$ est le nombre maximum de processeurs de P disponibles durant l'expérimentation. $Maximum(W,T)$ est donc le nombre maximum de processeurs travailleurs disponibles durant l'expérimentation.
$$Maximum(P,T) = Maximum\{Availability(P,t)/t \in T\}$$
- $Time(p,t_i,O)$ est le temps accordé par le processeur p durant la période $[t_i, t_{i+1}[$ pour le traitement de l'ensemble des opérations de O. L'ensemble O peut contenir des opérations de $Pliage$, $Depliage$, $Communication$, $Exploration$ ou $Toutes$. L'opération $Toutes$ désigne la totalité des traitements.

- $Time(P,T,O)$ est le temps accordé par un ensemble de processeurs P durant toute l'expérimentation pour le traitement des opérations O.
$$Time(P,T,O) = \sum_{p \in P} \sum_{t \in T} Time(p,t_i,O)$$
$Time(W,T,\{Fold, \text{Unfold}\})$, par exemple, est le temps consacré dans toute l'expérimentation à faire du pliage et du dépliage d'intervalles.

- $Time(P,T,*)$ est le temps total d'utilisation des CPUs de l'ensemble des processeurs P durant l'expérimentation. $Time(W,T,*)$ est donc le temps total de l'utilisation des CPUs de l'ensemble des processeurs travailleurs durant l'expérimentation.
$$Time(P,T,*) = \sum_{t \in T} Availability(P,t) * Period$$

Durant une résolution, les statistiques peuvent être consultées à l'aide de plusieurs fichiers. Ces fichiers sont mis à jour par le processus *monitor*. Les données de ces fichiers

sont disposées en deux colonnes et sur plusieurs lignes. Chacune des lignes a la forme $< t_i\ d_i >$. La valeur de t_i donne la période $[t_i, t_{i+1}[$ à laquelle d_i est prélevée. La valeur de t_i est exprimée en nombre de secondes passées par rapport au premier janvier 1970. La colonne d_i diffère d'un fichier à un autre. Selon le fichier, d_i peut signifier :

- nombre de problèmes traités au total,
- nombre de problèmes traités redondants,
- nombre de solutions qui restent à explorer dans l'espace de recherche,
- $Availability(W, t_i)$: nombre de processus travailleurs impliqués dans la résolution,
- $Power(W, t_i)$: puissance totale en $MIPS$,
- $Rate(W, t_i)$: taux moyen d'utilisation de la CPU des machines travailleurs. La moyenne est retrouvée en pondérant le taux d'utilisation de chaque machine avec sa puissance,
- $Rate(c, t_i)$: taux d'utilisation de la machine fermier,
- $Time(W, t_i, \{Fold\})$: temps total consacré à l'opération de pliage ($Fold$),
- $Time(W, t_i, \{Unfold\})$: temps total consacré à l'opération de dépliage ($Unfold$),
- $Time(W, t_i, \{Communication\})$: temps total consacré à la communication,
- $Time(W, t_i, \{Exploration\})$: temps total consacré à l'exploration de l'arbre.

L'enregistrement de ces données est fait régulièrement. Toutes les $Period$, le processus moniteur procède à l'ajout d'une ligne dans chacun des onze fichiers. Ainsi, il est possible d'observer l'évolution des différentes données. Ceci peut se faire de trois manières différentes :

- Consulter directement les fichiers.
- Utiliser Gnuplot pour obtenir des courbes à partir de ces fichiers en en deux ou trois dimensions.
- Utiliser une interface Web. B&B@Grid proposera une interface Web, en cours de développement, pour suivre une résolution même à distance.

3.5 Expérimentations et résultats

Trois expérimentations ont été réalisées pour valider B&B@Grid sur les problèmes mono-critères et multi-critères. Dans les deux premières, nous avons considéré l'instance $Ta056$ qui est une instance mono-critère publiée dans [Tai93]. Il s'agit d'une instance du problème $(F/permut/C_{max})$. La solution optimale de cette instance n'était pas encore connue. Cette instance est définie par 50 tâches et 20 machines. La meilleure solution connue pour $Ta056$ a un coût de 3681. Elle a été trouvée dans [RS04] à l'aide d'une méta-heuristique gloutonne itérative. En plus de la validation de B&B@Grid, le défi de cette première expérimentation est de résoudre de façon exacte cette instance. Par contre, la deuxième expérimentation vise à prouver l'efficacité de B&B@Grid pour la

résolution de problèmes mono-critères. La troisième expérimentation s'intéresse à une instance bi-critère disponible à l'adresse [11]. C'est une instance du problème (F/d_i, $permut/C_{max}, \bar{T}$). Elle étend une instance publiée dans [Tai93] en définissant des dates de fin souhaitées pour chaque tâche. Cette instance bi-critère est définie par 50 tâches et 5 machines. Son front Pareto optimal n'était pas encore connu. En plus de résoudre cette instance, l'objectif de la troisième expérimentation est de montrer l'efficacité de B&B@Grid pour résoudre des problèmes bi-critères.

3.5.1 Résolution de l'instance mono-critère *Ta056*

Le défi de la première expérimentation est de trouver la solution exacte de $Ta056$. Cette instance nécessite une puissance de calcul considérable. L'initialisation d'un algorithme B&B avec une solution approchée permet de réduire, d'une façon significative, la puissance de calcul nécessaire à une résolution. Par conséquent, l'algorithme est initialisé par la meilleure solution connue de $Ta056$.

La grille d'expérimentation, présentée dans le premier chapitre, est constituée de machines de l'Université des Sciences et Technologies de Lille et de nœuds de Grid'5000. Au début de l'expérimentation, seules les trois grappes universitaires et une seule grappe Grid'5000, celle de Lille, sont exploitées. Les autres grappes de Grid'5000 ne sont intégrées à cette expérimentation que trois jours avant la fin de la résolution. L'expérience a permis de trouver la solution optimale au bout d'un mois et trois semaines environ. Le pic de processeurs enregistré est de 1245. La solution optimale de $Ta056$ a un coût égal à 3679. Dans cette solution, les tâches doivent être ordonnancées selon l'ordre suivant : (14, 37, 3, 18, 8, 33, 11, 21, 42, 5, 13, 49, 50, 20, 28, 45, 43, 41, 46, 15, 24, 44, 40, 36, 39, 4, 16, 47, 17, 27, 1, 26, 10, 19, 32, 25, 30, 7, 2, 31, 23, 6, 48, 22, 29, 34, 9, 35, 38, 12).

Le coût de la solution optimale est donc proche du coût de la solution trouvée par la méta-heuristique proposée dans [RS04]. Ce résultat montre donc l'efficacité de la méta-heuristique utilisée. Ainsi, les méthodes exactes sur grilles permettent de produire des solutions servant de références pour mesurer l'efficacité des méthodes approchées à résoudre des problèmes de grande taille.

Le but de la deuxième expérience est de récupérer davantage de statistiques durant la résolution de $Ta056$. Ces statistiques permettent de mieux connaître le comportement de B&B@Grid, et d'évaluer ses performances. Dans cette expérience, la totalité des machines disponibles dans les grappes universitaires et Grid'5000 sont exploitées.

L'algorithme est initialisé par une borne égale à 3.680. Cette borne est choisie pour réduire le temps de résolution de $Ta056$, par rapport à la première expérimentation, tout en permettant au B&B@Grid d'améliorer cette solution initiale. Le tableau TAB. 3.9 résume les statistiques enregistrées durant l'expérimentation. Comme l'indique ce

[11] http ://www.lifl.fr/OPAC/

tableau, la résolution a duré environ 25 jours avec un temps de résolution cumulé d'environ 22 ans. En moyenne, 328 processeurs sont utilisés. La figure FIG. 3.3 montre l'évolution du nombre de processeurs utilisés pendant la résolution. Le pic de processeurs enregistré est de 1.195.

Temps de résolution	25 jours
Temps cumulé	22 ans
Nombre moyen de travailleurs	328
Nombre maximal de travailleurs	1.195
Utilisation des processeurs des travailleurs	97%
Utilisation du processeur du fermier	1,7%
Opérations de sauvegarde	4.094.176
Allocations d'unités de travail	129.958
Nœuds explorés	6,50874 e+12
Nœuds redondants	0,39%

TAB. 3.9 – Statistiques enregistrées durant la deuxième expérimentation.

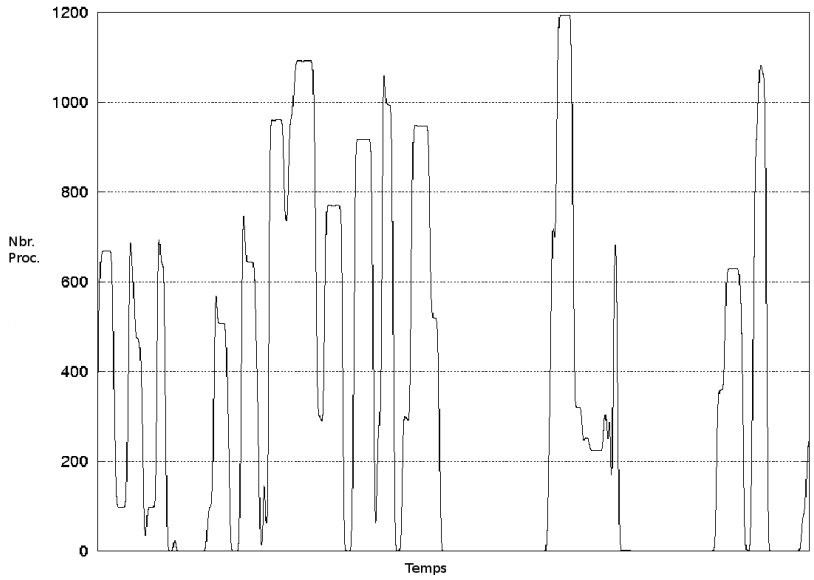

FIG. 3.3 – Evolution du nombre de processeurs utilisés durant la deuxième expérience.

Durant cette expérimentation, les processus B&B ont sollicité environ 130 milles fois le mécanisme d'équilibrage de charge du coordinateur, et effectué plus de 2 millions d'opérations de sauvegarde. Ceci démontre le fonctionnement robuste de ces deux mécanismes. Le coordinateur opère des sauvegardes toutes les 30 minutes. Plus de 6 milliards de nœuds ont été explorés. Certains nœuds sont explorés de manière redon-

dante, cela survient en phase de terminaison de la résolution. En effet, s'il ne reste plus que des intervalles d'une taille inférieure à un certain seuil, ces derniers sont dupliqués en réponse aux requêtes de demande de travail. Le travail spéculatif ainsi réalisé permet d'accélérer la résolution.

Les statistiques enregistrées montrent que le pourcentage de nœuds redondant est faible (inférieur à $0,4\%$). Les processeurs travailleurs sont exploités en moyenne à 97%, tandis que le processeur hébergeant le fermier est exploité à $1,7\%$. Ces deux pourcentages sont de bons indicateurs sur l'efficacité parallèle de l'approche et de sa qualité quant au passage à l'échelle. Dans le paradigme fermier-travailleur, une bonne approche parallèle doit maximiser l'utilisation des processeurs des travailleurs, et minimiser l'utilisation du processus fermier.

En terme de puissance de calcul utilisée, cette expérience se place en deuxième position des expérimentations à grande échelle menées sur les problèmes d'optimisation combinatoire. A notre connaissance, une seule autre expérience[12], a duré davantage de ressources de calcul. Le tableau TAB. 3.10 montre les principaux défis réalisés en optimisation combinatoire. $Sw24978$ est la seule instance qui a demandé davantage de puissance de calcul. Il s'agit d'une instance du problème de voyageur de commerce qui consiste à trouver le plus court chemin pour visiter 24.978 villes suédoises. La puissance de calcul utilisée pour résoudre cette instance est équivalente à environ 84 ans de calcul sur un processeur Intel Xenon 2.8 Ghz.

Pos.	**Prob.**	**Instance**	**Description**	**Puiss. de calcul**
1	TSP	$Sw24978$	24.978 villes de Suède	84 ans/Intel 2.8 GHz
2	FS	$Ta056$	50 tâches sur 20 machines	22 ans
3	TSP	$D15112$	15.112 villes d'Allemagne	22 ans/ Alpha 500 MHz
4	QAP	$Nug30$	Instance du QAP de taille 30	7 ans/HP-C3000 400MHz
5	TSP	$Usa13509$	13.509 villes des USA	4 ans

TAB. 3.10 – Principaux défis en optimisation combinatoire.

3.5.2 Application à un problème multi-critère

Pouvoir résoudre des instances multi-critères est l'une des contributions de B&B@Grid en comparaison avec les autres plate-formes. En guise de validation, la plate-forme a été utilisée pour résoudre l'instance bi-critère présentée dans la section 3.5. L'expérimentation a été réalisée sur une grille composée des trois grappes universitaires, et de quatre grappes de Grid'5000. Il s'agit des grappes de Lille, Rennes, Sophia Antipolis et Toulouse. Le tableau TAB. 3.11 résume les statistiques les plus importantes enregistrées pendant la résolution, et la figure FIG. 3.3 montre l'évolution du nombre de processeurs

[12] http ://www.tsp.gatech.edu/world/swlog.html

utilisés. Comme l'indique le tableau TAB. 3.9, la résolution a duré environ une semaine avec une moyenne de 334 processeurs, un pic de 704 processeurs, et un temps de calcul cumulé d'environ 5 ans. Durant la résolution, les processus B&B ont sollicité le mécanisme de régulation de charge environ 2 millions de fois. Au total, plus de 2000 milliards de nœuds de l'arbre ont été explorés. Le tableau TAB. 3.11 montre que le taux de nœuds redondants est inférieur à 0,02%. Plus de 2 millions de sauvegardes ont été faites par les processus B&B, tandis que le coordinateur a effectué une sauvegarde toutes les 30 minutes. Ces sauvegardes ont permis au mécanisme de tolérance aux fautes de gérer les centaines de pannes survenues. En effet, plus de 1.000 pannes des hôtes travailleurs et une panne volontaire du hôte fermier ont été enregistrées.

Les processeurs travailleurs ont été exploités en moyenne à 98,1 % tandis que le processeur fermier a été exploité à 2,4 %. Ces deux taux confirment l'efficacité parallèle de cette approche, ainsi que sa capacité quant au passage a l'échelle.

Début	28 Mars 11 :01 :02 2006
Fin	3 Avril 04 :08 :10 2006
Durée	137 :07 :08
Temps de calcul cumulé	5 ans et 80 jours
Average number of workers	334.37
Maximum number of workers	704
Cpu travailleurs	98,1%
Cpu coordinateur	2,4%
Sauvegardes	2.608.612
Pannes de travailleur	1.082
Pannes du fermier	1
Allocations de travail	1.721.564
Nœuds traités	2,23739 e+12
Nœuds redondants	0,014%
processeurs rejoignant le calcul	1.665

TAB. 3.11 – Statistiques enregistrées lors de la résolution bi-critère.

L'expérimentation a permis de résoudre pour la première fois l'instance bi-critère. Le front Pareto exact de cette instance est constitué de 21 solutions. Ce sont les solutions : (2834, 2770), (2836, 2549), (2837, 2518), (2838, 2345), (2839, 2343), (2840, 2316), (2844, 2285), (2845, 2270), (2848, 2065), (2849, 2058), (2851, 2025), (2857, 2020), (2859, 1980), (2862, 1961), (2865, 1943), (2866, 1891), (2872, 1884), (2876, 1843), (2877, 1838), (2879, 1806) et (2902, 1792). La première composante de chaque point représente le coût du *makespan*, tandis que la deuxième composante donne le coût du *tardiness*. La figure FIG. 3.4 illustre le front Pareto exact de cette instance.

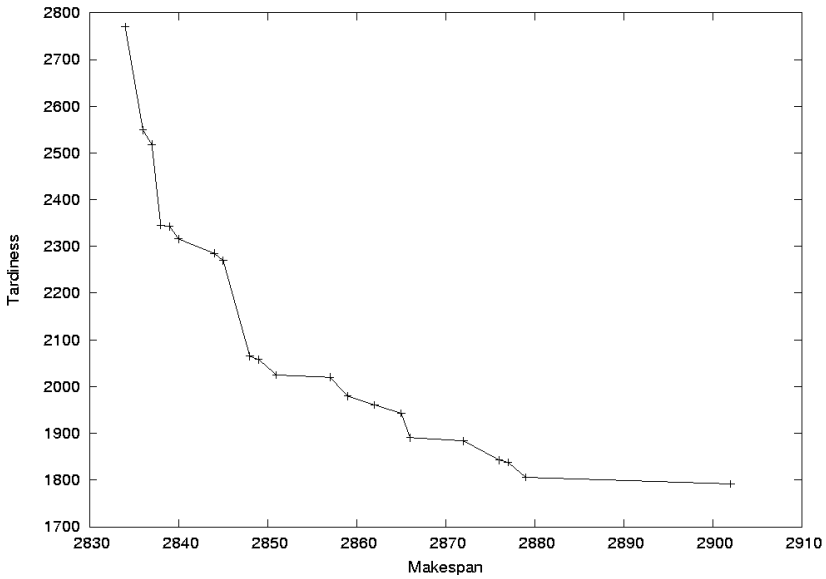

FIG. 3.4 – Front Pareto obtenu pour l'instance bi-critère considérée.

Conclusion

Dans ce chapitre, nous avons présenté la plate-forme B&B@Grid qui implémente l'approche présentée dans le chapitre précédent (voir l'annexe B). De nombreuses plates-formes dédiées à la parallélisation des méthodes exactes sont décrites dans la littérature. Ces plates-formes résolvent des problèmes mono-critères, diffèrent par les méthodes exactes prises en compte, sont basées sur le paradigme *fermier-travailleur* ou *fermier-travailleur hiérarchique*, et adoptent souvent le mode de communication *push*. Contrairement aux autres plates-formes, B&B@Grid est pensée dès le départ pour les grilles. Son objectif est donc de mieux gérer les caractéristiques de cet environnement. Etant donnée l'hétérogénéité des grilles, B&B@Grid utilise son propre environnement parallèle pour établir les communications entre les processus. Ceci permet d'exploiter cette plate-forme sur toute machine où le noyau de base Linux est installé. Cette nouvelle plate-forme adopte le mode de communication *pull*. Ce mode convient davantage aux environnements où des pare-feux sont présents. En plus des problèmes mono-critères, B&B@Grid permet de résoudre également les problèmes multi-critères. Cette plate-forme est dédiée à la gridification de toute méthode exacte de type B&B. B&B@Grid supporte le paradigme *fermier-travailleur*, et peut être étendue pour être utilisée avec les autres paradigmes.

Dans le but de valider B&B@Grid, trois expérimentations ont été effectuées. Les deux premières sont réalisés sur une instance mono-critère du problème ($F/permut/C_{max}$). Cette instance, connue sous le nom de *Ta056*, est publiée dans [Tai93]. La solution optimale de *Ta056* est restée inconnue pendant plusieurs années. Le défi de la première

expérimentation est de résoudre d'une façon exacte cette instance. L'expérimentation a permis de trouver la solution optimale de *Ta056* après plus d'un mois de résolution. L'objectif de la deuxième expérimentation est de prouver l'efficacité de B&B@Grid pour la résolution de problèmes mono-critères. En terme de puissance de calcul utilisée, la deuxième résolution de *Ta056* se classe en deuxième position des expérimentations à grande échelle menées sur les problèmes d'optimisation combinatoire. Elle a nécessité environ 328 processeurs en moyenne pendant plus de 25 jours. Le pic de processeurs, atteint pendant cette résolution, est d'environ 1200 processeurs. La troisième expérimentation a permis également de résoudre pour la première fois une instance bi-critère, et de valider la plate-forme sur ce type de problèmes.

Deuxième partie

Coopération avec les méta-heuristiques sur grilles de calcul

Chapitre 4

Coopération avec les méta-heuristiques parallèles

Les deux approches hybrides, présentées dans ce chapitre, et leurs déploiements avec le modèle de coordination, décrits dans le chapitre suivant, sont publiés dans la revue internationale *Journal of Mathematical Modelling and Algorithms* **[MMT07a].**

Introduction

A l'instar de B&B@Grid, les approches, proposées pour la parallélisation des algorithmes B&B sur les grilles de calcul, sont basées souvent sur la stratégie *profondeur d'abord*. L'intérêt de cette stratégie est de réduire le nombre de sous-problèmes gérés simultanément par chaque processus. Par contre, son inconvénient est d'explorer davantage de sous-problèmes que la stratégie *meilleur d'abord*. Toutefois, il est possible de diminuer considérablement le nombre de sous-problèmes explorés. Il suffit d'initialiser les algorithmes B&B avec une (ou plusieurs) bonne(s) solution(s). Cependant, une (de) telle(s) solution(s) n'est (sont) souvent pas disponible(s) pour certaines instances de problèmes. C'est le cas notamment des instances non standards ainsi que des instances réelles. Il est donc indispensable d'hybrider tout B&B parallèle, dont la stratégie d'exploration est de type *profondeur d'abord*, avec une méthode approchée. Cette méthode peut être, par exemple, une méta-heuristique. Le rôle de la méta-heuristique est de fournir à l'approche exacte des solutions de bonne qualité afin de lui permettre d'éliminer le plus grand nombre de sous-problèmes avant de les explorer. Par conséquent, nous proposons et décrivons deux approches pour hybrider B&B@Grid avec une méta-heuristique parallèle.

Ce chapitre est organisé en trois sections. Dans la **section 4.1**, nous présentons les modèles parallèles de la classe des méta-heuristiques à population de solutions. Dans cette section, seuls les algorithmes évolutionnaires (AE) sont considérés. Néanmoins, le

principe de fonctionnement des modèles reste valable pour les autres méta-heuristiques à population de solutions. Cette section présente également le modèle parallèle à population de solutions utilisé par nos approches. Dans la **section 4.2**, nous analysons les modèles parallèles des méta-heuristiques à solution unique et décrivons aussi le modèle parallèle à solution unique exploité dans nos deux approches. Dans la **section 4.3**, nous justifions l'intérêt de combiner différentes méthodes d'optimisation, et présentons les deux schémas hybrides utilisés dans nos approches.

4.1 Modèles parallèles pour les méta-heuristiques à population de solutions

Les AE [Hol75] sont des techniques de recherche stochastique ayant été appliquées avec succès à différents problèmes réels et complexes. Leur principe de fonctionnement (voir l'algorithme 2) est basé sur l'application itérative d'opérateurs stochastiques à une population de solutions (ou individus). A chaque génération du processus d'évolution, des individus de la population sont sélectionnés (*politique de sélection*) et re-combinés (*opérateurs de variation* tels que le *croisement* ou *crossover* et la *mutation*) de manière à générer de nouvelles solutions pouvant en remplacer d'autres (*stratégie de remplacement*). Le processus d'évolution s'arrête lorsqu'un critère défini est avéré (*critère d'arrêt*).

Algorithm 2 Pseudo-code d'un AE.

```
Générer(P(0)) ;
t := 0 ;
Evaluer(P(t)) ;
Tant que Terminaison_non_avérée(P(t)) Faire
  P'(t) := Sélectionner(P(t)) ;
  P'(t) := Appliquer_op_variation(P'(t)) ;
  Evaluer(P'(t)) ;
  P(t+1) := Remplacer(P(t), P'(t)) ;
  t := t+1 ;
Fin TantQue
```

La population initiale est souvent générée de manière *aléatoire ou gloutonne*. La sélection peut être *aléatoire* ou *élitiste* utilisant une politique basée sur le *tournoi*, la *roulette*, la *sélection par rang*, etc. Le tournoi consiste à tirer uniformément T individus de la population et à en sélectionner le meilleur. La roulette consiste à donner à chaque individu une probabilité d'être sélectionné proportionnelle à sa performance (ou *fitness*). La sélection par rang est une roulette dans laquelle la performance d'un individu est mesurée non pas par sa qualité mais par son rang dans la population (compris entre 1 et la taille de la population). Trois stratégies de remplacement sont souvent

utilisées : *plus*, *générationnelle* et *stationnaire* ou *steady-state*. La stratégie *plus* consiste à fusionner les individus parents avec les enfants issus de leur reproduction, et éliminer de manière aléatoire ou élitiste une partie suffisante de la population résultante pour la ramener à la taille constante fixée. Dans l'approche générationnelle, les parents sont tous remplacés par les enfants générés. Dans le schéma stationnaire, un individu est sélectionné, généralement par tournoi, un second si le croisement doit être appliqué, et l'enfant résultant (après croisement et mutation éventuels) est réinséré dans la population en remplacement d'un parent sélectionné par tournoi inversé (le moins performant est retenu). Le critère d'arrêt peut être simplement un nombre fixé de générations, ou basé sur la progression de la qualité des solutions pendant un certain nombre de générations. Par exemple, le processus d'évolution s'arrête si aucune progression de la qualité des solutions n'est avérée après un seuil fixé de générations.

Dans le contexte multi-critère, la structure de l'algorithme reste inchangée. Par contre, des adaptations sont nécessaires dans les phases d'évaluation et de sélection. L'étape d'évaluation comprend, en plus du calcul des valeurs associées aux différents critères, l'obtention à partir de celles-ci d'une valeur d'efficacité. Une fonction (*e.g. d'agrégation*) de transformation du vecteur de fonctions objectifs (les termes objectif et critère sont des synonymes) en une valeur scalaire doit être définie. Cette dernière est utilisée dans les autres phases de l'algorithme, notamment la sélection (*e.g. ranking*). Le processus de sélection est basé sur deux mécanismes appelés *élitisme* et *partage* ou *sharing*. Ces derniers permettent respectivement de converger vers le meilleur front Pareto [OTT98, PM98, Meu02] et de maintenir une diversité des individus dans la population et dans le front Pareto. L'élitisme consiste à maintenir une population (appelée *archive Pareto*) autre que la population courante en archivant toutes les solutions Pareto optimales générées pendant le processus d'évolution. Cette archive, mise à jour à chaque itération, est utilisée dans le processus de sélection. En effet, différentes stratégies de sélection peuvent être envisagées selon que les individus sont choisis à partir de la population courante, à partir de l'archive ou à partir des deux. L'opérateur de partage maintient la diversité en considérant le degré de similitude d'un individu vis à vis des autres individus, souvent basé sur une distance à définir dans l'espace objectif.

Pour une conception et une mise en œuvre réutilisable d'un AE, on peut distinguer deux parties : une *partie invariante* et une *partie spécifique au problème*. Cette dernière comprend principalement le codage des solutions, la définition de la fonction d'évaluation ou objectif, et la spécification des opérateurs de variation notamment les opérateurs de recombinaison et de mutation. Par contre, la partie invariante est complètement indépendante du problème traité. Elle comporte essentiellement la stratégie de sélection des individus (la roulette, le rang, le tournoi, etc.), la procédure de remplacement (générationnelle, élitiste, par tournoi, etc.), et enfin le critère de décision d'arrêt/continuation (l'atteinte d'un nombre prédéfini d'itérations, la notification de convergence exprimée par un nombre d'itérations sans amélioration, etc.).

L'information définissant la progression du processus d'évolution à une itération

donnée se résume à la génération courante (une population de solutions) ainsi qu'à l'état du critère de décision d'arrêt/continuation. Par exemple, il peut s'agir pour ce dernier d'un compteur d'itérations contrôlant l'avancement de la recherche. Dans le cas multi-critère, il faut ajouter l'archive qui stocke le front Pareto courant.

On distingue trois modèles parallèles pour les méta-heuristiques à population de solutions : le *modèle insulaire*, le *modèle d'évaluation parallèle de la population* et le *modèle d'évaluation parallèle d'une solution.*

4.1.1 Modèle insulaire

Dans le modèle insulaire [CHMR87], inspiré de certains comportements observés dans les niches écologiques, plusieurs AE sont déployés pour faire évoluer simultanément différentes populations (îles) de solutions. Dans la suite de ce document, on confondra parfois une île avec son AE ou sa population. Les AE peuvent être *homogènes ou hétérogènes* selon qu'ils utilisent des paramètres (opérateurs de variation et autres) identiques ou différents. Comme le montre la figure FIG. 4.1, ces AE échangent de manière *synchrone ou asynchrone* du matériel génétique. Cet échange a pour objectif de diversifier l'espace de recherche et de retarder la convergence du processus d'évolution. Cela permet l'obtention de solutions de meilleure qualité et une exécution *plus robuste* i.e. minimisant la déviation en terme de qualité d'une exécution à une autre.

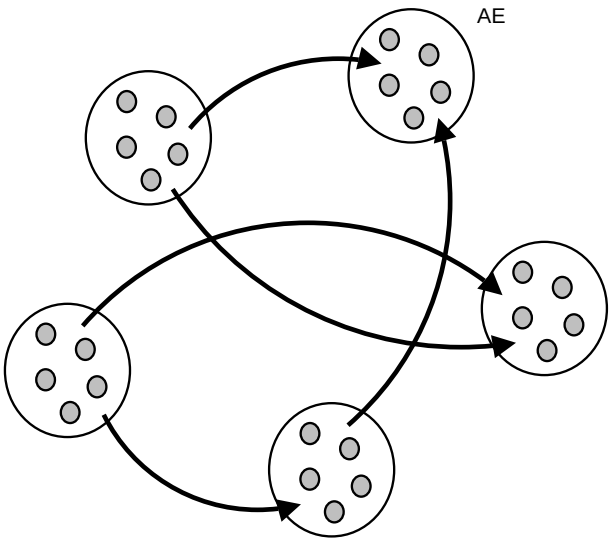

FIG. 4.1 – Le modèle coopératif insulaire d'AE.

Pour chacune des populations, le processus de gestion des migrations synchrones ou asynchrones intervient au terme de chaque génération de l'AE standard, succédant à la phase de remplacement. La politique de migration d'individus entre îles est définie par

les paramètres suivants : le critère de décision de migration, la topologie des échanges d'individus, le nombre d'émigrants intervenant dans une opération de migration et la politique d'intégration des immigrants. Tous ces paramètres sont indépendants du problème traité, ce qui rend le modèle générique.

Critère de décision de migration : la migration d'individus d'une île vers une autre peut être décidée suivant un critère aveugle ou intelligent. La migration aveugle peut être périodique ou probabiliste. La première intervient sur chaque AE après un nombre de générations fixé par l'utilisateur (fréquence de migration). La seconde consiste à effectuer la migration après chaque génération avec une probabilité définie par l'utilisateur. A l'inverse, la migration intelligente est dirigée par des critères d'amélioration de la qualité des solutions. Un seuil d'amélioration de cette qualité doit être fourni, et si l'amélioration entre deux générations est inférieure à ce seuil, une opération de migration est déclenchée. Ces différents paramètres (fréquence, probabilité, seuil de pression) doivent être fixés en fonction de la puissance des machines pour prendre en compte le caractère hétérogène de la grille.

La topologie des échanges d'individus : la topologie indique pour chaque île ses voisins au regard de la migration i.e. l'île (ou les îles) de destination/provenance de ses émigrants/immigrants. De nombreux travaux ont été menés pour étudier l'impact de la topologie sur la qualité des résultats obtenus [CHMR87, SWM91]. Il ressort de ces études que les graphes cycliques sont préférables, les modèles en anneau et hypercube sont d'ailleurs largement utilisés. Il faut noter qu'un nombre trop élevé de jonctions entre îles se révèle inefficace, l'ensemble des populations distribuées se comportant alors comme une seule population globale. Sur une grille de calcul (caractère volatile), notons que la disparition d'une île requiert une reconfiguration dynamique de la topologie. Cette reconfiguration peut s'avérer très coûteuse et rendre le processus de migration inefficace. Belding [Bel95] a expérimenté une coopération d'AE sans topologie prédéfinie, signifiant que la population cible de chaque migration est choisie de manière aléatoire. Les résultats ont montré que ce type de topologie contribue à améliorer de manière significative la robustesse. Une topologie aléatoire est donc envisageable dans le contexte des grilles de calcul. Nos travaux confirment les résultats de Belding.

Le nombre d'émigrants : ce paramètre peut être défini comme un nombre fixé ou variable d'individus, ou comme un pourcentage d'individus de la population et/ou de l'archive Pareto (dans le contexte multi-critère). Le choix de ce paramètre est crucial. En effet, s'il est trop faible les îles auront tendance à évoluer de manière indépendante et la migration aura moins d'impact sur le retard de la convergence et donc sur la qualité des solutions obtenues. Si le nombre d'émigrants est trop élevé le coût de communication sera plus important surtout sur une grille, et les îles auront tendance à converger vers les mêmes solutions. Un compromis est donc à trouver pour allier à la fois l'exploration et l'intensification de la recherche.

Politique de sélection des émigrants : la politique de sélection des émigrants

indique pour chaque île de manière *élitiste* ou *aléatoire* les individus à migrer. La stratégie aléatoire ne garantit pas la sélection des meilleurs individus, mais elle a un coût de calcul plus faible. La stratégie élitiste, souvent basée sur la roulette, le rang, le tournoi ou l'échantillonnage uniforme, tente de sélectionner les meilleurs individus de la population. Dans le contexte multi-critère, les individus peuvent être sélectionnés de la population courante, de l'archive Pareto ou des deux.

Politique de remplacement/intégration des immigrants : de manière symétrique, la politique de remplacement/intégration des immigrants indique de manière élitiste ou aléatoire les individus de la population de destination à remplacer par les nouveaux arrivants. Dans le modèle insulaire multi-critère, plusieurs politiques peuvent être envisagées. Par exemple, les solutions Pareto immigrantes remplacent de manière aléatoire les individus de la population n'appartenant pas au front Pareto local. On peut également faire un classement de tous les individus de la population en fronts Pareto en utilisant la relation de dominance. Le plus mauvais front est remplacé par les solutions immigrantes. Une autre solution consiste à fusionner le front Pareto immigrant avec le front local pour constituer la nouvelle archive locale.

L'implémentation du modèle insulaire peut être *asynchrone* ou *synchrone*. Le modèle asynchrone associe à chaque algorithme insulaire un critère de décision d'immigration. Il est réévalué au terme de chaque itération à partir de l'état de la population courante et/ou de l'archive Pareto. S'il est avéré, des requêtes sont émises vers les populations voisines, lesquelles les traiteront dans un temps indéterminé. La réception et intégration de nouvelles solutions interviendra lors des itérations ultérieures. Cependant, en raison de l'hétérogénéité logicielle et/ou matérielle, il se peut que les populations soient à différents stades d'évolution, entraînant le problème de *non-effet* ou du *super individu*. L'arrivée de solutions de mauvaise qualité dans une population à un stade d'évolution avancé n'apporte aucune contribution. Dans la situation opposée, la coopération insulaire se traduit par une convergence prématurée. Le modèle est non bloquant et donc plus performant et plus tolérant aux pannes jusqu'à un certain seuil autorisé de pertes de migrations.

Dans le modèle synchrone, les algorithmes insulaires procèdent à une opération de synchronisation à chaque itération. Cette opération se traduit par un échange d'individus entre les différentes îles. La synchronisation garantit que ces îles sont au même stade d'évolution évitant ainsi les problèmes évoqués ci-dessus. Par contre, elle pose des problèmes de performance en terme de temps de calcul sur les grilles en raison de leur hétérogénéité. En effet, le processus d'évolution est souvent suspendu sur les machines les plus puissantes en attente des machines les moins puissantes. En outre, le modèle synchrone est non tolérant aux pannes puisque la perte d'une île entraîne le blocage du modèle. Ceci le rend difficile à exploiter dans un environnement volatile. En résumé, le modèle synchrone est plus complexe à mettre en œuvre sur grilles de calcul de manière efficace.

Les données (ou la mémoire) du modèle coopératif insulaire nécessaires au mécanisme de sauvegarde/restauration comprennent (comprend) uniquement les individus en cours de migration d'une population vers une autre. On note que, dans le mode asynchrone, cette information n'est pas critique à l'exécution du modèle parallèle puisque les AE sont des processus stochastiques.

Dans nos travaux, l'algorithme évolutionnaire instancié sur chaque île est l'Algorithme Hybride Génétique Mimétique (AGMA) proposé dans [Bas05]. Comme son nom l'indique, AGMA est une méta-heuristique qui combine un algorithme génétique et un algorithme mimétique. La figure FIG. 4.2 décrit la parallélisation d'AGMA selon le modèle insulaire. Ce modèle est exploité pour la parallélisation de la partie algorithme génétique d'AGMA. Chaque AGMA fait évoluer la population de son île et stocke dans une archive Pareto locale les solutions non dominées. L'échelle et les délais de communication dans les grilles obligent à réduire le nombre de migrants. Par conséquent, la politique de sélection des émigrants est basée sur leur qualité à leur quantité. Une île échange donc avec les autres AGMA son archive Pareto. Cet échange se fait de manière asynchrone en raison de la volatilité des machines de la grille. Dans nos travaux, aucune topologie pour la migration d'individus n'est utilisée. A chaque migration, la population cible est choisie de manière aléatoire. L'absence de topologie s'impose à cause de la nature volatile des machines de la grille. En effet, il est difficile de respecter une topologie prédéfinie dans un environnement où les machines tombent en panne fréquemment. Le critère de migration est périodique en raison de la nature hétérogène de la grille. Certaines îles évoluent plus vite que d'autres. Un critère de migration basé sur le nombre de générations, par exemple, avantagerait les AGMA déployés sur les machines les plus rapides. Ces AGMA submergeraient alors les autres AGMA par leurs patrimoines génétiques et rendrait les populations moins diversifiées. Dans nos travaux, la migration se fait donc toujours au bout d'une certaine période de temps.

4.1.2 Modèle d'évaluation parallèle de la population

L'évaluation de la population est la phase la plus coûteuse en temps CPU d'un AE, en particulier lorsqu'il s'agit de résoudre des problèmes réels. Succédant à la phase de transformation (par application des opérateurs de variation), l'évaluation consiste à déterminer la qualité (ou *fitness*) de chaque nouvelle solution produite. Les différentes évaluations étant indépendantes les unes des autres, leur parallélisation est donc naturelle et est basée sur le partitionnement des individus suivant le modèle *fermier-travailleur* (FIG. 4.3).

Dans le contexte multi-critère, l'approche d'implémentation souvent utilisée est celle qui consiste à distribuer les individus de la population entre les différents travailleurs. Chaque travailleur évalue le vecteur de toutes les fonctions objectifs sur les individus reçus. Une autre approche, également envisageable, consiste à répartir les fonctions objectifs entre les travailleurs. Chacun d'entre eux évalue sa propre fonction sur tous les

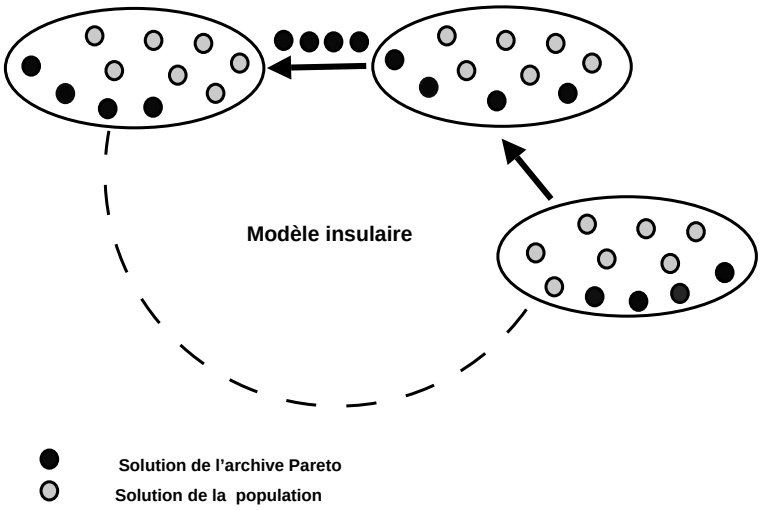

FIG. 4.2 – Coopération insulaire entre les AG de l'algorithme AGMA.

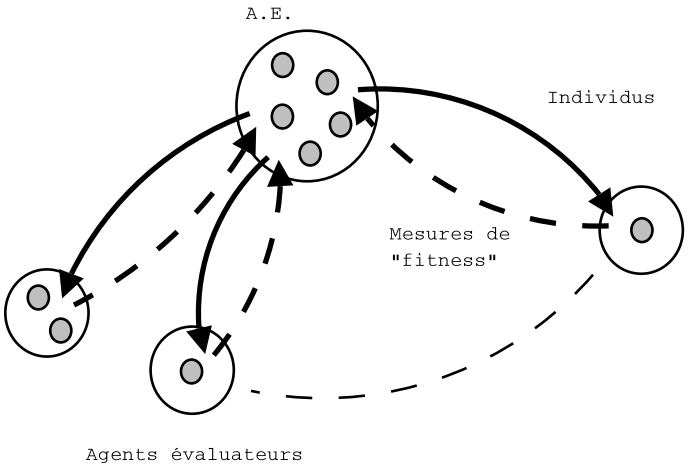

FIG. 4.3 – Illustration du modèle d'évaluation parallèle.

individus de la population. Le fermier se chargera de l'agrégation des différents résultats sur toute la population. Cette approche offre un degré de parallélisme et une extensibilité limités par le nombre de critères considérés dans la fonction évaluant les critères, c'est à dire souvent 2 à 3.

Selon l'ordre d'exécution de la phase d'évaluation par rapport au reste de l'AE, le modèle a deux modes : *synchrone* et *asynchrone*. Dans le mode synchrone, le processus *fermier* gère l'évolution de la population, exécute en séquence les phases de sélection, transformation et remplacement. A chaque itération, il distribue l'ensemble des nouvelles solutions générées entre différents agents évaluateurs (travailleurs), et *se met en attente des résultats*. Après collecte des résultats, le processus d'évolution se poursuit à nouveau. Le modèle n'altère en rien le comportement de l'AE original.

L'exécution du modèle synchrone est toujours synchronisée avec le retour de la dernière solution évaluée. Il est donc bloquant sur une grille en raison de l'hétérogénéité de celle-ci. De plus, le modèle n'est pas tolérant aux pannes, aussi la notification de disparition d'un agent évaluateur nécessite une redistribution des solutions lui ayant été affectées vers d'autres agents évaluateurs. Il convient donc de mémoriser l'ensemble des solutions non encore évaluées. Par ailleurs, l'extensibilité de ce modèle est limitée par la taille de la population.

Dans le mode asynchrone, la phase d'évaluation n'est pas synchronisée avec le reste de l'AE. Le processus fermier n'attend pas le retour de toutes les évaluations pour exécuter les phases de sélection, transformation et remplacement. L'AE *Steady-State* est le meilleur exemple illustrant le modèle asynchrone. Il offre divers avantages importants en comparaison du modèle synchrone. Il est naturellement non bloquant et tolérant aux pannes (à un taux de perte raisonnable) car il n'y a pas de contrôle quant au retour des solutions émises pour évaluation. De plus, ce modèle est bien adapté, en terme d'efficacité, à l'ensemble des applications caractérisées par un coût d'évaluation irrégulier. En outre, il n'est pas nécessaire de mémoriser les solutions traitées par les processus évaluateurs, ce qui réduit le taux d'occupation de la mémoire. Enfin, l'extensibilité de ce modèle n'est pas limitée par la taille de la population.

Par ailleurs, le modèle d'évaluation parallèle (a)synchrone, étant indépendant du problème traité, sa conception est générique. Cependant, son efficacité à l'exécution dépend fortement de la granularité des évaluations i.e. le nombre de solutions qu'un travailleur doit évaluer à chaque envoi. Les délais de communication étant importants, le problème de granularité de parallélisme est accentué sur une grille (grande échelle). Le choix de la taille du grain i.e. du nombre pertinent de solutions affectées à chaque travailleur est un paramètre de performance crucial pour le modèle.

4.1.3 Modèle d'évaluation parallèle d'une solution

L'évaluation de chaque solution de la population est effectuée de manière parallèle centralisée. Dans ce modèle de type *fermier-travailleurs*, une même solution est répliquée sur les différents sites évaluateurs (travailleurs). Les valeurs de qualité partielles retournées par ces travailleurs seront ensuite agrégées par le fermier.

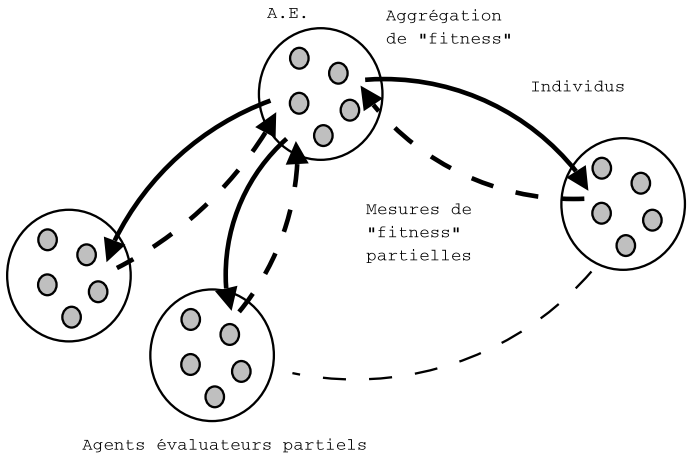

FIG. 4.4 – L'évaluation parallèle d'une solution unique.

Ce modèle est nécessaire à la résolution de problèmes réels dont la fonction objectif requiert l'accès à des bases de données volumineuses ne pouvant être manipulées sur un seul site. A cause de cette contrainte matérielle, elles sont distribuées entre différents sites. Le parallélisme de données est exploité lors de l'évaluation de la fonction objectif. La parallélisation de la fonction objectif est également intéressante lorsque l'évaluation d'une seule solution s'avère coûteuse. Même si son extensibilité est limitée, son exploitation conjointe avec le modèle d'évaluation parallèle en particulier synchrone de la population permet d'étendre le degré de parallélisme de celui-ci et améliorer son extensibilité. En optimisation multi-critère, toutes les fonctions objectifs peuvent être parallélisées simultanément.

Le modèle nécessite de nouveaux composants spécifiques au problème traité notamment le vecteur de fonctions d'évaluation partielles ainsi qu'une fonction d'agrégation de celles-ci. L'implémentation de ce modèle étant toujours synchrone, il est indispensable de mémoriser les valeurs de qualité partielles de la solution évaluée pour gérer la tolérance aux pannes et la disponibilité variable des machines sur une grille. Le modèle d'évaluation parallèle d'une solution est souvent ignoré dans les taxinomies [CP98]. Cependant, son utilisation conjointe avec d'autres modèles améliore de manière siginificative le degré de parallélisme.

4.2 Modèles parallèles pour les méta-heuristiques à solution unique

Les méta-heuristiques à solution unique peuvent être vues comme des marches à travers des voisinages i.e. des trajectoires dans l'espace de recherche du problème traité [CT02]. Ces marches sont conduites par des procédures itératives (voir l'algorithme 3) permettant de passer d'une solution à une autre dans l'espace de recherche. Nous appellerons *mouvement* l'opération de base permettant de transformer une solution en une autre solution se trouvant dans son voisinage. Par exemple, dans le problème du voyageur de commerce le mouvement peut être une permutation de deux arcs du circuit représentant la solution courante. La sélection d'un mouvement acceptable peut être basée sur la génération d'un voisinage partiel ou total, déterministe (e.g. élitiste) ou stochastique.

Algorithm 3 Pseudo-code d'une recherche locale.

```
Générer(s(0)) ;
t := 0 ;
Tant que Terminaison_non_avérée(s(t)) Faire
  m(t) := SélectionnerMouvementAcceptable(s(t)) ;
  s(t+1) := AppliquerMouvement(m(t), s(t)) ;
  t := t+1 ;
Fin TantQue
```

La conception et réalisation d'une méta-heuristique à solution unique comporte deux parties : une partie invariante et une partie spécifique au problème traité. Dans la partie spécifique, il convient de définir le codage d'une solution et le(s) voisinage(s) directement lié(s) au problème traité. Au contraire, les composants énoncés ci-après sont génériques, et diffèrent selon la méta-heuristique utilisée : le choix de la solution initiale, l'ensemble des données mémorisées au cours de l'exploration (meilleure solution courante, liste de mouvements tabous, divers paramètres guidant la trajectoire, etc.), la stratégie de génération du voisinage candidat (partiel ou complet, déterministe ou non, etc.), la stratégie de sélection d'une solution voisine (élitiste ou stochastique), et enfin le critère d'arrêt (identification d'un optima local, nombre d'itérations fixé atteint, etc.).

La configuration des différents composants génériques ou spécifiques ont une grande influence sur l'efficacité des méta-heuristiques à solution unique en termes de temps d'exécution et de qualité de la solution produite. Un autre moyen d'améliorer cette efficacité est d'utiliser le parallélisme. Il existe trois modèles parallèles principaux pour les méta-heuristiques à solution unique : le *modèle parallèle multi-départ*, le *modèle d'évaluation parallèle du voisinage* et le *modèle d'évaluation parallèle d'un mouvement*.

4.2.1 Modèle parallèle multi-départ

Le modèle parallèle multi-départ consiste à déployer simultanément plusieurs marches afin d'obtenir des solutions meilleures et plus robustes. Les algorithmes à base de recherche locale déployés peuvent être de nature *homogènes ou hétérogènes* (démarrant d'une même solution ou non, associés aux mêmes paramètres ou non guidant la trajectoire.), *indépendants ou coopératifs* (FIG. 4.5). Les marches indépendantes n'échangent pas d'information durant leur exécution, les résultats sont donc identiques à ceux que l'on obtiendrait par de mêmes algorithmes séquentiellement exécutés. Dans ce contexte particulier, le parallélisme est trivial, facile à implémenter, et totalement indépendant du problème traité. Cela justifie les nombreuses implémentations proposées et expérimentées dans la littérature.

Dans le modèle multi-départ parallèle coopératif, des informations sont échangées entre les différentes méthodes de recherche locale. Contrairement au modèle multi-départ indépendant, l'implémentation est généralement spécifique au problème. En effet, la nature des informations échangées ne peut être définie de manière générique. Ces informations peuvent être les bonnes solutions visitées, des mouvements effectués et/ou d'autres paramètres exprimant la trajectoire accomplie dans l'espace de recherche, etc.

La communication peut être mise en œuvre par le biais d'un processeur central contrôlant les échanges entre les différentes méthodes, ou selon un modèle totalement décentralisé de type pair-à-pair. Le choix de la topologie adoptée sera déterminante à l'extensibilité du modèle déployé. L'échange centralisé se montre généralement moins extensible que celui totalement distribué. D'autre part, le mode de communication peut-être synchrone ou non. Le premier se distingue par une fiabilité accrue mais une efficacité moindre à l'exécution.

L'algorithme mimétique d'AGMA[Bas05] est une combinaison d'un algorithme génétique et d'une recherche locale. La recherche locale remplace l'opérateur de mutation de l'algorithme génétique. Dans nos travaux, le modèle multi-départ est utilisé pour paralléliser la recherche locale. La figure FIG. 4.6 décrit la parallélisation de la recherche locale d'AGMA selon le modèle multi-départ. A chaque itération de la partie mimétique, l'archive Pareto est exploitée et intensifiée par un processus de recherche locale. Chaque solution de l'archive Pareto constitue la solution initiale d'une recherche locale dont le but est de calculer le voisinage de celle-ci. Les solutions non dominées de ce voisinage sont intégrées dans l'archive Pareto de l'AGMA.

4.2.2 Modèle d'évaluation parallèle du voisinage

Le modèle d'évaluation parallèle du voisinage est un modèle de type *fermier-travailleur* (FIG. 4.7) qui n'altère en rien le comportement de l'heuristique parallélisée. Il se caractérise principalement par une distribution des calculs, une même recherche séquentielle

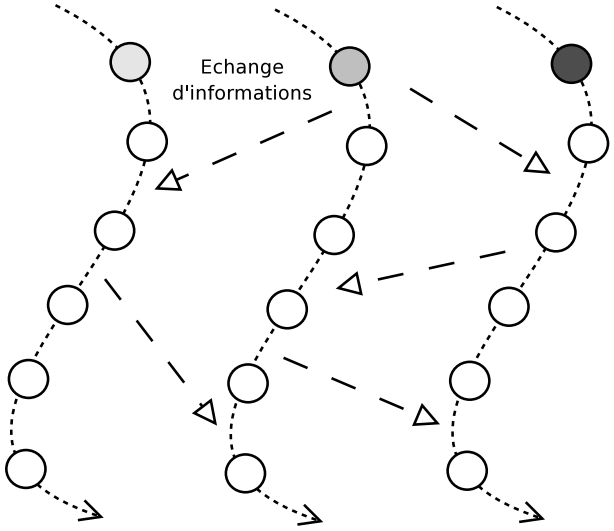

FIG. 4.5 – Coopération de recherches locales concurrentes.

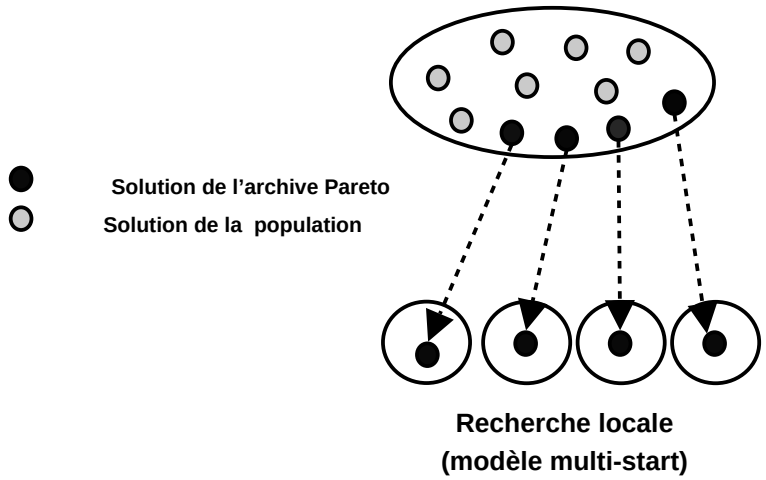

FIG. 4.6 – Parallélisation de la recherche locale avec le modèle multi-départ.

produirait les mêmes résultats. Au début de chaque itération, le fermier duplique la solution courante qui sera émise vers différents nœuds travailleurs distribués, lesquels considéreront un voisinage partiel de solutions candidates. Le fermier se met en attente des résultats avant de poursuivre sa marche.

L'implémentation parallèle de ce modèle peut être réalisée de manière indépendante. Il convient cependant de définir les mécanismes afin de partitionner le voisinage d'une solution. Ce modèle de parallélisation de *bas-niveau* se montre efficace si le temps d'évaluation d'une solution voisine est important et/ou le nombre de solutions voisines candidates à considérer est élevé. Selon la stratégie de sélection du prochain voisin à évaluer (*e.g.* le choix du premier meilleur voisin), l'efficacité du modèle n'est pas toujours avérée.

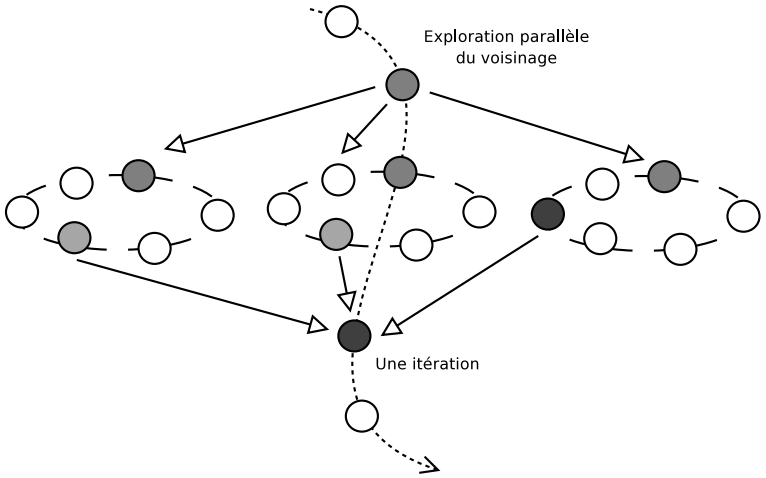

FIG. 4.7 – La décomposition et l'exploration parallèle du voisinage d'une solution.

Dans le contexte multi-critère, l'exploration parallèle du voisinage est souvent appliquée à un front Pareto entier. Dans ce cas, le modèle d'exploration parallèle du voisinage est combiné avec le modèle multi-départ. Le modèle hiérarchique obtenu consiste à appliquer simultanément une exploration parallèle du voisinage à chaque solution du front Pareto. Ce modèle s'est révélé efficace dans nos travaux sur les applications bi-critères d'ordonnancement de type Flow-Shop.

4.2.3 Modèle d'évaluation parallèle d'un mouvement

Le principe du modèle est analogue à celui de l'évaluation parallèle de la fonction objectif décrite précédemment dans le cadre des AE. Ce modèle s'avère particulièrement intéressant lorsque l'évaluation incrémentale d'un seul mouvement est coûteuse.

La mise en œuvre de ce modèle nécessite l'implémentation d'un ensemble de nou-

velles fonctions d'évaluation partielles du mouvement appliqué à une solution donnée. Il convient également de définir un opérateur d'agrégation de telles valeurs de qualité.

Ce modèle est caractérisé par sa granularité très fine. C'est pourquoi la plupart de ses implémentations sont réalisées sur des architectures parallèles à mémoire partagée [KR87].

4.3 Modèles de coopération entre méthodes d'optimisation

Afin de tirer avantage des bénéfices apportés par les différentes méthodes d'optimisation, il est souvent nécessaire de les combiner. Aujourd'hui, les méthodes hybrides permettent d'obtenir les meilleurs résultats sur la plupart des problèmes, qu'ils soient académiques (voyageur de commerce, affectation quadratique, etc.) ou pratiques (issus du monde réel) [WM95]. Nous nous sommes intéressés à deux types d'hybridation : l'hybridation entre méta-heuristiques (à population de solutions et à solution unique) et l'hybridation entre méta-heuristiques et méthodes exactes.

La motivation du premier type d'hybridation est d'exploiter à la fois le pouvoir d'exploration des méta-heuristiques à population de solutions et le pouvoir d'intensification des méta-heuristiques à solution unique pour produire des solutions diversifiées et de meilleure qualité. D'autre part, le deuxième type d'hybridation permet d'exploiter la complémentarité entre méta-heuristiques et méthodes exactes : (1) les méthodes exactes permettent de prouver l'optimalité des solutions pour des instances de taille raisonnable, alors que les méthodes approchées trouvent de bonnes solutions pour des instances de taille nettement supérieure et/ou avec un nombre de contraintes beaucoup plus important ; (2) Les solutions optimales trouvées par les méthodes exactes servent souvent d'étalon pour mesurer l'efficacité des méthodes approchées ; (3) Les bonnes solutions obtenues par les méthodes approchées sont quant à elles d'excellentes bornes pour élaguer des branches des méthodes exactes pendant le parcours arborescent et réduire ainsi considérablement le temps de calcul.

Une taxinomie hiérarchique et à plat des différents schémas d'hybridation est proposée dans [Tal02] et illustrée par la figure FIG. 4.8. Dans la classification à plat, trois critères sont considérés : la composition des méthodes d'optimisation hybridées (homogène ou hétérogène), leur fonction (généraliste ou spécifique) et la portée du domaine du problème considéré (globale ou partielle). Les hybridations dites homogènes, par exemple le modèle insulaire, se basent sur le couplage de méta-heuristiques de même type, mais généralement associées à des paramètres différents. Les hybridations hétérogènes, comme la méthode GRASP, utilisent des méta-heuristiques de types différents. Les hybridations générales visent à résoudre un même problème d'optimisation. Au contraire, les hybridations spécialistes regroupent des méthodes dédiées chacune à un problème différent. Les algorithmes impliqués dans des hybridations globales considèrent

tout l'espace décisionnel dans sa totalité. Dans des hybridations partielles, le problème à résoudre est préalablement décomposé en sous-problèmes. Chacune des méthodes de résolution hybridées est associée à l'exploration d'un sous-espace.

Dans la classification hiérarchique, deux niveaux (*Haut* et *Bas*), et deux modes (*relais* et *co-évolutionnaire*) d'hybridation sont considérés permettant d'identifier quatre classes : *hybride bas-niveau relais (HBR)*, *hybride bas-niveau co-évolutionnaire (HBC)*, *Hybride haut-niveau relais (HHR)* et *hybride haut-niveau co-évolutionnaire (HHC)*.

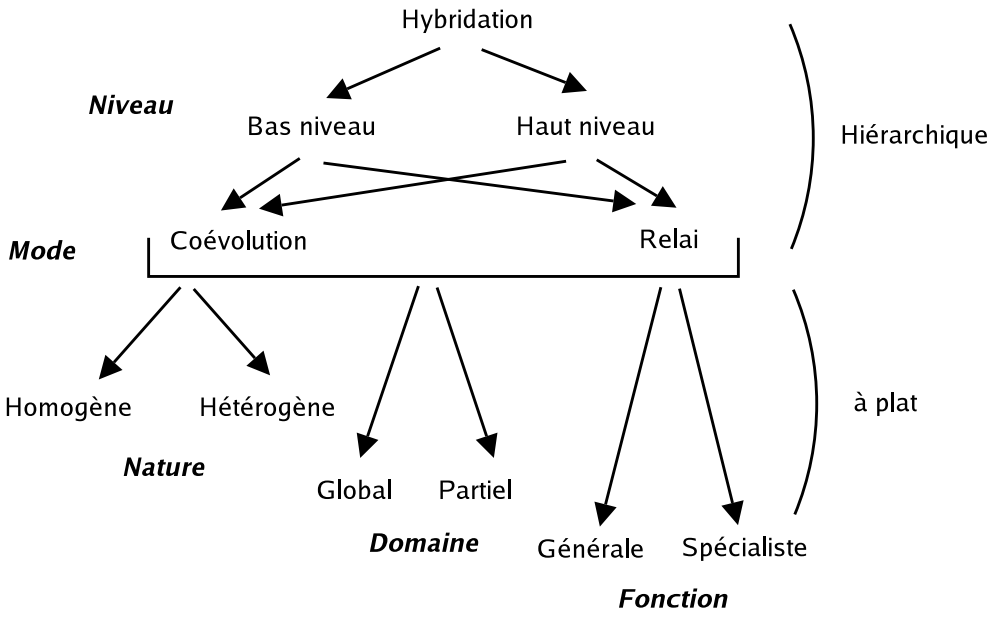

FIG. 4.8 – Classifications *hiérarchique* et *à plat* des schémas d'hybridation de méthodes d'optimisation combinatoire.

4.3.1 Hybride bas-niveau relais (HBR)

La classe des hybrides de bas niveau en mode relais regroupe les méthodes hybrides souvent constituées d'une méta-heuristique à solution unique dans laquelle est insérée une autre méthode de résolution. Il existe très peu de méthodes hybrides appartenant à cette classe. Par exemple, dans [MOF92] une méthode de recuit simulé est combinée avec une méthode de descente. A chaque itération du recuit simulé, la solution localement optimale courante est perturbée pour obtenir la solution initiale de la méthode de descente. Celle-ci remplace la solution localement optimale courante du recuit simulé si elle est meilleure.

4.3.2 Hybride bas-niveau co-évolutionnaire (HBC)

La classe des hybrides de bas niveau en mode co-évolutionnaire regroupe les méthhodes hybrides constituées en général d'une méthode de résolution insérée dans une méta-heuristique à population de solutions. La meilleure illustration de cette classe est la substitution d'un opérateur génétique par une méta-heuristique à solution unique. En effet, un opérateur de recombinaison ou de mutation pourrait être remplacé par une méthode de recherche locale ou une méthode exacte considérant le(s) individu(s) en entrée comme des (une) solution(s) initiale(s). Les nouvelles solutions générées intégreraient ensuite la population de la méta-heuristique à population de solutions. Par exemple, dans [CANT95], un algorithme $B\&B$ explore les solutions potentielles issues de la recombinaison de deux parents afin d'en déduire la plus intéressante.

Ce type d'hybridation transforme des opérateurs de granularité fine en opérateurs de moyenne voire de grosse granularité. La phase de transformation devient ainsi coûteuse en temps d'exécution, ce qui rend l'utilisation du parallélisme inévitable. L'hybridation parallèle peut être assimilée au modèle parallèle multi-départ sans coopération si plusieurs recombinaisons sont appliquées simultanément. Les mêmes considérations peuvent donc être appliquées quant à son implémentation sur un support de type grille de calcul. Les résultats expérimentaux présentés dans le dernier chapitre montrent que, combinées aux méta-heuristiques suivant ce schéma, les méthodes exactes sont nettement plus efficaces.

4.3.3 Hybride haut-niveau relais (HHR)

Dans les hybrides de haut niveau en mode relais, les méthodes de résolution conservent leur intégrité et sont exécutées en séquence. Dans ce schéma d'hybridation, la sortie d'une méthode de résolution est utilisée comme entrée d'une autre méthode de résolution. Par exemple, on peut utiliser un ou plusieurs algorithme(s) glouton(s) afin de générer une bonne solution initiale ou une population initiale de bonnes solutions, exploitée(s) ensuite par une méta-heuristique. On peut également appliquer une méta-heuristique à solution unique (un recuit simulé [MG95] ou une recherche tabou [TMS94]) ou une méthode exacte sur l'ensemble des solutions d'une population obtenue par une méta-heuristique à population de solutions pour améliorer leur qualité. Dans le contexte multi-critère, le front Pareto obtenu par une méta-heuristique (un AG par exemple) est amélioré par une autre méta-heuristique à solution unique (la recherche tabou par exemple) ou une méthode exacte. Le schéma HHR peut être également assimilé au modèle parallèle multi-départ sans coopération. Par conséquent, son étude dans le contexte de grilles de calcul se ramène à l'étude du modèle multi-départ dans le même contexte. Les résultats expérimentaux présentés au dernier chapitre montrent que, combinées aux méta-heuristiques en mode HHR, les méthodes exactes sont nettement plus efficaces.

Un critère majeur ayant un impact considérable sur l'efficacité des algorithmes B&B

est le choix de la valeur initiale de la meilleure solution trouvée. En effet, ce paramètre est fondamental et critique pour les opérations de séparation et du test d'élagage. Le choix judicieux de celui-ci permet de réduire de manière significative la taille de l'espace de recherche exploré. Cependant, la meilleure solution est souvent initialisée à la pire des valeurs i.e. l'infini. C'est pourquoi, nous avons proposé une approche dans laquelle la valeur initiale de la meilleure solution trouvée est fournie à la méthode exacte par une méta-heuristique. Dans nos travaux, la méta-heuristique utilisée est AGMA. L'intérêt de cette approche serait avéré si le coût d'exécution de la méta-heuristique est largement inférieur au coût d'exploration des nœuds élagués grâce à la valeur initiale qu'elle fournit. Autrement dit, le coût cumulé d'exécution de la méta-heuristique et de la méthode exacte est largement inférieur au coût de la même méthode exacte dont la meilleure solution est initialisée à l'infini. Il faudrait donc que la méta-heuristique produise la meilleure solution possible en moins de temps possible. La production de la meilleure solution possible permettra à la méthode exacte d'élaguer le plus de nœuds possibles. La minimisation du temps d'exécution de la méta-heuristique est primordial pour ne pas compromettre l'intérêt de celle-ci. Comme le montre la figure FIG. 4.9, la méta-heuristique et la méthode exacte sont exécutées en séquentiel dans le mode *HHR*.

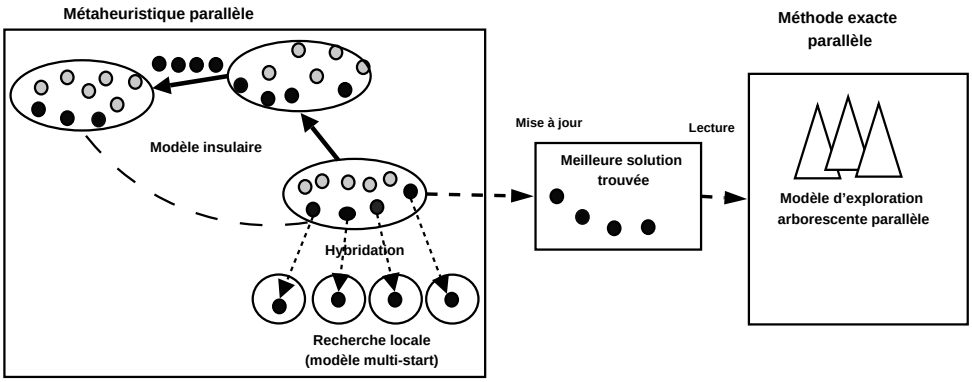

FIG. 4.9 – Coopération parallèle entre méta-heuristiques et méthodes exactes en mode HHR.

4.3.4 Hybride haut-niveau co-évolutionnaire (HHC)

Dans les hybridations de haut niveau co-évolutionnaires, la structure interne des méthodes de résolution hybridées n'est pas modifiée. Ces dernières sont exécutées simultanément et coopèrent pour résoudre un problème donné. Le meilleur exemple illustratif de ce schéma d'hybridation est le modèle insulaire. Dans ce cas, nos travaux dédiés à l'évaluation du schéma HHC montrent que celui-ci permet d'améliorer la robustesse des méta-heuristiques et la qualité des solutions produites. Ces travaux montrent également que l'utilisation de ce schéma pour faire coopérer les méta-heuristiques et les méthodes exactes est source d'efficacité mais pose des problèmes de gestion de la concurrence de

ces méthodes.

Les schémas d'hybridation généralistes sont indépendants du problème traité. Pour assurer leur généricité à l'implémentation, il faut faire en sorte que la sémantique des opérateurs substitués soit identique à celle des substituants. Par exemple, la sémantique d'un opérateur de mutation doit être la même que celle d'une méta-heuristique à solution unique.

Dans nos travaux, nous avons également proposé l'hybridation parallèle des méta-heuristiques et des méthodes exactes selon le mode HHC. La figure FIG. 4.10 illustre ce mode d'hybridation. A l'instar du mode HHR, le rôle de la méta-heuristique dans le mode HHC est également de fournir des solutions Pareto à la méthode exacte. Dans le contexte des grilles, des questions se posent sur la répartition des ressources entre les deux méthodes. Nos expérimentations montrent qu'au départ il est préférable de favoriser la méta-heuristique. Au fil du calcul, de plus en plus de ressources doivent être attribuées à la méthode exacte.

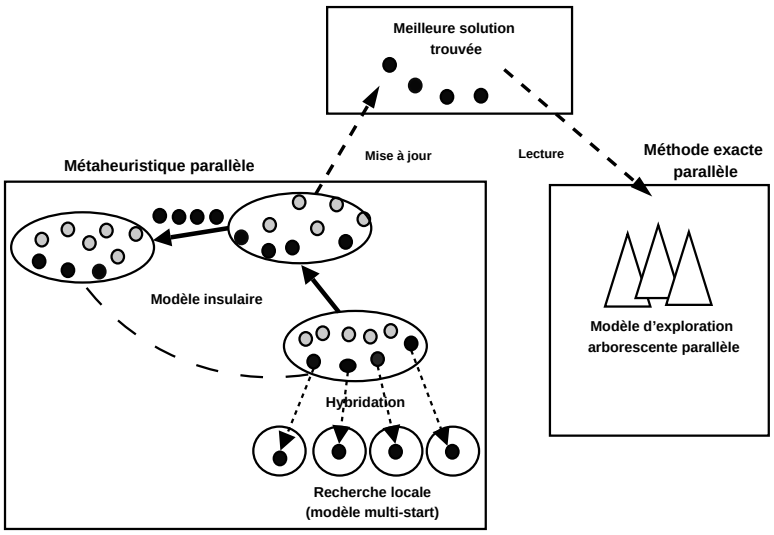

FIG. 4.10 – Coopération parallèle entre méta-heuristiques et méthode exactes avec le mode HHC.

Conclusion

Dans ce chapitre, nous avons proposé une analyse des méthodes d'optimisation combinatoire parallèles hybrides en vue de leur gridification. Pour supporter le coût exorbitant des mécanismes d'hybridation, il est nécessaire de les combiner avec les modèles parallèles. Ainsi, certains schémas d'hybridation, notamment HBC et HHR, devraient être exploités suivant le modèle parallèle multi-départ. Par exemple, sur chacune des

solutions d'une population finale ou du front Pareto produits par un AG, on peut appliquer simultanément une recherche locale. Ensuite, pour chaque modèle parallèle, nous avons défini les données (ou mémoire) caractérisant l'état courant de l'exécution pour la gestion transparente de la volatilité. Cet état permet de minimiser le coût de gestion de la volatilité en évitant de reprendre l'exécution depuis le début à l'instar de ce qui est souvent fourni par les intergiciels.

Dans ce chapitre, nous nous sommes intéressés à l'algorithme AGMA proposé dans [Bas05]. Nous avons proposé une parallélisation de cet algorithme selon deux modèles parallèles : le modèle insulaire et le modèle multi-départ. Le modèle insulaire est exploité pour la parallélisation de l'algorithme génétique d'AGMA, et le modèle multi-départ pour la parallélisation de la recherche locale de l'algorithme mimétique d'AGMA. La granularité du problème combinatoire considéré est fine pour exploiter les autres modèles parallèles. Ensuite, nous avons proposé d'hybrider la méta-heuristique parallèle ainsi obtenue avec un algorithme B&B parallélisé à l'aide de l'approche B&B@Grid. Deux schémas d'hybridation sont proposés : *Hybridation Haut-niveau Relais (HHR)* et *Hybridation Haut-niveau Co-évolutionnaire*. L'objectif est de réduire le temps de résolution de l'algorithme B&B malgré la stratégie *profondeur d'abord* utilisée. Cependant, les intergiciels dédiés au déploiement d'applications sur les grilles de calcul sont limités quant à leur capacité de supporter la coopération et l'hybridation entre différentes méthodes de résolution des problèmes d'optimisation combinatoire. Par conséquent, il est nécessaire de les étendre par une couche de coordination.

Chapitre 5

Modèle de coordination pour les méthodes hybrides sur grilles de calcul

Une version préliminaire du modèle de coordination, présenté dans ce chapitre, est publiée dans la conférence internationale *LNCS EGC'2005* **[MMT05], et une version plus complète et finale est publiée dans la revue internationale** *Parallel Computing* **[MMT06].**

Introduction

L'utilisation d'un intergiciel grille pour implémenter les méthodes hybrides parallèles, telles que celles que nous avons proposées dans le chapitre précédent, permet de résoudre une partie des problèmes rencontrés dans cet environnement. Toutefois, ces intergiciels ne sont pas totalement adaptés au parallélisme coopératif. Par exemple, les intergiciels de type *coordinateur-travailleur* tels que *XtremWeb* [Fed03], ne prennent souvent pas en compte la génération dynamique de tâches et leur coopération. Dans ce chapitre, nous proposons d'étendre ce type d'intergiciels avec un modèle de coopération inspiré de Linda [Gel85] et adapté à l'optimisation multi-critère sur grilles. Ce nouveau modèle vise à être une couche de coordination pour ce type d'intergiciel, et à servir à la mise en œuvre des approches hybrides que nous avons proposé.

Ce chapitre est organisé en trois sections. La **section 5.1** présente le principe du modèle de coordination Linda, ses primitives, et un exemple montrant ses qualités expressives. La **section 5.2** identifie les limites de ce modèle quant à son utilisation dans les grilles de calcul. Cette section présente alors le nouveau modèle de coordination Linda étendu, et décrit l'implémentation de ce modèle sur XtremWeb [Fed03]. La section explique également l'utilisation de Linda étendu pour la mise en œuvre de trois modèles parallèles. Il s'agit des modèles insulaire, multi-départ et d'exploration arbo-

rescente parallèle. Cette section se termine en décrivant la mise en œuvre, à l'aide de Linda étendu, de deux schémas d'hybridation. Dans la **section 5.1**, nous présentons deux expérimentations. En plus de la validation du modèle Linda étendu, ces deux expérimentations visent à démontrer l'intérêt des deux approches hybrides présentées dans le chapitre précédent.

5.1 Le modèle de coordination Linda

Historiquement, Linda [Gel85] est l'un des premiers modèles de coordination proposés dans la littérature. Des extensions de Linda ont été proposées par la suite telles que *JavaSpace*. Le modèle Linda est basé sur le paradigme de communication générative. Dans ce paradigme, l'échange de messages entre les processus se fait via une mémoire partagée associative, dans laquelle un processus expéditeur dépose son message, et de laquelle un processus récepteur le récupère. L'intérêt de ce paradigme est d'assurer une indépendance spatiale et temporelle entre les processus. Autrement dit, chaque processus ignore l'endroit, i.e. la position dans le réseau, et le moment auquel s'exécutent les autres processus.

Dans Linda, la mémoire partagée et un message sont appelés respectivement *espace de tuples* (ET) et *tuple*. L'espace de tuples est une collection de tuples, chaque tuple étant une suite finie et ordonnée de champs typés dont le premier est généralement une étiquette. Chaque champ contient soit une valeur typée, soit un appel de processus appelé *champ processus*. Un tuple qui ne contient que des valeurs typées est appelé *tuple de données*, ou *tuple passif*. Un tuple qui contient au moins un champ processus est appelé *tuple de processus*, ou *tuple actif*. Le résultat de l'exécution d'un processus est affecté au champ-processus correspondant. Dès qu'un tuple de processus termine son exécution, il se transforme en un tuple de données en remplaçant les champs processus par les valeurs de retour de leurs processus respectifs. Contrairement à un tuple de données, qui est une entité passive, un tuple de processus est une entité active qui échange des tuples en générant, lisant et consommant d'autres tuples.
Comme l'illustre la figure FIG. 5.1, l'action des processus sur ET se fait par l'intermédiaire de quatre *primitives*, ou *opérations*, définies dans le modèle.

- *out(T)* est utilisée pour déposer un tuple passif T dans ET.
- *eval(F)* est utilisée pour déposer un tuple actif T dans ET.
- *in(P)* (resp. *rd(P)*) est employée pour prendre (resp. lire), de ET, un tuple quelconque correspondant au *pattern* P. Contrairement à un tuple lu, un tuple pris est supprimé de ET.

Un pattern est lui-même une sorte de tuple passif contenant au moins un champ

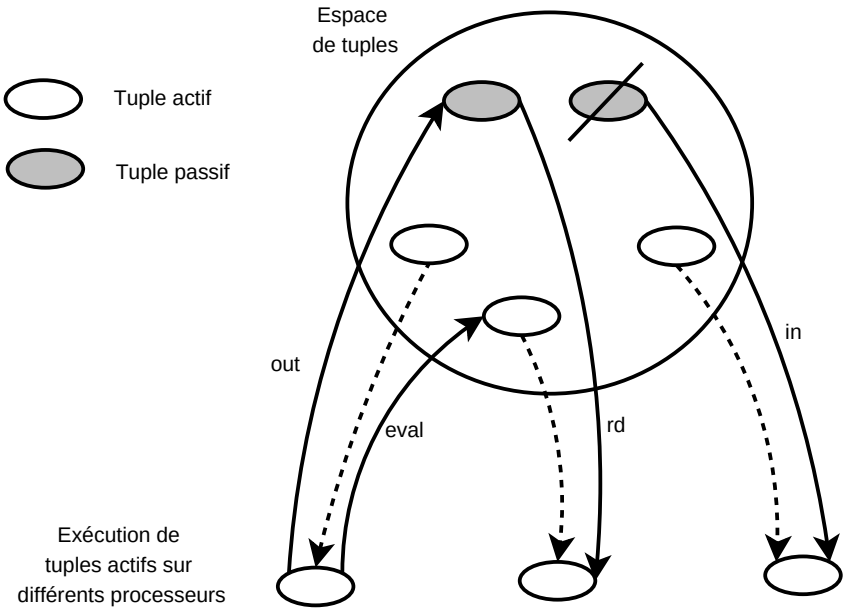

FIG. 5.1 – Illustration du modèle de coordination Linda.

vide qu'on appelle *champ formel*. Par convention, le nom d'un champ formel commence par un point d'interrogation (?). Un pattern P correspond à un tuple T si un champ non formel (respectivement formel) de P et un champ de T ont le même ordre, alors ils ont la même valeur (respectivement sont du même type).

Le pseudo-code de l'algorithme 4 illustre le fonctionnement de *Linda*. Il s'agit de trouver tous les nombres premiers inférieurs à N. Les *N-1* processus lancés, sont synchronisés : si l'entier I est inférieur à J alors le processus $est\_premier(I)$ va finir avant le processus $est\_premier(J)$. Cette solution utilise le fait que i est un nombre premier si et seulement si i n'est pas divisible par j, où j est un nombre premier plus petit que la racine carré de i.

5.2 Le modèle de coordination Linda étendu

L'exemple précédent donne une idée de la capacité expressive du modèle ainsi que de son pouvoir de synchronisation. En effet, quatre primitives suffisent pour gérer la synchronisation et les communication entre processus. Cependant, la nature volatile, hétérogène, et à grande échelle des grilles, nous ont conduit à étendre le modèle avec trois types de primitives qui sont des versions modifiées des primitives de base de *Linda*.

Algorithm 4 Calcul de tous les nombres premiers inférieurs à N avec Linda.

```
// fonction pour vérifier si un nombre est premier
function est_premier(i : entier) : booléen
for j :=2 to sqrt(i) do
  rd("premier",i,?ok)
  if ok et (i mod j=0) then return false
end for
return true ;
end function
// procédure principale
procedure main()
for i :=2 to N do
  eval("premier",i,est_premier(i)) // Lancement de N-1 tuples de processus
end for
for i :=2 to N do
  rd("premier",i,?ok) // Lecture des résultats
end for
end procedure
```

5.2.1 Primitives non bloquantes

Les primitives *rd* et *in* sont des primitives bloquantes, à l'inverse de *eval* et *out*. Autrement dit, après chaque appel, le processus appelant est bloqué tant qu'aucun tuple correspondant au tuple recherché n'est pas trouvé. Le processus appelant peut se bloquer indéfiniment si le processus, censé déposer le tuple recherché, tombe en panne. Les pannes de processus sont relativement fréquentes dans **un environnement volatile** tel que celui des grilles de calcul. Pour cette raison, nous avons repris les primitives non bloquantes, pour *rd* et *in*, proposées dans certaines extensions de Linda [ACG86].

- *inp(P)* (resp. *rdp(P)*) est la primitive non bloquante de *in* (resp. *rd*). Le processus appelant *in* (resp. *rd*) se bloque jusqu'à ce qu'un tuple correspondant au pattern *P* soit trouvé. Par contre, la primitive *inp* (resp. *rdp*) ne se bloque pas et renvoie un tuple NULL en cas d'absence de tuple correspondant au pattern *P*.

5.2.2 Primitives de groupe

Dans les applications parallèles, souvent la même primitive est invoquée plusieurs fois de suite avec des arguments différents. Étant donnés, **les délais de communication**, relativement importants dans les grilles de calcul, une extension de *Linda* par des versions manipulant un groupe de tuples rendrait les applications exploitant le modèle plus performantes. De plus, l'optimisation multi-critère manipule des fronts (groupes) de solutions Pareto contrairement à l'optimisation mono-critère qui s'intéresse à une

seule solution. Ceci nous a donc conduit à définir pour chacune des primitives précédentes une primitive de groupe équivalente.

- *ing(P)* (resp. *rdg(P)*) est la primitive de groupe de *in* (resp. *rd*). La primitive *in* (resp. *rd*) prend (resp. lit) de l'ET un seul tuple correspondant au pattern P, et le renvoie au processus appelant. Par contre, la primitive *ing* (resp. *rdg*) prend (resp. lit) de l'ET tous les tuples dont le pattern correspond à P, et les renvoie au processus appelant.

- *rdpg(P)* (resp. *outpg(P)*) est la primitive non bloquante pour lire (resp. prendre) un groupe de tuples ayant le pattern P.

- *outtg(groupe de tuples)* est la primitive de groupe de *out*. Elle ajoute à ET, en un seul appel, plusieurs tuples passifs.

- *evalg(groupe de tuples)* est la primitive de groupe de *eval*. Elle ajoute à ET, en un seul appel, plusieurs tuples actifs.

5.2.3 Primitives de mise à jour

Les primitives de base de *Linda* sont soit dédiées à consulter (prendre ou lire) ou à ajouter des tuples. Une modification d'un tuple se traduit donc par l'exécution de deux primitives successives : *in* suivi de *out*. Ce qui est, dans une application grille, très **coûteux en temps de communication**, sans oublier tous les problèmes de synchronisation qu'engendrerait **une machine qui tomberait en panne** après un appel de *in* et avant un appel de *out*. Il est donc nécessaire d'enrichir le modèle par des primitives atomiques dédiées à la mise à jour d'un tuple.

- *updateg(P, T)* met à jour tous les tuples dont le pattern correspond à P en leur affectant la valeur T.

- *update(P, T)* met à jour un tuple quelconque dont le pattern correspond à P en lui affectant la valeur T.

5.2.4 Implémentation du modèle et son intégration dans XtremWeb

Le modèle Linda étendu proposé, illustré par la figure FIG. 5.2, pourrait être intégré à n'importe quel intergiciel de type *coordinateur travailleur*. L'implémentation du modèle est composée de deux parties : une interface de coordination et son implémentation au sein des processus travailleurs, et un médiateur de coordination ou un bus logiciel

de requêtes de coordination (ou *BRC*). L'espace des tuples est une partie de la base de données de l'intergiciel, et chaque tuple est un enregistrement de la base.

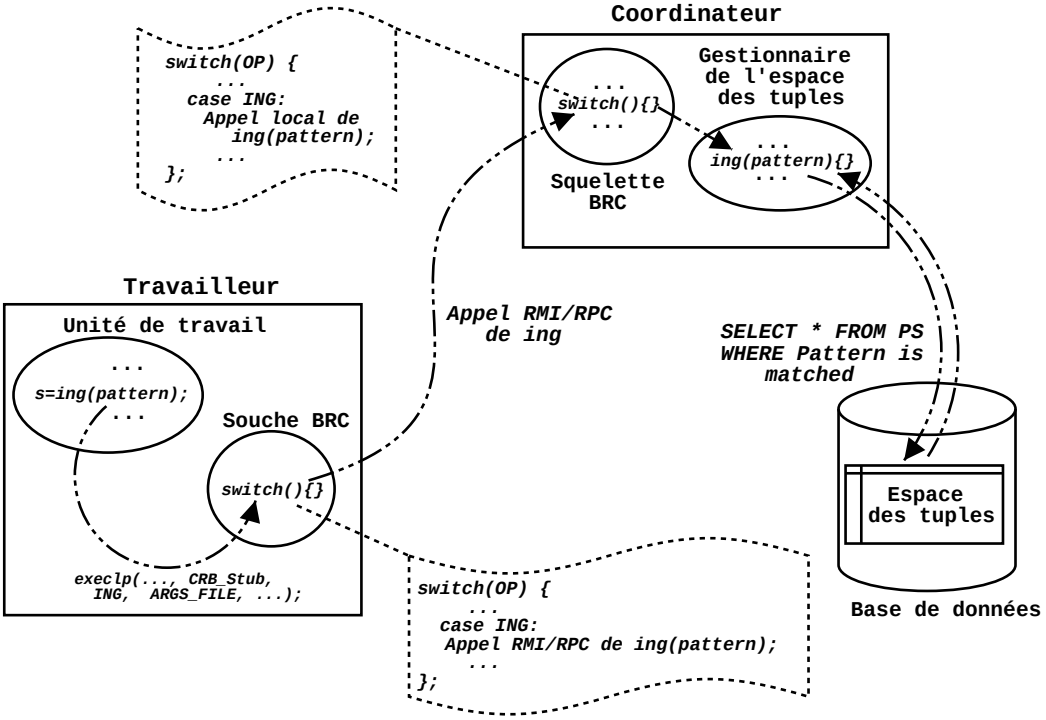

FIG. 5.2 – Intégration du modèle Linda étendu dans un intergiciel de type coordinateur-travailleur.

L'interface de coordination contenant les opérations du modèle Linda étendu est implémentée sous forme de librairie en C/C++ et en Java. Cette librairie doit être incluse dans les applications déployées au niveau des travailleurs. Au niveau du coordinateur, l'interface de coordination est implémentée en Java et C++ comme un gestionnaire de l'espace des tuples. Le BRC permet le transport vers le coordinateur des appels des travailleurs aux opérations de coordination. Il a deux composants : un squelette du côté du coordinateur et une souche du côté des travailleurs. Le rôle de la souche est de transformer en appels RMI ou RPC les appels locaux aux opérations de coordination effectués par les unités de travail exécutées par les travailleurs. Le rôle du squelette est de transformer les appels RMI ou RPC en appels locaux aux opérations de coordination exécutées par le gestionnaire de l'espace des tuples. Ces appels locaux sont traduits en requêtes SQL adressées à l'espace Pareto stocké dans la base de données de l'intergiciel.

XtremWeb [Fed03] est un intergiciel de calcul global développé en Java à l'Université de Paris-Sud. Il s'agit d'un intergiciel de type *coordinateur travailleur*. *XtremWeb* distingue trois rôles pour une machine : client, coordinateur ou travailleur. Le rôle principal du **coordinateur** est d'assurer la mise en relation entre les demandes de ressources de

calcul émanant des clients et les ressources mises à disposition par les travailleurs. Le rôle du **travailleur** est de fournir les ressources de la machine d'un utilisateur pour l'exécution d'un calcul *XtremWeb*. Un **client** soumet une tâche en fournissant la référence de l'application et l'ensemble des paramètres qui la définissent. Le coordinateur utilise une BD MySql pour stocker notamment les résultats des applications et des informations relatives aux travailleurs et au déploiement de ces applications. Le modèle de coordination proposé a été intégré à XtremWeb. La version obtenue a été utilisée pour implémenter et expérimenter des métaheuristiques et méthodes exactes parallèles multi-critères et leur coopération.

5.2.5 Mise en œuvre du modèle insulaire

Comme indiqué dans le chapitre précédent, le modèle insulaire est inspiré de certains comportements observés dans les niches écologiques. Dans ce modèle, plusieurs algorithmes évolutionnaires sont déployés pour faire évoluer simultanément différentes populations de solutions, souvent appelées îles. Ces îles coopèrent par échange de solutions pour améliorer la diversité et la robustesse. Cet échange a pour objectif de retarder la convergence du processus évolutionnaire et ainsi d'explorer davantage de zones dans l'espace des solutions.

Pour chacune des îles, un opérateur de migration intervient au terme de chaque génération. Son rôle consiste notamment à décider de l'opportunité d'opérer une migration, à sélectionner la population source d'immigrants ou destinataire d'émigrants, à choisir les solutions émigrantes et à intégrer les solutions immigrantes. Sa mise en œuvre est basée sur l'utilisation de trois types de tuples : *tuples-insulaires*, *tuples-migrations* et *tuples-pannes*.

Tuples-insulaires : initialement, un programme principal dépose dans l'espace de tuples autant de *tuples-insulaires* que d'îles à déployer. Un *tuple-insulaire* est un tuple de processus constitués d'un seul champ. Ce programme reçoit deux paramètres : le nombre d'îles à déployer et leurs numéros. Une fois déposés dans l'espace de tuples, les *tuples-insulaires* sont déployés par un intergiciel de calcul sur grilles, par exemple *XtremWeb*. En plus du *tuple-insulaire*, deux autres tuples sont associés à chaque île. Ce sont le *tuple-migration* et le *tuple-panne*. Les deux sont des tuples de données et servent respectivement à l'élaboration de la migration dans le modèle insulaire ainsi qu'au mécanisme de tolérance aux pannes.

Tuples-migrations : un *tuple-migration* a la forme [N, MIGRANTS], où N est le numéro d'une île et MIGRANTS contient l'ensemble des solutions émigrantes de cette île. Ces tuples servent à l'importation et l'exportation de migrants entre les îles. L'exportation des migrants se fait en deux étapes : d'abord, sélectionner de l'archive Pareto les solutions à exporter, ensuite les déposer dans le *tuple-migration* associé à l'île. L'importation de migrants se fait en deux étapes également : d'abord, choisir, selon la

topologie de migration, l'île de laquelle les solutions seront importées. Ceci revient à déterminer son numéro, lire le *tuple-migration* portant ce numéro et insérer les migrants dans la population locale selon une certaine stratégie d'intégration.

Tuples-pannes : la tolérance aux pannes peut être mise en œuvre au niveau applicatif ou au niveau de l'intergiciel de calcul sur grilles utilisé. Dans notre approche, les deux niveaux sont utilisés. Au niveau de l'intergiciel choisi, la stratégie adoptée par celui-ci consiste à relancer, avec les mêmes paramètres, sur une autre machine toute île tombée en panne. Au niveau applicatif, la tolérance aux pannes est assurée par un mécanisme de check-pointing manipulant des *tuples-pannes*. A toute île, un seul *tuple-panne* est associé, et il a la structure suivante [N, GENERATION, POPULATION, PARETO]. Les quatre champs désignent respectivement le numéro de l'île, le numéro de sa génération actuelle, sa population courante et son front Pareto. Périodiquement, les différentes îles sauvegardent leur état en mettant à jour les champs du *tuple-panne* qui leur est associé. A son lancement, la première opération d'une île est une tentative de lecture de son *tuple-panne*. L'existence de ce tuple signifie qu'une île de même numéro s'exécutait auparavant, et donc l'île déployée n'est qu'un re-lancement, par l'intergiciel, d'une île tombée en panne. Dans ce cas, le numéro de la génération, la population, et l'archive Pareto de l'île sont mis à jour selon les valeurs du *tuple-panne*. Dans le cas contraire, l'île est lancée avec ses valeurs initiales.

5.2.6 Mise en œuvre du modèle multi-départ

Le modèle parallèle multi-départ consiste à lancer simultanément plusieurs tâches et à fusionner leurs résultats. Ce modèle est exploité pour la parallélisation des recherches locales. Une recherche locale consiste à générer de nouvelles solutions à partir d'une archive de solutions, explorer simultanément les voisinages de ces solutions initiales, fusionner les solutions voisines obtenues avec les solutions initiales, ne conserver que les solutions Pareto optimales, explorer de nouveau les voisinages des solutions Pareto conservées et ainsi de suite. Une recherche locale s'arrête lorsque les solutions voisines n'améliorent plus les solutions initiales. Une seule recherche locale se traduit donc par le lancement d'une série de liste de tâches où chaque tâche est une exploration de voisinage. Le déploiement de chaque liste de tâches est fait selon le modèle *multi-départ*.

Un seul type de tuples est utilisé pour l'élaboration de ce modèle. Ce sont les *tuples-explorations*, i.e. de processus contenant deux champs : le numéro de la recherche locale concernée, et celui de l'appel au programme de l'exploration. Ce programme reçoit, comme arguments, les solutions pour lesquelles les voisinages sont visités, et renvoie, comme résultats, les solutions voisines. Etant donnée la durée relativement courte d'une exploration de voisinage dans nos travaux, aucun mécanisme de tolérance aux pannes au niveau applicatif n'a été mis en œuvre. En cas de panne d'une machine durant un processus d'exploration, l'intergiciel assure son redéploiement sur une autre machine avec les mêmes paramètres.

5.2.7 Mise en œuvre du modèle d'exploration arborescente parallèle

Le modèle d'exploration parallèle consiste à parcourir en parallèle différents nœuds de sous-arbres définissant des sous-espaces de recherche. Cela signifie que les opérateurs de séparation, sélection, évaluation et élagage sont exécutés en parallèle de manière synchrone ou asynchrone par différents processus explorant ces sous-espaces.
Dans la majorité des approches de parallélisation de l'algorithme B&B, l'unité de travail est un ensemble de nœuds. Que ce soit pour l'équilibrage de charge, la tolérance aux pannes, le passage à l'échelle, la gestion de la granularité ou la détection de la fin, cet échange de nœuds induit des limites et des coûts, en communication et stockage, relativement pénalisant sur les grilles de calcul. Dans le but de pallier à ces limites, l'approche présentée au chapitre 2 est utilisée. Trois types de tuples sont utilisés pour le déploiement d'un algorithme B&B selon l'approche présentée : *tuples-B&B*, *tuples-travaux* et *tuples-solutions*.

Tuples-B&B : contrairement aux deux autres tuples qui sont des tuples de données, les *tuples-B&B* sont des tuples de processus. Le déploiement de l'algorithme se fait en déposant autant de *tuples-B&B* que de processus B&B participant au calcul. Comme pour les *tuples-insulaires*, l'intergiciel se charge de les déployer sur les machines de la grille.

Tuples-travaux : les *tuples-travaux* sont associés aux différents intervalles. Un *tuple-travail* a la forme [N,X,Y], où N est le numéro d'un intervalle, X son début et Y sa fin. Au départ, l'espace de tuples est initialisé par un seul *tuple-travail* couvrant la totalité des nœuds de l'arbre. Il correspond à la portée du nœud racine de l'arbre. Il est accordé au premier processus B&B rejoignant le calcul.

Lorsqu'un *tuple-travail* $[N_i,X_i,Y_i]$, exploré par un processus i, se termine, autrement dit X_i est supérieur ou égal à Y_i, le processus i adresse une requête de demande de travail à l'espace de tuples. Ce dernier lui renvoie le plus grand *tuple-travail* non encore alloué s'il existe. Dans le cas contraire, l'espace de tuples applique une opération de division au tuple d'un processus j, idéalement celui correspondant au plus grand intervalle. Sa division donne deux tuples $[N_j,X_j,Z]$ et $[N_i,Z,Y_j]$. Le processus i obtient le deuxième tuple-travail et j garde le premier car il a déja commencé son exploration à partir de X_j. Pour éviter l'allocation d'unités de granularité trop fine, on définit une taille seuil en dessous de laquelle, au lieu d'être divisé, un tuple est dupliqué.

La détection de la fin des traitements se fait naturellement. En effet, au cours des traitements, un tuple peut disparaître si la valeur de son champ X est supérieure ou égale à celle de Y. Ainsi, une fois qu'il ne reste plus de *tuples-travaux* dans l'espace de tuples le programme s'arrête.

En plus de l'équilibrage de charge et de la détection de la terminaison, cette approche facilite également la gestion de la tolérance aux pannes. Périodiquement, chaque processus envoie à l'espace de tuples un compte rendu de l'avancement de l'exploration de son *tuple-travail*. Si [N,X_1,Y_1] et [N,X_2,Y_2] désignent respectivement le même *tuple-travail* avant et en cours d'exploration, l'espace de tuples met à jour son intervalle en appliquant une opération de fusion de tuples qui donne le tuple [N,Max(X_1, X_2),Min(Y_1, Y_2)].

Tuples-solutions : les tuples-solutions sont constitués de deux champs, désignant le code de la solution et le vecteur de ses coûts. Sur ces tuples, une opération de suppression est définie. Un *tuple-solution* disparaît de l'espace de tuples si le champ de son vecteur coût est dominé, au sens Pareto, par le vecteur coût d'un autre *tuple-solution*. Cette opération de suppression assure que seules les solutions Pareto restant dans l'espace de tuples. Toute nouvelle solution Pareto retrouvée par un processus, qu'il soit de type B&B ou insulaire, sera immédiatement déposée dans l'espace de tuples afin qu'elle soit récupérée par les autres processus. Les processus B&B lisent périodiquement l'ensemble des tuples-solutions Pareto pour permettre à l'opérateur d'élagage de s'en servir.

5.2.8 Mise en œuvre de l'hybridation

A l'aide des *tuples-solutions*, les deux modes d'hybridation sont réalisés d'une façon très simple. Pour le mode relais, il suffit de lancer les processus insulaires, de les arrêter lorsque aucune amélioration des solutions Pareto n'est avérée. Les solutions approchées produites servent de solutions de départ (bornes inférieures) à l'algorithme B&B parallèle qui prend le relais pour produire le ou les solution(s) optimale(s). Comme, à la terminaison des processus insulaires, l'espace de tuples n'est pas vidé de ses solutions Pareto, les processus B&B sont initialisés par les solutions Pareto déjà retrouvées. Pour le mode coopératif, les processus insulaires et B&B sont lancés en même temps. Toutefois, lorsque le modèle des îles converge, les processus insulaires sont remplacés par des processus B&B afin d'exploiter pleinement la puissance de calcul de la grille. Comme les deux types de processus partagent le même espace de tuples, les solutions retrouvées par tout processus, qu'il soit de type insulaire ou B&B, sont utilisées par les autres processus.

5.3 Expérimentations et résultats

Deux expérimentations sont présentées dans cette section. Elles visent principalement à - *démontrer l'apport des méta-heuristiques pour réduire le temps de résolution des méthodes exactes*. Le but est donc de - *prouver l'intérêt des deux approches hybrides que nous avons proposées* dans le chapitre précédent. L'autre objectif de ces expérimentations est de - *montrer l'intérêt des grilles de calcul pour les méthodes combinatoires parallèles coopératives*. Ces expérimentations visent également à - *valider le*

modèle Linda étendu et l'implémentation réalisée sur *XtremWeb*.

5.3.1 Coopération parallèle entre méta-heuristiques

En plus de la validation de Linda étendu et de son implémentation, la première expérimentation vise à montrer qu'en utilisant davantage de ressources de calcul, sur une période encore plus longue, le front Pareto obtenu par une méta-heuristique bi-critère parallèle peut être considérablement amélioré.

Dans cette expérimentation, nous avons considéré une instance bi-critère du problème (F/d_i, $permut/C_{max}, \bar{T}$) définie par 200 tâches et 10 machines. En définissant des retards pour chaque machine, cette instance [13], dont le front Pareto n'a jamais été trouvé, étend une autre instance publiée dans [Tai93]. La grille d'expérimentation est constituée de trois grappes de l'Université des Sciences et Technologies de Lille. Au moment de cette expérimentation, les grappes de Grid'5000 n'étaient pas encore opérationnelles.

Dans la première expérimentation, nous avons considéré trois versions d'AGMA notées *Vers.0*, *Vers.1* et *Vers.2*. La version *Vers.0* est un AGMA séquentiel. La version *Vers.2*, déployable sur une machine parallèle, est un AGMA implémenté en MPI. Cette version exploite uniquement le modèle insulaire et n'intègre aucun mécanisme de tolérance aux pannes. La version *Vers.2*, déployable sur une grille de calcul, exploite les deux modèles insulaire et multi-départ, et intègre le mécanisme de tolérance aux pannes décrit dans la section précédente. Il s'agit de l'AGMA parallélisé avec le modèle Linda étendu. Dans la version *Vers.2*, la migration intervient toutes les 10 minutes. Lorsque la taille du front Pareto est inférieure ou égale à 20, la population émigrante est composée de la totalité des solutions du front Pareto. Dans le cas contraire, la population émigrante n'est composée que de 20 solutions choisies aléatoirement parmi les solutions du front Pareto.

Les résolutions effectuées à l'aide de ces trois versions sont notées *Run.0.0*, *Run.1.0*, *Run.2.0*, *Run.2.1*, et *Run.2.2*. La résolution *Run.0.0* est faite avec la version séquentielle *Vers.0*. La résolution *Run.1.0* est celle effectuée dans [Bas05] avec la version MPI *Vers.1*. Dans *Run.1.0*, 8 îles sont déployées pendant 24 heures sur une machine parallèle IBM-SP3. Les trois résolutions *Run.2.0, Run.2.1, et Run.2.2* sont effectuées sur notre grille d'expérimentation avec la version *Ver.2*. Chacune des résolutions a duré plus de 110 heures. Dans *Run.2.0*, une seule île est déployée, 10 îles sont déployées dans *Run.2.1*, et 30 îles dans *Run.2.2*. La résolution *Run.2.0* n'exploite donc que le modèle multi-départ, et les deux autres exploitent les deux modèles parallèles.

Le tableau TAB. 5.1 donne le nombre d'occurrences de chacune des opérations de redémarrage, de recherche locale, d'exploration, de sauvegarde, et de migration enregistrées dans les résolutions *Run.0.0, Run.2.0, Run.2.1, et Run.2.2*. Le nombre d'opérations

[13] http ://www.lifl.fr/OPAC/

Nombre	Run.0.0	Run.2.0	Run.2.1	Run.2.2
Redémarrage	0	13	118	793
Recherche locale	131	282	3224	11,931
Explorations	10,890	23,596	304,011	1,006,793
Sauvegarde & Migration	0	279	3,135	11,280

TAB. 5.1 – Nombre d'occurrences des différentes opérations.

de sauvegarde et de migration est le même. Dans notre implémentation, chaque opération de migration est suivie par une opération de sauvegarde. Le nombre d'opérations de sauvegarde démontre que le mécanisme de tolérance aux pannes a bien fonctionné.

La S-métrique mesure la taille de l'hyperespace compris entre un point de référence de l'espace de coût et les points coûts des solutions Pareto. Plus cette valeur est grande, meilleur est le front Pareto. La figure FIG. 5.3 montre l'évolution de la S-métrique durant les trois résolutions effectuées avec la version *Vers.2*, ainsi que la résolution réalisée avec la version séquentielle *Vers.2* . La figure FIG. 5.4 est un agrandissement de la figure FIG. 5.3 sur les 60 dernières heures. Elle montre la variation de la S-métrique pour les expérimentations *Run.2.1* et *Run.2.2*. Ces deux figures montrent l'intérêt de laisser AGMA tourner plus longtemps, ainsi que celui de l'utilisation davantage de ressources de calcul. En effet, que ce soit pour *Run.2.1* ou *Run.2.2*, même après quasiment cinq jours de résolution, le front Pareto n'a pas encore convergé.

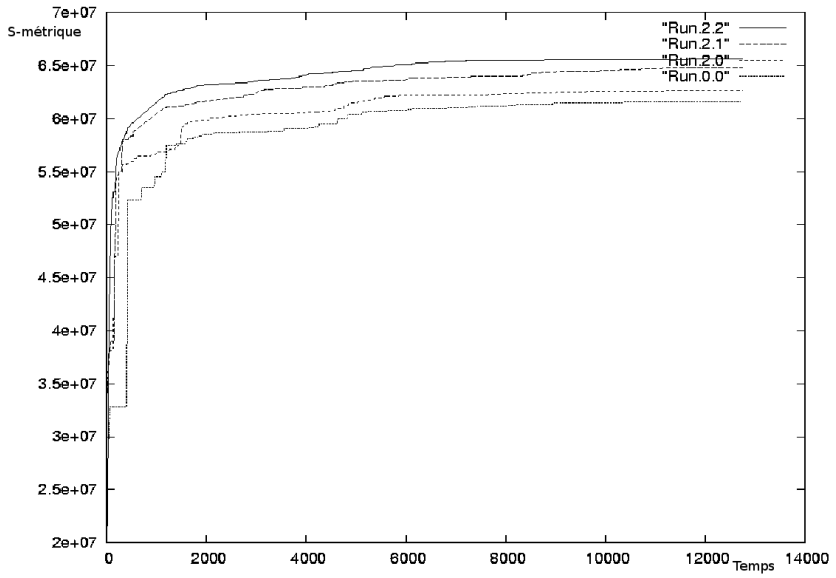

FIG. 5.3 – Evolution de la S-métrique durant les différentes résolutions.

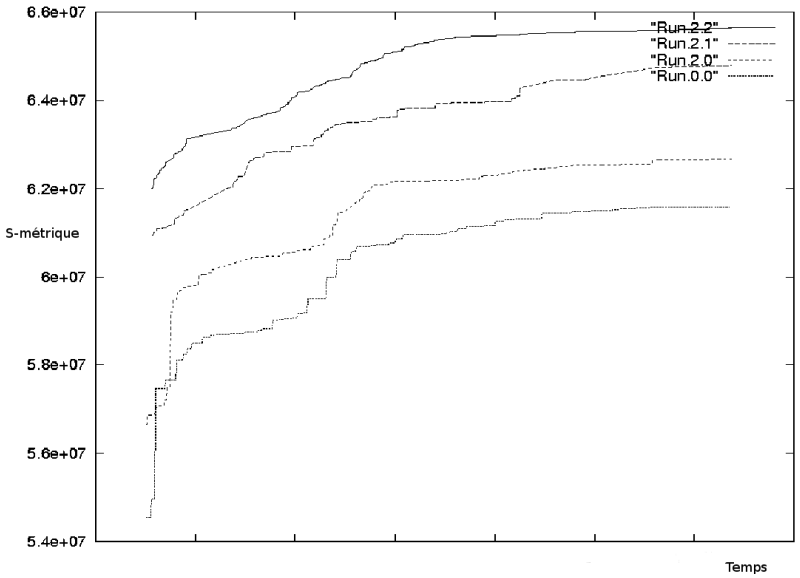

FIG. 5.4 – Evolution de la S-métrique durant les 60 dernières heures.

La figure FIG. 5.5 donne les fronts Pareto obtenus par les différentes résolutions. Cette figure montre notamment que le front Pareto obtenu par la résolution *Run.2.1* de la version *Vers.2* domine et améliore tous les autres fronts. De plus, à tout instant, le front Pareto de la résolution *Run.2.2* est meilleur, au sens de la S-mértrique, que celui de *Run.2.1*. Ce qui démontre que le modèle insulaire n'a pas été encore pleinement exploité, puisque le modèle à 30 îles donne un front meilleur que celui à 10 îles. Comme il est possible de le voir sur la figure FIG. 5.6, *Run.2.2* exploite nettement plus de machines que *Run.2.1*. Il faudrait certainement davantage de puissance de calcul pour pouvoir explorer les limites du modèle insulaire quant à l'amélioration apportée.

5.3.2 Hybridation de méthodes d'optimisation

Dans cette section, les expérimentations concernent les deux méthodes hybrides présentées dans le chapitre précédent et dont la mise en œuvre est décrite dans la section 5.2.8. Il s'agit donc de l'hybridation d'une méthode exacte et d'une méta-heuristique, en l'occurrence AGMA. L'hybridation est faite selon les modes relais et co-évolutionnaire. L'objectif principal des expérimentations présentées dans cette section est de prouver l'intérêt des deux approches hybrides. Le but est donc de montrer l'apport des méta-heuristiques pour réduire le temps de résolution des méthodes exactes. L'autre objectif de ces expérimentations est, d'un part, de montrer l'intérêt des grilles de calcul pour la parallélisation des méthodes hybrides, et, d'autre part, de valider le modèle Linda étendu et son implémentation sur ce type de méthodes.

Nous avons considéré, dans ces expérimentations, une autre instance bi-critère du

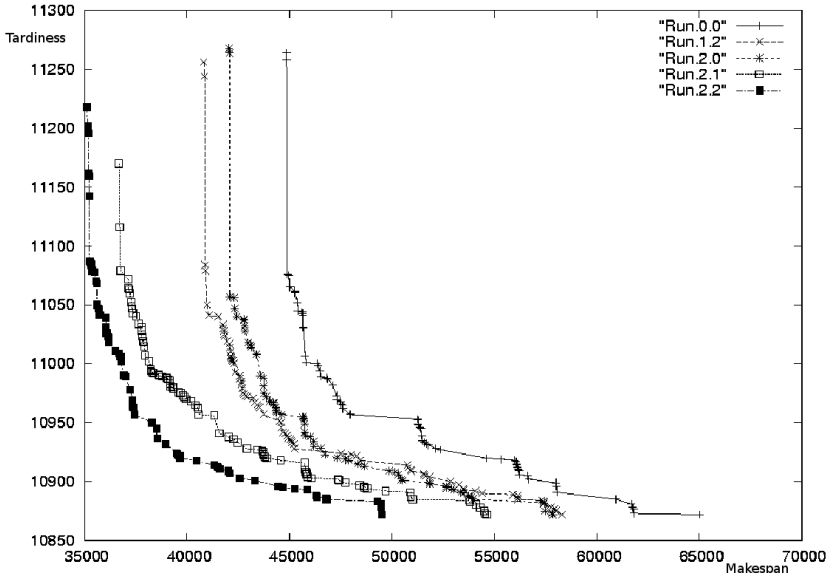

FIG. 5.5 – Les fronts Pareto obtenus dans les différentes résolutions.

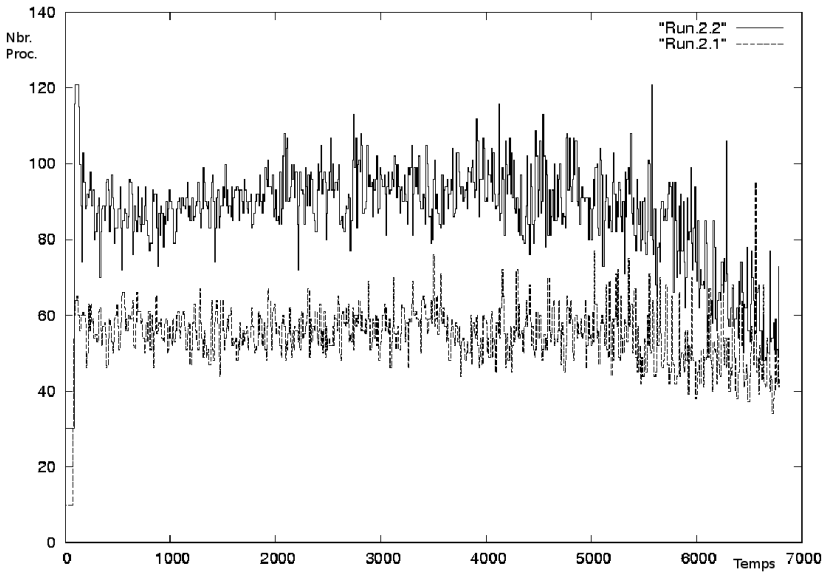

FIG. 5.6 – Evolution du nombre de travailleurs exploités durant Run. 2.1 et Run. 2.2.

problème $(F/d_i, permut/C_{max}, \bar{T})$ définie par 50 tâches et 5 machines. Cette instance[14], dont le front Pareto n'a jamais été trouvé, étend une autre instance publiée dans [Tai93] en définissant des retards pour chaque machine. La grille utilisée dans nos expérimentations est constituée de trois grappes de l'Université des Sciences et Technologies de Lille. Les paramètres d'expérimentation sont fixés de la manière suivante :

- *Modèle insulaire :* la migration et le mécanisme de sauvegarde de l'état d'une île interviennent toutes les 2 minutes. L'échange de migrants se fait selon la topologie aléatoire. La population migrante est l'ensemble du front Pareto, si celui-ci ne contient pas plus de 20 solutions, et seulement 20 solutions sélectionnées, au hasard, toujours du front, dans le cas contraire.

- *Modèle multi-départ :* chaque exploration visite les voisinages de 11 solutions à la fois.

- *Exploration arborescente parallèle :* les processus B&B contactent l'espace de tuples toutes les 3 minutes à la fois pour sauvegarder l'état de l'avancement de leur travail et pour lire les solutions déposées par les autres processus insulaires ou B&B.

Une série de 14 résolutions ont été réalisées sur la grille universitaire. La première résolution est un déploiement de processus B&B seuls sans aucune hybridation. Comme l'indique le tableau TAB. 5.4, le front Pareto exact a été trouvé après plus de 6 jours de calcul. Ensuite, 11 déploiements insulaires, d'une durée d'environ 2 heures chacun, ont été effectués (voir TAB. 5.2 et TAB. 5.3). Ces déploiements diffèrent seulement en nombre d'îles. Ceux-ci visent principalement à estimer, pour la version co-évolutionnaire, la répartition des machines entre les processus insulaires et les processus B&B. Tous les déploiements supérieurs à 60 îles retrouvent le front exact quasiment au même temps. Pour l'instance considérée, la résolution avec 60 îles semble donc donner le meilleur compromis entre le nombre de machines utilisées et la qualité des solutions obtenues. Les autres déploiements n'obtiennent toujours pas le front exact après 2 heures de calcul. Néanmoins, il est intéressant de constater que, sauf pour le cas à 30 îles, plus le nombre d'îles est grand, plus la S-métrique du front Pareto obtenu est meilleure.

Les deux dernières résolutions concernent l'hybridation avec le B&B en mode relais et co-évolutionnaire en déployant 60 îles. Il ressort que l'hybridation en mode relais est plus rapide que celle en mode co-évolutionnaire. Ces deux résolutions montrent la capacité des méta-heuristiques utilisées dans les deux modes à accélérer la méthode exacte. En effet, les deux résolutions parallèles hybrides sont plus rapides que la résolutions d'un B&B parallèle seul.

[14] http ://www.lifl.fr/OPAC/

Nombre d'îles	Temps (secondes)
1	7200
10-50	7200
60	6231
70	6242
80	6244
90	6247
100	6231

TAB. 5.2 – Variation du temps de résolution selon le nombre d'îles.

Nombre d'îles	S-métrique
1	1,086,366
10	1,123,495
20	1,123,602
30	1,123,519
40	1,123,617
50	1,123,617
60-100	1,123,654(Exact)

TAB. 5.3 – Variation de la S-métrique selon le nombre d'îles.

Déploiement	Méta. (60 îles)	B&B	Méta.+B&B
Méta. Seule	1h43	0	1h43
B&B. Seul	0	152h03	152h03
Méta. et B&B en Relais	1h43	116h26	118h09
Méta. et B&B en Co-évolutionnaire	1h44	128h40	128h40

TAB. 5.4 – Durées des différentes résolutions.

Conclusion

Une partie des problèmes rencontrés lors de la gridification des méthodes d'optimisation combinatoire peut être résolue en utilisant un intergiciel grille. Cependant, souvent ces intergiciels ne supportent pas les méthodes parallèles coopératives, telles que les méthodes hybrides parallèles. Ils ne peuvent pas générer dynamiquement des tâches et leur fournir un mécanisme de coopération et de coordination entre elles. Dans ce chapitre, nous avons donc proposé un modèle de coordination Linda étendu dédié à l'optimisation combinatoire multi-critère parallèle sur grilles. Le nouveau modèle peut s'intégrer comme une couche de coordination sur tout intergiciel gille de type

coordinateur-travailleur, et une première implémentation a été réalisée sur *XtremWeb*.

Dans ce chapitre, nous avons également expliqué la mise en œuvre, à l'aide de Linda étendu, de l'approche proposée dans le chapitre précédent. Deux expérimentations sont présentées et les résultats préliminaires ont montré l'efficacité et la robustesse apportée aux méta-heuristiques par les modèles parallèles combinés aux mécanismes d'hybridation. En effet, la méta-heuristique parallèle hybride permet de fournir des fronts Pareto proches des fronts exacts. D'autre part, l'hybridation de la méthode exacte avec la méta-heuristique permet à la deuxième méthode de fournir à la première une solution de départ de nature à réduire de manière importante son temps d'exécution. L'analyse des résultats et du comportement des deux schémas d'hybridation HHR et HHC à l'exécution a soulevé certains problèmes notamment : à quel moment doit être arrêtée la méta-heuristique en mode relais, comment gérer la répartition des ressources entre les deux types de méthodes dans le mode co-évolutionnaire.

Conclusions et perspectives

Dans ce document, nous avons présenté nos contributions sur la gridification des méthodes exactes pour la résolution des problèmes d'optimisation combinatoire mono et multi-critères. La gridification d'une méthode de résolution, telle qu'elle est définie dans nos travaux, signifie la prise en compte des caractéristiques des grilles, notamment l'hétérogénéité, la volatilité, la nature multi-domaine d'administration et la grande échelle, en vue de l'adaptation de cette méthode aux grilles informatiques. Dans ce document, nous avons commencé par une analyse des différents modèles de parallélisation des méthodes exactes dans le contexte des grilles de calcul. Cette étude a mis en évidence l'importance du degré de parallélisme du modèle d'exploration arborescente parallèle, et l'intérêt de son utilisation sur les grilles. En se basant sur ce modèle, nous avons présenté une nouvelle approche, appelée B&B@Grid, pour la gridification des méthodes exactes mono et multi-critères.

L'approche B&B@Grid est basée sur une stratégie de parcours de type profondeur d'abord. Cette approche assigne un numéro à chaque sous-problème de l'arbre exploré, et définit une relation d'équivalence entre le concept d'intervalle de numéros de sous-problème, d'une part, et celui d'ensemble de sous-problèmes, d'autre part. Afin de passer d'un concept à l'autre, B&B@Grid étend les algorithmes B&B par deux opérateurs de pliage et dépliage. L'approche B&B@Grid peut être utilisée pour la gridification des méthodes exactes de type B&B avec différents paradigmes. Dans cette thèse, nous avons également proposé des stratégies de tolérance aux pannes, d'équilibrage de charge, de détection de terminaison, et de partage des solutions trouvées. Toutefois, l'apport le plus significatif de B&B@Grid concerne la stratégie de répartition de la charge entre les processus B&B. Contrairement aux autres approches, cette stratégie est la seule, à notre connaissance, à prendre en compte la nature hétérogène et volatile des grilles de calcul, ainsi que leur échelle.

Dix expérimentations, d'une heure chacune sur le problème du Flow-Shop avec une grille de 1111 processeurs en moyenne, ont été effectuées pour valider B&B@Grid. Les résultats montrent que le codage de B&B@Grid réduit de 766 fois en moyenne la taille des sous-problèmes communiqués. Ces expérimentations ont également montré que le codage des sous-problèmes communiqués, réalisé à l'aide de B&B@Grid, est en moyenne 92 fois plus petit que celui de l'approche à jetons de PICO [EPH00], et 465 fois plus petit que celui de l'approche à variables publiée dans [IF00]. Ces expérimentations prouvent

l'intérêt d'utiliser B&B@Grid dans les grilles pour réduire la taille de l'information communiquée, et par conséquent les délais de communication. Par ailleurs, la sauvegarde de l'ensemble des sous-problèmes en cours d'exploration permet à une stratégie de tolérance aux pannes de relancer une résolution en cas de panne. Les résultats démontrent que B&B@Grid améliore les taux de compression des approches à jetons et à variables, et diminue la taille du codage des sous-problèmes en cours de traitement, dans les mêmes proportions qu'elle le fait pour le codage des sous-problèmes communiqués. Ceci prouve l'efficacité de la stratégie de tolérance aux pannes de B&B@Grid. En outre, malgré le nombre considérable de processeurs de notre grille expérimentale, i. e. 1111 en moyenne, les expérimentations, réalisées avec le paradigme *fermier-travailleur*, ont montré que le fermier exploite son processeur à $1,29\%$ en moyenne, et que les travailleurs passent en moyenne $99,94\%$ de leur temps à résoudre des sous-problèmes. Ces deux taux sont de bons indicateurs sur la qualité de la stratégie d'équilibrage de charge de B&B@Grid et sur sa capacité quant au passage à l'échelle.

Nous avons implémenté cette nouvelle approche sous forme d'une plate-forme appelée aussi B&B@Grid. Contrairement aux autres plates-formes, B&B@Grid est pensée dès le départ pour les grilles, et son objectif est donc de prendre en compte les caractéristiques de ce type d'environnement. La plate-forme B&B@Grid adopte le mode de communication *pull*, qui convient davantage aux environnements où des pare-feux sont présents, résout les problèmes mono et multi-critères, "gridifie" toute méthode exacte de type B&B, et supporte actuellement le paradigme *fermier-travailleur* toute en restant ouverte aux autres paradigmes. La plate-forme a été utilisée pour résoudre une instance connue sous le nom de *Ta056* (50 tâches et 20 machines) et publiée dans [Tai93] n'ayant jamais été résolue de manière optimale. Le défi d'une de nos expérimentations était de résoudre d'une façon exacte cette instance. L'expérimentation a permis de trouver la solution optimale de *Ta056*. En terme de puissance de calcul utilisée, la résolution de *Ta056* se classe en deuxième position parmi les défis à grande échelle réalisés sur les problèmes d'optimisation combinatoire. En moyenne, 328 processeurs ont été utilisés pendant plus de 25 jours, et un pic d'environ 1200 processeurs a été enregistré pendant cette résolution. La première instance bi-critère[15] du problème du Flow-Shop, définie par 50 tâches et 5 machines, a été également résolue avec cette plate-forme. Ces expérimentations valident à la fois la plate-forme et l'approche B&B@Grid proposée sur des problèmes mono et multi-critères.

Nos contributions, dans cette thèse, ont porté également sur la coopération des méthodes exactes avec les méta-heuristiques dans le contexte des grilles informatiques. Nous avons notamment présenté une analyse des méthodes d'optimisation combinatoire parallèles hybrides en vue de leur gridification. L'hybridation de tout B&B parallèle, dont la stratégie d'exploration est de type *profondeur d'abord*, par une méthode approchée, telle qu'une méta-heuristique, permet de réduire le nombre de sous-problèmes traités. Nous avons donc proposé une approche qui combine une méthode exacte avec

[15] http ://www.lifl.fr/OPAC/

une méta-heuristique. Dans une telle méthode hybride, la méta-heuristique fournit à la méthode exacte des solutions de bonne qualité, et lui permet ainsi d'éliminer un grand nombre de sous-problèmes avant de les explorer. L'utilisation d'un intergiciel pour l'implémentation de notre approche hybride sur la grille permet de résoudre une partie des problèmes rencontrés dans cet environnement. Toutefois, les intergiciels de type coordinateur-travailleur, tels que *XtremWeb*, sont limités quant à leur capacité de supporter l'hybridation et la coopération entre méthodes de résolution. En effet, ils ne permettent pas la communication entre les tâches d'une application parallèle. Des primitives non bloquantes, asynchrones, et manipulant des groupes de données (front Pareto), sont indispensables dans un environnement volatile, ayant des délais de communication considérables et s'intéressant à l'optimisation multi-critère. Nous avons donc proposé un modèle de coordination, basé sur Linda [Gel85] et dédié à l'optimisation combinatoire multi-critère sur grilles. Ce modèle peut être utilisé pour rajouter une couche de coordination à tout intergiciel de type *coordinateur-travailleur*. Nous avons fait une première implémentation du modèle de coordination proposé sur *XtremWeb*. Les expérimentations réalisées ont montré la capacité de la méta-heuristique utilisée à réduire le temps de résolution des méthodes exactes.

Le domaine des grilles informatiques est récent, et les perspectives de nos travaux sont donc nombreuses. D'ailleurs, thèse de M. Mahdi, en co-tutelle entre l'Université des Science et Technologie de Lille et l'Université du Luxembourg, démarre cette année en vue notamment de poursuivre les travaux faits sur B&BGrid. Une des principales perspectives de nos travaux est de trouver des approches permettant d'estimer le nombre de sous-problèmes restant à traiter, et d'affiner cette estimation tout au long d'une expérimentation. Au cours d'une résolution, de nombreuses informations utiles sont collectées, notamment le nombre de sous-problèmes traités, leurs positions dans l'arbre exploré, et le nombre de solutions explicitement ou implicitement visitées, l'objectif est d'exploiter ces informations pour estimer la durée d'une résolution. Une telle approche permettrait de poursuivre ou d'arrêter une résolution en fonction de cette estimation. D'autres perspectives concernent notamment l'utilisation et l'étude de B&BGrid avec d'autres (1) paradigmes parallèles comme les paradigmes fermier-travailleur hiérarchique et pair-à-pair, (2) algorithmes de type B&B et (3) problèmes d'optimisation combinatoire tels que le problème Q3AP considéré dans le cadre de la thèse de M. Mehdi.

Le problème du Flow-Shop, à l'instar de nombreux problèmes d'optimisation combinatoire, est un problème de permutation. Pour ce type de problèmes, les formules présentées dans cette thèse permettent de trouver efficacement le *numéro* et le *poids* d'un sous-problème. Par contre, des questions restent posées sur l'utilisation des formules génériques, données dans le deuxième chapitre, pour déterminer le poids et le numéro d'un sous-problème pour d'autres types de problèmes. De ces formules dépend en grande partie l'efficacité de B&B@Grid. L'objectif, en s'intéressant à d'autres problèmes, est de déterminer les caractéristiques des problèmes d'optimisation combinatoire pour lesquels B&B@Grid est efficace.

Dans l'algorithme B&B de base, utilisé dans nos expérimentations, l'information échangée, entre les processus parallèles, est constituée essentiellement de sous-problèmes et beaucoup plus rarement de solutions. Par conséquent, B&B@Grid a permis de réduire considérablement la taille de l'information échangée. Cependant, les processus parallèles, dans d'autres algorithmes B&B, tels que l'algorithme B&C, échangent d'autres types d'information, tel que les plans de coupe. Des travaux utilisant B&B@Grid pour la gridification de ces algorithmes permettraient d'évaluer le taux compression de l'information communiquée, et d'améliorer cette nouvelle approche pour s'adapter davantage à ce type de méthodes.

Aujourd'hui, B&B@Grid a été expérimentée avec le paradigme *fermier-travailleur*. Il serait intéressant d'analyser B&B@Grid avec d'autres paradigmes parallèles, notamment le paradigme pair-à-pair. L'utilisation de Chord [SMLN$^+$03], protocole totalement distribué et qui passe fortement à l'échelle, avec le paradigme pair-à-pair permettrait de proposer une approche qui supporte des grilles de plus grande taille.

Annexe A

Problèmes d'ordonnancement et bornes pour le problème du Flow-Shop

A.1 Problèmes d'ordonnancement

Les travaux de recherche se sont intéressés aux problèmes d'ordonnancement depuis plus d'un demi-siècle. Pour appréhender leur nature, plusieurs définitions sont proposées dans la littérature. De nos jours, la définition donnée dans [Pin95] est l'une des plus admises. Dans [Pin95], un problème d'ordonnancement est défini comme "... l'affectation de ressources limitées aux tâches dans le temps ...". Un ordonnancement est donc une affectation dans le temps d'un ensemble de ressources à des tâches. Comme le montre la figure FIG. A.1, un ordonnancement est une association de trois éléments qui sont les tâches, les ressources et le temps. Une tâche se décompose en une ou plusieurs opérations. Ainsi, une tâche peut être mono-opération ou multi-opération. Les ressources peuvent être renouvelables ou consommables. Les ressources renouvelables redeviennent disponibles après leur utilisation. Tandis que les ressources consommables ne sont utilisables qu'une seule fois. Les ressources renouvelables peuvent être disjonctives ou cumulatives. Contrairement à une ressource disjonctive, une ressource cumulative peut être utilisée par plusieurs opérations à la fois.

Une solution d'un problème d'ordonnancement est un placement des travaux selon la double dimension spatiale et temporelle. La dimension spatiale est représentée par l'ensemble des ressources. Le diagramme de Gantt est souvent utilisé pour visualiser une solution. Ce diagramme, proposé en 1910 par l'ingénieur américain Henry L. Gantt, représente une tâche par un segment ou une barre horizontale dont la longueur est proportionnelle à la durée de l'opération. Comme l'indique la figure FIG. A.2, l'abscisse représente les ressources et l'ordonnée représente le temps.

A l'instar de tout problème d'optimisation, les solutions d'un problème d'ordon-

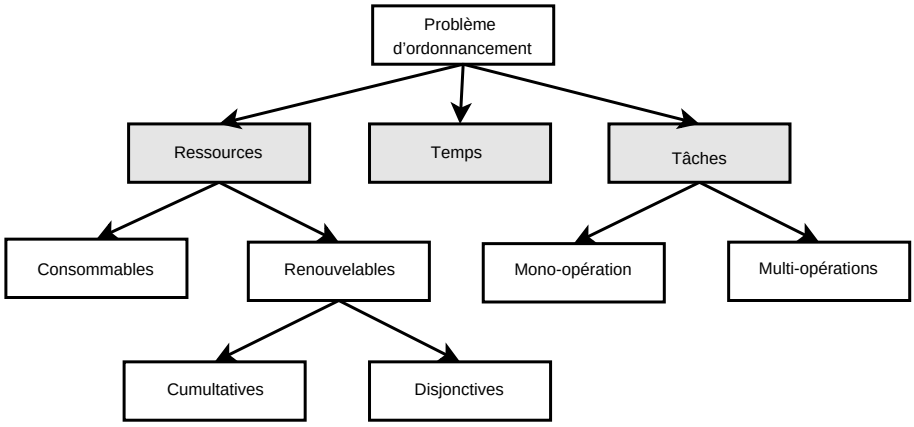

FIG. A.1 – Les éléments d'un problème d'ordonnancement

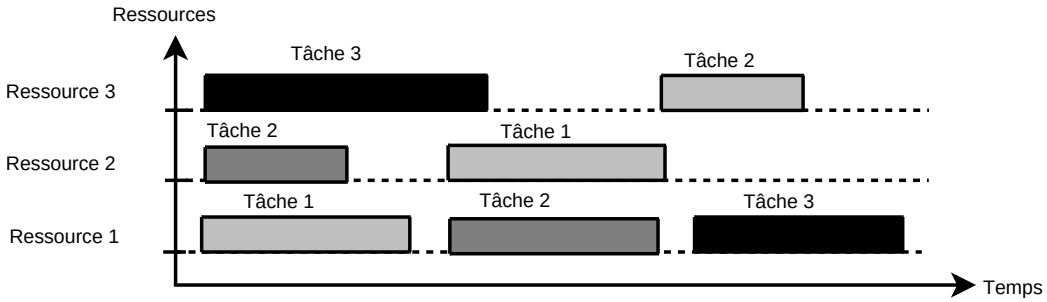

FIG. A.2 – Exemple de diagramme de Gantt

nancement doivent souvent respecter des contraintes, et sont toujours évaluées par un ou plusieurs critères. La frontière entre la notion de critère et celle de contrainte est mince. Une contrainte représente une propriété que tout ordonnancement doit impérativement respecter, tandis qu'un critère autorise une certaine liberté. Les contraintes séparent les solutions réalisables des solutions non réalisables, et les critères distinguent les solutions réalisables entre elles. Résoudre un problème d'ordonnancement consiste à dégager, parmi l'ensemble des solutions possibles, celles qui respectent les contraintes et qui optimisent les critères.

Un grand nombre de problèmes d'ordonnancement sont identifiés dans la littérature. Ces problèmes diffèrent selon leur structure, l'ensemble de leur contraintes et l'ensemble des critères pris en compte. Ces trois éléments sont notés respectivement α, β, et γ. Un problème d'ordonnancement peut donc être noté $\alpha/\beta/\gamma$. Cette façon de distinguer et de noter ces problèmes est présentée pour la première fois dans [GLLK79]. Actuellement, cette approche demeure la plus utilisée pour référencer les problèmes d'ordonnancement. Cependant, d'autres classifications et notations plus anciennes existent notamment celle présentée dans [CMM67]. Dans ce document, un problème d'ordonnancement est noté $(\alpha/\beta/\gamma)$ au lieu de $\alpha/\beta/\gamma$.

A.1.1 Structure d'un problème

La partie α du problème renvoie à la structure et à la typologie du problème. Comme toute typologie, la classification se fait selon plusieurs dimensions. Chacune de ces dimensions admet plusieurs valeurs. L'intersection de ces valeurs forme un espace où chaque point est considéré comme une classe. La dimension la plus importante de cette typologie est la gamme des tâches. Dans un problème d'ordonnancement, la gamme d'une tâche est l'ordre selon lequel elle utilise les ressources. La gamme permet de distinguer, par exemple, le Flow-Shop, le Job-Shop et l'Open-Shop. Pour ces trois problèmes la valeur de α est notée respectivement F, J et O. Dans le Flow-Shop , les tâches doivent avoir la même gamme. Autrement dit, les tâches doivent utiliser les ressources dans le même ordre. Dans le Job-Shop, toutes les tâches ont également une gamme mais qui n'est pas forcément la même d'une tâche à une autre. Contrairement au Flow-Shop et au Job-Shop, un problème de type Open-Shop n'impose aucune gamme aux tâches. En d'autres termes, l'Open-Shop n'exige aucune contrainte pour les tâches sur l'ordre d'utilisation des ressources. Un problème de type mixte, noté X, est également défini. Comme son nom le suggère, certaines tâches ont une gamme tandis que d'autres n'en ont pas. Dans le cas des tâches mono-opérations, le problème d'ordonnancement peut concerner un problème à une machine ou à plusieurs machines parallèles.

A.1.2 Contraintes

La partie β indique l'ensemble des contraintes que doit respecter toute solution. Une contrainte peut être implicite ou explicite. Les contraintes implicites sont définies dans la structure du problème, autrement dit, dans sa partie α. La gamme, par exemple, est une contrainte implicite. Les contraintes explicites peuvent concerner les ressources, les tâches ou les deux à la fois. Parmi les contraintes les plus rencontrées, *permu*, *pmtn*, *prec* et *no-wait* sont souvent citées dans la littérature.

La contrainte *permu* est généralement associée au Flow-Shop. L'association du Flow-Shop et de *permu* définit le Flow-Shop de permutation qui est l'une des variantes du Flow-Shop les plus étudiées. La contrainte *permu* exige que les tâches doivent être affectées aux ressources dans le même ordre. Soient J_1 et J_2 deux tâches qui utilisent les ressources R_1 et R_2. Si la ressource R_1 est utilisée par la tâche J_1 avant la tâche J2, alors la ressource R_2 doit également être utilisée par la tâche J_1 avant la tâche J_2. *pmtn* est une contrainte qui autorise la préemption dans une tâche. Par défaut, les problèmes d'ordonnancement supposent que les opérations ne peuvent être interrompues. L'ajout de la contrainte *pmtn* à l'ensemble β indique que les opérations, qui définissent des tâches, peuvent interrompre leur utilisation d'une ressource et reprendre plus tard. La contrainte *prec* ne concerne que les tâches. Cette contrainte définit des relations de précédence entre tâches. Ainsi pour chaque tâche J_i, la contrainte précise, par exemple, l'ensemble des tâches qui doivent finir avant que la tâche J_i puisse utiliser les ressources. Une structure arborescente est souvent utilisée pour définir cette contrainte. La contrainte *no-wait* ne concerne également que les tâches, ou plus précisément que l'ensemble des opérations d'une tâche. Cette contrainte impose que les opérations d'une même tâche se succèdent sans aucune attente.

Il existe également des contraintes purement temporelles. A titre d'exemple, il est possible de mentionner celles relatives aux dates de début et de fin des tâches. La contrainte sur les dates de début, notée r_i, précise pour chaque tâche la date à partir de laquelle elle peut utiliser les ressources. Les dates de fin peuvent être une contrainte impérative ou souhaitée. Une date impérative doit être respectée. Dans le cas contraire, la solution obtenue est considérée non réalisable. Lorsque il s'agit d'une date souhaitée, les tâches peuvent utiliser les ressources au-delà de ces dates. Cependant, les solutions trouvées sont pénalisées par des critères définis dans la partie γ. Les contraintes sur les dates de fin souhaitées et impératives sont notées respectivement d_i et $\tilde{d}_i$.

A.1.3 Critères

La partie γ définit l'ensemble des critères d'optimisation pris en compte dans un problème d'ordonnancement. Cet ensemble peut éventuellement être vide. Dans ce cas, il s'agit d'un problème d'existence ou de décision. L'objectif est seulement d'identifier les solutions satisfaisant les contraintes. Certains critères, utilisés dans les problèmes

d'ordonnancement, sont équivalents. Optimiser selon l'un ou l'autre conduit aux mêmes solutions optimales, même si le coût d'une même solution peut être différent d'un critère à un autre.

Les critères, rencontrés dans les problèmes d'ordonnancement, se répartissent en deux familles. Ce sont les critères de type *minimax* et ceux de type *minisum*. Les critères *minimax* cherche la valeur maximale d'une ou de plusieurs fonctions à minimiser, tandis que les critères *minisum* cherche la valeur minimale de la somme de plusieurs fonctions à minimiser. Les critères *minimax* sont également appelés critères de type maximum, et les critères *minisum* sont appelés critères de type somme.

Le plus classique des critères de type maximum est le makespan. Ce critère, noté C_{max}, mesure la date de fin de l'ensemble des tâches. En plus du makespan, deux critères relatifs au plus grand retard d'une tâche sont souvent étudiés. Il s'agit du plus grand retard algébrique d'une tâche, et de son plus grand vrai retard. Dans un retard algébrique, la valeur d'une tâche qui se termine en avance est négative. Dans un vrai retard, le retard d'une tâche est toujours une valeur positive ou nulle. Le plus grand retard algébrique est noté L_{max}, tandis que le plus grand vrai retard est noté T_{max}. Au lieu de s'intéresser au retard des tâches, il est possible également de considérer leurs avances. Ainsi, un critère sur la plus grande avance d'une tâche est également défini. Ce critère est noté E_{max}. Tous ces critères portent sur les tâches. D'autres critères de type *minimax* sont définis sur les ressources. Le plus grand temps d'inactivité d'une ressource, noté I_{max}, est certainement l'un des plus étudiés. Soient J l'ensemble des tâches d'un problème d'ordonnancement, et S l'ensemble de ses solutions possibles. Ces critères *minimax* peuvent être définis de la façon suivante :

$$C_{max} = \max_{j \in J}(C_{j,s}) \tag{A.1}$$

Où $C_{j,s}$ est le temps de fin de la tâche j dans un ordonnancement s.

$$L_{max} = \max_{j \in J}(L_{j,s}) \tag{A.2}$$

Où $L_{j,s}$ est le retard algébrique de la tâche j dans un ordonnancement s.

$$T_{max} = \max_{j \in J}(T_{j,s}) \tag{A.3}$$

Où $T_{j,s}$ est le vrai retard de la tâche j dans un ordonnancement s.

$$E_{max} = \max_{j \in J}(E_{j,s}) \tag{A.4}$$

Où $E_{j,s}$ est l'avance de la tâche j dans un ordonnancement s.

$$I_{max} = \max_{j \in J}(I_{j,s}) \tag{A.5}$$

Où $I_{j,s}$ est le temps d'inactivité de la tâche j dans un ordonnancement s.

Les critères de type somme sont généralement plus difficiles à optimiser que les critères de type maximum. Dans [Ehr97], ce constat est confirmé avec une analyse théorique sur certains problèmes. Le retard total des tâches, leur temps de traitement total ou leur avance totale sont trois critères de somme souvent rencontrés. Ils sont notés respectivement $\bar{T}$, $\bar{C}$ et $\bar{E}$.

$$\bar{T} = \sum_{j \in J}(T_{j,s}) \tag{A.6}$$

Où $T_{j,s}$ est le retard de la tâche j dans un ordonnancement s.

$$\bar{C} = \sum_{j \in J}(C_{j,s}) \tag{A.7}$$

Où $C_{j,s}$ est le temps de traitement de la tâche j dans un ordonnancement s.

$$\bar{E} = \sum_{j \in J}(E_{j,s}) \tag{A.8}$$

Où $E_{j,s}$ est l'avance de la tâche j dans un ordonnancement s.

A.2 Borne inférieure pour le problème $(F/permut/C_{max})$

Dans [LLK78], une borne inférieure est proposée pour $(F/permut/\ C_{max})$ qui est un problème de Flow-Shop de permutation où l'objectif est de minimiser le makespan. Ce problème est décrit en donnant tous les $P(j,m)$ associés à tous les couples de machine

M_m et de tâche J_j. $P(j,m)$ est le temps de traitement de la tâche J_j par la machine M_m.

La borne proposée dans [LLK78] date de la fin des années soixante-dix, et reste toujours la plus utilisée. Son principe est basée sur une réduction de $(F/permut/C_{max})$ en $(F2|l_j|C_{max})$. $(F2|l_j|C_{max})$ est un problème de Flow-Shop de permutation à deux machines avec lags. Dans ce problème, un lag $L(j)$ est associé à chaque tâche J_j. $L(j)$ est le temps minimal qui doit s'écouler entre la fin de la tâche J_j sur la machine M_1 et son début sur la machine M_2. Ce temps peut être éventuellement négatif.

La réduction de $(F/permut/C_{max})$ en $(F2/l_j/C_{max})$ se fait suivant un couple de machines M_l et M_k où l'indice l est supposé strictement inférieur à celui de k. $(F2|l_j|C_{max})$ est défini en donnant pour chaque tâche J_j son temps d'exécution sur la machine M_1, son temps d'exécution sur la machine M_2 et le laps de temps que la tâche doit respecter entre les machines M_1 et M_2. Ces trois valeurs sont notées respectivement $P'(j,1)$, $P'(j,2)$ et $L'(j)$. Elles sont définies de la façon suivante :

$$P'(j,1) = P(j,l) \tag{A.9}$$

$$\begin{array}{rl} & L'(j) = S(j,l,k) \\ avec & S(j,l,k) = \sum_{m=l+1}^{m=k-1} P(j,m) \end{array} \tag{A.10}$$

$$P'(j,2) = P(j,k) \tag{A.11}$$

Soient P une instance du problème $(F/permut/C_{max})$, et $P'(l,k)$ l'instance du problème $(F2|l_j|C_{max})$ obtenue après réduction selon les machines M_l et M_k. Le makespan de $P'(l,k)$, noté $C_{max}(P'(l,k))$, est toujours inférieur au makespan de P, noté $C_{max}(P)$. Dans ce qui suit, le makespan d'une instance d'un problème désigne le makespan de sa solution optimale. Ainsi, la formule (A.12) est toujours vérifiée :

$$\forall (k,l),\ \ 0 < k < l < J,\ \ C_{max}(P) \geq C_{max}(P'(k,l)) \tag{A.12}$$

Pour tout couple de machines M_l et M_k, $C_{max}(P'(k,l))$ constitue donc une borne inférieure de $C_{max}(P)$. Il est possible d'améliorer ces bornes pour s'approcher davantage de $C_{max}(P)$. En effet, la première tâche d'une solution ne peut démarrer sur la machine M_l avant $min_{0<j<J} S(j,0,k)$ unités de temps. De même, la dernière tâche ne peut se terminer avant $min_{0<j<J} S(j,l,J)$ unités de temps. De plus, une tâche ne peut se trouver à la fois au début et à la fin d'une solution, puisqu'une tâche ne figure qu'une et une seule fois. Ainsi, la valeur de $C_{max}(P)$ peut être bornée de la façon suivante :

$$\begin{array}{rl} & C_{max}(P) \geq \max_{1 \leq k < l \leq M} (C_{max}(P'(l,k)) + Q(l,k)) \\ avec & Q(k,l) = min_{i \neq j} [S(i,0,k) + S(j,l,J)] \end{array} \tag{A.13}$$

Les travaux de [Jac56, Mit59] montrent que le problème $(F2|l_j|C_{max})$ est équivalent au problème $(F2||C_{max})$. $(F2||C_{max})$ est un Flow-Shop avec seulement deux machines

et sans lags. En effet, $(F2|l_j|C_{max})$ peut être transformé en un problème $(F2||C_{max})$ qui lui est équivalent. La transformation s'opère selon les formules (A.14) et (A.15). Dans les formules (A.14) et (A.15), $P"(j,1)$ et $P"(j,2)$ donnent respectivement les temps de traitement de J_j sur les machines M_1 et M_2.

$$P"(j,1) = P'(j,l) + L(j) \tag{A.14}$$

$$P"(j,2) = P'(j,2) + L(j) \tag{A.15}$$

Soit $P"(l,k)$ l'instance du problème $(F2||C_{max})$ obtenue après transformation de l'instance $P'(l,k)$. Comme l'indique la formule (A.16), le makespan de $P"(l,k)$, noté $C_{max}(P"(l,k)$, est toujours égal à celui de $P'(l,k)$.

$$C_{max}(P"(k,l)) = C_{max}(P'(k,l)) \tag{A.16}$$

Dans [Joh54], un algorithme polynomial est donné pour retrouver la solution optimale du problème $(F2||C_{max})$. Ce article introduit une relation d'ordre, notée $\leq$, entre les tâches de $(F2||C_{max})$. La formule (A.17) définit cette relation.

$$J_i \leq J_j \Longleftrightarrow min(P"(i,1), P"(j,2)) \leq min(P"(i,2), P"(j,1)) \tag{A.17}$$

[LLK78] énonce un théorème sur la solution optimale des problèmes de type $(F2|$ $permut|C_{max})$. Selon ce théorème, si $J_i \leq J_j$ alors la tâche J_i doit se trouver, dans une solution optimale, avant la tâche J_j. Pour retrouver une solution optimale, il suffit donc de trier les tâches. Ceci peut se faire avec un quelconque algorithme de trie en $O(J\ log(J))$ où J est le nombre de tâches. Pour ordonner les tâches, l'algorithme de trie utilisé doit se baser sur la relation d'ordre émise dans [Joh54]. Ainsi, il est donc possible de calculer $C_{max}(P"(l,k))$ en temps polynomial pour toute instance $P"(l,k)$. La borne de [LLK78] peut donc être formulée de la façon suivante :

$$\max_{1 \leq k < l \leq M} (C_{max}(P''(l,k)) + Q(l,k)) \leq C_{max}(P) \tag{A.18}$$

A.3 Borne inférieure pour le problème $(F/d_i, permut/\bar{T})$

$(F/d_i, permut/\bar{T})$ est un Flow-Shop de permutation où l'objectif est d'optimiser le retard total. $(F/d_i, permut/\bar{T})$ est défini en précisant, pour chaque tâche j et pour chaque machine m, les valeurs de $R(m)$, $d(j)$ et $P(j,m)$. Ces trois valeurs sont définies comme suit :

- $R(m)$: date de début de disponibilité d'une machine M_m. $R(m)$ est généralement supposée égale à 0

- $d(j)$: date due d'une tâche J_j.
- $P(j,m)$: temps d'exécution d'une tâche J_j sur une machine M_m.

Cette section présente deux bornes pour le problème $(F/d_i, permut/\bar{T})$. Les deux bornes se basent sur la réduction du problème $(F/d_i, permut/\bar{T})$ en problème à une machine $(1/d_i/\bar{T})$. Les problèmes $(1/d_i/\bar{T})$ et $(1/d_i, permut/\bar{T})$ sont équivalents. En effet, le Flow-Shop est équivalent au Flow-Shop de permutation lorsqu'il est défini avec une seule machine. Dans $(1/d_i/\bar{T})$, l'objectif est de minimiser le retard total. La réduction peut se faire selon un certain couple de machines M_k et M_l. La valeur de k est supposée strictement inférieure à celle de l. Le problème $(1/d_i/\bar{T})$ obtenu est défini en précisant, pour chaque tâche j, les valeurs de $R'(1)$, $d'(j)$ et $P'(j,1)$. Ces valeurs ont respectivement la même sémantique que $R(1)$, $d(j)$ et $P(j,1)$. Elles sont déduites de $(F/d_i, permut/\bar{T})$ à l'aide des formules (A.19), (A.20) et (A.21).

$$R'(1) = R(k) \tag{A.19}$$

$$d'(j) = d(i) - \sum_{m=l+1}^{M} P(m,j) \tag{A.20}$$

$$\begin{array}{c} P'(1,j) = P(k,j) + Cste \\ Avec \quad Cste = \min_{1\leq i\leq J}(\sum_{m=k+1}^{l} p(m,i)) \end{array} \tag{A.21}$$

Soient P une instance du problème $(F/d_i, permut/\bar{T})$ original, et $P'(k,l)$ l'instance du problème $(1/d_i/\bar{T})$ obtenue après une réduction selon les machines M_k et M_l. Le retard total de la solution optimale de P, noté $\bar{T}(P)$, est toujours supérieur ou égal au retard total de la solution optimale de $P'(k,l)$, noté $\bar{T}(P(k,l))$. Par conséquent, pour tout problème P la règle suivante est toujours vérifiée :

$$\max_{1\leq k<l\leq M}(\bar{T}(P(k,l))) \leq \bar{T}(P) \tag{A.22}$$

[DL90a] démontre que le problème $(1/d_i/\bar{T})$ est NP-difficile si les dates de fin ne sont pas toutes dépassées. [Kim95, Lem06] donnent deux bornes inférieures des solutions de ce problème. L'idée de [Kim95] se base sur une séparation des dates dues et des temps d'exécution. La j^{eme} plus petite date due est donc affectée au j^{eme} plus petit temps d'exécution. Ainsi, le nouveau problème $(1/d_i/\bar{T})$ obtenu peut être formulé de la façon suivante :

- $C1(1) = R'(1)$
- $D1(j)$= la j^{eme} plus petite date due.
- $P1(1,j)$= le j^{eme} plus petit temps d'exécution.

La solution optimale du nouveau problème $P'(k,l)$ obtenu est toujours inférieure à celle du problème $P1(k,l)$ original. Il suffit, pour obtenir la solution optimale de $P1(k,l)$, d'ordonnancer les tâches par ordre croissant de leur temps d'exécution. Le retard total

ainsi obtenu est toujours inférieur au retard total de la solution optimale. La borne proposée dans [Kim95] peut être formulée à l'aide de la formule (A.23).

$$\begin{array}{rl} & \bar{T}(P1(k,l)) = \sum_{j=1}^{j=J}(\max(F1(j) - D1(j), 0)) \\ Avec & F1(j) = C1(1) + \sum_{i=1}^{i=j} P1(1,i) \end{array} \quad (A.23)$$

Dans [LDT07, Lem06], une deuxième borne est donnée pour délimiter le retard total de la solution optimale. Cette borne est proposée par Julien Lemesre, dans le cadre de sa thèse. Le calcul de cette borne est basé également sur une transformation du problème $P'(k,l)$ en problème $P2(k,l)$. $P2(k,l)$ se contente seulement de changer la numérotation des tâches de $P'(k,l)$. $P2(k,l)$ numérote les tâches selon l'ordre croissant des dates dues. Ainsi, $P'(k,l)$ est transformé en $P2(k,l)$ de la façon suivante :

- $C2(1) = R'(1)$
- $D2(j)$= la j^{eme} plus petite date due.
- $P2(1,j)$= le j^{eme} plus petit temps d'exécution.

Contrairement à la borne de [Kim95], la borne de [Lem06] ne sépare donc pas les dates dues et les temps d'exécution. Pour que le retard total obtenu soit toujours inférieur au retard de la solution optimale, la borne de [Lem06] impose que le retard d'une tâche ne peut être supérieur à son temps d'exécution. La borne proposée dans [Lem06] peut être formulée avec la formule (A.24).

$$\begin{array}{rl} & \bar{T}(P2(k,l)) \geq \sum_{j=1}^{j=J}(\min(F2(j), P(j,1))) \\ Avec & F2(j) = C1(1) + \sum_{i=1}^{i=j} P1(1,i) \end{array} \quad (A.24)$$

Des tests, effectués et publiés dans [Lem06], démontrent qu'aucune des deux bornes ne donne toujours de meilleurs résultats que l'autre. Les deux bornes peuvent donc être utilisées en ne retenant que le plus grand retard total. Ainsi, le retard de la solution optimale de P'(k,l) peut être borné à l'aide de la formule (A.25) :

$$\bar{T}(P'(k,l)) > \max(\bar{T}(P1(k,l), \bar{T}(P2(k,l)) \quad (A.25)$$

La formule (A.26) donne donc une borne du retard total du problème $(F/d_i, permut/\bar{T})$.

$$\bar{T}(P) > \max_{1 \leq k < l \leq M}(\max(\bar{T}(P1(k,l), \bar{T}(P2(k,l))) \quad (A.26)$$

Annexe B

Utilisation de B&B@Grid

B&B@Grid est une plate-forme *white box* logicielle *open source*. Une fois adaptée au problème à traiter et compilée, la plate-forme se présente sous la forme d'un seul exécutable *bbGRID*. Le lancement de B&B@Grid se fait à l'aide de la ligne de commande suivante :

bbGRID $-t$ *type* [$-c$ *coordinator*] [$-m$ *monitor*] [$-s$ *solutions*] [$-i$ *intervals*] [$-p$ *port*] [$-t$ *period*] [$-f$ *fault*]

L'option *type* indique le rôle du processus *bbGRID*. La valeur de *type* peut être égale à w, c ou m. Ces trois lettres indiquent à un processus *bbGRID* de se lancer respectivement comme travailleur, coordinateur ou moniteur. De toutes les options de *bbGRID*, *type* est le seul qui est obligatoire. La présence des autres options dépend de la valeur de *type*.

L'argument *port*, qui est facultatif, donne le port sur lequel un processus *bbGRID* se met à l'écoute pour recevoir les requêtes des autres processus et d'y répondre, sa valeur par défaut est 56789. Préciser un numéro de port est indispensable seulement si plusieurs processus *bbGRID* sont lancés sur une même machine. Sur une machine bi-processeur, par exemple, il est plus intéressent de lancer deux processus *bbGRID* de type travailleur. Les options *coordinator* et *monitor* donnent respectivement le nom du hôte coordinateur, et du hôte moniteur.

Les options *solutions* et *intervals* ne peuvent être précisées que lors du lancement d'un processus de type coordinateur. Ces deux arguments donnent respectivement le nom du fichier des solutions et celui des intervalles. Le fichier des solutions contient les solutions par lesquelles une résolution est initialisée. Le fichier des intervalles contient, quant à lui, les intervalles de numéros qui doivent être explorés. Si le fichier des intervalles n'existe pas, tous les noeuds de l'arbre doivent être explorés, i. e. l'intervalle global. Si le fichier des solutions n'existe pas, la résolution n'est initialisée par aucune solution. Lors d'une résolution, le processus coordinateur modifie le contenu de ces deux

fichiers. En effet, il sauvegarde régulièrement les intervalles non encore explorés dans le fichier des intervalles. Et le fichier des solutions est actualisé immédiatement à chaque fois que de nouvelles solutions sont trouvées. En cas de panne, il suffit de relancer le processus coordinateur en lui passant comme arguments les noms des deux fichiers des solutions et des intervalles.

Les deux dernières options utilisées sont *period* et $fault$. Ils sont facultatifs et correspondent à des valeurs temporelles exprimées en secondes. Si leur valeur n'est pas indiquée, les processus *bbGRID* prennent des valeurs par défaut. Ces valeurs sont de 3 minutes pour *period* et de 30 minutes pour $fault$. Les deux valeurs par défaut sont fixées d'une façon arbitraire. La valeur de *period* indique, pour les processus travailleurs, la période de mise à jour de leur intervalle. Comme indiqué auparavant, les processus travailleurs doivent régulièrement contacter le processus coordinateur. Il est donc indispensable de préciser, au lancement de ces processus, cette période de mise à jour de leur intervalle. L'option $faut$ peut être utilisée seulement au lancement d'un processus coordinateur. Il indique à celui-ci la période de mise à jour du fichier d'intervalles. Comme indiqué auparavant, le fichier des intervalles doit être mis à jour régulièrement par les intervalles non encore explorés. Plus cette période est petite, moins le travail refait en cas de panne est minime. Cependant, une petite période augmenteraient la charge du processus coordinateur. Par conséquent, une petite période augmenteraient le risque du goulot d'étranglement et limiterait le passage à l'échelle. Trois schémas de lancement d'un processus B&B@Grid sont donc possibles. Il s'agit du lancement d'un coordinateur, d'un moniteur et d'un processus travailleur.

- $bbGRID\ -t\ w\ -c\ coordinator\ -m\ monitor\ [-p\ port]\ [-t\ period]$
- $bbGRID\ -t\ I\ -m\ monitor\ -s\ solutions\ -i\ intervals\ [-p\ port]\ [-t\ period]\ [-f\ fault]$
- $bbGRID\ -t\ m\ [-p\ port]\ [-t\ period]$

Bibliographie

[AAB+02] E. Alba, F. Almeida, M. Blesa, C. Cotta, M. Diaz, I. Dorta, J. Gabarro, J. Gonzalez, C. Leon, L. Moreno, J. Petit, J. Roda, A. Rojas, , and F. Xhafa. Mallba : A library of skeletons for combinatorial optimisation. In B. Monien and editors R. Feldman, editors, *EuroPa 2002 Parallel Processing, volume 2400 of Lecture Notes in Computer Science*, pages 927–932, Berlin Heidelberg, 2002. SpringerVerlag.

[ABGL02] K. Anstreicher, N. Brixius, J.P. Goux, and J. Linderoth. Solving large quadratic assignment problems on computational grids. *Mathematical Programming*, 91(3) :563–588, 2002.

[ACG86] S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *Computer*, 19(8) :26–34, 1986.

[ACK+02] D.P. Anderson, J. Cobb, E. Korpela, , M. Lepofsky, and D. Werthimer. SETI@home : An Experiment in Public-Resource Computing. *Communications of the ACM*, Vol. 45(No. 11) :56–61, Nov. 2002.

[AF02] K. Aida and Y. Futakata. High-performance parallel and distributed computing for the BMIeigenvalue problem. *Parallel and Distributed Processing Symposium., Proceedings International, IPDPS 2002, Abstracts and CD-ROM*, pages 71–78, 2002.

[ANF03] K. Aida, Wataru N., and Y. Futakata. Distributed computing with hierarchical master-worker paradigm for parallel branch and bound algorithm. *Proceedings oft the 3rd IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid 2003)*, pages 156–163, 2003.

[AT02] E. Alba and M. Tomassini. Parallelism and Evolutionary Algorithms. *IEEE Transactions on Evolutionary Computation*, 6(5) :443–462, October 2002.

[Bas05] M. Basseur. *Conception d'algorithmes coopératifs pour l'optimisation multi-objectif : Application aux problèmes d'ordonnancement de type Flow-Shop.* PhD thesis, Université des Sciences et Technologies de Lille, France, 2005.

[BBL02] M. Baker, R. Buyya, and D. Laforenza. Grids and grid technologies for wide-area distributed computing. *Software - Practice & Experience*, 32(15) :1437–1466, 2002.

[BCD+95] M. Benaichouche, V.D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Bob : une plateforme unifiée de développement pour les

algorithmes de type branch-and-bound. RR 95/12, Laboratoire PRiSM, Université de Versailles Saint Quentin en Yvelines, May 1995.

[BdKT95] P. Bouvry, J.C. de Kergommeaux, and D. Trystram. Efficient Solutions for Mapping Parallel Programs. *European Conference on Parallel Processing*, pages 379–390, 1995.

[Bel95] T. Belding. The distributed genetic algorithm revisited. In D. Eshelmann editor, editor, *Sixth Int. Conf. on Genetic Algorithms*, San Mateo, CA, 1995. Morgan Kaufmann.

[BKR97] R.E. Burkard, S.E. Karisch, and F. Rendl. QAPLIB–A Quadratic Assignment Problem Library. *Journal of Global Optimization*, 10(4) :391–403, 1997.

[BL02] C.A. Bohn and G.B. Lamont. Load balancing for heterogeneous clusters of PCs. *Future Generation Computer Systems*, 18(3) :389–400, 2002.

[BLDGS03] L. Bote-Lorenzo, A. Dimitriadis, and E. Gómez-Sánchez. Grid Characteristics and Uses : A Grid Definition. In *European Across Grids Conference, LNCS 2970*, Lecture Notes in Computer Science, pages 291–298, 2003.

[BMF+99] A. Brunnger, A. Marzetta, K. Fukuda, , and J. Nievergelt. The parallel search bench zram and its applications. *Annals of Operations Research*, 90 :45–63, 1999.

[CANT95] C. Cotta, J.F. Aldana, A.J. Nebro, and J.M. Troya. Hybridizing Genetic Algorithms with Branch and Bound Techniques for the Resolution of the TSP. In *Artificial Neural Nets and Genetic Algorithms 2, D.W. Pearson, N.C. Steele, R.F. Albrecht (eds.), Springer-Verlag*, pages 277–280, Wien New York, 1995.

[CDC+94] V.D. Cung, S. Dowaji, B. Le Cun, T. Mautor, and C. Roucairol. Parallel and distributed branch-and-bound/A* algorithms. Technical Report 94/31, Laboratoire PRISM, Université de Versailles, 1994.

[CHMR87] J.P. Cohoon, S.U. Hedge, W.N. Martin, and D. Richards. Punctuated equilibria : A parallel genetic algorithm. In J.J. Grefenstette, editor, *Proceedings of the Second International Conference on Genetic Algorithms*, page 148. Lawrence Erlbaum Associates, 1987.

[CMM67] R. Conway, W. Maxwell, and L. Miller. *Theory of scheduling*. Addison-Wesley, 1967.

[CMRR02] V.D. Cung, S.L. Martins, C.C. Ribeiro, and C. Roucairol. Strategies for the Parallel Implemetation of Metaheuristics. *Essays and Surveys in Metaheuristics, C.C. Ribeiro, P. Hansen (Eds.), Kluwer Academic Publishers, Norwell, MA*, pages 263–308, 2002.

[CP98] E. Cantu-Paz. A survey of parallel genetic algorithms. *Calculateurs Parallèles, Réseaux et Systèmes Répartis*, 10(2) :141–171, 1998.

[CR95] B. Le Cun and C. Roucairol. BOB : A unified platform for implementing branch-and-bound like algorithms. Research Report 95/16, PRiSM Laboratory, University of Versailles - St. Quentin en Yvelines, 1995.

[CT02] T.G. Crainic and M. Toulouse. *Parallel Strategies for Meta-heuristics*. In F. Glover and G. Cochenberger, eds., State-of-the-Art handbook in Meta-heuristics, Kluwer Academic Publishers, Norwell, MA, 2002.

[CZ06] S. Climer and W. Zhang. Cut-and-solve : an iterative search strategy for combinatorial optimization problems, artificial intelligence. 170 :714–738, 2006.

[DFP+96] T.A. DeFanti, I. Foster, M.E. Papka, R. Stevens, and T. Kuhfuss. Overview of the I-WAY : Wide-Area Visual Supercomputing. *The Intl. Journal of Supercomputer Applications and High Performance Computing*, 10(2/3) :123–131, Summer/Fall 1996.

[DL90a] J. Du and J. Y.-T. Leung. Minimizing total tardiness on one machine is NP-hard. *Mathematics of Operations Research*, 15 :483–495, 1990.

[DL90b] J. Du and Y.T. Leung. Minimizing Total Tardiness on One Machine is NP-hard. *Mathematics of operations research*, 15 :483–495, 1990.

[Ehr97] M. Ehrgott. *Multiple criteria optimization : Classificaiton and methodology*. PhD thesis, University of Kaiserslautern, Germany, 1997.

[EPH00] J. Eckstein, C. A. Phillips, and W. E. Hart. PICO : An object-oriented framework for parallel branch-and-bound. Research Report 40–2000, RUTCOR, 2000.

[Fed03] G. Fedak. *XtremWeb : une plate-forme pour l'étude expérimentale du calcul global pair-à-pair*. PhD thesis, Université Paris XI, 2003.

[FK97] I. Foster and C. Kesselman. Globus : A Metacomputing Infrastructure Toolkit. *Intl. J. of Supercomputer Applications*, 11(2) :115–128, 1997.

[FK99] I. Foster and C. Kesselman. *The Grid : Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1999.

[FKT01] I. Foster, C. Kesselman, and S. Tuecke. The anatomy of the Grid : Enabling Scalable Virtual Organizations. *Int. J. High Perform. Comput. Appl.*, 15(3) :200–222, 2001.

[FM87] R. Finkel and U. Manber. DIB-a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 9(2) :235–256, 1987.

[Fos02] I. Foster. What is the Grid ? A Three Point Checklist. *Grid Today*, 1(6), July 22 2002.

[FVW99] A. Fink, S. Voß, and D. Woodruff. Building reusable software components for heuristic search. *Operations Research Proceedings*, pages 210–219, 1999.

[GC94] B. Gendron and T.G. Crainic. Parallel Branch and Bound Algorithms : Survey and Synthesis. *Operations Research*, 42 :1042–1066, 1994.

[Gel85] D. Gelernter. Generative Communication in Linda. *ACM Transactions on Programming Languages and Systems*, Vol. 7(No. 1) :80–112, jan. 1985.

[GGS04] B. Goldengorin, D. Ghosh, and G. Sierksma. Branch and peg algorithms for the simple plant location problem. *Computers & Operations Research*, 31 :241–255, 2004.

[GHJV94] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns, Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

[GJS76] M.R. Garey, D.S. Johnson, and R. Sethi. The complexity of flow-shop and job-shop scheduling. *Mathematics of Operations Research*, 1 :117–129, 1976.

[GKYL00] J-P. Goux, S. Kulkarni, M. Yoder, and J. Linderoth. An Enabling Framework for Master-Worker Applications on the Computational Grid. In *HPDC'00 : Proc. of the* 9^{th} *IEEE Intl. Symposium on High Performance Distributed Computing (HPDC'00)*, page 43, Washington, DC, USA, 2000. IEEE Computer Society.

[GLLK79] L. Graham, E. Lawler, J. Lenstra, and A. Rinooy Kan. Optimization and approximation in deterministic sequencing and scheduling : a survey. *Annals of Discrete Mathematics*, 5 :287–326, 1979.

[Gri02] A.S. Grimshaw. What is a Grid ? *Grid Today*, 1(26), 2002.

[GWB+04] M. Gerndt, R. Wismüller, Z. Balaton, G. Gombás, Z. Németh, N. Podhorski, H.-L. Truong, T. Fahringer, M. Bubak, E. Laure, and T. Margalef. *Performance Tools for the Grid : State of the Art and Future*, chapter Volume 30 of LRR-TUM Research Report Series35. Shaker Verlag, Aachen, 2004.

[HJMS+00] W. Hoscheck, J. Jaen-Martinez, A. Samar, H. Stockinger, and K. Stockinger. Data management in an international data Grid project. In *Proc. of the* 1^{st} *IEEE/ACM Intl. Work. on Grid Computing (Grid 2000)*, pages 77–90, Déc. 2000.

[Hol75] J.H. Holland. *Adaptation in natural and artificial systems.* Ann Arbor, MI, USA, The University of Michigan Press, 1975.

[IF00] A. Iamnitchi and I. Foster. A Problem-Specific Fault-Tolerance Mechanism for Asynchronous, Distributed Systems. *29th International Conference on Parallel Processing (ICPP), Toronto, Canada, August*, pages 21–24, 2000.

[Jac56] S. M. Jackson. An extension of johnson's results on job-lot scheduling. *Naval Research Logistics Quartely*, 3, 1956.

[JAM88] V.K. Janakiram, D.P. Agrawal, and R. Mehrotra. A Randomized Parallel Branch-and-bound Algorithm. In *in Proc. of Int. Conf. on Parallel Processing*, pages 69–75, Aug. 1988.

[JF88] R. Johnson and B. Foote. Designing Reusable Classes. *Journal of Object-Oriented Programming*, 1(2) :22–35, June/July 1988.

[Joh54] S. M. Johnson. Optimal two- and three-stage production schedules with setup times included. *Naval Research Logistics Quarterly*, 1 :61–68, 1954.

[KBH88] L.H. Keel, S.P. Bhattacharyya, and J.W. Howze. Robust control with structure perturbations. *IEEE Trans. on Automatic Control*, 33 :68–78, 1988.

[KBM02] K. Krauter, R. Buyya, and M. Maheswaran. A taxonomy and survey of grid resource management systems for distributed computing. *Software - Practice & Experience*, 32(2) :135–164, 2002.

[Kim95] Y.D. Kim. Minimizing total tardiness in permutation flowshops. *European Journal of Operational Research*, 33 :541–551, 1995.

[KK84] V. Kumar and L. Kanal. Parallel Branch-and-Bound Formulations For And/Or Tree Search. *IEEE Trans. Pattern. Anal. and Machine Intell.*, PAMI–6 :768–778, 1984.

[KR87] S.A. Kravitz and R.A. Rutenbar. Placement by simulated annealing on a multiprocessor. *IEEE Transactions in Computer Aided Design*, 6 :534–549, 1987.

[LDT04] J. Lemesre, C. Dhaenens, and E.G. Talbi. A parallel exact method for a bicriteria permutation flow-shop problem. In *In Proc. of Project Management and Scheduling (PMS'04)*, pages 359–362, Jul. 2004.

[LDT07] J. Lemesre, C. Dhaenens, and E.G. Talbi. An exact parallel method for a bi-objective permutation flowshop problem. *European journal of operational research*, 177(3) :1641–1655, 2007.

[Lem06] J. Lemesre. *Méthodes exactes pour l'optimisation combinatoire multi-objectif : conception et application.* PhD thesis, Université des Sciences et Technologies de Lille, 2006.

[LLK78] J. K. Lenstra, B. J. Lageweg, and A. H. G. Rinnooy Kan. A general bounding scheme for the permutation flow-shop problem. *Operation Research*, 26(1) :53–67, 1978.

[LLM88] M. Litzkow, M. Livny, and M. Mutka. Condor - A Hunter of Idle Workstations. In *Proc. of the 8th Intl. Conf. of Distributed Computing Systems*, Jun. 1988.

[Mel05] Nouredine Melab. Contributions à la résolution de problèmes d'optimisation combinatoire sur grilles de calcul. LIFL, USTL, Novembre 2005. Thèse HDR.

[Meu02] H. Meunier. *Algorithmes évolutionnaires parallèles pour l'optimisation multi-objectif de réseaux de télécommunications mobiles.* PhD thesis, Université des Sciences et Technologies de Lille, 2002.

[MG95] S.W. Mahfoud and D.E. Goldberg. Parallel recombinative simulated annealing : A genetic algorithm. *Parallel Computing*, 21 :1–28, 1995.

[Mit59] L. G. Mitten. Sequencing n jobs on two machines with arbitrary time lags. *Management Science*, 5(3), 1959.

[MMT05] M. Mezmaz, N. Melab, and E. G. Talbi. Towards a Coordination Model for Parallel Cooperative P2P Multi-objective Optimization. *LNCS, Advances in Grid Computing - EGC 2005*, 3470/2005 :305–314, 2005.

[MMT06] N. Melab, M. Mezmaz, and E.-G. Talbi. Parallel cooperative meta-heuristics on the computational grid : A case study : the bi-objective flow-shop problem. *Parallel Comput.*, 32(9) :643–659, 2006.

[MMT07a] M. Mezmaz, N. Melab, and E. G. Talbi. Combining metaheuristics and exact methods for solving exactly multi-objective problems on the grid. *Journal of Mathematical Modelling and Algorithms*, 6(3) :393–409, 2007.

[MMT07b] M. Mezmaz, N. Melab, and E. G. Talbi. An efficient load balancing strategy for grid-based branch and bound algorithm. *Parallel Comput.*, 33(4-5) :302–313, 2007.

[MMT07c] M. Mezmaz, N. Melab, and E-G. Talbi. A grid-enabled branch and bound algorithm for solving challenging combinatorial optimization problems. In *Proc. of 21th IEEE Intl. Parallel and Distributed Processing Symp.*, Long Beach, California, March 2007.

[MOF92] O. Martin, S.W. Otto, and E.W. Felten. Large-step markov chains for the tsp : Incorporating local search heuristics. *Operation Research Letters*, 11 :219–224, 1992.

[MP93] D.L. Miller and J.F. Pekny. The Role of Performance Metrics for Parallel Mathematical Programming Algorithms. *ORSA J. Computing*, 5(1) :26–28, 1993.

[MS05] R. Moe and T. SØREVIK. Parallel Branch and Bound Algorithms on Internet Connected Workstations. *High Performance Computing and Communcations*, 3726/2005 :768–775, 2005.

[NGB04] Z. Németh, G. Gombás, and Z. Balaton. Performance Evaluation on Grids : Directions, Issues and Open Problems. In *12th Euromicro PDP*, pages 290–297. IEEE Service Center, Feb. 2004.

[OTT98] S. Obayashi, S. Takahashi, and Y. Takeguchi. Niching and Elitist Models for MOGAs. In *PPSN'98, Springer Verlag, LNCS 1498*, pages 260–269, 1998.

[Par12] V. Pareto. Manuel d'économie politique. *Bull. Amer. Math. Soc. 18 (1912), 462-474.*, 1912.

[PC04] R. Pastor and A. Corominas. Branch and win : Or tree search algorithms for solving combinatorial optimisation problems. *Top*, 1 :169–192, 2004.

[Pin95] M. Pinedo. *Scheduling - Theory, Algorithms, and Systems*. Prentice Hall, 1995.

[PM98] G.T. Parks and I. Miller. Selective Breeding in a Multiobjective Genetic Algorithm. In A. E. Eiben, M. Schoenauer, and H.-P. Schwefel, editors, *Parallel Problem Solving From Nature — PPSN V*, pages 250–259, Amsterdam, Holland, 1998. Springer-Verlag.

[Rei90] G. Reinelt. *TSPLIB-A Traveling Salesman Problem Library.* Inst. für Mathematik, 1990.

[RG05] T.K. Ralphs and M. Guzelsoy. The symphony callable library for mixed integer programming. In *The Proceedings of the Ninth INFORMS Computing Society Conference*, 2005.

[RS04] R. Ruiz and T. Stutzle. A Simple and Effective Iterative Greedy Algorithm for the Flowshop Scheduling Problem. *Technical Report, submitted to European Journal of Operational Research*, 2004.

[SHH95] Y. Shinano, M. Higaki, , and R. Hirabayashi. A generalized utility for parallel branch-and-bound algorithms. In *Proceedings of the 7nd IEEE Symposium on Parallel and Distributed Processing (SPDP'95)*, pages 858–865, 1995.

[Sin98] D. Sinclair. The GST load balancing algorithm for parallel and distributed systems. *International Journal of Approximate Reasoning*, 19(1-2) :39–56, 1998.

[SMLN+03] I. Stoica, R. Morris, D. Liben-Nowell, D.R. Karger, M.F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord : a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking (TON)*, 11(1) :17–32, 2003.

[SWM91] T. Starkweather, D. Whitley, and K. Mathias. Optimization using distributed genetic algorithms. In H.-P. Schwefel and R. Manner, editors, *Parallel Problem Solving from Nature*, volume 496, page 176. Lecture Notes in Compuer Science, Springer-Verlag, 1991.

[Tai93] E. Taillard. Benchmarks for basic scheduling problems. *European Journal of Operational Research*, 64 :278–285, 1993.

[Tal02] E-G. Talbi. A Taxonomy of Hybrid Metaheuristics. *Journal of Heuristics, Kluwer Academic Publishers*, Vol.8 :541–564, 2002.

[TAML05] E-G. Talbi, E. Alba, N. Melab, and G. Luque. *Metaheuristics and Parallelism*, chapter 4, pages 79–103. In Wiley Book on Parallel Metaheuristics : A New Class of Algorithms, 2005.

[TMS94] E-G. Talbi, T. Muntean, and I. Samarandache. Hybridation des algorithmes génétiques avec la recherche tabou. In *Evolution Artificielle EA'94*, Toulouse, Sept. 1994.

[TO95] O. Toker and H. Ozbay. On the NP-hardness of solving bilinear matrix inequalities andsimultaneous stabilization with static output feedback. In *Proceedings of the American Control Conference*, Seattle, Washington, 1995.

[TTL02] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid computing : Making the Global Infrastructure a Reality.* John Wiley & Sons Inc, Dec 2002.

[Wal99] M. Wall. GAlib : A C++ Library of Genetic Algorithms Components., 1999. http ://lancet.mit.edu/ga/.

[WM95] D.H. Wolpert and W.G. Macready. No free lunch theorems for search. Technical Report SFI-TR-95-02-010, Santa Fe Institute, 1995.

[XRLS05] Y. Xu, T..K. Ralphs, L. Ladányi, and M.J. Saltzman. Alps : A framework for implementing parallel search algorithms. In *The Proceedings of the Ninth INFORMS Computing Society Conference*, 2005.

Table des figures