Parallélisme en programmation par contraintes

Mohamed Rezgui

UNIVERSITE NICE SOPHIA ANTIPOLIS

ECOLE DOCTORALE STIC
SCIENCES ET TECHNOLOGIES DE L'INFORMATION ET DE LA COMMUNICATION

T H E S E

pour obtenir le titre de

Docteur en Sciences

de l'Université Nice Sophia Antipolis
Mention : INFORMATIQUE

Présentée et soutenue par

Mohammed REZGUI

Parallélisme en programmation par contraintes

Thèse dirigée par Jean-Charles RÉGIN

préparée au Laboratoire I3S
Informatique, Signaux et Systèmes de Sophia-Antipolis
UMR7271 - UNS CNRS

soutenue le 8 Juillet 2015

Jury :

Rapporteurs :	Christophe LECOUTRE	-	Université d'Artois
	Laurent PERRON	-	Google
	Xavier LORCA	-	Ecole des Mines de Nantes
Directeur :	Jean-Charles RÉGIN	-	Université Nice Sophia Antipolis
Président :	Michel RUEHER	-	Université Nice Sophia Antipolis
Examinateurs :	Arnaud MALAPERT	-	Université Nice Sophia Antipolis
	Michel RUEHER	-	Université Nice Sophia Antipolis
Invités :	Benoit ROTTEMBOURG	-	Euro Décision
	Bertrand LECUN	-	Université de Versailles

Remerciements

En premier lieu, je tiens à remercier à mon directeur de thèse Jean-Charles Régin qui m'a encadré depuis ma dernière année en école d'ingénieur jusqu'à la fin de cette thèse. Je ne saurai jamais lui exprimer toute ma gratitude pour ses qualités humaines, son aide, son soutien, sa grande disponibilité durant ces années de thèse. Sans lui cette thèse n'aurai pas abouti. Il m'a toujours assuré la possibilité de travailler dans les meilleures conditions. Au cours de ces dernières années, il s'est impliqué dans l'orientation de mes travaux et il s'est également investi dans celle de mon avenir. Il n'y était pas obligé mais je l'en suis infiniment reconnaissant. Je suis fier et heureux qu'il fasse ainsi partie aujourd'hui des personnes qui ont marqué ma vie. Merci infiniment Jean-Charles.

Un très grand merci à Arnaud Malapert qui m'a grandement aidé durant cette thèse. Ses aides techniques, ses conseils, son soutien et sa gentillesse. Nous avons passé de bons moments ensemble avec des conversations intéressantes et enrichissantes. Merci infiniment Arnaud.

Merci à Carine Fédèle et à Michel Rueher pour avoir pris le temps de lire mon manuscrit et d'avoir grandement contribué à son amélioration par leurs remarques et commentaires. Je les remercie également pour leur accueil chaleureux à chaque fois que j'ai sollicité leur aide, ainsi que pour leurs multiples encouragements. Je remercie également Claude Michel pour les débats intéressants.

Je souhaite aussi exprimer ma gratitude à mes amis Moussa et à mes co-bureaux Mohammed Saïd Belaid, Raylen La, Youssef Bennani, Guillaume Perez, Aïtem Zitoun, Sébastien Autran, Anthony Palmieri, Emilien Cornillon et tant d'autres doctorants ou non. Je tiens à les remercier de m'avoir aidé, soutenu et encouragé pendant cette thèse.

Je tiens à remercier les membres de mon jury qui ont accepté de s'intéresser à mon travail malgré leurs emplois du temps surchargés. Je leur en suis très reconnaissant.

Je remercie tous les membres du pôle MDSC du laboratoire I3S sans citation de leurs noms pour ne pas en oublier, pour leur gentillesse et leur accueil.

Je remercie également à tous mes anciens étudiants. Je leur souhaite bonne continuation dans leurs études et dans leur parcours professionnel.

Enfin, je remercie du fond du cœur et avec un grand amour ma mère pour sa présence, ses sacrifices consentis à mon éducation, sa patience et son interminable soutien qui me sont d'une immense source d'énergie et d'enthousiasme. Je remercie également mes frères Ahmed, Yacine et Abdoullah qui m'ont soutenu sans relâche et encouragé durant toutes ces années. Je remercie également aux membres de la "grande" famille (les tantes, les oncles, les cousins, les cousines) pour leurs soutiens et leurs encouragements.

Table des matières

CHAPITRE 1

Introduction

Dans cette thèse, nous nous intéressons aux apports du parallélisme pour la Programmation Par Contraintes (PPC). L'objectif est de résoudre plus rapidement de grands problèmes. Nous rappelons tout d'abord succinctement les concepts de base de la programmation par contraintes et du parallélisme. Nous présentons également l'état de l'art concernant l'application du parallélisme en PPC. Ensuite, nous donnons les motivations et les contributions apportées par notre étude. Enfin, nous détaillons le plan du manuscrit.

Présentation de la PPC, domaines d'application

La programmation par contraintes est une technique de résolution de problèmes qui vient de l'intelligence artificielle et qui est née dans les années 1970. On peut dire que la PPC telle qu'elle existe s'est principalement inspirée :

- des travaux sur les *"General Problem Solvers"*, notamment ceux de Jean-Louis Laurière qui a proposé le système ALICE [Laurière 1976, Laurière 1978] et ceux de John McCarthy [McCarthy 1977, McCarthy 1987] ;
- de la programmation logique avec contraintes et principalement de Prolog [Colmerauer 1990] ainsi que des travaux de Mehmet Dincbas et Pascal van Hentenryck qui ont développé un langage Prolog nommé CHIP (Constraint Handling In Prolog), un langage inspiré d'ALICE [Van Hentenryck 1989] ;
- des problèmes de satisfaction de contraintes (en anglais *Constraint Satisfaction Problem*) [Waltz 1975, Montanari 1974, Freuder 1978] ;
- de la recherche opérationnelle dont elle a emprunté de nombreux algorithmes pour la résolution de contraintes globales.

C'est avec le premier de ces quatre domaines que la PPC est le plus proche. En PPC, un problème est défini à partir de *variables* et de *contraintes*, dans le but de trouver un ensemble de solutions. Chaque variable est munie d'un domaine définissant l'ensemble des valeurs possibles pour cette variable. Une contrainte exprime une propriété qui doit être satisfaite par un ensemble de variables. Une contrainte portant sur 2 (resp n) variables est une contrainte binaire (resp n-aire). Une *affectation* entre l'ensemble des variables et des valeurs respectant toutes les contraintes forme une *solution*.

En PPC, un problème est aussi vu comme une *conjonction de contraintes* pour lesquels nous disposons de méthodes efficaces de résolution. Ces contraintes peuvent être très

simples comme $x < y$ ou complexes comme la recherche du plus court chemin dans un graphe.

La programmation par contraintes va utiliser pour chaque contrainte, une méthode de résolution spécifique afin de supprimer les valeurs des domaines des variables impliquées dans la contrainte qui, compte tenu des valeurs des autres domaines, ne peuvent appartenir à aucune solution de cette contrainte. Ce mécanisme est appelé *filtrage*. En procédant ainsi pour chaque contrainte, les domaines des variables vont se réduire. Pour illustrer le fonctionnement d'un algorithme de filtrage, nous utilisons l'une des propriétés les plus intéressantes qui est la cohérence d'arc. Par abus de langage, nous parlerons de consistance d'arc. Un algorithme de filtrage associé à une contrainte établit la consistance d'arc s'il supprime toutes les valeurs des variables impliquées dans la contrainte qui ne peuvent pas appartenir à une solution de la contrainte. Par exemple, considérons la contrainte $x+3 = y$ avec le domaine de la variable x, noté $D(x)$, égal à $\{1, 3, 4, 5\}$ et le domaine de la variable y, noté $D(y)$, égal à $\{4, 5, 8\}$. Établir la consistance d'arc conduira à modifier les domaines avec $D(x) = \{1, 5\}$ et $D(y) = \{4, 8\}$ où il n'existe pas de solution pour $x \in \{3, 4\}$ et $y \in \{5\}$.

Après chaque modification du domaine d'une variable, il est nécessaire d'étudier à nouveau l'ensemble des contraintes impliquant cette variable, car la modification peut conduire à de nouvelles déductions. Autrement dit, la réduction du domaine d'une variable peut permettre de déduire la non-appartenance de certaines valeurs d'autres variables à une solution. Ce mécanisme est appelé *propagation*.

Ensuite et afin de parvenir à une solution, l'espace de recherche va être parcouru en essayant d'affecter successivement une valeur à toutes les variables. Ce parcours suit une stratégie de recherche. Elle revient à caractériser des critères permettant de choisir la prochaine variable et la prochaine valeur qui sera affectée à cette variable. Les mécanismes de filtrage et de propagation étant bien entendu relancés après chaque essai puisqu'il y a modification de domaines. Parfois, une affectation peut entraîner la disparition de toutes les valeurs d'un domaine : on dit alors qu'un *échec* se produit ; le dernier choix d'affectation est alors remis en cause, il y a *backtrack* (retour en arrière) et une nouvelle affectation est tentée.

La programmation par contraintes est donc basée sur trois principes : filtrage, propagation et recherche de solutions. On pourrait représenter ces principes en reformulant la définition du célèbre Kowalski de l'algorithme (Algorithme = Logique + Contrôle) [Kowalski 1979] de la manière suivante :

$$PPC = \underbrace{Filtrage + Propagation}_{\text{Logique}} + \underbrace{Recherche\ de\ solutions}_{\text{Contrôle}} \tag{1.1}$$

où le filtrage et la propagation correspondent à la logique et la recherche de solutions au contrôle.

La programmation par contraintes permet une grande souplesse pour énoncer et résoudre les problèmes. Une contrainte peut être définie en extension ou en intention.

Une contrainte définie en extension est représentée par un ensemble des combinaisons de valeurs autorisées ou interdites. Chaque combinaison de cet ensemble est appelée un tuple. Par exemple, si les domaines des variables x et y contiennent les valeurs 0, 1 et 2,

alors on peut définir la contrainte "x est plus petit que y" en extension par "(x=0 et y=1) ou (x=0 et y=2) ou (x=1 et y=2)", ou encore par "(x,y) élément de $\{(0,1),(0,2),(1,2)\}$". Toute contrainte peut être représentée par une conjonction de contraintes binaires en extension [Rossi 1990]. Cependant, l'espace mémoire occupé par les tuples autorisés (ou interdits) peut être exponentiel, car c'est un sous-ensemble du produit cartésien des domaines des variables. Précisément, le nombre de tuples représenté en mémoire est borné par d^n, où n est le nombre maximal de variables et d la plus grande taille des domaines des variables. Malheureusement, nous retrouvons en temps la complexité en espace, car la complexité de test de consistance ou de filtrage dépend du nombre de tuples.

On peut éviter cette complexité en mémoire exponentielle en représentant la contrainte en intention. Une contrainte en intention correspond à la représentation en compréhension de l'ensemble de ses tuples autorisés : les tuples autorisés sont exprimés à l'aide de propriétés mathématiques. Par exemple, soient les variables x et y, on peut définir la contrainte "x est plus petit que y" par la contrainte en intention $x < y$.

Contrairement à une contrainte en extension, une contrainte définie en intention ne demande pas beaucoup d'espace mémoire, mais demande l'écriture d'un algorithme de filtrage qui soit capable d'exploiter la sémantique véhiculée par sa définition.

Nous montrons l'avantage d'utiliser la structure d'une contrainte en intention par rapport à une contrainte en extension avec la contrainte $x \leq y$. Soit $min(D)$ et $max(D)$ la valeur minimale et la valeur maximale d'un domaine D. Il est facile d'établir que les valeurs de x et y qui sont celles de l'intervalle $[min(D(x)), max(D(y))]$, satisfont la contrainte. Par exemple, en prenant le domaine $[10, 100000]$ pour la variable x et $[0, 90000]$ pour la variable y, la contrainte $x \leq y$ en intension met à jour les bornes des deux intervalles à $[10, 90000]$; alors que la contrainte en extension doit énumérer tous les tuples vérifiant la contrainte $x \leq y$ entre 10 et 90000.

Entre autres, cela signifie que la consistance d'arc peut être efficace et facilement mise en place en retirant les valeurs qui ne sont pas dans l'intervalle ci-dessus. En outre, l'utilisation de la structure est souvent le seul moyen d'éviter d'avoir des problèmes de mémoire lorsqu'il s'agit de contraintes non binaires. Une contrainte en intention permet d'éviter de représenter explicitement toutes les combinaisons de valeurs permises par la contrainte.

Ainsi, des algorithmes de filtrage spécifiques pour la plupart des contraintes en intention simples (comme $=$, $<$, $\leq$, etc.) ont été conçus. Des algorithmes génériques permettant d'exploiter efficacement une certaine connaissance des contraintes binaires (AC3 [Mackworth 1977], AC4 [Mohr 1986], AC5 [Hentenryck 1992], AC6 [Bessiere 1994] et AC2001 [Bessière 2001]) ont aussi été développés.

Cependant, deux nouveaux problèmes sont apparus : le manque d'expressivité de ces contraintes simples et la faiblesse de la réduction des domaines par les algorithmes de filtrage associés à ces contraintes simples. Il est, en effet, très pratique lors de la modélisation d'un problème en PPC d'avoir des contraintes correspondant à un ensemble de contraintes. Ces contraintes complexes peuvent soit être exprimées par une combinaison logique de plusieurs contraintes en utilisant les opérateurs OR, AND, NOT, XOR ou soit exprimées par des problèmes plus complexes comme la recherche du plus court chemin dans un graphe. Ces dernières contraintes exploitent la particularité du problème, on dit qu'elles exploitent la structure du problème. Ces contraintes, appelées contraintes globales, permettent de dé-

finir des algorithmes de filtrage efficaces, car elles peuvent tenir compte de la présence simultanée de contraintes simples pour réduire encore plus les domaines des variables.

Une des contraintes globales les plus célèbres est la contrainte `alldiff` [Régin 1994], car l'algorithme de filtrage associé à cette contrainte est en mesure d'établir la consistance d'arc d'une manière très efficace. Une contrainte `alldiff` définie sur un ensemble de variables X établit que les valeurs prises par des variables doivent être toutes différentes. Cette contrainte peut être représentée par un ensemble de contraintes binaires. Dans ce cas, une contrainte binaire est construite pour chaque paire de variables de X. Pour une telle contrainte binaire entre deux variables x et y, la consistance d'arc supprime une valeur du domaine de x seulement lorsque le domaine de y est réduit à une valeur unique. Supposons que nous avons un problème avec 3 variables x, y, z et une contrainte `alldiff` impliquant ces variables avec $D(x) = \{a, b\}, D(y) = \{a, b\}$ et $D(z) = \{a, b, c\}$. Établir la consistance d'arc pour cette contrainte `alldiff` supprime les valeurs a et b du domaine de z, tandis que la consistance d'arc pour la contrainte `alldiff` représentée par des contraintes binaires de différence ne supprime aucune valeur.

Utilisation de la PPC en pratique

La programmation par contraintes a pour ambition de résoudre n'importe quel type de problème combinatoire. Elle est et a été utilisée pour une très grande variété d'applications réelles. Les programmes permettant la modélisation et la résolution des problèmes en PPC sont appelés des solveurs. Dans cette thèse, nous utilisons `OR-tools`, solveur de Google [Perron 2012], `Gecode` [Schulte 2006] et `Choco2` [Choco 2010]. Une illustration des différents domaines où la PPC a énormément contribué, est donnée dans le mémoire de HDR (Habilitation à Diriger des Recherches) de [Régin 2004] par l'utilisation du solveur industriel, ILOG Solver [Puget 1994] :

- Planification et gestion : affectation des tâches aux ressources, de missions satellitaires et de réseaux, dimensionnement et modélisation, gestion de la chaîne logistique et d'entrepôts, la configuration et diagnostic d'équipements, l'ordonnancement de lignes de production, l'affectation de fréquences et de bande passante et l'optimisation de charge.

- Production industrielle : l'entreprise Chrysler a mis en place un système de planification de la production de ses véhicules. L'application gère la séquence des opérations de peinture des véhicules et améliore la productivité de 15 usines du groupe en Amérique du Nord, au Mexique et en Europe. Ce système a permis au producteur automobile de réduire ses coûts de production de 500000 dollars par an et par usine, soit une économie totale de 7 à 9 millions de dollars par an.

- Transport : dans le domaine des transports, la PPC a fait ses preuves pour des applications telles que l'affectation d'équipages, de comptoirs, de portes d'embarquement et de tapis à bagages, la gestion de flottes et la planification du trafic.

- Commerce en ligne : la PPC est utilisée pour résoudre des problèmes liés à la gestion

des commandes et des approvisionnements, le service de voyages, la gestion des crédits ou de conseils financiers.

- Défense : la PPC a été utilisée pour planifier la formation des 85000 militaires de la British Army.

- Agencement ou aménagement : la PPC a fait ses preuves dans de nombreuses variétés de problèmes de placement d'une façon générale, d'aménagement d'intérieur, en passant par l'assemblage, jusqu'à l'agencement des véhicules et bateaux.

Parallélisme

Le parallélisme est utilisé depuis longtemps en informatique pour résoudre des problèmes scientifiques (simulation, météorologie, biologie, jeux vidéo) plus rapidement. Le principe de base du parallélisme est d'utiliser plusieurs ressources (processeurs) fonctionnant concurremment pour accroître la puissance de calcul [Almasi 1989]. L'objectif du parallélisme est non seulement de résoudre les problèmes le plus rapidement, mais aussi de pouvoir résoudre des problèmes de plus grande taille.

En informatique, un programme est constitué d'une suite d'instructions exécutables. Un processus est une instance d'exécution d'un programme dans un certain contexte pour un ensemble particulier de données. Depuis les débuts de l'informatique, la plupart des programmes ont été définis pour s'exécuter sur une seule unité de calcul, c'est-à-dire qu'ils s'exécutent sur un processeur avec un seul cœur. De tels programmes sont dits séquentiels. Durant l'exécution d'un programme séquentiel, une seule instruction est exécutée et une seule donnée est traitée à un moment donné.

Il y a 20 ans, les machines parallèles (possédant plusieurs processeurs) étaient très coûteuses et réservées à l'usage de grandes entreprises ou de certains centres de recherche. Depuis une dizaine d'années, les ordinateurs intègrent des processeurs possédant plusieurs cœurs (processeurs multi-cœurs). Paralléliser les programmes ou les méthodes de résolution est nécessaire pour exploiter la puissance des architectures matérielles actuelles.

Comment paralléliser un programme ?

La mise au point des programmes parallèles de façon efficace dépend du problème à résoudre. Si certains problèmes sont simplement divisibles en plusieurs sous-problèmes indépendants qui sont ensuite résolus par différentes unités de calcul, ce n'est pas le cas de tous. Nous prenons trois problèmes pour l'illustrer :

- compter les maisons dans une zone géographique : il est simple de paralléliser efficacement ce problème en attribuant une partie de la zone géographique à chaque unité de calcul afin de compter le nombre de maisons présentes et en regroupant les résultats à la fin ;

- trier des nombres : on peut découper le problème en tâches indépendantes et les regrouper, mais regrouper en parallèle n'est pas simple bien que faisable, car on a

besoin de mettre en place un mécanisme de coopération entre les unités de calcul. Dans la littérature, le tri parallèle a fait l'objet de nombreux travaux sur différentes architectures ; [Blelloch 1998, Cérin 2006, Jeon 2002, Sanders 2004], notamment basé sur une parallélisation du tri par fusion.

- trouver le plus court chemin de Nice à Lille : ce problème est difficile à paralléliser ; il est difficile de le découper en tâches indépendantes car on doit partager les ressources. De plus, il faut veiller à ne pas calculer inutilement des chemins qui ne peuvent pas participer à une solution.

On dit que deux tâches sont concurrentes lorsqu'elles demandent un accès simultané à la même ressource. L'utilisation de la concurrence entre les tâches dépend du type de problème à résoudre. Ainsi, l'écriture d'un programme parallèle nécessite de faire le choix entre trois types de concurrence existants :

- disjointe : les tâches concurrentes ne communiquent et n'interagissent pas, l'écriture d'un programme est grandement simplifiée ;
- compétitive : les tâches concurrentes sont en compétition pour l'accès à certaines ressources partagées (par exemple le temps CPU, un port d'entrées/sorties, une zone mémoire) ;
- coopérative : les tâches concurrentes coopèrent pour atteindre un objectif commun ; des échanges de données ont lieu entre elles.

Le problème du comptage des maisons peut être facilement parallélisable en utilisant la concurrence disjointe parce qu'il est facile de découper en sous-zones. Les deux autres problèmes vont devoir utiliser les trois types de concurrence en fonction de la nature des sous-problèmes que les unités de calcul vont devoir traiter (s'il y a communication ou non, s'il y a un objectif commun).

Les problèmes induits par la concurrence se manifestent dans les cas de la concurrence compétitive et coopérative. Dans ces cas, la mémoire est partagée, plusieurs tâches d'un programme peuvent modifier exactement la même donnée en même temps. Cela pourrait invalider le programme. Afin d'écrire des programmes parallèles cohérents, nous avons donc besoin de mettre en place des accès exclusifs aux données. En outre, nous avons besoin d'empêcher toutes les autres tâches du programme qui sont concurrentes, de lire ou d'écrire sur des données verrouillées. Pour cela, on a historiquement utilisé différentes primitives de synchronisation comme les *mutex*, les moniteurs ou encore les sémaphores (calqués sur le principe de la signalisation ferroviaire). Ces différentes primitives sont une forme plus ou moins évoluée de verrouillage qui sert à mettre en place la synchronisation des tâches concurrentes (sur une ressource ou plus généralement sur une section critique). Cependant, la synchronisation induite par ces primitives implique une perte de gain dans le parallélisme, car elle est consommatrice de temps. Plus les tâches du programme auxquelles nous accordons un accès exclusif prennent du temps, plus le risque d'attente augmente. Ainsi, réduire la nécessité d'un accès exclusif est nécessaire pour optimiser les performances.

Par exemple, considérons que nous voulions peindre les chambres d'une maison de dix pièces, où une seule personne est autorisée à être présente dans une chambre à tout moment, et chaque chambre prend la même quantité de temps pour être peinte. Si une personne peut peindre la maison en dix heures, dix personnes pourront peindre la maison en une heure. Mais si on ajoute une chambre, alors cela prendra deux fois plus de temps de peindre la maison avec le même nombre de personnes.
Deux autres problèmes spécifiques sont également à considérer lorsque les primitives de synchronisation sont utilisées :

- interblocage (ou *deadlock*) : Le problème se produit lorsque deux tâches concurrentes s'attendent mutuellement (chacune attendant que l'autre libère les ressources). Les unités de calcul bloquées dans cet état le sont définitivement, il s'agit donc d'une situation catastrophique, car le programme ne s'arrêtera jamais, parce qu'il n'arrive pas à démarrer.

- famine (ou *livelock*) : ce problème se produit lorsqu'un programme n'est pas équitable, c'est-à-dire qu'il ne garantit pas un accès en un temps fini à l'ensemble des unités de calcul souhaitant accéder à des ressources verrouillées.

Traiter ces problèmes liés à la synchronisation ne suffit pas pour être efficace, car il faut aussi prendre en compte les problèmes globaux de répartition de charge. En effet, un programme parallèle doit définir un ensemble de tâches. Chacune de ces tâches contient les données à traiter et le traitement à effectuer sur ces données. L'écriture d'un programme parallèle implique la décomposition de l'ensemble du calcul en sous-tâches et l'affectation de celles-ci aux différentes unités de calcul. Le but dans la répartition de charge est de répartir de manière équilibrée le travail entre les unités de calcul, et de réduire au minimum les communications entre les différentes unités de calcul. On peut distinguer deux types d'approche de décomposition d'un problème en sous-problèmes :

- la répartition statique distribue les tâches sur la base de la quantité de travail *a priori* de chaque tâche (au début de la résolution). Ce type d'algorithme donne de bons résultats lorsque la durée d'exécution de chaque tâche est prévisible,

- la répartition dynamique vise à obtenir de meilleurs résultats en effectuant continuellement des mesures du taux d'occupation de chaque unité de calcul, mais cela peut entraîner des ralentissements à cause de la création de nouvelles tâches à effectuer et à l'augmentation de la communication due à la gestion des requêtes.

PPC et parallélisme

Lorsque nous voulons paralléliser la résolution d'un problème en PPC, il faut diviser le travail nécessaire en plusieurs tâches et les répartir entre plusieurs solveurs. Précédemment, nous avons décrit trois principes définissant la PPC : filtrage, propagation et recherche de solutions. Il est possible de paralléliser chacune de ces étapes :

- filtrage : paralléliser les algorithmes de filtrage. Par exemple, dans certains algorithmes de consistance d'arc, comme l'algorithme AC-2001 [Bessière 2001], la recherche de support d'une valeur d'un domaine d'une variable, c'est-à-dire la recherche d'un tuple autorisé et valide, est indépendante des autres recherches de supports et donc on peut faire un traitement parallèle de chaque recherche de support.

- propagation : évaluer les contraintes en parallèle. Depuis quelques années apparaissent des travaux sur ce type de parallélisme avec les problèmes de satisfaction de contraintes distribués (CSP distribués) [Yokoo 1990, Yokoo 1998, Hamadi 1998, Hamadi 2002, Wahbi 2011] qui utilisent la programmation par agent. Chaque agent, associé à une unité de calcul, gère au niveau local une partie distribuée du CSP correspondant au problème initial. Comme l'intégration des agents dans le CSP distribué implique une redéfinition du problème initial, plusieurs CSP distribués [Hirayama 1997, Chong 2006, Léauté 2009] ont été proposés. L'une des difficultés majeures des CSP distribués survient, lorsqu'à partir des solutions locales calculées par les agents, il faut constituer une solution globale. Un exemple d'utilisation des CSP distribués est la parallélisation de l'algorithme de consistance d'arc de chaque contrainte que l'on appelle consistance parallèle. Cela peut être très utile si la consistance d'arc prend beaucoup de temps.

- recherche : décomposer l'espace de recherche en plusieurs tâches disjointes. On l'appelle recherche parallèle ; on peut alors résoudre les tâches avec des solveurs utilisables en parallèle. Plusieurs méthodes de parallélisation de l'espace de recherche ont été proposées utilisant la décomposition statique simple [Bordeaux 2009] ou le *work stealing* qui s'appuie sur la répartition dynamique [Michel 2009, Perron 1999, Schulte 2000, Zoeteweij 2004, Jaffar 2004, Chu 2009].

Une autre façon de paralléliser la résolution d'un problème en PPC est d'utiliser la méthode du portefeuille (*portfolio* en anglais). La méthode *portfolio* est très utilisée dans le domaine financier. Un conseiller financier doit choisir pour ses clients (club d'investissement) un certain nombre d'actions dans lesquelles investir. En diversifiant ainsi les investissements, il espère ainsi limiter les risques et avoir une rentabilité en tenant compte de la durée prévue du placement. Par analogie, le principe du *portfolio* en PPC consiste à exécuter en parallèle plusieurs stratégies de recherche différentes (diversification) sur le même espace de recherche. Lorsqu'on utilise une seule stratégie de recherche, le risque d'avoir des cas pathologiques est grand. Le *portfolio* en PPC limite ce risque et permet rapidement de trouver une solution. Cette idée a été exploitée de manière séquentielle avec la méthode du redémarrage (*restart* en anglais) [Gomes 2000, Xu 2008], qui est

une méthode utilisant des redémarrages du parcours de l'espace de recherche afin d'éviter d'explorer des parties non prometteuses de l'espace de recherche, qui ne semblent pas conduire rapidement à une solution. Plusieurs travaux ont été effectués autour de la méthode du *portfolio* [Luby 1993, Gomes 1997, Gomes 1999, Gomes 2001, Kautz 2002, Hamadi 2009, Bordeaux 2009, Amadini 2012]

Objectifs et contributions

L'objectif de cette thèse et de proposer une nouvelle méthode de parallélisation de la résolution d'un problème PPC. Cette méthode doit être simple et facile à implémenter. Elle doit également obtenir des gains linéaires avec des centaines d'unités de calcul voire des milliers. Les contributions apportées par ce travail de recherche sont les suivantes :

- La méthode EPS (pour *Embarrassingly Parallel Search*) [Régin 2013, Régin 2014] qui part du constat que plus on génère de sous-problèmes au début de la résolution, mieux on résout les problèmes de répartition de charge et donc de famine. On réduit également les communications.
- Plusieurs algorithmes qui permettent la décomposition du problème PPC en plusieurs sous-problèmes. Ces algorithmes ont été implémentés pour plusieurs solveurs PPC : `OR-tools`, `Gecode` et `Choco2`.
- Une évaluation expérimentale de la méthode EPS et les différents algorithmes proposés ainsi qu'une comparaison avec les autres approches de parallélisme (*work stealing*, *portfolio*).
- Un environnement d'expérimentation avec notamment une interface de programmation (API) permettant d'implémenter aisément la méthode EPS sur plusieurs modes de communication. Cette API a été implémentée en C++ avec le solveur `Gecode`.

Les résultats de cette thèse ont été publiés dans des conférences nationales (Journées Francophones de Programmation par Contraintes) [Rezgui 2013, Rezgui 2014a] et internationales (*CP* et *HLPP*) [Régin 2013, Menouer 2014b, Régin 2014] qui concernent l'apport de la méthode EPS sur différentes architectures. Les problèmes liés au *cloud computing* et de l'utilisation de la méthode EPS dans cet environnement ont été présentés dans un *workshop* de la conférence internationale *CP* [Rezgui 2014b]. L'apport d'EPS dans l'amélioration du *work stealing* [Menouer 2015] a été publié dans la revue *International Journal of Parallel Programming*.

Organisation du manuscrit

L'organisation de ce document est la suivante :

- Le chapitre 2 présente un état de l'art sur le parallélisme en programmation par contraintes, à savoir les différentes méthodes existantes comme les CSP distribués, le *portfolio* et les décompositions statiques et dynamiques de l'espace de recherche.
- Le chapitre 3 détaille les principes d'EPS et ces différents algorithmes.
- Le chapitre 4 présente la partie expérimentale décrivant le comportement d'EPS lors de la recherche de toutes les solutions ou lors de la résolution de problèmes d'optimisation. Nous comparerons également les résultats avec le *work stealing* et le *portfolio*.
- Dans le chapitre 5, nous proposons une interface de programmation (API) générique qui permet une implémentation simple d'EPS quelle que soit l'architecture utilisée (architecture à mémoire partagée avec les *threads* ou architecture distribuée avec MPI et programmation réseau).
- Enfin le chapitre 6 conclut cette thèse et donne quelques perspectives.

CHAPITRE 2

Principes et méthodes de la PPC et du parallélisme

Sommaire

Nous avons présenté de manière non formelle les principes de la PPC au début de l'introduction. Dans ce chapitre, nous allons définir plus formellement un problème en PPC. Nous donnons ensuite les principes du parallélisme et enfin son utilisation avec la PPC. L'ensemble des notations utilisées dans cette thèse sont regroupées dans l'annexe A.

2.1 Principes de la PPC

2.1.1 Définition d'un graphe

Les définitions qui suivent sont extraites des livres [Berge 1983, Harary 1969, Gondran 2009].

Définition 1 *Un graphe $G = (X, E)$ est déterminé par la donnée de :*

- *un ensemble X dont les éléments sont appelés sommets ou nœuds.*
- *un ensemble E dont les éléments sont des paires de sommets appelées arêtes.*

Une arête du graphe est une paire $e = (x, y)$ de sommets. Les sommets x et y sont les extrémités de l'arête. Un graphe simple est sans boucle (arête dont les extrémités coïncident) et ne possède jamais plus d'une arête entre deux sommets quelconques. Deux sommets sont voisins ou adjacents s'ils appartiennent à la même arête. Les deux sommets d'une arête sont appelés extrémités de l'arête. L'ensemble des voisins d'un sommet w est noté $\Gamma(x)$. Le degré d'un sommet x, noté $deg(x)$ est le nombre de ses voisins.

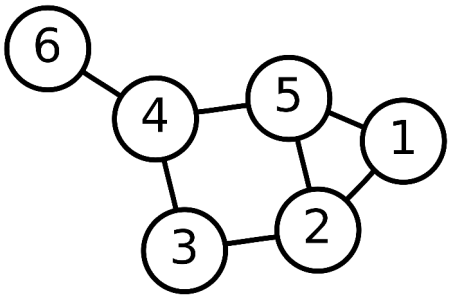

FIGURE 2.1 – Un exemple de graphe

La figure 2.1 représente un exemple de graphe ayant 6 nœuds et 7 arêtes.

Un graphe simple non orienté est parfaitement déterminé par la donné de l'ensemble X de ses sommets et par l'application Γ, qui à tout élément de X fait correspondre l'ensemble de ses voisins. On pourra noter un graphe simple non orienté aussi bien $G = (X, \Gamma)$ que $G = (X, E)$. Dans la suite, et par abus de langage, nous emploierons le terme graphe pour graphe simple.

Définition 2

- *Une chaîne est une séquence $(u_1, \ldots, u_q)$ d'arêtes de G telles que chaque arête de la séquence ait une extrémité en commun avec l'arête qui la précède et l'autre extrémité en commun avec l'arête suivante. On peut également décrire une chaîne par une séquence de sommets $(x_{1_1}, x_{1_2}, \ldots, x_{q_1}, x_{q_2})$ ordonnée suivant la séquence d'arêtes définissant la chaîne. Une chaîne qui n'utilise pas deux fois le même sommet est dite élémentaire.*

- *Un cycle est une chaîne ne contenant pas deux fois la même arête dans la séquence et telle que les deux sommets aux extrémités de la chaîne coïncident.*
- *Un graphe G est connexe si pour toute paire de sommets x, y il existe au moins une chaîne dans G qui relie x et y. L'existence d'une chaîne entre deux sommets quelconques d'un graphe définit une relation d'équivalence sur les sommets de ce graphe. Les classes d'équivalence de cette relation constituent les composantes connexes du graphe.*

2.1.2 Réseau de contraintes

Nous limitons notre étude aux réseaux de contraintes à domaines finis. Un réseau de contraintes $\mathcal{N}$ est défini par un triplet $(\mathcal{X}, \mathcal{D}, \mathcal{C})$ où :

- $\mathcal{X} = \{x_1, \ldots, x_n\}$, est un ensemble de variables,
- $\mathcal{D} = \{D(x_1), \ldots, D(x_n)\}$, est un ensemble de domaines avec $D(x_i)$, l'ensemble fini des valeurs possibles pour la variable x_i. On définit l'ensemble d'entiers $[a, b]$, tous les entiers compris entre a à b, a et b inclus.
- $\mathcal{C} = \{C_1, \ldots, C_e\}$, est un ensemble de contraintes entre les variables x_i où chaque contrainte définit les combinaisons de valeurs des variables autorisées.

Définition 3 *Une* **contrainte** *C définie sur l'ensemble ordonné de variables $X(C) = (x_{i_1}, \ldots, x_{i_r})$ est un sous-ensemble $T(C)$ du produit cartésien $D(x_{i_1}) \times \ldots \times D(x_{i_r})$ qui spécifie les combinaisons* **autorisées** *de valeur pour les variables dans $X(C)$. Un élément de $T(C)$ est appelé un* **tuple** *de $X(C)$ et $|X(C)|$ représente* **l'arité** *de C. C est dite binaire si $|X(C)| = 2$ et n-aire si $|X(C)| > 2$.*

Définition 4 *Une valeur a pour une variable x est souvent désignée par (x, a). (x, a) est* **valide** *si $a \in D(x)$, et un tuple t sur $X(C)$ est* **valide** *si $\forall (x, a) \in t$, $a \in D(x)$.*
Une affectation $x = a$ réduit le domaine de x à $\{a\}$.

Définition 5 *[Dechter 1992] Soit $\mathcal{X}'$ un sous-ensemble des variables de $\mathcal{X}$.*
Une instanciation partielle I sur $\mathcal{X}'$ associe à chaque variable de $\mathcal{X}'$ une valeur de son domaine.

2.1.3 Filtrage

Définition 6 *Une contrainte C est consistante si et seulement s'il existe un tuple t de $T(C)$ valide. Une valeur $a \in D(x)$ est consistante avec C si et seulement si $x \notin X(C)$ où il existe un tuple t valide de $T(C)$ avec $(x, a) \in t$ (t est appelé support de (x, a) sur C).*

Un algorithme de filtrage est associé à chaque contrainte C. Son rôle est de supprimer des valeurs des domaines des variables de C pour lesquelles il n'est pas possible de satisfaire C. Par exemple, pour la contrainte $x < y$ avec $D(x) = [10, 20]$ et $D(y) = [0, 15]$, un algorithme de filtrage associé à cette contrainte pourra supprimer les valeurs de 15 à 20 de

D(x) et les valeurs de 0 à 10 de D(y), ainsi les domaines résultants sont $D(x) = [10, 14]$ et $D(y) = [11, 15]$. Lorsqu'un algorithme de filtrage supprime toutes les valeurs des variables impliquées dans la contrainte qui ne sont pas consistantes avec la contrainte, on dit qu'il réalise la cohérence d'arc [1]. Par exemple, pour la contrainte $x + 5 = y$ avec les domaines $D(x) = \{3, 4, 6, 7\}$ et $D(y) = \{6, 7, 11\}$, un algorithme de filtrage établissant la consistance d'arc modifiera les domaines pour obtenir $D(x) = \{3, 7\}$ et $D(y) = \{6, 11\}$.

2.1.4 Mécanisme de propagation

Dès lors qu'un algorithme de filtrage associé à une contrainte modifie le domaine d'une variable, les conséquences de cette modification sont étudiées pour les autres contraintes impliquant cette variable, car de nouvelles déductions peuvent être éventuellement faites. Autrement dit, les algorithmes de filtrage des autres contraintes sont appelés afin de déduire éventuellement d'autres suppressions. On dit alors qu'une modification a été propagée. Ce mécanisme de **propagation** est répété jusqu'à ce qu'il n'apparaisse plus aucune modification. Comme les domaines sont finis et que les algorithmes de filtrage sont appelés au plus une fois pour chaque modification, ce processus se termine nécessairement. L'idée sous-jacente à ce mécanisme est d'essayer d'obtenir des déductions globales. En effet, on espère que la conjonction des déductions obtenues pour chaque contrainte prise indépendamment conduira à un enchaînement de déductions, c'est-à-dire que cette conjonction soit plus forte que l'union des déductions obtenues indépendamment les unes des autres.

2.1.5 Mécanisme de recherche de solutions

Une solution est une instanciation de l'ensemble de variables satisfaisant toutes les contraintes.

Définition 7 *Le problème de la recherche de l'existence d'une ou plusieurs solutions dans un réseau de contraintes est appelé problème de satisfaction de contraintes, de l'anglais Constraint Satisfaction Problem (CSP).*

Définition 8 *Le problème de la recherche de l'existence d'une solution optimale dans un réseau de contraintes est appelé problème d'optimisation, de l'anglais Constraint Optimization Problem (COP). On associe à ce problème une fonction, appelée fonction objective, qui permet de minimiser ou de maximiser une variable qui contient la valeur de l'objectif.*

Lors de la résolution de problèmes d'optimisation, on distinguera deux types de solutions : les solutions du CSP et les solutions optimales (solutions du COP) c'est-à-dire celles qui intègrent l'objectif en cherchant à le minimiser ou maximiser.

Ainsi, le mécanisme de recherche de solutions a pour but de trouver une solution, éventuellement optimale, et met en œuvre les différents moyens qui vont permettre au solveur d'atteindre des solutions. Parmi ces moyens, nous pouvons citer :

[1] Par abus de langage, nous utilisons également le terme consistance d'arc au lieu de la cohérence d'arc. La définition originale est *arc consistency* en anglais

- les stratégies de choix de variables et de valeurs. Elles définissent les critères qui vont permettre de déterminer la prochaine variable qui sera instanciée ainsi que la valeur qui lui sera affectée.
- les méthodes de décomposition. Lorsque le problème est trop gros, il est souvent nécessaire de le décomposer en plusieurs parties, puis de résoudre ces parties de façon plus ou moins indépendante et enfin de les recombiner.
- les améliorations itératives. Il est souvent illusoire de vouloir trouver et prouver l'optimalité d'un problème de grande taille. Aussi, bien souvent, on recherche quelques "bonnes" solutions puis on essaie de les améliorer à l'aide de techniques d'améliorations locales.

2.1.6 Arbre de recherche

Définition 9 *Un arbre est un graphe connexe et sans cycle dont le nœud initial est appelé racine. Soit un arbre T de racine r.*

- *La racine est le seul nœud qui n'a pas de prédécesseur.*
- *Le père d'un nœud x est l'unique voisin de x sur le chemin de la racine à x. La racine r est le seul nœud sans père.*
- *Les fils d'un nœud x sont les voisins de x autres que son père.*
- *Une feuille est un nœud sans fils. Les feuilles correspondent aux nœuds de degré 1.*
- *La hauteur $h(T)$ de l'arbre T est alors la profondeur maximale de ses nœuds, c'est-à-dire la profondeur à laquelle il faut descendre dans l'arbre pour trouver le nœud le plus loin de la racine.*
- *L'arité de l'arbre T est le nombre maximum de successeurs d'un nœud x peut avoir.*
- *Un sous-arbre T' de T est un arbre tel que tous les nœuds et toutes les arêtes de T' sont aussi des nœuds ou des arêtes de T.*

La figure 2.2 montre un exemple d'un arbre. Ici, le nœud n_1 est la racine de l'arbre, n_5, n_6, n_7, n_9, n_{10}, n_{11} sont les feuilles, n_1, n_2, n_3, n_4, n_8 les nœuds internes. Plus généralement, l'ensemble des nœuds est constitué des nœuds internes et des feuilles.

L'application d'une procédure de recherche de solution peut être représentée par un arbre de recherche.

On peut toujours représenter un nœud de l'arbre de recherche par l'affectation d'une valeur à une variable. Si cette affectation a lieu au niveau k, alors le nœud apparaitra à la profondeur k dans l'arbre. À tout nœud n_k de niveau k est donc associé à une instanciation partielle I_k définie sur un ensemble de k variables. La racine de cet arbre est un nœud qui ne correspond à aucune instanciation, mais qui a pour fils tous les nœuds de la profondeur 1. Le nombre maximum de feuilles d'un arbre associé à une procédure de recherche est égal au nombre d'instanciations possibles. L'affectation de valeurs à des variables s'apparente à

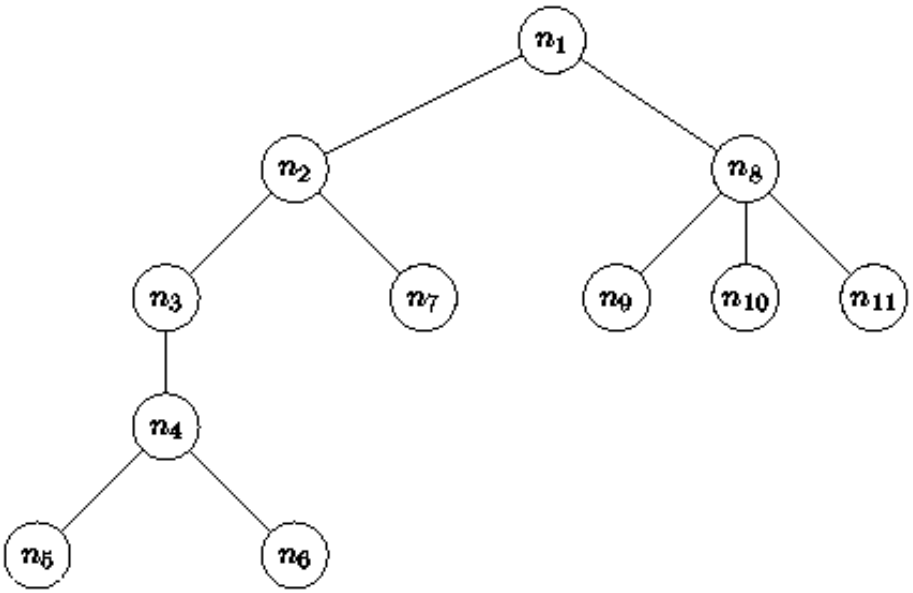

FIGURE 2.2 – Un exemple d'arbre

une descente dans l'arbre de recherche, tandis que la remise en cause d'une affectation est vue comme une remontée (*backtrack* en anglais).

La figure 2.3 donne un exemple d'arbre de recherche pour un problème avec deux variables, x_1 et x_2, dont leurs domaines ont pour valeurs $\{a, b, c\}$. Les nœuds à la profondeur 1 n'ont seulement que x_1 qui est instanciée et les nœuds à la profondeur 2 ont les deux variables qui sont instanciées. La feuille la plus à gauche correspond à l'affectation des valeurs $\{a, b\}$.

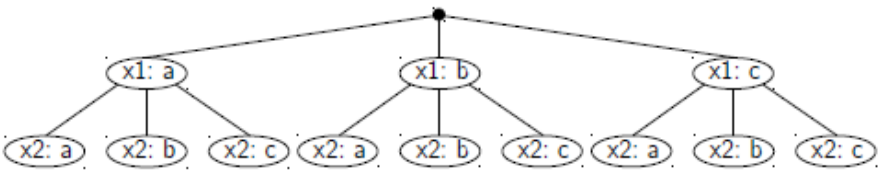

FIGURE 2.3 – Un exemple d'arbre de recherche

2.1.7 Modélisation

Pour résoudre un problème, il faut :

- utiliser des contraintes qui permettent de parvenir rapidement à une solution. Cependant, l'identification de ces contraintes est difficile. D'une part, il faut que chaque algorithme de filtrage associé à chacune de ces contraintes puisse éliminer un grand nombre de valeurs qui ne peuvent pas appartenir à une solution. D'autre part, les modifications dues au filtrage d'une contrainte doivent permettre aux autres contraintes de faire de nouvelles déductions permettant d'éliminer un grand nombre de valeurs ne pouvant pas appartenir à une solution. Il faut également prendre en compte le temps de filtrage d'une contrainte, car le parcours des structures de données et les raisonnements sur les modifications des domaines peuvent influencer grandement sur le temps de résolution. Il faut donc avoir un compromis entre temps de filtrage et efficacité du filtrage sur le nombre de valeurs filtrées.

- utiliser une stratégie de recherche qui minimise le nombre de nœuds à parcourir lors de l'exploration de l'arbre de recherche.

La figure 2.4 montre la modélisation du problème de sudoku en programmation par contraintes en utilisant Minizinc [Nethercote 2007], un langage de modélisation en programmation par contraintes et une solution de ce problème. Un problème de sudoku contient 81 variables représentant les cases du jeu ou on peut mettre les chiffres de 1 à 9, ce qui définit le domaine des variables $D = [1, 9]$. On doit mettre les chiffres dans les cases tel que chaque horizontale, chaque verticale et chaque bloc de 9 cases contiennent des chiffres différents. Pour avoir des chiffres différents, on a deux choix de modélisation. On peut utiliser soit des contraintes binaires de différence ou la contrainte `alldiff` [Régin 1994] qui établit que les valeurs prises par des variables doivent être toutes différentes.

2.1.8 Contraintes

Dans cette section, nous étudions les contraintes complexes comme les contraintes de table et les contraintes globales qui s'appliquent sur un ensemble de variables.

Contrainte de table

Lorsqu'on définit une contrainte C explicitement, on énumère l'ensemble des valeurs qui satisfont cette contrainte : on dit que la contrainte est définie en extension. On l'appelle communément contrainte de table. Elle peut être exprimée :

- soit positivement en listant les tuples autorisés par C ;

- soit négativement en listant les tuples interdits par C.

Par exemple, si les domaines des variables x et y contiennent les valeurs 0, 1 et 2, alors on peut définir la contrainte $x < y$ en extension par l'ensemble des tuples autorisés $\{(0, 1), (0, 2), (1, 2)\}$;

```
% On inclut la contrainte alldiff
include "all_different.mzn";
% On crée 81 variables avec un domaine de valeurs de 1 à 9
array [1..9, 1..9] of var 1..9: sudoku;

% On pose la contrainte alldiff pour chaque ligne
constraint
  forall (row in 1..9)
    (all_different (col in 1..9) (sudoku[row, col]));
% On pose la contrainte alldiff pour chaque colonne
constraint
  forall (col in 1..9)
    (all_different (row in 1..9) (sudoku[row, col]));
% On pose la contrainte alldiff pour chaque carré 3x3
constraint
  forall (row, col in {0, 3, 6})
    (all_different (i, j in 1..3) (sudoku[row+i, col+j]));

solve satisfy; % On lance la recherche d'une solution
output [ show(sudoku) ]; % On affiche la solution
```

5	3	4	6	7	8	9	1	2
6	7	2	1	9	5	3	4	8
1	9	8	3	4	2	5	6	7
8	5	9	7	6	1	4	2	3
4	2	6	8	5	3	7	9	1
7	1	3	9	2	4	8	5	6
9	6	1	5	3	7	2	8	4
2	8	7	4	1	9	6	3	5
3	4	5	2	8	6	1	7	9

FIGURE 2.4 – La modélisation du problème de sudoku en programmation par contraintes avec Minizinc et une solution de ce problème

Les contraintes de table sont utiles et indispensables lorsqu'il est difficile d'établir une sémantique dans les relations entre les variables. L'algorithme de filtrage utilisé dans les contraintes de table réalise la consistance d'arc.

Plusieurs algorithmes de filtrage existants établissent la fermeture par consistance d'arc. Dans le cas binaire, nous avons AC3 [Mackworth 1977], AC4 [Mohr 1986], AC5 [Hentenryck 1992], AC6 [Bessiere 1994] et AC2001 [Zhang 2001]. Pour le cas n-aires, on peut citer GAC-4 [Mohr 1988], STR1 [Ullmann 2007], STR2 [Lecoutre 2011] STR3 [Lecoutre 2012], GAC-4R [Perez 2014] ou GAC-Schema [Bessière 1997]. Tous ces algorithmes peuvent être décrits par l'algorithme AC* [Régin 2005]. L'avantage de cet algorithme générique est qu'il permet de combiner ou sélectionner plusieurs algorithmes de consistance d'arc pour obtenir les meilleures performances en exploitant la structure des contraintes.

D'autres algorithmes de filtrage dans le cas n-aire, représentent les contraintes de table par des *Multi-Valued Decision Diagram* [Cheng 2010, Perez 2014] et sont compétitifs aux autres algorithmes précités.

Contraintes globales

L'un des moyens les plus efficaces pour améliorer les performances dans la résolution en PPC est d'utiliser les contraintes dont le filtrage réalise la consistance d'arc. Ces contraintes, appelées contraintes globales, s'appliquent notamment sur un ensemble de variables. Elles permettent d'obtenir un niveau d'abstraction dans la modélisation des problèmes d'une façon beaucoup plus concise et proche de la réalité. Cependant, l'inconvénient des contraintes globales est que leurs algorithmes de filtrage utilisent des structures complexes dont les calculs prennent plus de temps et ont une complexité de calcul élevée dans la plupart des cas. Ainsi, représenter une contrainte globale par un ensemble de contraintes binaires réduit la puissance du filtrage, cependant seules les contraintes dont les variables changent doivent être recalculées, ce qui réduit le temps de propagation. Les contraintes globales doivent donc être utilisées avec parcimonie, car il faut définir un compromis optimal entre le temps de filtrage et le temps de propagation de ces contraintes.

Une panoplie de contraintes globales ont été découvertes durant ces vingt dernières années et elles sont référencées dans un catalogue consultable en ligne [Beldiceanu 2005]. Nous en présentons quelques-unes :

- `alldiff` [Régin 1994] : cette contrainte établit que les valeurs prises par des variables doivent être toutes différentes. Elle peut être traitée en utilisant la théorie des couplages dans les graphes bipartis. Elle est souvent utilisée en pratique dans les problèmes d'affectation et de permutation.

- `gcc` [Régin 1996] : la contrainte de cardinalité globale impose que le nombre de fois qu'une valeur a puisse apparaître dans une solution, soit compris entre une borne inférieure notée l_a, et une borne supérieure notée u_a. Le nombre de variables, ayant pour valeur a dans une solution, doit donc appartenir à l'intervalle $[l_a, u_a]$. Elle est traitée par l'utilisation d'un algorithme de flot maximum dans un graphe. Cette contrainte est une généralisation du `alldiff` ; on fixe les bornes l_a à 0 et u_a à 1 pour `alldiff`. Elle est souvent utilisée en pratique dans les problèmes d'affectations complexes et de distribution.

- `cumulative` [Aggoun 1993] : cette contrainte exprime le fait qu'à tout instant, le total des ressources utilisées par un ensemble de tâches pour une machine donnée ne dépasse pas une certaine limite. Elle permet de résoudre des problèmes d'ordonnancement tels que l'affectation des tâches aux processeurs, la gestion de projet, la gestion d'ateliers dans les usines, la gestion des emplois du temps ou la rotation et affectation du personnel.

- `disjunctive` [Baptiste 1996] : cette contrainte exprime le fait qu'à tout instant, un ensemble de tâches ne peuvent se chevaucher, c'est-à-dire tous les intervalles de temps représentant les tâches sont disjoints de sorte qu'une ressource n'exécute qu'une tâche à tout instant. Elle est très utilisée en ordonnancement.

- `sequence` [Régin 1997] : cette contrainte est adaptée pour modéliser les contraintes réglementaires comme les contraintes légales de gestion des ressources

humaines (obligation d'un jour de repos après chaque période de 5 journées consécutives travaillées)

- `cycle` [Beldiceanu 1994] : cette contrainte exprime le nombre de tournées à calculer, les contraintes de poids sur chaque tournée, les incompatibilités entre des sites et les fenêtres de temps de visite par site. Elle est complémentaire à la contrainte `cumulative` dans les applications d'ordonnancement où elle permet d'exprimer des contraintes de délais dues à des changements d'outils.

- `among` [Beldiceanu 1994] : cette contrainte permet de restreindre le nombre de variables pouvant être assignées à des valeurs d'un ensemble donné. Elle est utilisée dans de nombreux problèmes d'allocation de ressources tels que la rotation et affectation du personnel ou le *car sequencing* qui consiste à déterminer l'ordre dans lequel un ensemble de voitures seront fabriquées sur une chaîne d'assemblage.

- `tree` [Beldiceanu 2005] : cette contrainte qui sépare les nœuds d'un graphe orienté en un ensemble de nœuds disjoints qui peuvent définir des anti-arborescences (arbres orientés où tout point possède au plus un successeur) pour lesquelles seuls certains nœuds peuvent être des racines. Elle est utilisée dans la biologie moléculaire et a des applications dans la linguistique.

- `regular` [Pesant 2004] : cette contrainte impose que la suite de valeurs prises par les variables corresponde à un mot du langage régulier décrit par un automate fini déterministe.

2.1.9 Stratégies de recherche

Lorsqu'on utilise la propagation des contraintes, il n'est pas garanti que toutes les variables auront été instanciées. En effet, certains domaines peuvent encore contenir plusieurs valeurs. Puisque la propagation n'est en pratique jamais suffisante pour déterminer des solutions réalisables, il est généralement nécessaire d'entreprendre une recherche de ces solutions. Ce processus s'effectue par l'ajout dynamique de contraintes et la génération d'un arbre de recherche où chaque feuille représente une solution réalisable. Lorsque plusieurs variables restent non instanciées, il est nécessaire de choisir la prochaine variable qui fera l'objet d'une instanciation et d'une propagation des contraintes. L'ensemble des décisions opérées sur l'arbre de recherche est défini par une stratégie de recherche (stratégie de branchement). Les stratégies de recherche déterminent ainsi :

- l'ordre des variables à instancier ;

- l'ordre des valeurs à affecter.

L'efficacité d'une stratégie de recherche dépend du nombre de nœuds parcourus durant la recherche de solutions. Il y a deux types de choix pour la variable à affecter et les valeurs à tenter :

- le choix statique qui sélectionne les variables à instancier et les valeurs de leurs domaines dans un ordre prédéfini ;

- le choix dynamique qui sélectionne une variable et une valeur de son domaine en fonction d'un ou plusieurs critères (taille des domaines, nombre de contraintes violées par le choix d'une précédente affectation, etc.).

On peut ainsi définir des stratégies de recherche qui utilisent le choix statique comme la stratégie `lex` qui sélectionne la première variable non instanciée et tente les valeurs du domaine dans l'ordre croissant. On peut également définir d'autres stratégies de recherche qui utilisent le choix dynamique comme la stratégie `dom` [Haralick 1980] qui sélectionne la variable ayant le plus petit domaine ou la stratégie `random` qui sélectionne aléatoirement une variable non instanciée.

Il y a également d'autres stratégies de recherche plus complexes telles que :

- `dom/ddeg` [Bessiere 1996, Beck 2005] sélectionne la variable ayant le ratio le plus petit entre la taille du domaine courant et du degré de la variable. Le degré d'une variable est le nombre de contraintes auxquelles la variable est impliquée. `dom/ddeg` choisit donc la variable qui est responsable du plus grand nombre de variables non affectées après chaque propagation.

- `dom/wdeg` et `dom/bwdeg` : [Boussemart 2004] sélectionne la variable ayant le ratio le plus petit entre la taille du domaine courant et du degré pondéré de la variable. Le poids est incrémenté de 1 à chaque fois qu'une contrainte est violée durant la résolution. Le degré pondéré d'une variable est donc la somme des poids sur l'ensemble des contraintes auxquelles cette variable est impliquée. `dom/bwdeg` est une variante de `dom/wdeg` en utilisant un arbre de recherche binaire.

- `impact` [Refalo 2004] utilise l'importance des variables lors de la réduction de l'espace de recherche. S'inspirant des calculs de coûts utilisés en *Mixed Integer Programming* (MIP), cette stratégie calcule l'impact d'une instanciation $x = a$, c'est-à-dire la réduction des domaines courants des autres variables. L'impact calculé permet l'importance de cette instanciation. Ainsi, les impacts calculés durant le parcours de l'arbre permettent de réduire l'espace de recherche en prenant de meilleures décisions.

- `activity` [Michel 2012] utilise la réduction des domaines pour définir les activités des variables et faire un choix sur la variable qui a la plus grande activité. L'avantage est qu'elle permet d'avoir une vision plus fine d'apprentissage au niveau des contraintes n-aires et des contraintes globales contrairement au calcul du poids effectué par `dom/wdeg` et `dom/bwdeg`.

2.2 Principes du parallélisme

Pour résoudre efficacement un problème long et coûteux en temps de calcul, nous pouvons améliorer le temps de résolution en changeant de modèle. L'identification de nouvelles contraintes devient alors de plus en plus difficile dans la recherche d'une meilleure expression du modèle et dans l'efficacité des algorithmes de filtrage. Une autre solution est

d'utiliser, pour un modèle donné, plusieurs unités de calcul qui vont résoudre chacune une partie du problème. Chaque partie du problème représente une tâche à réaliser par une unité de calcul.

2.2.1 Entités de traitement de calcul

Afin d'avoir une meilleure compréhension du comportement et des interactions entre les unités de calcul, nous nous proposons d'utiliser des termes plus génériques et abstraits. Un *worker* est une unité de calcul chargée de résoudre un sous-problème. La plupart du temps, il correspond à un cœur d'un processeur. Un *master* est une unité de calcul chargée de contrôler la résolution. En général, il n'y a qu'un seul *master*.

2.2.2 Efficacité du parallélisme

Il est important de mesurer l'efficacité d'une méthode en parallélisme, car cela permet de connaître le gain apporté de son utilisation et de son effet sur les performances. L'intérêt de l'étude de performance permet de comparer les différents algorithmes utilisés, de savoir les endroits du code où l'on doit optimiser, d'évaluer l'effet des topologies des architectures (réseaux de communication) et d'estimer le coût de la mise en œuvre. Il y a plusieurs métriques de performance comme le temps d'exécution, la mémoire utilisée, les entrées-sorties (temps de transfert et volume) et le coût de la mise en œuvre. Les choix de ces métriques dépendent de l'application. Dans notre étude, nous traitons le temps d'exécution et le facteur de gain (en anglais *speedup*).

Notation 1 *Soit t_0 le temps de la résolution séquentielle et t le temps de résolution en parallèle. On note $su = \frac{t_0}{t}$ le facteur de gain par rapport au temps de résolution séquentielle.*

Le facteur de gain est une mesure indiquant le nombre de fois l'algorithme parallèle est plus rapide que l'algorithme séquentiel. Le facteur de gain maximal possible de la parallélisation d'un programme est défini par la loi d'Amdahl [Amdahl 1967]. Celle-ci établit qu'une partie du programme non parallélisable limite l'accélération globale possible lors de la parallélisation Par exemple, si le programme contient (en temps d'exécution) la moitié de code non parallélisable, le facteur de gain maximum est de 2. Si le code est à 90% parallélisable, le facteur de gain maximal est de 10. Soit $B \in [0, 1]$, le taux représentant la partie non parallélisable d'un algorithme. On note w le nombre de *workers*. Soit t(w) le temps d'exécution d'un algorithme sur w *workers*. Il correspond à : $t(w) = t(1)\left(B + \frac{1}{n}(1 - B)\right)$ Le facteur de gain théorique $su(c)$ est donc de :

$$su(w) = \frac{1}{B + \frac{1}{n}(1 - B)}$$

Dans le cas idéal, on dit que le facteur de gain est linéaire si $su = k * w$ avec k proche de 1, c'est-à-dire qu'on a divisé le temps de résolution séquentielle par le nombre de *workers* avec la résolution en parallèle. On dit qu'il est supra-linéaire si $su > w$, on parle alors d'hyper-accélération. Cela peut se produire par exemple lors de la recherche de la première solution trouvée dans l'arbre de recherche.

2.2.3 Communication

Lorsqu'on choisit de faire de la concurrence compétitive ou coopérative, on utilise la communication entre les unités de calcul. C'est un mécanisme complexe à mettre en œuvre, qui a les inconvénients suivants :

- c'est particulièrement lent, surtout en comparaison avec des calculs internes ;
- cela demande de suivre des protocoles complexes ;
- cela implique bien souvent de mettre en œuvre des principes comme l'exclusion mutuelle. Les unités de calcul peuvent accéder à des données communes et les modifier. Il faut ainsi définir une section critique pour assurer la cohérence des données en utilisant les mécanismes de verrous et d'exclusions mutuelles. L'exclusion mutuelle permet à un *worker* de verrouiller certaines ressources pour y obtenir un accès exclusif ; tout *worker* qui tentera d'accéder à ces ressources alors qu'elles sont verrouillées sera bloqué jusqu'à ce que le *worker* en question la libère.

2.2.4 Répartition de charge

La charge d'un *worker* correspond à sa quantité de travail, mesurée comme un temps de calcul. Les performances d'un programme parallèle dépendent fortement de la qualité de la répartition de charge, car le temps d'exécution global est égal à celui du plus lent des *workers*. Un bon équilibrage des charges sera réalisé si chaque worker termine ses travaux à un delta temporel près. Il est difficile de bien équilibrer les charges de travail. On peut décomposer la tâche d'origine en sous-tâches statiquement ou dynamiquement. La répartition statique est faite *a priori* et il est très difficile de trouver une bonne répartition dans ce cas, car connaître à l'avance le temps de calcul n'est pas une mince affaire. Donc, faire une répartition dynamique durant la résolution permet de résoudre ce problème. Cependant, la répartition dynamique pose des problèmes comme la famine. La famine survient lorsqu'un *worker* est exclu de l'accès aux ressources (mémoire, CPU, etc.). Les figures 2.5 et 2.6 illustrent respectivement une bonne et une mauvaise répartition.

2.3 Utilisation du parallélisme en PPC

2.3.1 CSP distribués

Un CSP distribué est un CSP dans lequel les variables et les contraintes sont distribuées entre plusieurs agents, c'est-à-dire plusieurs *workers*. Le problème consiste alors à trouver une combinaison consistante d'actions des agents. Différents modèles de résolution de ces CSP ont été récemment proposés. Les agents s'échangent des messages pour informer les différentes consistances locales trouvées. Ainsi, l'une des difficultés majeures des CSP distribués est de constituer une solution globale à partir de solutions locales. Il y a également une autre difficulté dans la modélisation des CSP distribués qui est la définition des sous-problèmes et l'affectation de ces sous-problèmes aux différents agents.

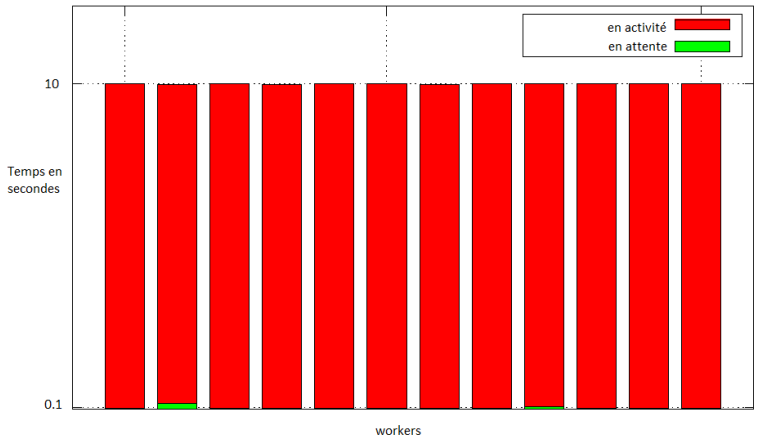

FIGURE 2.5 – Un exemple concret d'une bonne répartition de charge, tous les *workers* travaillent

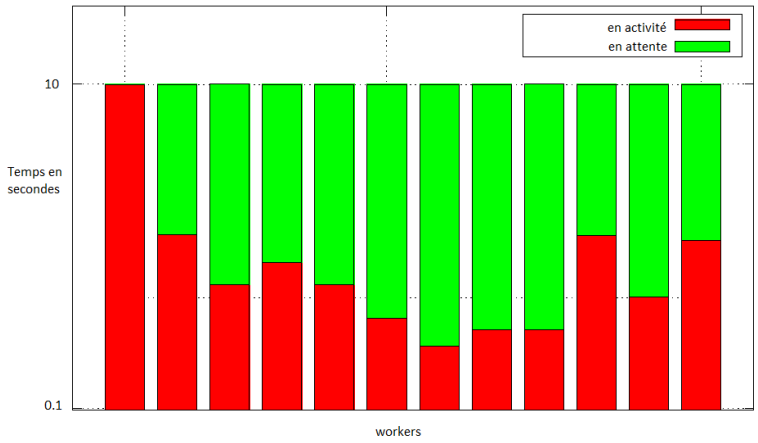

FIGURE 2.6 – Un exemple concret d'une mauvaise répartition de charge, un *worker* travaille en permanence, les autres *workers* sont en attente

Pour cela, le premier *framework* sur les CSP distribués introduit par [Yokoo 1990] permet de découper naturellement le problème entre les agents, par exemple pour des raisons de confidentialité. D'autres *frameworks* CSP distribués ont été proposés tels que [Hirayama 1997, Chong 2006, Ezzahir 2007, Léauté 2009, Wahbi 2011].

2.3.2 Méthode du portefeuille

La méthode du portefeuille en PPC consiste à exécuter en parallèle plusieurs stratégies de recherche différentes sur le même espace de recherche. Lorsqu'on utilise une seule stratégie de recherche, le risque d'avoir des cas pathologiques est grand. Ainsi, la méthode du portefeuille en PPC limite ce risque et permet rapidement de trouver une solution en exploitant les performances des autres différentes stratégies de recherche. Au départ, la méthode

du portefeuille a été utilisée par les premiers algorithmes *Las Vegas* qui ont commencé à exploiter le parallélisme en raisonnant sur la structure du problème, par exemple l'utilisation de différents algorithmes ou le réglage de plusieurs paramètres (*tuning parameters* en anglais). Cette idée a également été exploitée dans un contexte non parallèle [Gomes 2000]. Aucune communication n'est requise et un excellent niveau de l'équilibrage de charge est atteint (ils ont tous le même espace de recherche). Même si cette approche entraîne un niveau élevé de redondances entre les unités de calcul, elle donne de bonnes performances, car elle prend le temps minimum de résolution des différentes stratégies utilisées. L'approche a été grandement améliorée en utilisant des redémarrages aléatoires de parcours de l'arbre de recherche [Luby 1993]. Cependant, l'efficacité de la méthode du portefeuille est limitée par l'utilisation de stratégies qui ont des temps de résolution similaires (appelées stratégies non orthogonales). Certains auteurs visent à accroître l'efficacité de la méthode en permettant aux unités de calcul de partager des informations apprises au cours de la recherche [Hamadi 2009]. Le défi de la méthode du portefeuille est de trouver différents points de vue orthogonaux afin de fournir des performances complémentaires.

2.3.3 Décomposition de l'arbre de recherche

Il existe deux méthodes de décomposition d'un arbre de recherche en parallélisme :

- la décomposition statique ;
- la décomposition dynamique.

La première génère au début de la résolution l'ensemble des sous-problèmes à traiter alors que la seconde génère des sous-problèmes durant la résolution si besoin est.

Dans un premier temps, nous rappelons quelques concepts des arbres de recherche [Perron 1999] qui sont la base des procédures de décomposition que nous allons introduire plus tard.

Nœuds ouverts et expansion de nœuds Lors d'une résolution, les nœuds de l'arbre de recherche peuvent être dans 3 états différents : ouvert, fermé et non exploré. Ces états ont les propriétés suivantes :

- Tous les nœuds qui sont les ancêtres d'un nœud ouvert sont fermés.
- Chaque nœud inexploré a exactement un nœud ouvert comme ancêtre.
- Aucun nœud fermé n'a un nœud ouvert comme ancêtre.

L'ensemble des nœuds ouverts est appelé *frontière de l'arbre de recherche*. La figure 2.7 illustre ce concept.

La *frontière de l'arbre de recherche* évolue simplement à travers un processus connu comme l'expansion de nœud. Il supprime un nœud ouvert de la frontière, ferme ce nœud (état fermé), et ajoute les nœuds enfants inexplorés de ce nœud de la frontière. L'expansion de nœud est la seule opération qui se produit lors de la recherche.
La frontière de recherche est une décomposition *non triviale* du problème où chaque nœud

ouvert est associé à un sous-problème. La décomposition est *efficace* si et seulement si la stratégie de recherche est efficace. Nous remarquons qu'une décomposition basée sur des affectations peut être considérée comme un cas particulier de la décomposition avec les nœuds.

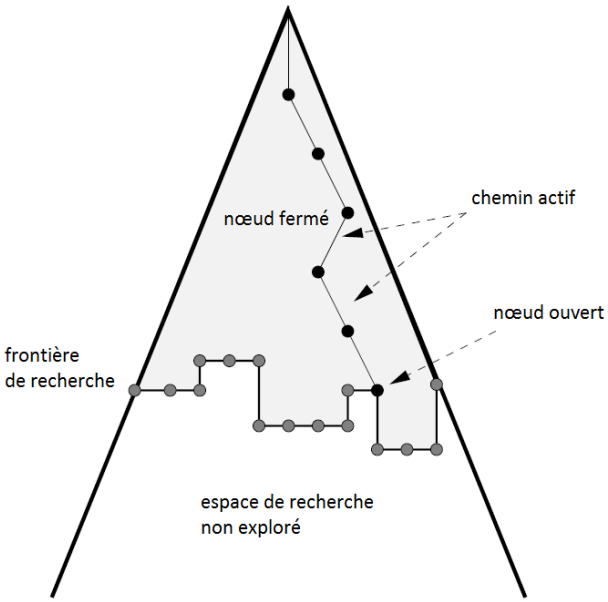

FIGURE 2.7 – Présentation de la frontière dans un arbre de recherche

Chemins actifs et sauts dans l'arbre de recherche Faire une expansion d'un nœud peut exiger un changement d'état des variables (les domaines des variables du problème). Cependant, d'un point de vue de la consommation de la mémoire, il est irréaliste de maintenir l'état des domaines des variables associé à tous les nœuds ouverts de l'arbre de recherche. Par conséquent, le *worker* chargé d'explorer un nœud ouvert doit reconstruire l'état des domaines des variables associé au nœud ouvert. On utilise pour cela un chemin actif et une opération de saut.
Lorsqu'on descend dans l'arbre de recherche, on construit un chemin actif. Un chemin actif est la liste des ancêtres du nœud ouvert courant, comme illustré dans la figure 2.7. Lorsque la procédure de recherche se déplace d'un nœud à un autre, elle effectue une opération de saut.

Exemple de décomposition statique : la décomposition statique simple

La méthode de décomposition statique simple consiste à découper ce problème en autant de sous-problèmes disjoints que l'on dispose de *workers* de façon à donner exactement un sous-problème par *worker*. Une solution d'un sous-problème étant une solution du problème initial, car les sous-problèmes sont disjoints. On recueille ensuite l'ensemble des solutions calculées par les *workers* pour obtenir l'ensemble des solutions du problème initial.

L'avantage de cette méthode est sa simplicité. Un inconvénient majeur survenant fréquemment dans la pratique concerne la répartition de charge souvent mal équilibrée, car le temps de résolution de chaque sous-problème n'est pas homogène. En effet, l'homogénéité de la charge dépend directement de l'homogénéité du découpage. Pour un meilleur équilibrage de charge, certains travaux ont été réalisés sur la décomposition de l'arbre de recherche [Knuth 1975, Cornuéjols 2006, Kilby 2006]. Toutefois, la taille des arbres est approximative et n'est pas strictement corrélée avec le temps de résolution. Ainsi, comme mentionné par [Bordeaux 2009], il est assez difficile de s'assurer que chaque *worker* recevra la même quantité de travail. Afin de remédier à ces problèmes, une approche dynamique a été popularisée : l'idée du *work stealing*.

Exemple de décomposition dynamique : *work stealing*

L'approche du *work stealing* ("vol de ressources") a été proposée à l'origine par [Burton 1981] et implémentée pour la première fois sur une machine parallèle Lisp [Halstead 1984].

Habituellement, il est mis en œuvre comme suit : quand un *worker* w_1 n'a plus de travail, il demande du travail à un autre *worker* w_2. Si la réponse est positive, alors w_2 découpe le problème qu'il résout actuellement en deux sous-problèmes et en donne un à w_1. Dans le cas contraire, w_1 demande du travail à d'autres *workers*, jusqu'à obtenir une réponse positive.

Le *work stealing* résout en partie le problème d'équilibrage de la méthode de décomposition statique simple, en introduisant une décomposition dynamique ; par conséquent, il n'est pas nécessaire d'être en mesure de découper un problème en plusieurs parties équilibrées.

Il faut cependant noter que tous les sous-problèmes donnés à un *worker* ne sont pas égaux. Quand un *worker* est en famine, il doit éviter de voler trop de sous-problèmes faciles à résoudre, car dans ce cas, il devra demander un autre travail presque immédiatement. Cela se produit fréquemment à la fin de la résolution, lorsqu'un grand nombre de *workers* n'ont plus de travail, et demandent en continu des sous-problèmes à traiter, ce qui augmente le temps de communication entre *workers* et de fait ralentit la fin de la résolution. Ainsi, pour un petit nombre de *workers*, on obtient de bonnes performances. Cependant, il est difficile de maintenir un gain linéaire, lorsque le nombre de *workers* devient plus grand.

Un autre inconvénient du *work stealing* qui n'est pas souvent mentionné est que sa mise en œuvre est intrusive. Elle est ainsi fortement dépendante du solveur et oblige d'avoir une très bonne connaissance des fonctions internes et d'y avoir accès.

Il existe des méthodes qui tentent de répondre à ce problème comme le *framework* `Bobpp` [Le Cun 2007]. `Bobpp` fournit une interface entre les solveurs et les machines parallèles. La figure 2.8 montre son fonctionnement. Le *framework* initialise une file d'attente globale de sous-problèmes appelée *GPQ* (de l'anglais *Global Priority Queue*), par laquelle les *workers* ont un accès par exclusion mutuelle. L'arbre de recherche est découpé et réparti entre les différents *workers*. Périodiquement, chaque *workers* qui est en train de résoudre un sous-problème, vérifie s'il y a présence de *workers* en attente. Si c'est le cas, le *worker* arrête la résolution en cours et donne une partie de son travail à effectuer en construisant

bob-node qui contient la tâche correspondante. Généralement, cela correspond au parcours de l'arbre de recherche à cet instant en réfutant la décision en cours. Ce *bob-node* est inséré dans la file *GPQ* puis le *worker* poursuit ce qu'il lui reste à résoudre. Les *workers* inactifs sont notifiés par l'insertion d'un nouveau *bob-node* dans *GPQ*. Le premier *worker* inactif qui a accès à la file d'attente *GPQ* prend le *bob-node* disponible et commence à le résoudre.

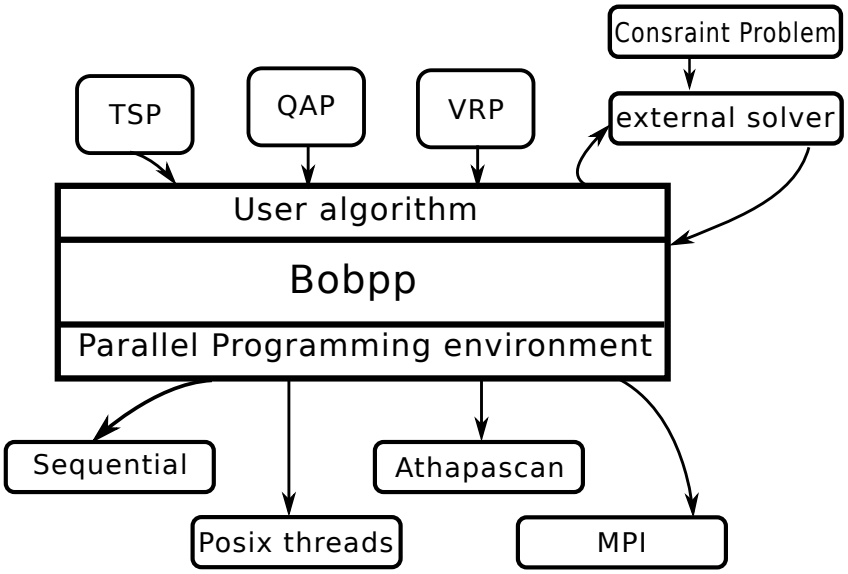

FIGURE 2.8 – Schéma du fonctionnement `Bobpp`

La méthode du *work stealing* a été implémentée sur plusieurs solveurs (`Comet` [Michel 2009] ou `ILOG Solver` [Perron 1999] par exemple) et de différentes façons [Schulte 2000, Jaffar 2004, Zoeteweij 2004, Chu 2009] selon la centralisation du travail, la décomposition de l'arbre de recherche ou encore selon la méthode de communication entre les *workers*.

Certaines méthodes [Xie 2010, Jaffar 2004] ont été développées pour tenter d'améliorer la répartition de charge. Dans [Xie 2010], les auteurs proposent d'utiliser une approche impliquant plusieurs *masters* et *workers*. Chaque *master* gère ses *workers*. L'espace de recherche est divisé en différents sous-arbres de recherche entre les différents *masters*, et chaque *master* met ses sous-arbres de recherche dans une queue qui seront distribués aux *workers*. Lorsqu'un sous-arbre est détecté difficile à parcourir, les *workers* découpent un grand nombre de sous-arbres et les ajoutent dans la queue du *master*. Dans [Jaffar 2004], les auteurs expérimentent sur 64 cœurs en utilisant le *work stealing*. Un *master* centralise toutes les informations (bornes, solutions et requêtes). Le *master* estime quel *worker* possède la plus grande quantité de travail afin de redistribuer le travail aux *workers* en attente.

2.4 Discussion

La répartition de charge est fortement dépendante de la décomposition du problème. La décomposition statique simple permet de mettre en place facilement une solution parallèle, mais ne traite pas le problème de la répartition de charge, ce qui met la plupart des *worker* en attente de la fin de la résolution. La décomposition dynamique par la méthode du *work stealing* remédie à ce problème en décomposant les sous-problèmes difficiles durant la résolution. Néanmoins, le *work stealing* pose des problèmes lorsqu'on augmente le nombre de *workers*, car il a besoin de plus en plus de communication.

CHAPITRE 3

Méthode EPS

Sommaire

3.1 Introduction à EPS

Nous rappelons d'abord dans ce chapitre les principes de base du modèle *Embarrassingly Parallel* (EP) [Wilkinson 2005]. Puis nous montrons comment nous avons adapté le modèle EP pour développer la méthode *Embarrassingly Parallel Search* (EPS). Les points clés de cette adaptation sont :

- une décomposition du problème en un grand nombre de sous-problèmes,
- une répartition dynamique des sous-problèmes aux *workers*, c'est-à-dire une fois qu'un sous-problème est résolu, le *worker* piochera dans la liste des sous-problèmes disponibles pour résoudre à nouveau.

Nous verrons que la répartition dynamique des sous-problèmes est simple à expliquer, mais que le difficulté de la méthode réside essentiellement sur la génération des sous-problèmes, à savoir, la décomposition. Nous proposons 4 différentes décompositions et nous les comparerons.

Les 3 premières méthodes sont exactes et statiques, c'est-à-dire qu'elles génèrent le nombre de sous-problèmes fixé au départ et elles génèrent toujours les mêmes sous-problèmes. Elles utilisent le même procédé pour estimer la profondeur de l'arbre de recherche à partir de laquelle elles généreront les sous-problèmes. Un sous-problème est caractérisé par un ensemble d'affectations. Par exemple, soit un problème avec 3 variables x, y et z dont leur domaines sont constitués des valeurs a, b et c. On peut avoir les sous-problèmes suivants : $\{(x, a), (y, b), (z, b)\}$, $\{(x, a), (y, b), (z, c)\}$ et $\{(x, b), (y, a), (z, c)\}$.

Pour estimer la profondeur de l'arbre de recherche, les 3 premières décompositions fixent l'ordre des variables et utilisent le produit cartésien des domaines courants pour estimer la profondeur de l'arbre de la recherche. Cette profondeur correspond au nombre de variables à partir desquelles les 3 décompositions vont générer l'ensemble des affectations possibles, c'est-à-dire des sous-problèmes. La première décomposition est séquentielle et les deux autres sont des variantes parallèles de la première. Ces variantes parallèles ont été élaborées, car la décomposition séquentielle peut prendre un temps considérable lorsqu'on souhaite générer un très grand nombre de sous-problèmes.

La dernière décomposition est une décomposition dynamique séquentielle, basée sur une estimation de la profondeur proposée par [Cornuéjols 2006]. Elle relâche le problème de la décomposition en ne générant pas exactement le nombre de sous-problèmes souhaité afin de les servir plus rapidement aux *workers* dès qu'un sous-problème est généré.

3.1.1 Modèle Embarrassingly Parallel

Le modèle *Embarrassingly Parallel* [Wilkinson 2005] permet de diviser le problème à résoudre en un certain nombre de tâches indépendantes, ce qui permet aux *workers* d'exécuter leurs tâches avec très peu, voire aucune communication.

Certaines applications bien connues sont basées sur des calculs *embarrassingly parallel*, comme les projets `BOINC`, `Seti@home`, `Distributed.net` et `Sharcnet` ou le traitement d'image de bas niveau ainsi que l'ensemble de Mandelbrot (Fractales) et les calculs de Monte Carlo [Wilkinson 2005].

On notera qu'un autre avantage intéressant du modèle *Embarrassingly Parallel* est la possibilité d'observer une résolution en parallèle en sauvegardant l'ordre d'exécution des tâches, ce qui est très utile dans le débogage d'applications. Les applications utilisant le modèle *Embarrassingly Parallel* sont nombreuses comme :

- les bases de données distribuées ;
- les serveurs de fichiers statiques pour plusieurs utilisateurs ;
- les rendus des images en infographie. Dans l'animation assistée par ordinateur, chaque image peut être rendue indépendamment ;
- les simulations informatiques comparant plusieurs scénarios indépendants, tels que les modèles climatiques ;
- les recherches par brute-force en cryptographie.

Comme le montre la figure 3.1, la performance d'un algorithme *embarrassingly parallel* dépend directement de la qualité de la répartition de charge. Une mauvaise affectation des tâches indépendantes à un ensemble de *workers* conduit à une mauvaise répartition de charge alors qu'une bonne affectation de ces tâches permet de mieux répartir la charge.

En partant d'un problème que l'on va diviser en un grand nombre de tâches indépendantes pour lesquelles on ne connaît pas à l'avance leur temps de résolution, le modèle EP

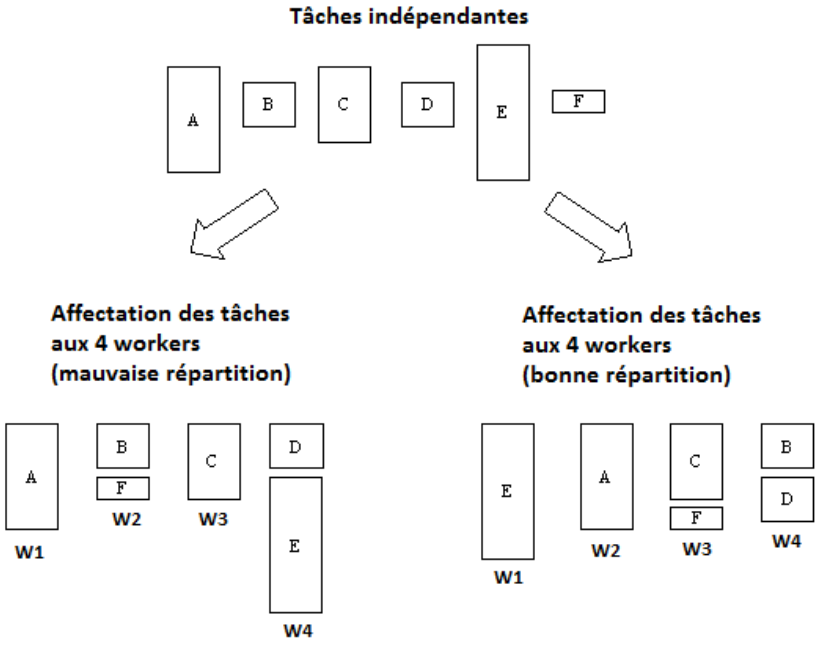

FIGURE 3.1 – Exemple d'une affectation de tâches aux *workers* menant à une mauvaise et à une bonne répartition de charge.

amène de manière statistique à un équilibrage des temps de résolution par *worker*, conduisant à une bonne répartition de charge.

D'après [Wilkinson 2005], un algorithme qui utilise le modèle EP est *embarrassingly parallel*. Un algorithme EP se décompose en 3 parties :

1. TaskDefinition : la définition des tâches ;
2. TaskAssignment : la répartition de ces tâches aux *workers* ;
3. TaskResultGathering : la collecte des résultats.

La première partie, TaskDefinition, dépend du problème à traiter, car les tâches sont définies en fonction de la structure interne du problème, tandis que la seconde, TaskAssignment, est générique : on pourra utiliser au choix une répartition statique ou dynamique. Lors d'une répartition statique, on affecte au début de la résolution l'ensemble des tâches indépendantes aux *workers*. *A contrario*, lors d'une répartition dynamique, une liste de tâches indépendantes à traiter est maintenue, dans laquelle les *workers* vont venir prendre du travail ; éventuellement, ils peuvent ajouter de nouvelles tâches dans cette liste en découpant leurs tâches courantes.

Finalement, la dernière partie, *i.e.* TaskResultGathering, permet de définir la manière de collecter les résultats de chacune des tâches résolues par les *workers*, qui dépend notamment de la recherche d'une ou plusieurs solutions.

3.1.2 Embarrassingly Parallel Search

Dans cette section, nous introduisons la méthode EPS, qui est une méthode de résolution de problèmes en PPC utilisant le modèle EP.

Deux étapes définissent la méthode EPS :

1. la décomposition du problème en sous-problèmes ;
2. la résolution des sous-problèmes par les *workers*.

Pendant la première étape, on va s'intéresser à la définition des tâches alors que pendant la seconde, on va s'intéresser à l'étape TaskResultGathering.

De manière informelle, le principe est le suivant : nous avons vu que lors d'une décomposition statique simple avec k *workers*, on découpait le problème en k sous-problèmes équivalents ; ici, nous proposons de découper le problème en un grand nombre de sous-problèmes, par exemple $30k$ sous-problèmes, puis de donner de façon dynamique ces sous-problèmes aux *workers* lorsqu'ils sont inactifs. On espère ainsi arriver à un équilibrage des temps de résolution par *worker*, comme énoncé dans le modèle EP.

Cependant, la décomposition en sous-problèmes doit être faite avec soin. En effet, nous ne devons pas générer de sous-problèmes où, après la propagation, il existe au moins une variable dont le domaine est vide, c'est-à-dire nous devons générer seulement les sous-problèmes consistants avec la propagation. Ici, il faut noter que la génération des sous-problèmes consistants avec la propagation respecte un principe du parallélisme stipulant qu'il faut éviter de faire en parallèle ce que l'on ne fait pas en séquentiel, car sinon on aurait du travail inutile à donner aux *workers*.

On peut envisager deux types de décomposition : la décomposition séquentielle et la décomposition parallèle. Cependant, selon le type de décomposition choisi (séquentielle ou parallèle), TaskAssignment sera effectué dans les deux étapes ou seulement dans la deuxième étape.

Dans les parties suivantes, nous verrons comment les trois parties (TaskDefinition, TaskAssignment et TaskResultGathering) peuvent être définies de manière à être en mesure de lancer la recherche en parallèle et de manière efficace.

3.2 Principes de la méthode EPS

Pour être bien équilibrés, les sous-problèmes doivent être équivalents en temps de résolution, or on ne sait pas faire cela. Précédemment, nous avons vu que la décomposition d'un problème initial en k sous-problèmes, traités par k *workers*, peut mener à un mauvais équilibrage de charge. Notre idée consiste à augmenter fortement le nombre de sous-problèmes considérés, afin de définir un calcul *embarrassingly parallel*, dont le but est d'obtenir de bonnes performances. Dans notre approche, nous nous basons sur la remarque suivante :

Remarque : *Les temps d'activité de tous les workers peuvent être bien équilibrés, même si le temps de résolution de chaque sous-problème n'est pas bien équilibré.*

Exemple : Considérons un problème dont le temps de résolution est de 100 secondes. Ce problème peut, par exemple, être découpé en 8 sous-problèmes ayant des temps de résolution différents : $25, 20, 15, 10, 10, 10, 5$ et 5 secondes. Si on disposait de 4 *workers*, une répartition de charge pourrait être la suivante :

1. "worker1" : 25 ;
2. "worker2" : $20 + 5$;
3. "worker3" : $15 + 10$;
4. "worker4" : $10 + 10 + 5$

Cette répartition de charge serait alors bien équilibrée bien que les temps de résolution de chaque sous-problème ne le soient pas.

Avant d'entrer dans les détails de mise en œuvre, nous donnons certaines définitions. Durant la résolution d'un problème, nous appelons :

1. le *temps d'activité d'un worker* la somme des temps de résolution d'un *worker*.
2. le *temps d'inactivité d'un worker* la différence entre le temps écoulé pour résoudre le problème en entier et le temps d'activité dudit *worker*.

On appellera respectivement un *worker* actif ou inactif, s'il est en train ou non de résoudre ou non un sous-problème.

Propriété 1 Si un problème est découpé en plusieurs sous-problèmes dont le temps maximal de résolution est *tmax*, alors

1. Le temps minimal de résolution du problème est *tmax* ;
2. Le temps maximal d'inactivité d'un *worker* est inférieur ou égal à *tmax*.

Preuve 1 Supposons qu'il existe un *worker* i dont le temps d'inactivité soit supérieur à *tmax*. Considérons le moment où i est inactif. À ce moment, il n'y a plus de sous-problèmes disponibles à résoudre, autrement i aurait été actif. De plus, un autre *worker* j était déjà en charge du sous-problème le plus long à résoudre ou bien ce dernier a déjà été résolu. Ainsi, la durée maximale de la résolution de *worker* j est inférieure à *tmax*.

La principale difficulté d'une décomposition n'est donc pas de définir des problèmes équivalents en temps de résolution, mais d'éviter d'avoir des *workers* inactifs. De plus, il n'est pas nécessaire de connaître à l'avance le temps de résolution de chaque sous-problème. Nous espérons juste que les *workers* aient des temps d'activité équivalents. Pour atteindre cet objectif, nous proposons de décomposer le problème initial en un grand nombre de sous-problèmes pour les raisons suivantes :

1. L'augmentation du nombre de sous-problèmes augmente les chances d'obtenir des temps d'activité bien équilibrés pour chacun des *workers* (la probabilité que la somme des temps de résolution des sous-problèmes par *worker* soit équivalente étant plus élevée).

2. Dans le cas où l'arbre de recherche tend à ne pas être équilibré, les sous-problèmes prendront plus de temps à être résolus ; augmenter le nombre de sous-problèmes revient à augmenter la probabilité de les découper en plusieurs parties ayant des temps de résolution comparables et ainsi obtenir une bonne répartition de charge.

3. Dans le cas où un sous-problème a un temps de résolution plus long que n'importe quel autre sous-problème, il est difficile d'obtenir une répartition de charge équilibrée. La décomposition de ce sous-problème va donc mieux répartir la charge en la distribuant à d'autres *workers*.

Par exemple, considérons un problème dont la résolution est de 140 secondes et pour lequel nous disposons de 4 *workers*. Une décomposition statique simple génère 4 sous-problèmes avec les temps de résolution suivants : 20, 80, 20 et 20 secondes. Ainsi, en parallèle, 80 secondes seront nécessaires pour résoudre ces sous-problèmes alors qu'une résolution séquentielle aurait nécessité 140 secondes. Par conséquent, nous obtenons un facteur de gain de $140/80 = 1,75$. À présent, en séparant à nouveau chaque sous-problème en 4 sous-problèmes, nous pourrions obtenir les sous-problèmes suivants représentés par leurs temps de résolution en secondes : $(5 + 5 + 5 + 5) + (20, 10, 10, 40) + (2, 5, 10, 3) + (2, 2, 8, 8)$. Les temps de résolution sont regroupés en référence à la décomposition statique simple (chaque sous-problème est décomposé à nouveau). Dans ce cas, une affectation possible des sous-problèmes aux *workers* pourrait être la suivante :

- "worker1" : $5 + 20 + 2 + 8 = 35$;
- "worker2" : $5 + 10 + 2 + 10 = 27$;
- "worker3" : $5 + 10 + 5 + 3 + 2 + 8 = 33$;
- "worker4" : $5 + 40 = 45$.

Le temps de résolution total est désormais de 45 secondes et le facteur de gain de $140/45 = 3,1$. En découpant encore les sous-problèmes, nous réduirons leur temps de résolution moyen et espérons couper le plus long sous-problème (40 secondes) en plusieurs sous-problèmes plus courts. Ainsi, en découpant les sous-problèmes les plus longs (difficiles à résoudre), on diminue le temps total de résolution, car il y aura une meilleure distribution des sous-problèmes.

3.2.1 Décomposition en sous-problèmes

La décomposition dans la méthode EPS est principalement responsable de la qualité de la répartition de charge. Nous proposons donc deux types de décomposition : la décomposition statique et la décomposition dynamique. La décomposition statique permet de décomposer le problème en un nombre de sous-problèmes fixé à l'avance. Elle est statique, c'est-à-dire qu'elle décompose le problème sur un sous-ensemble de ses variables et elle est également exacte, car elle génère le nombre de sous-problèmes souhaités en parcourant intégralement l'arbre de recherche à une profondeur estimée. Si elle n'atteint pas le bon

nombre de sous-problèmes, elle décomposera à nouveau à une profondeur plus élevée jusqu'à atteindre le nombre de sous-problèmes souhaité. Nous utilisons seulement la stratégie de recherche `lex` dans le parcours de l'arbre de recherche. Nous proposons trois différents algorithmes de cette méthode de décomposition : la première est séquentielle et les deux autres parallèles. Quant à la décomposition dynamique, elle génère les sous-problèmes en parcourant l'arbre de recherche avec n'importe quelle stratégie de recherche, contrairement à la décomposition statique. Cependant, elle n'est pas exacte, car elle ne génère pas le nombre de sous-problèmes souhaité. En effet, elle estime *a priori* la profondeur de l'arbre à laquelle elle s'arrête pour générer les sous-problèmes sans connaitre exactement le nombre de sous-problèmes. L'avantage de la décomposition dynamique est qu'elle permet de ne pas attendre la génération de tous les sous-problèmes pour commencer la résolution, le premier sous-problème généré est directement mis à disposition des *workers*.

3.2.1.1 Décomposition statique

Principes de la décomposition statique

Soit un problème P que l'on divise en q sous-problèmes avec :

1. $\{x_1, \ldots, x_n\}$ un ensemble ordonné de n variables
2. A_k le produit cartésien $D(x_1) \times \ \ldots \ \times D(x_k)$, $k \leq n$
3. $|A_k|$ représente la cardinalité de A_k

On calcule la valeur k telle que $|A_{k-1}| < q \leq |A_k|$, où $|A_{k-1}|$ et $|A_k|$ représentent respectivement les cardinalités de A_{k-1} et A_k.

Chaque tuple de A_k définit un sous-problème et donc A_k, qui contient l'ensemble des sous-problèmes, est la décomposition recherchée de P en q sous-problèmes.

Afin de générer q sous-problèmes, on peut utiliser l'algorithme suivant, nommé SIMPLEDECOMPOSITION.

Algorithm 1: SimpleDecomposition

```
1 SIMPLEDECOMPOSITION(N, q)
  N : un réseau de contraintes, q : le nombre de sous-problèmes à générer
2   calculer la valeur k tel que |A_{k-1}| < q ≤ |A_k| ;
3   générer tous les tuples du produit cartésien des domaines de x_1 à x_k ;
4   répartir les tuples afin d'obtenir q sous-problèmes et les mettre dans S ;
5   retourner (S,k) ;
```

On notera que lors de l'étape 4, on s'autorise à regrouper les tuples dans un même sous-problème si seulement le nombre de tuples est conséquent.
Cette méthode, qui consiste à utiliser le produit cartésien, obtient un bon facteur de gain pour certains problèmes comme les n-reines ou la règle de *Golomb* ; en revanche, le facteur de gain est très mauvais pour des problèmes où un grand nombre de tuples de A_k est trivialement inconsistant.

Considérons, par exemple, trois variables x_1, x_2 et x_3 qui possèdent trois valeurs $\{a, b, c\}$ dans leurs domaines ; on applique une contrainte `alldiff` [Régin 1994] sur ces variables. Le produit cartésien des domaines des variables contient 27 tuples. Parmi eux, seulement 6 (les permutations de $\{a, b, c\}$) sont consistants avec la contrainte `alldiff` : $\{(a, b, c), (a, c, b), (b, a, c),(b, c, a),(c, a, b), (c, b, a)\}$.

Ainsi, seulement $6/27$ soit $2/9$ des problèmes générés ne sont pas trivialement inconsistants. Il est important de noter que la plupart des problèmes inconsistants pourraient ne jamais être considérés lors d'une résolution séquentielle. Pour plusieurs problèmes, nous observons que plus de 99% des problèmes générés sont inconsistants après avoir lancé le mécanisme de propagation. La méthode que nous présentons nous permet donc de **générer uniquement les sous-problèmes qui ne sont pas détectés inconsistants avec la propagation**, appelés sous-problèmes consistants avec la propagation.

La génération de sous-problèmes consistants avec la propagation est plus complexe, car le nombre de ces sous-problèmes peut ne pas être égal au nombre de tuples constituant le produit cartésien de certains domaines. La réduction des domaines par l'utilisation de la propagation ne donne aucune information sur le nombre de tuples valides après application de la propagation.

Pour générer ces sous-problèmes consistants avec la propagation, on parcourt l'arbre de recherche niveau par niveau. Un algorithme simple pourrait consister à effectuer un parcours en largeur (BFS), jusqu'au moment où le nombre q de sous-problèmes consistants avec la propagation est atteint. Malheureusement, exécuter une BFS sur un arbre de recherche a une complexité en mémoire exponentielle, car il faut stocker tous les nœuds de l'arbre à chaque profondeur. Par ailleurs, un parcours en profondeur (DFS) ne permet pas d'avoir une bonne estimation sur le reste de l'arbre à parcourir.

Par conséquent, nous proposons d'utiliser un parcours en profondeur borné, DBDFS (de l'anglais *Depth-bounded Depth First Search*), qui se comporte comme une DFS ne visitant jamais les nœuds au-delà d'une profondeur donnée.

Notation 2 *Soit C_k l'ensemble des tuples consistants avec la propagation à une profondeur k. Pour générer q sous-problèmes, nous exécutons la DBDFS jusqu'à atteindre la profondeur k telle que $|C_{k-1}| < q \leq |C_k|$.*

Décomposition statique séquentielle

Une première mise en œuvre d'une décomposition utilisant la DBDFS se fait de manière séquentielle :

- il faut essayer d'estimer une bonne valeur pour k pour éviter de répéter inutilement l'algorithme DBDFS en utilisant le produit cartésien des domaines des variables. **Exemple :** Soit une profondeur k' et $q/1000$ sous-problèmes générés après avoir effectué une DBDFS à la profondeur k'. Soit $|D(x_{k'+1})| = |D(x_{k'+2})| = |D(x_{k'+3})| = 10$. Le produit cartésien des domaines des variables entre la profondeur $k'+1$ et la profondeur $k'+3$ est donc égal à 1000. Comme on veut générer q sous-problèmes, il faut donc exécuter la DBDFS jusqu'à la profondeur $k'+3 = k$.

- afin d'éviter de répéter inutilement l'exploration de l'arbre sur les variables précédemment explorées, nous sauvegardons les tuples calculés dans une contrainte de table. Plus précisément, nous utilisons une contrainte de table contenant C_k pour calculer C_l avec $l > k$.

Notation 3 *Soit Q un problème, nous notons $D(Q, x)$ le domaine résultant de la variable x après application du mécanisme de propagation sur Q.*

Nous introduisons l'algorithme 2 qui nous permet de faire une décomposition séquentielle. La fonction COMPUTEDEPTH permet de calculer la profondeur d telle que le nombre de tuples constituant le produit cartésien des domaines des variables de x_1 à x_d soit égal ou supérieur au nombre q sous-problèmes. La fonction GETDOMAINS permet de récupérer l'ensemble des domaines courants de toutes les variables après application du mécanisme de propagation sur S. La fonction GENERATESUBPROBLEMS permet de lancer une DBDFS à la profondeur d et récupère l'ensemble des tuples valides d'arité d qui constituent l'ensemble des sous-problèmes consistants avec la propagation. Dans le cas où on ne génère pas de sous-problèmes (ligne 14), on renvoie une liste de sous-problèmes vide et la résolution du problème est terminée.

Algorithm 2: SEQUENTIALDECOMPOSITION

1 COMPUTEDEPTH($\mathcal{N}, cardS, \delta, q$)
2 renvoyer d tel que $cardS \times |D(x_{\delta+1})| \times \ldots \times |D(x_{d-1})| < qetq \leq cardS \times |D(x_{\delta+1})| \times \ldots \times |D(x_{d-1})| \times |D(x_d)|$;
3 GETDOMAINS(S)
4 renvoyer l'ensemble des domaines de $\mathcal{D} = \{D(S, x_1), D(S, x_2), \ldots, D(S, x_n)\}$ tel que $\forall x \in \mathcal{X}\ D(S, x) = \cup_{P \in S} D(P, x)$;
5 GENERATESUBPROBLEMS($\mathcal{N}, S, d$)
6 lancer une recherche de solutions basée sur une DBDFS avec d la profondeur limite sur le réseau de contrainte formé par $\mathcal{N}$ et la table de contraintes définie à partir des éléments de S;
7 renvoyer l'ensemble des tuples valides d'arité d;
8 SEQUENTIALDECOMPOSITION($\mathcal{N}$, q)
9 $\mathcal{N}$ est un réseau de contraintes ; q le nombre de sous-problèmes à générer;
10 $S \leftarrow \emptyset$; $d \leftarrow 0$;
11 **while** $|S| < q$ **do**
12 $d \leftarrow$COMPUTEDEPTH($\mathcal{N}, |S|, d, q$);
13 $S \leftarrow$GENERATESUBPROBLEMS($\mathcal{N}, S, d$);
14 **if** $S = \emptyset$ **then return** $\emptyset$;
15 $\mathcal{N} \leftarrow (\mathcal{X}$,GETDOMAINS$(S), \mathcal{C})$
16 **return** S;

Décompositions statiques parallèles

Une idée naïve de décomposition parallèle de l'arbre de recherche consisterait, en utilisant w *workers*, à découper l'arbre de recherche en w sous-problèmes consistants avec la propagation, où chacun de ces sous-problèmes va être affecté à un *worker*. Ensuite, chaque *worker* va générer à son tour q/w sous-problèmes qu'il va mettre dans une file d'attente. Si un *worker* est dans l'incapacité de générer q/w sous-problèmes, il va prendre un sous-problème dans la file d'attente et réessayer jusqu'à obtenir q/w sous-problèmes.

On considérera désormais qu'une répartition de charge prend en compte à la fois le temps de décomposition et le temps de résolution des tâches.

Considérant cette répartition de charge, la décomposition parallèle naïve souffre d'un problème majeur : il est possible qu'un des sous-problèmes donné à un *worker* soit celui qui va générer le plus de sous-problèmes lors de sa décomposition. Ainsi, le *worker* en question aura une charge de travail plus conséquente menant à une très mauvaise répartition de charge.

Pour remédier à ce problème, nous proposons une méthode de décomposition parallèle efficace qui va essayer de contrôler la répartition de charge durant la décomposition.

En s'intéressant à t_w, des expérimentations préliminaires de la méthode EPS sur un centre de calcul avec 512 *workers* ont montré qu'on obtenait un bon facteur de gain sans tenir compte de t_d. Il faut noter que la décomposition devient de plus en plus difficile en augmentant le nombre de *workers*, ce qui de fait augmente le temps total de résolution. Afin d'améliorer le facteur de gain en augmentant le nombre de *workers*, il est nécessaire de se concentrer sur la décomposition.

Pour améliorer la décomposition, notre première approche a consisté à trouver une valeur de q/w afin de parvenir à une bonne répartition de charge. La machine utilisée pour l'étude expérimentale possédait 40 *workers*. On a constaté que pour $q/w \geq 30$, on obtenait un bon facteur de gain. Cependant, en passant sur un centre de calcul avec 512 *workers*, le facteur de gain était limité, suite à l'augmentation du temps de décomposition. Nous avons également constaté que :

1. L'écart entre les facteurs de gain diminue lorsque le nombre de sous-problèmes augmente jusqu'à ce qu'un certain nombre de sous-problèmes $q_{critique}$ est atteint. À partir de ce nombre de sous-problèmes, on ne peut plus obtenir d'améliorations.
2. Une synchronisation se produit lorsqu'on attend que tous les sous-problèmes soient générés par les *workers* à une profondeur estimée. Elle permet de distribuer à nouveau les charges de travail, mais elle coûte du temps. Il faut donc trouver un bon compromis. Plus les synchronisations sont faites en haut de l'arbre de recherche, moins elles coûtent, car il y a peu de travail à faire.
3. Lorsqu'on décompose le problème vers le haut de l'arbre de recherche, il y a moins de sous-problèmes à générer. Par conséquent, il y a plus de chances d'avoir un déséquilibre dans la répartition de charge. Pour éviter ce déséquilibre, on utilise des synchronisations en haut de l'arbre de recherche.
4. Pour que les *workers* commencent à décomposer efficacement, il faut donner à chacun des sous-problèmes consistants avec la propagation.

5. En deçà d'un certain nombre de sous-problèmes par *worker*, on ne génère pas assez de sous-problèmes, par conséquent on aura une mauvaise répartition durant la décomposition. Il faut donc fixer ce nombre pour obtenir le meilleur facteur de gain lors de la parallélisation de la décomposition. En sélectionnant quelques problèmes représentatifs afin de fixer le bon nombre de sous-problèmes par *worker*, le tableau 3.1 montre que 5 est un bon choix. Ainsi, au-delà de 5 sous-problèmes par *worker*, le facteur de gain relatif à t_d va augmenter.

6. Même si une décomposition ne doit pas générer de sous-problèmes inconsistants, ce qui aurait un impact négatif sur les performances, découper le problème initial en un petit nombre de sous-problèmes sans en vérifier l'inconsistance prend un temps raisonnable par rapport au temps total de décomposition et peut donc être fait de façon séquentielle.

Instance	**nombre de sous-problèmes**				
	3	4	5	6	7
`sb_sb_13_13_6_4`	0,8	0,5	0,7	1,1	1,0
`sugiyama`	2,5	1,4	1,3	1,5	1,5
`patternSetMiningGerman`	1,4	1,4	0,9	0,9	1,1
`radiation_03`	0,6	0,6	0,7	0,4	0,6
temps total(s)	5,3	3,8	**3,6**	3,9	4,2

TABLE 3.1 – Comparaison des temps de décomposition (en secondes) en fonction d'un nombre de sous-problèmes fixé pour décomposer à nouveau des sous-problèmes durant la seconde phase.

Suite à ces observations, la seconde approche consiste à trouver un processus itératif décomposant le problème. Ainsi, nous proposons une méthode en 3 phases, qui utilise 1 *master*, w *workers* et deux queues S et S' qui peuvent contenir des sous-problèmes. Les phases sont les suivantes :

- une phase initiale où le *master*, à partir du problème initial, va générer w sous-problèmes et les mettre dans S ; chacun de ces sous-problèmes va être traité par un *worker* ;

- une phase principale qui génère 5 sous-problèmes par *worker*, consistants avec la propagation. Pour cela, le *master* estime une profondeur k ; chaque *worker* prend un sous-problème dans S (on vide ainsi S) et le décompose en sous-problèmes jusqu'à la profondeur k, puis tous les sous-problèmes engendrés par le *worker* sont mis dans S'. Lorsque tous les *workers* ont fini leur décomposition (phase de synchronisation), le *master* vérifie le nombre total de sous-problèmes contenus dans S' : s'il y a moins de 5 sous-problèmes par *worker*, le *master* estime à nouveau une profondeur $k' > k$ puis met tous les sous-problèmes de S' dans S (on vide ainsi S') afin que S soit

disponible pour les *workers* qui vont à nouveau décomposer les sous-problèmes. Ce processus est répété jusqu'à qu'on dépasse 5 sous-problèmes.

- une phase finale qui génère 30 sous-problèmes par *worker*, qui s'opère de la même manière que lors de la phase principale.

L'algorithme 3 est une possible implémentation de la nouvelle décomposition parallèle.

Algorithm 3: ParallelDecomposition

On définit les paramètres suivants utilisés dans les fonctions ci-dessous :

- $\mathcal{N}$, un réseau de contraintes
- Q et S, des ensembles contenant des sous-problèmes
- $nbspb$ et q, des nombres de sous-problèmes à atteindre
- d, une profondeur dans l'arbre de recherche
- $numStep$, le nombre d'étapes dans la décomposition parallèle
- nbPbStep ; un tableau de valeurs définissant le nombre de problèmes requis pour chaque étape

WORKERDEC($\mathcal{N}$, Q, d)
- $S \leftarrow \emptyset$;
- **run in parallel**
  - **while** $Q \neq \emptyset$ **do**
    - prendre $P \in Q$ et supprimer P dans Q;
    - $S' \leftarrow$GENERATESUBPROBLEMS$(\mathcal{N}, P, d)$;
    - $S \leftarrow S \cup S'$
- **return** S;

DECOMPOSE($\mathcal{N}$, S, $nbspb$)
- **while** $|S| < nbspb$ **do**
  - $d \leftarrow$COMPUTEDEPTH$(\mathcal{N}, |S|, d, nbspb)$;
  - $S \leftarrow$WORKERDEC$(\mathcal{N}, S, d)$;
  - **if** $S = \emptyset$ **then return** $\emptyset$ $\mathcal{N} \leftarrow (\mathcal{X}$,GETDOMAIN$(S), \mathcal{C})$
- **return** S;

PARALLELDECOMPOSITION($\mathcal{N}$, nbPbStep, $numStep$, q)
- $(S, d) \leftarrow$SIMPLEDECOMPOSITION$(\mathcal{N}$, nbPbStep$[0])$;
- $S \leftarrow$WORKERDEC$(\mathcal{N}, S, d)$;
- $\mathcal{N} \leftarrow (\mathcal{X}$,GETDOMAIN$(S), \mathcal{C})$;
- **for** *i=0* **to** *numStep-1* **do**
  - $S \leftarrow$DECOMPOSE$(\mathcal{N}, S,$ nbPbStep$[i])$;
  - **if** $S = \emptyset$ **or** $|S| \geq q$ **then return** S
- **return** S;

3.2.1.2 Décomposition dynamique

Nous étudions une autre décomposition qui respecte les principes d'EPS. Les décompositions que nous avons vu précédemment sont statiques, c'est-à-dire que nous fixons l'ordre et le nombre de variables sur lesquels nous décomposons le problème. De plus, nous utilisons la stratégie de recherche `lex` pour énumérer les sous-problèmes consistants avec la propagation. Pour les problèmes où des stratégies de recherche plus spécifiques sont plus efficaces, nous proposons un algorithme de décomposition qui l'exploite pleinement, c'est-à-dire que nous pouvons appliquer n'importe quelle stratégie de recherche pour générer les sous-problèmes. Une telle décomposition est dynamique, car elle génère les sous-problèmes qui seront immédiatement mis à disposition aux *workers*.

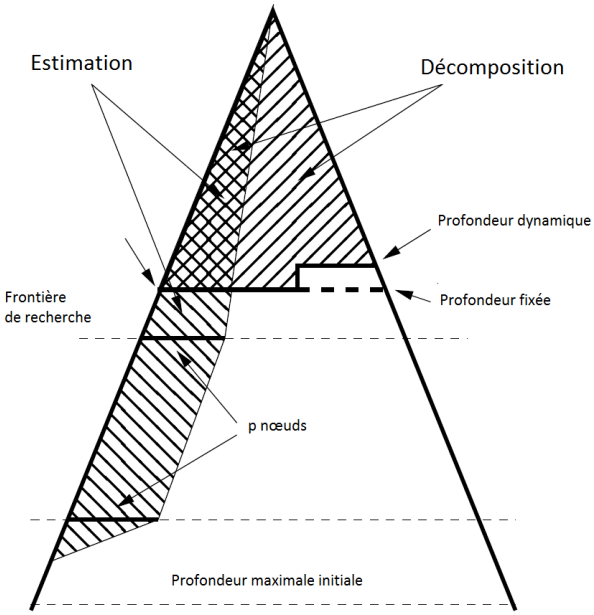

FIGURE 3.2 – Utilisation de l'estimation dans la décomposition dynamique de l'arbre de recherche

Nous utilisons la notion de frontière dans un arbre recherche vue dans le chapitre 2. La décomposition calcule la frontière à une profondeur donnée de l'arbre de recherche illustrée dans la figure 3.2. Si la profondeur est fixée, chaque nœud ouvert de la frontière correspond à un sous-problème consistant avec la propagation peu importe la distance entre le nombre réel de sous-problèmes $q(d)$ et le nombre de sous-problèmes souhaités q. Nous remarquons qu'un sous-problème non consistant avec la propagation ne peut pas exister dans cette décomposition contrairement à la décomposition simple. En effet, nous utilisons le parcours de l'arbre de recherche qui permet d'éviter de les générer. Si la profondeur est dynamique, alors la profondeur peut être réduite durant la décomposition pour réduire la distance entre q et $q(d)$. La profondeur est diminuée de 1 si le nombre de sous-problèmes courant dépasse une limite donnée. Cette limite est initialisée à $2 \times p$ et elle augmente de manière à être multipliée par deux après chaque diminution de la profondeur.

Dans sa forme la plus simple, la profondeur de la décomposition peut être fournie *a priori* avec une bonne connaissance du problème. Cependant, il est préférable de le calculer automatiquement pour déterminer la profondeur de décomposition. C'est pour cette raison que la décomposition proposée vise à identifier la plus haute frontière avec un nombre proche de q nœuds ouverts. La décomposition est divisée en trois phases : une phase d'échantillonnage au niveau du sommet de l'arbre de recherche avec un temps imparti (5 secondes), une phase d'estimation des largeurs de niveaux, et enfin une phase qui permet d'estimer la profondeur avec un algorithme glouton.
Lors de l'exploration du sommet de l'arbre de recherche, on fixe une borne supérieure de la profondeur de décomposition. La profondeur de décomposition maximale doit être choisie en ce qui concerne le nombre de *workers* et le nombre de sous-problèmes souhaités par *worker*. Dans nos expériences, le niveau de décomposition maximale est limité à 20 ce qui donnerait plus de 1 million de feuilles dans un arbre binaire parfait. En pratique, la profondeur de décomposition est diminuée jusqu'à ce que le produit cartésien du domaine de variable le plus petit devient inférieur à $10 \times p$ pour prendre en compte l'arité de l'arbre de recherche. À la fin de cette phase, nous déterminons la profondeur maximale de la décomposition.
Durant la phase d'échantillonnage, la profondeur maximale maximum est réduite à chaque fois que les q nœuds ouverts ont été visités à une profondeur donnée. Si l'échantillonnage se termine dans le temps imparti, alors le sommet de l'arbre de recherche a été entièrement visité et aucune estimation n'est nécessaire. Une heuristique détermine le niveau de décomposition en minimisant la distance entre le nombre de nœuds et le nombre de sous-problèmes souhaité. On note qu'un léger biais est introduit pour générer plus de sous-problèmes afin de ne pas en avoir moins.
Dans le cas où la phase d'échantillonnage a été interrompue en raison du temps imparti, il faut estimer les largeurs des meilleurs niveaux de l'arbre de recherche. L'estimation est plus ou moins adaptée de [Cornuéjols 2006] pour des arbres de recherche *n-aires*. En pratique, le principal problème est que plus l'arité est élevée, plus la précision de l'estimation est basse. Par conséquent, une heuristique doit déterminer la profondeur de décomposition sur la base du nombre estimé des nœuds par niveau et également du nombre des nœuds visités.

3.2.2 Résolution

La partie résolution de la méthode EPS est simple : une fois la décomposition exécutée, elle nous fournit une queue S de sous-problèmes à résoudre qui vont être traités par les *workers*. Chaque *worker* donne le sous-problème au solveur et le résout. Une fois le sous-problème résolu, il envoie les résultats au *master* et récupère un nouveau sous-problème dans la queue S. Si la queue S est vide, il ne fait plus rien. À la fin de la résolution, le *master* affiche les résultats.

CHAPITRE 4

Évaluation de la méthode EPS

Sommaire

Nous présentons les expérimentations que nous avons réalisées pour évaluer les différentes implémentations d'EPS proposées dans le chapitre 3 avec le *work stealing* et le *portfolio*. Nous décrivons d'abord le protocole expérimental puis nous analysons la décomposition dans EPS. Nous décrivons ensuite les résultats obtenus sur les différentes architectures. Enfin, nous décrivons et évaluons *Embarrassingly Distributed Search* (EDS), une variante de la méthode EPS qui utilise l'outil de planification fourni par le centre de calcul afin de distribuer les différents sous-problèmes aux *workers*.

4.1 Protocole expérimental

4.1.1 Instances

Il existe un large éventail de problèmes qui sont représentatifs des types de problèmes résolus en PPC. Le défi est de sélectionner des instances de problème variées, pertinentes et difficiles. Seuls trois types de problèmes sont étudiés : les problèmes d'énumération (rechercher toutes les solutions), les problèmes n'ayant aucune solution et les problèmes d'optimisation. Il faut noter que les tests de performance de notre méthode parallèle sont plus difficiles sur un problème d'optimisation ; en effet, la chance peut jouer un rôle (car l'espace de recherche peut être réduit radicalement si on trouve rapidement une solution optimale). Pour cela, nous avons sélectionné 4 listes d'instances. Toutes ces instances sont listées dans l'annexe B. Les deux premières listes `fzn 1` et `fzn 2` sont choisies entre les 5000 instances venant soit du blog de Hakan Kjellerstrand [Kjellerstrand 2014] ou directement de la distribution *minizinc* [MiniZinc 2012] modélisées avec le langage FlatZinc [Nethercote 2007]. La première liste, appelée `fzn 1` est une sélection de 18 instances composée de 6 problèmes d'énumération de solutions, et 12 problèmes d'optimisation. Ces instances sont résolues en séquentiel entre 20 secondes et 700 secondes (11,6 minutes) avec `Gecode`. La deuxième liste, appelée `fzn 2` est une sélection de 18 instances composée d'un problème n'ayant pas de solution, 6 problèmes d'énumération de solutions, et 11 problèmes d'optimisation. Ces instances sont résolues en séquentiel entre 500 secondes et 1 heure avec `Gecode`. On note `fzn`, l'union des `fzn 1` et `fzn 2`.
Les deux autres listes d'instances proviennent des catégories ACAD et REAL de la distribution XCSP 2.1 [XCSP 2008]. On note que le langage XCSP 2.1 ne traite pas les problèmes d'optimisation. Ces deux listes d'instances contiennent des instances réputées difficiles qui ont pu être résolues sur `Choco2` [Malapert 2014]. La première liste appelée `xcsp 1` est composée de 5 instances ne possédant aucune solution et 5 instances d'énumération alors que la seconde liste `xcsp 2` contient 11 instances ne possédant aucune solution et 3 instances d'énumération. `xcsp 1` est composée d'instances plus faciles à résoudre que `xcsp 2`. On note `xcsp`, l'union des `xcsp 1` et `xcsp 2`.
De plus, nous considérons deux problèmes classiques venant de [Gent 1999], les problèmes des *n-queens* et de la règle de *Golomb* qui ont été largement utilisés dans la littérature au niveau de la parallélisation de leurs résolutions.

`OR-tools` et `Gecode` intègrent un parseur FlatZinc et `Choco2` possède un parseur

XCSP. La conversion du format XCSP au format FlatZinc avec le programme *xcsp2fzn* fonctionne, mais la conversion du format FlatZinc au format XCSP échoue en raison des incompatibilités de traduction et du comportement instable du programme *fzn2xcsp* entre les deux formats. Ces deux programmes de conversion proviennent de la distribution *minizinc* [MiniZinc 2012].

Pour chaque instance, les meilleurs facteurs de gain et les meilleurs temps sont indiqués en gras dans les tableaux. Les moyennes arithmétiques sont calculées pour les temps de résolution, alors que les moyennes géométriques sont calculées pour les facteurs de gain. On note également que les valeurs manquantes sont ignorées lors du calcul des statistiques.

Nous rappelons que dans le cadre de notre étude, la recherche de la première solution est exclue, car l'interprétation des résultats serait compliquée en raison de grandes différences en termes de performance et de comportement entre les algorithmes parallèles et séquentiels. De plus, la variabilité de la recherche de la première solution peut être observée à une moindre mesure, pour des problèmes d'optimisation, car la preuve d'optimalité est nécessaire. Cependant, nous pouvons considérer les problèmes ne possédant aucune solution en raison de leur importance pratique, par exemple dans les tests de logiciels.

Utilisation des diagrammes en boîtes

Dans nos analyses, nous utilisons aux diagrammes en boîtes. Les diagrammes en boîtes montrent les différences entre plusieurs populations sans faire de prédictions sur la loi statistique de distribution qui en régit : ces différences sont non paramétriques. Un diagramme en boîtes est un moyen pratique de représenter graphiquement des groupes de données numériques à travers leurs quartiles. Une boîte dans ce diagramme s'étend toujours sur la plage de valeurs contenant le premier quartile jusqu'au troisième quartile. Les marqueurs dans une boîte s'étendent de chaque extrémité de la boîte pour une plage de valeurs égale à 1,5 fois l'écart interquartile qui est une mesure de dispersion qui s'obtient en faisant la différence entre le troisième et le premier quartile. Les espacements entre les différentes parties de la boîte permettent d'indiquer le degré de dispersion et l'asymétrie dans les données. En général, l'asymétrie d'une distribution est positive si la partie haute de la courbe est plus longue ou large, et elle est négative si la partie basse de la courbe est plus longue ou large. Tous les points qui se trouvent hors de la plage des boîtes sont considérés comme des valeurs à ignorer (ces points sont entourés par des cercles).

4.1.2 Environnements d'exécution

L'ensemble des expérimentations a été effectué sur trois types d'environnements :

- Environnement multi-cœurs (**fourmis**) : une machine Dell fonctionnant sous Scientific Linux 6,0 avec 256 Go de mémoire et quatre processeurs Intel E7-4870, ayant chacun 10 cœurs. En exécutant le même programme 10 fois sur chaque cœur de la machine, nous avons pu atteindre un facteur de $37,5$ par rapport à une résolution séquentielle. Par conséquent, c'est le gain maximum que nous pouvons espérer sur cette machine.

- Environnement *cluster* (**cicada**) : un *cluster* est une grappe de serveurs (ou « ferme de calcul ») constituée de deux machines au minimum (appelés aussi nœuds). Le *cluster* utilisé est le Centre de Calculs Interactifs de l'Université Nice-Sophia Antipolis (`http://calculs.unice.fr/`) fonctionnant sous Linux avec 4608 Go de mémoire et 1152 cœurs répartis sur 72 nœuds avec 144 processeurs Intel E5-2670 (chaque processeur possède 8 cœurs). Le centre de calcul utilise un outil de planification *OAR* [Capit 2005] qui s'occupe de la gestion et de la soumission des travaux (l'exécution des programmes).

- Environnement *cloud* (**azure**) : la plate-forme de *cloud computing* produite par l'entreprise Microsoft qui permet de déployer des applications avec l'environnement Windows Server [Li 2009]. On peut allouer à la demande des nœuds de calcul. Chaque nœud possède 56 Go de RAM et un processeur Intel Xeon E5-2690E avec 8 cœurs physiques cadencés à 2,6 GHz . Nous avons été autorisés à utiliser trois nœuds (24 cœurs) qui sont gérés simultanément avec l'outil de planification Microsoft Cluster HPC 2012 [MS-MPI 2015]. Il faut noter que la plate-forme de *cloud* impose d'utiliser MS-MPI [Krishna 2010, Lantz 2008], une implémentation MPI sur Windows.

Toutes les architectures étudiées fournissent des technologies d'*hyperthreading*. L'*hyperthreading* améliore la parallélisation des calculs. Pour chaque cœur de processeur physiquement présent, le système d'exploitation crée deux cœurs logiques, et partage la charge de travail entre ces deux cœurs lorsque cela est possible. En effet, le temps d'accès à la mémoire est plus lent que le temps de calcul dans un processeur. Par conséquent, lorsqu'un cœur logique est en attente de réception de données, on peut entre temps affecter un autre cœur logique au cœur physique pour faire seulement du calcul. Cependant, on peut avoir des pertes de performances lorsque les deux cœurs logiques accèdent en même temps à la mémoire.

4.1.3 Détails d'implémentation

Nous avons implémenté la méthode EPS dans trois solveurs : `Choco2` écrit en Java, `OR-tools` et `Gecode` écrits en C++. Nous utilisons deux technologies pour faire du parallélisme : *threads* [Mueller 1993, Kleiman 1996] et MPI [Lester 1993, Gropp 1993]. La différence entre ces deux technologies est que les *threads* d'un même processus s'exécutent dans un espace de mémoire partagée, alors que MPI est un modèle d'exécution standard et portable permettant le passage de messages pour échanger des informations entre les processus exécutés dans des espaces de mémoire séparés. La technologie *threads* ne gère pas plusieurs nœuds d'un *cluster* alors que MPI peut le faire.

Avec le langage de programmation C++, les *threads* sont implémentés par les *Pthreads*, une bibliothèque respectant la norme POSIX [Mueller 1993, Kleiman 1996] utilisée par tous les systèmes Unix. Concernant le langage de programmation Java, la technologie *threads* est standardisée [Hyde 1999].

Quant à MPI, il existe plusieurs implémentations comme OpenMPI [Gabriel 2004], Intel MPI [IntelMPI 2015], MPI-CH [MPICH 2015] et celle de Microsoft nommée MS-MPI

[Krishna 2010, Lantz 2008], car MPI est seulement un standard et non un système. Ainsi, les caractéristiques d'une machine et les différentes optimisations possibles ne peuvent pas être prises en compte dans le standard MPI. En général, les fournisseurs de machines comme Bull, IBM et Intel donnent leur propre implémentation de MPI selon les spécifications propres des machines. Le *cluster cicada* fourni par l'entreprise Bull a sa propre implémentation du MPI d'Intel [IntelMPI 2015] alors que Microsoft Azure prend uniquement en charge sa propre bibliothèque, appelée MS-MPI [Krishna 2010, Lantz 2008].

On rappelle que l'exécution des *threads* est limitée uniquement dans un seul nœud. Comme il n'y a pas d'implémentation du MPI en Java, `Choco2` ne peut utiliser que les *threads* et donc ne peut s'exécuter que dans un seul nœud alors que `Gecode` peut utiliser tous les nœuds disponibles de la machine grâce au MPI d'Intel ou Open MPI [Lester 1993, Gropp 1993].

4.1.4 Technologies et méthodes de décomposition utilisées

`OR-tools` utilise la technologie *threads* en C++. Seule la décomposition séquentielle présentée dans le chapitre 3 est implémentée dans ce solveur. `Gecode` utilise les technologies *threads* et MPI en C++. Les décompositions statiques (séquentielle et parallèle) sont implémentées dans ce solveur. Dans nos expérimentations, nous utilisons seulement la décomposition statique parallèle pour ce solveur. De ce fait, `Gecode` utilisera les *threads* sur **fourmis**, OpenMPI sur *cicada*, et MS-MPI sur le *cloud* de Microsoft Azure. `Choco2` utilise la technologie *threads* en Java [Hyde 1999], car comme dit précédemment, il n'y a pas d'implémentation standard de MPI en Java.

Nombre de *workers*

Nous rappelons que w représente le nombre de *workers* et c représente le nombre de cœurs. Si $w < c$, alors il y aura toujours des cœurs inutilisés. Habituellement, nous choisissons $w = c$, de sorte que tous les *workers* peuvent travailler simultanément. Aujourd'hui, le matériel et le système d'exploitation prennent en compte les systèmes multi-cœurs pour améliorer la parallélisation et la planification en temps réel des processus et des *threads*. Bien sûr, ces technologies, par exemple *l'hyper-threading*, sont rapidement surchargées par la croissance du nombre de *workers*. Ici, nous considérons deux alternatives : un ou deux *workers* par cœur ($w = c, 2c$).

Nombre de sous-problèmes par *worker*

Le nombre de sous-problèmes par *worker* doit être suffisamment grand pour avoir une bonne répartition de charge. Nous allons tester plusieurs valeurs pour connaître le bon nombre de sous-problèmes par *worker*, noté $\#sspw$.

4.1.5 Opérations exécutées

Les différentes implémentations d'EPS dans les solveurs suivent les opérations suivantes :

- créer un *master* ;

- le *master* a plusieurs choix pour charger un problème : soit il lit un fichier contenant un modèle FlatZinc ou un modèle XCSP, soit le problème est directement modélisé par l'API du solveur ;
- le *master* crée les *workers* ;
- le *master* décompose les sous-problèmes à générer et les met dans une file d'attente qui sera à la disposition des *workers* ;
- chaque *worker* récupère un sous-problème dans la file d'attente et le résout puis envoie les résultats au *master*. Ensuite, il récupère à nouveau un sous-problème, jusqu'à ce que la file d'attente soit vide ;
- une fois le dernier sous-problème résolu, le *master* affiche les résultats ;

4.2 Analyse de la décomposition

Dans cette section, nous évaluons les performances et les qualités des différentes méthodes de décomposition proposées dans le chapitre 3. Nous observons d'abord les limites de la décomposition statique simple qui montrent qu'il faut générer plus de sous-problèmes que le nombre de *workers*. Ensuite, nous étudions le rapport entre les sous-problèmes consistants avec la propagation et le reste des sous-problèmes. Enfin, nous allons comparer avec la décomposition dynamique et analyser l'influence de l'utilisation de stratégies de recherche dans la décomposition.

4.2.1 Décomposition statique séquentielle

Les auteurs des articles [Jaffar 2004, Bordeaux 2009] se sont intéressés au problème des *n-queens*, en utilisant une décomposition statique simple avec un sous-problème par *worker*. Les résultats donnés par la figure 4.1 sont les suivants : comparé à une méthode de décomposition séquentielle, le facteur de gain observé est borné à 30 avec 64 *workers*. En utilisant l'algorithme de décomposition statique séquentielle, nous obtenons un meilleur gain lorsque $\#sppw > 20$, qui n'est plus borné à 30. Ces résultats valident le choix de générer un grand nombre de sous-problèmes pour obtenir des gains de performance.

4.2.2 Sous-problèmes consistants avec la propagation

Sur **fourmis**, nous avons calculé le ratio entre le nombre de sous-problèmes consistants avec la propagation par rapport à l'ensemble des sous-problèmes, sur l'ensemble des instances sélectionnées, en utilisant la méthode de décomposition statique simple. Pour chaque instance, on va générer k sous-problèmes. La figure 4.2 donne les parts maximales, en moyenne et minimales (représentées par les trois courbes) des sous-problèmes consistants avec la propagation générés par la méthode de décomposition simple sur l'ensemble des instances `fzn 1` sur **fourmis**. On remarque que la part de sous-problèmes consistants avec la propagation ne cesse de diminuer avec l'augmentation du nombre de sous-problèmes.

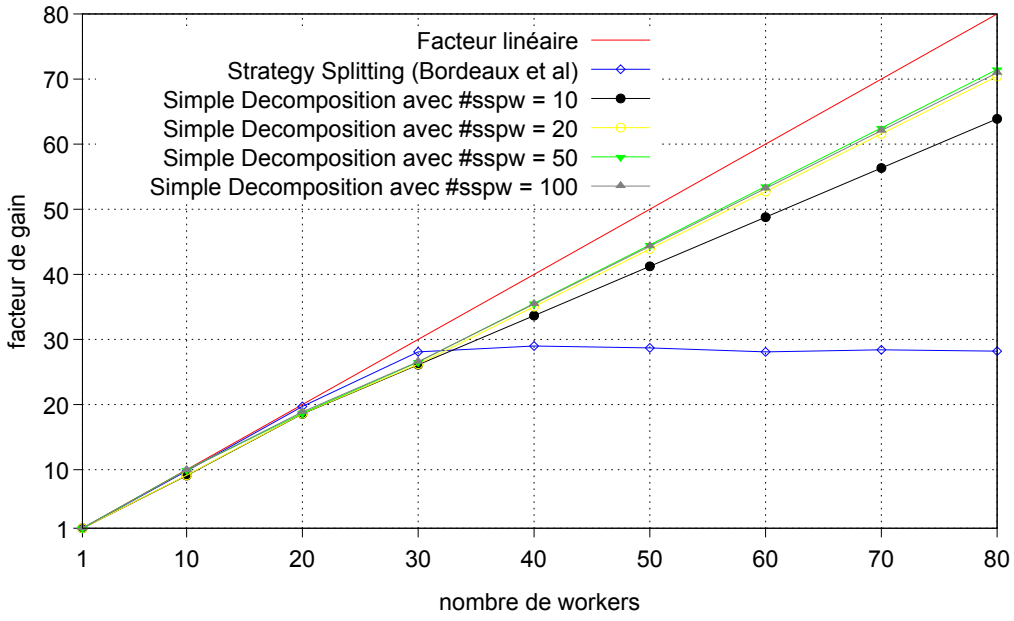

FIGURE 4.1 – Comparaison entre la décomposition statique séquentielle et l'algorithme de Bordeaux [Bordeaux 2009] sur le problème des *n-queens* de taille 17 : performance en fonction de #sppw.

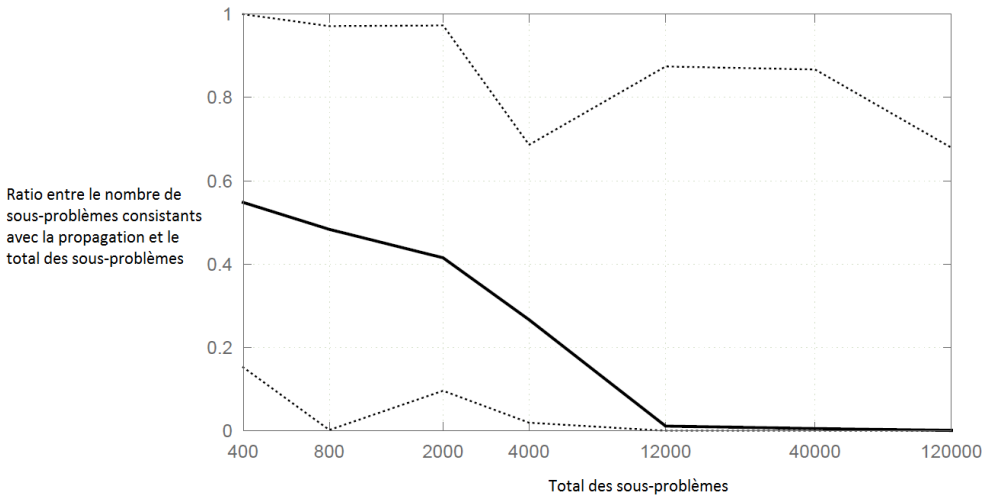

FIGURE 4.2 – Parts maximales, en moyenne et minimales des sous-problèmes consistants avec la propagation générés par la méthode de décomposition simple sur l'ensemble des instances `fzn 1` sur **fourmis**

C'est la raison pour laquelle nous avons développé la méthode de décomposition utilisant la DBDFS, générant uniquement des sous-problèmes consistants avec la propagation.

4.2.3 Temps d'inactivité selon le nombre de sous-problèmes par *worker*

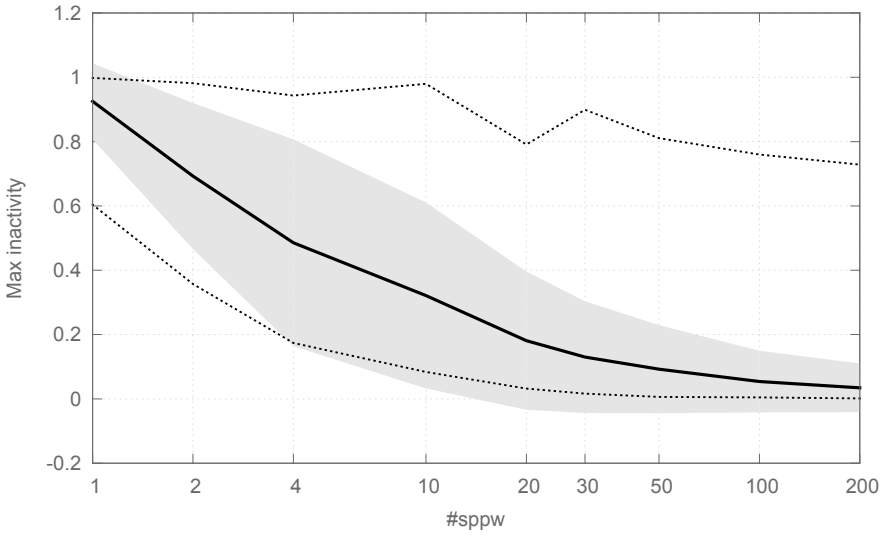

FIGURE 4.3 – Temps d'inactivité maximale des *workers* avec les instances `fzn 1` en fonction du nombre de sous-problèmes par *worker*.

Pour avoir une idée du nombre minimal de sous-problèmes par *worker*, menant à une bonne répartition de charge, nous avons calculé le temps d'inactivité de chaque *worker* avec la méthode EPS sur les instances `fzn 1`, puis nous prenons les temps d'inactivité maximale dans la figure 4.3, c'est-à-dire le temps d'inactivité du premier *worker* qui n'a plus de sous-problèmes à résoudre. Les courbes de la figure 4.3 représentent le maximum, la moyenne et le minimum de temps d'inactivité maximale des *workers* sur l'ensemble des instances `fzn 1`. L'aire autour de la moyenne des temps d'inactivité maximale représente l'écart-type. On remarque que ces temps d'inactivité diminuent lorsque le nombre de sous-problèmes par *worker* augmente. De plus, la répartition de charge s'améliore lorsque nous avons plus de 10 sous-problèmes par *worker*. Ces résultats valident l'hypothèse qui stipule que la génération d'un grand nombre de sous-problèmes permet de réduire l'inactivité des *workers*. De plus, ils permettent indirectement de montrer que la méthode amène à un équilibrage des temps de résolution de chaque *worker* lorsque l'inactivité des *workers* baisse. Un autre point intéressant est qu'il y a dans les sous-problèmes restants des sous-problèmes difficiles qui prennent un temps considérable à résoudre lors de la fin de la résolution ; ces derniers retiennent les *workers* qui les résolvent et excluent les autres *workers* qui n'ont plus de sous-problèmes à résoudre (*workers* inactifs). Par conséquent, l'inactivité est due à cause de ces sous-problèmes. En générant plus de sous-problèmes, on augmente également les chances de décomposer les sous-problèmes difficiles à résoudre. Ainsi, nous obtenons une meilleure répartition de charge et nous réduisons les temps d'inactivité des *workers*.

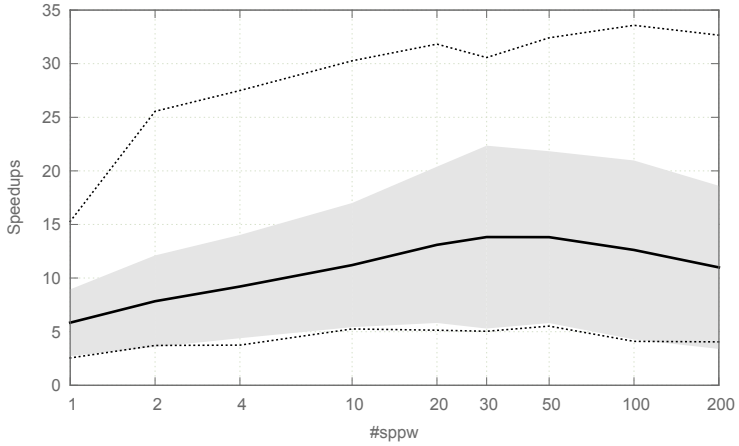

FIGURE 4.4 – Facteur de gain de la méthode EPS par rapport à une résolution séquentielle en fonction du nombre de sous-problèmes par *worker* avec les instances `fzn 1` sur une machine multi-cœurs (**fourmis**).

4.2.4 Influence du nombre de sous-problèmes

Nous calculons le facteur de gain de la méthode EPS par rapport à une résolution séquentielle en fonction du nombre de sous-problèmes par *worker*. La figure 4.4 donne l'évolution des facteurs de gain en fonction du nombre de sous-problèmes par *worker* avec les instances `fzn 1` sur une machine multi-cœurs (**fourmis**). Dans cette figure, nous donnons les facteurs de gain maximaux, moyens et minimaux. L'aire grise représente l'écart-type autour de la moyenne. Nous observons que générer 30 sous-problèmes par *worker* permet en moyenne de bons facteurs de gain pour la résolution des instances avec la méthode EPS. La réduction du facteur de gain lors de l'augmentation de la valeur de *#sspw* vient du fait que le processus de décomposition commence à résoudre le problème, qui est beaucoup plus lent qu'une procédure de résolution dédiée.

4.2.5 Parallélisation de la décomposition

Les expérimentations suivantes utilisent 96 *workers* avec `Gecode` car ces conditions d'expérimentation ont montré que la décomposition séquentielle présentait ses limites en terme d'efficacité. En effet, les premières expérimentations étaient effectuées avec une décomposition séquentielle et nous étions satisfaits des résultats sur la machine multi-cœurs (**fourmis**). Cependant, lorsque nous lançons nos expérimentations sur le centre de calcul (**cicada**) qui possèdent plusieurs centaines de cœurs, les performances baissent radicalement (à partir d'une centaine de cœurs). L'origine du problème est que la décomposition séquentielle prenait de plus en plus de temps pour générer les sous-problèmes. Plus nous augmentons le nombre de cœurs et plus le temps de générer les sous-problèmes augmente. C'est dans ce contexte-là que nous avons développé deux versions parallèles de la décomposition statique séquentielle.

4.2.5.1 Impact de la décomposition séquentielle

Instance	$\textbf{ratio}_{dec/totalres}$	su_w	su
allinterval_15	17%	83,5	69,1
magicsequence_40000	76%	87,8	28,2
sportsleague_10	49%	74,4	38,1
sb_sb_13_13_6_4	46%	82,4	44,6
quasigroup7_10	50%	76,5	38,1
non_non_fast_6	64%	80,9	28,8
golombruler_13	25%	94,2	71,0
warehouses	89%	219,9	25,1
setcovering	67%	74,6	24,4
depot_placement_att48_5	48%	76,9	32,1
depot_placement_rat99_5	58%	84,7	26,4
fastfood_ff58	69%	78,1	31,4
open_stacks_01_problem_15_15	60%	72,2	31,9
open_stacks_01_wbp_30_15_1	56%	80,5	29,7
sugiyama2_g5_7_7_7_7_2	63%	73,4	28,7
patternSetMiningGerman	61%	70,6	33,9
radiation_03	52%	97,8	18,7
talent_scheduling_film116	81%	84,50	33,50
Moyenne géométrique (M.G) en % ou ratio	54%	84,5	33,5

TABLE 4.1 – Décomposition séquentielle avec 96 *workers*.

Tout d'abord, nous observons l'impact de la décomposition statique séquentielle dans la résolution d'un problème avec une centaine de cœurs. Le tableau 4.1 donne pour 96 *workers*, la part de la décomposition séquentielle par rapport à la résolution ($ratio_{dec/totalres}$), le facteur de gain avec le temps de résolution des *workers* su_w, le facteur de gain global avec la décomposition séquentielle et celui avec la décomposition parallèle pour chaque instance. D'une part, nous constatons que la décomposition séquentielle prend un temps considérable par rapport à la durée totale de la résolution (une moyenne de 54%). D'autre part, nous observons que le facteur de gain moyen de la résolution par les *workers* est proche du facteur quasi-linéaire (84,5) et nous obtenons un facteur de gain moyen de la durée totale de la résolution à 33,5. Ainsi, EPS obtient des facteurs de gain linéaires seulement dans la résolution. Cependant, la génération de sous-problèmes prend beaucoup de temps, ce qui empiète sur les performances globales.

4.2.5.2 Profondeur de l'arbre de recherche avec la décomposition

Cet impact peut être expliqué également par la profondeur à laquelle la décomposition doit descendre dans l'arbre de recherche pour atteindre le nombre de sous-problèmes souhaité. La figure 4.5 montre l'évolution de la profondeur lorsqu'on utilise plus de *workers* avec

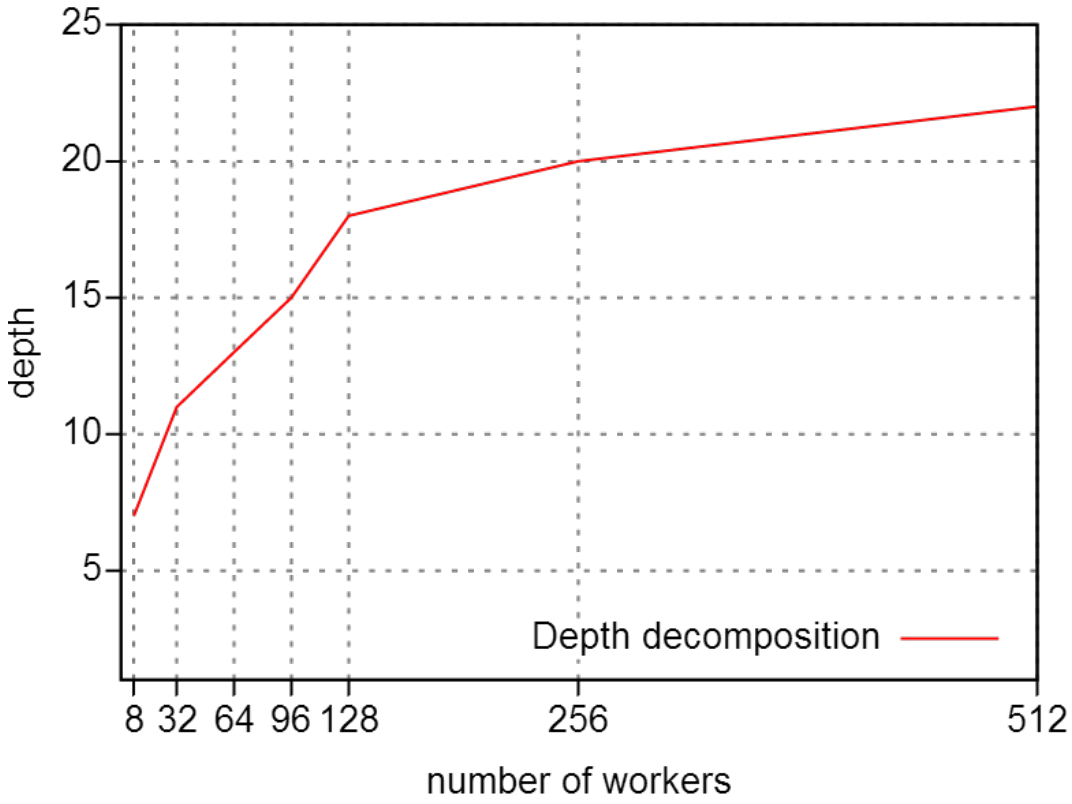

FIGURE 4.5 – Profondeur (*depth* en anglais) pour atteindre 30 sous-problèmes par *worker* en fonction du nombre de *workers* avec les instances `fzn 2`.

EPS. Comme prévu, la profondeur dans la décomposition croît avec le nombre de *workers*. Parfois, il peut être important avec certaines instances, comme *warehouses* ou *talent scheduling film116* ou non comme *allinterval 15* (voir tableau 4.2). Ainsi, l'amélioration de la décomposition peut avoir un impact sur t, notamment pour des problèmes faciles.

4.2.5.3 Comparaison des algorithmes de décomposition statiques

Nous comparons les trois algorithmes de décomposition statique que nous avons présentés dans le chapitre 3.

On note respectivement Dec_{seq}, $Dec_{//1}$ et $Dec_{//2}$ la décomposition séquentielle, la décomposition parallèle naïve et la décomposition parallèle efficace. On note que ces algorithmes donnent toujours la même décomposition. Le tableau 4.2 compare le temps de décomposition entre les différentes méthodes de décomposition avec 96 *workers* avec les instances `fzn 1` et le facteur de gain su de chaque algorithme de décomposition. On note que $Dec_{//1}$ améliore Dec_{seq} par un facteur de gain de 3,8 et $Dec_{//2}$ améliore $Dec_{//1}$ par un facteur de gain de 3,4. Ainsi, la nouvelle méthode de décomposition parallèle améliore clairement la décomposition.

Instance	Seq.		$Dec_{//2}$		$Dec_{//1}$		Dec_{seq}	
	t_0	su_w	t_d	su	t_d	su	t_d	su
	s	r	s	r	s	r	s	r
allinterval_15	220,0	83,5	0,3	75,1	0,6	69,1	3,1	38,2
magicsequence_40000	316,6	116,8	1,4	76,6	8,5	28,2	21,5	13,1
sportsleague_10	170,1	74,4	0,5	61,6	2,2	38,1	9,4	14,5
sb_sb_13_13_6_4	135,1	82,4	0,7	57,5	1,4	44,6	12,9	9,3
quasigroup7_10	287,0	76,5	0,6	65,6	3,8	38,1	13,1	17,0
non_non_fast_6	582,4	80,9	8,0	38,3	13,0	28,8	29,7	15,8
golombruler_13	1303,9	94,2	0,3	92,1	4,5	71,0	8,9	57,2
warehouses	139,4	219,9	0,7	108,3	4,9	25,1	14,8	9,1
setcovering	88,4	74,6	0,2	65,3	2,4	24,4	4,7	15,1
depot_placement_att48_5	113,3	74,9	0,4	60,3	1,4	38,9	6,2	14,7
depot_placement_rat99_5	20,3	76,9	0,4	30,9	0,4	32,1	6,9	2,9
fastfood_ff58	22,3	84,7	0,6	27,4	0,6	26,4	5,7	3,7
open_stacks_01_problem	99,1	78,1	0,3	62,1	1,9	31,4	14,1	6,4
open_stacks_01_wbp	180,7	72,2	0,9	53,0	3,2	31,9	21,7	7,5
sugiyama	237,5	80,5	1,3	56,5	5,0	29,7	33,0	6,6
patternSetMiningGerman	103,9	73,4	0,9	44,7	2,2	28,7	11,6	8,0
radiation_03	108,1	70,6	0,3	59,0	1,7	33,9	10,6	8,9
talent_scheduling_film	243,3	97,8	2,5	48,3	10,5	18,7	31,1	7,3
Temps total (s pour secondes) ou M.G (r pour ratio sans unité)	4371,1	85,8	**20,2**	**57,0**	68,1	33,5	259,0	10,7

TABLE 4.2 – Comparaison des différents algorithmes de décomposition avec 96 *workers* sur les instances `fzn 1`.

4.2.6 Décomposition statique et dynamique

La décomposition parallèle statique implémentée dans `Gecode` renvoie toujours le nombre de sous-problèmes souhaité $p(w) = \#sspw \times w$ ($\#sspw = 30$). À l'inverse, la décomposition dynamique implémentée dans `Choco2` ne le garantit pas, car les sous-problèmes sont générés durant le parcours de l'arbre de recherche. La figure 4.6 est un diagramme en boîtes présentant le nombre de sous-problèmes par *worker* ($p \div w$) avec `Choco2`.

Pour chaque nombre de *workers* $w \in \{16, 80, 512\}$, on effectue les décompositions des instances du `xcsp` avec les différentes stratégies de recherche, à savoir `lex`, `dom`, `dom/ddeg`,`dom/wdeg`, `dom/bwdeg`, et `impact`.

La décomposition de `Choco2` obtient des gains de performance satisfaisants (la plupart du temps entre 10 et 100 sous-problèmes par *worker*) tout en respectant autant que possible la stratégie de recherche assignée. Cependant, quelques anomalies se produisent. Tout d'abord, la décomposition est très sensible à la forme de l'arbre de recherche. Parfois, le modèle ne contient que quelques variables avec de grands domaines qui empêchent d'avoir une décomposition aboutissant à une bonne répartition de charge. Par exemple, le premier et le second niveau de l'arbre de recherche résultant de la décomposition avec l'instance `knights-80-5` contiennent respectivement plus de 6000 et 50000 nœuds. Il peut y avoir également une mauvaise surestimation de la taille de l'arbre de recherche, particulièrement si le branchement dans l'arbre de recherche donne un grand nombre de choix possibles. Par exemple, la taille du second niveau de l'arbre de recherche résultant de la décomposition avec l'instance `fapp07-0600-7` est estimée autour de 950 nœuds alors qu'il contient plus de 6000 nœuds. En revanche, une mauvaise surestimation peut se produire si les nœuds supérieurs sont éliminés d'un arbre de recherche correspondant à des

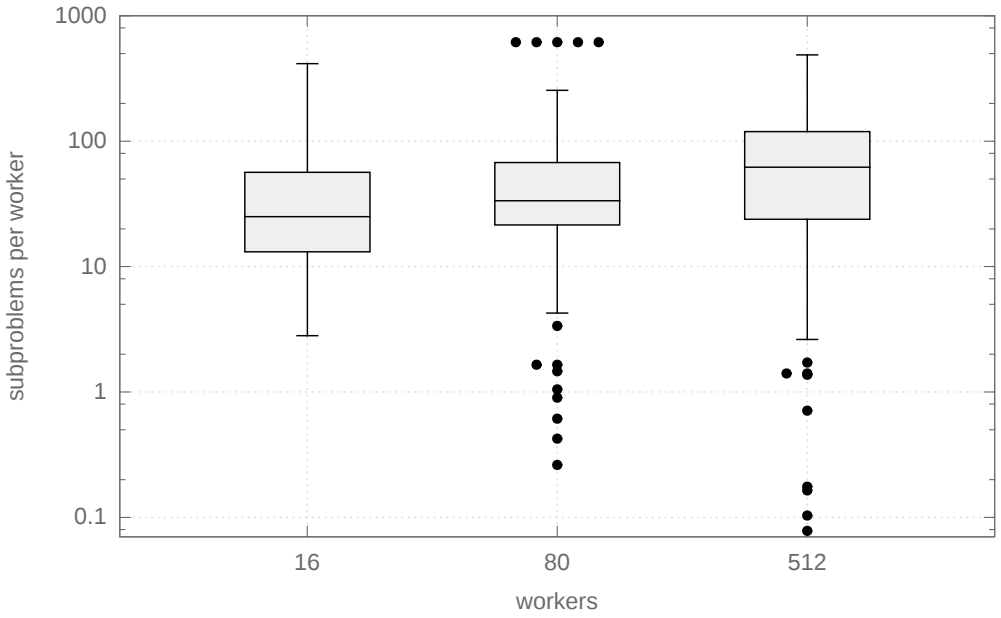

FIGURE 4.6 – Ratio des sous-problèmes par rapport au *workers* $p \div w$ sur `Choco2` sur les instances `xcsp` ($w = 16, 80, 512$).

sous-arbres ayant une faible arité. De plus, la décomposition est très précise pour les arbres de recherche à faible arité, par exemple avec la stratégie de recherche `dom/bwdeg`.
En conséquence, la décomposition parallèle de `Gecode` est statique et exacte alors que celle de `Choco2` est dynamique, mais surestime la plupart du temps.

La figure 4.7 donne la part des décompositions terminées en fonction du temps pour `Choco2` et `Gecode`. Les temps de `Choco2` sont reportés pour toutes les stratégies de recherche avec les instances `xcsp`. Les temps de `Gecode` sont reportés pour les instances `xcsp` et `fzn`.
À cause des différentes implémentations, les temps reportés de `Choco2` et de `Gecode` sont légèrement différents. En effet, seul le temps de décomposition t_d est donné pour `Gecode`. Dans `Gecode`, les sous-problèmes deviennent disponibles une fois que la décomposition est terminée.
Le temps de décomposition avec `Choco2` prend en compte l'estimation du temps t_e, le temps de remplir la queue de sous-problèmes, et le temps pris par les *workers* afin de vider la queue. Dans `Choco2`, les sous-problèmes deviennent disponibles immédiatement durant la décomposition. On rappelle que la décomposition démarre après que l'estimation soit terminée, c'est-à-dire lorsque le premier sous-problème devient disponible.
Dans ces deux cas, le temps reporté est une borne inférieure du temps de résolution pour les solveurs parallèles.

La décomposition de `Gecode` est plus rapide que celle de `Choco2` à l'exception des deux instances `crossword`, principalement à cause de leur parallélisation. En effet, la décomposition de `Gecode` est souvent plus rapide que l'estimation de `Choco2`. On rappelle que les *workers* avec `Choco2` sont inactifs seulement durant l'estimation de la profondeur

qui prend la plupart du temps 5 secondes.
On note que l'instance `knights-80-5` obtient le temps de décomposition le plus long avec `Choco2` (environ 800 secondes). En effet, la structure du problème est inadaptée pour l'algorithme de décomposition : il n'y a seulement que peu de variables ayant des domaines très grands (plus de 6000 valeurs) ; il n'y a pratiquement pas de réduction de domaine dans le haut de l'arbre de recherche, cependant la propagation est longue.
La décomposition statique de `Gecode` est efficace et exacte alors que la décomposition dynamique de `Choco2` offre plus de flexibilité, mais elle est moins robuste.

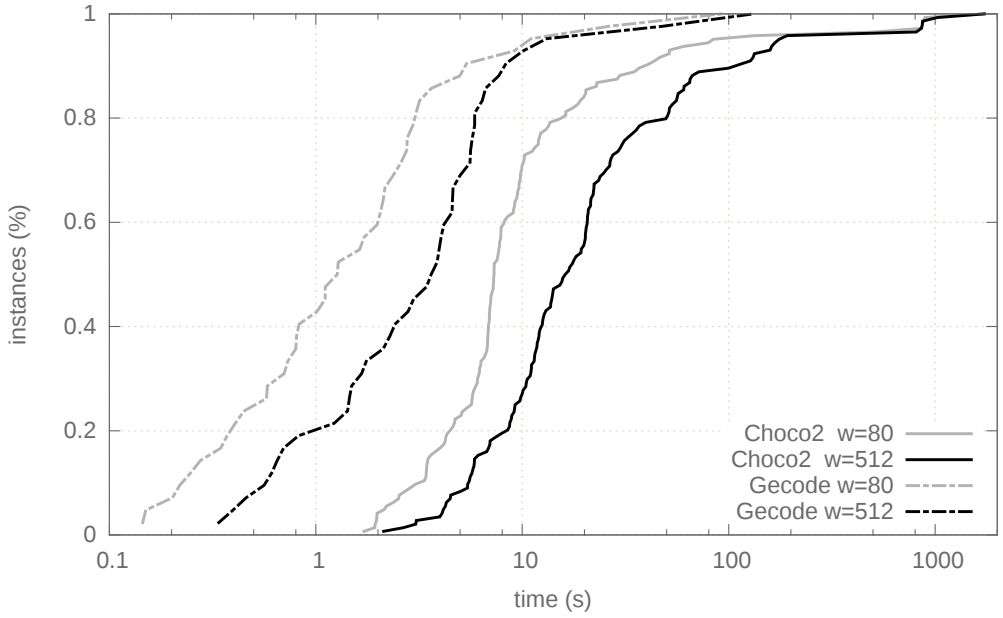

FIGURE 4.7 – Temps de décomposition ($w = 16, 80, 512$).

4.2.7 Influence des stratégies de recherche dans la décomposition

stratégie du *master*	lex		dom	
stratégie du *worker*	lex	dom	lex	dom
Instance	t	t	t	t
`costasArray-14`	**191,2**	191,4	240,9	240,0
`knights-80-5`	1138,3	1141,4	1137,9	**1133,1**
`latinSquare-dg-8_all`	479,4	470,6	**323,8**	328,1
`lemma-100-9-mod`	109,7	**101,8**	125,9	123,4
`ortholatin-5`	248,7	243,1	**242,8**	249,9
`pigeons-14`	1003,8	953,2	956,3	**899,1**
`quasigroup5-10`	182,2	188,5	125,3	**123,5**
`queenAttacking-6`	872,4	867,8	**598,3**	622,5
`series-14`	**39,3**	42,3	40,0	**39,3**
`squares-9-9`	**126,8**	127,8	1206,5	1213,0

TABLE 4.3 – Temps de résolution pour différentes stratégies au niveau de la décomposition (*master*) et de la résolution (*worker*) avec les instances `xcsp 1` ($w = 80$).

Dans le but d'analyser l'influence des stratégies de recherche dans la décomposition et dans la résolution, nous appliquons une stratégie dans la décomposition (au niveau du *master*) et une autre stratégie dans la résolution (au niveau du *worker*). Le tableau 4.3 montre les différents temps de résolution avec les instances `xcsp 1` sur 80 *workers*, en assignant une stratégie (`lex` ou `dom`) pour le *master* et une stratégie (`lex` ou `dom`) pour le *worker*. Nous observons d'abord que les instances suivantes `latinSquare-dg-8_all`, `lemma-100-9-mod`, `ortholatin-5` et `queenAttacking-6` réduisent leur temps de résolution en appliquant différentes stratégies pour la décomposition et la résolution (*master*-`lex`/ *worker*-`dom` et *master*-`dom`/ *worker*-`lex`). Pour les autres instances, nous remarquons qu'il y a peu d'influence des stratégies lorsqu'elles sont combinées, mais nous observons que la stratégie appliquée au niveau de la décomposition a une très grande influence dans le temps de résolution comme nous pouvons le constater avec les instances `costasArray-14`, `latinSquare-dg-8_all`, `pigeons-14`, `quasigroup5-10` et `queenAttacking-6` en utilisant `dom` au niveau de la décomposition. Seule l'instance `squares-9-9` a un meilleur avantage en temps de résolution d'utiliser `lex` au niveau de la décomposition.

4.3 Évaluation des performances d'EPS sur les différentes architectures

Nous avons analysé la décomposition et comparé les différents algorithmes d'EPS ainsi que l'influence des stratégies de recherche. Nous analysons maintenant les performances d'EPS sur les différentes architectures utilisées. Pour chaque architecture, nous comparons EPS avec le *work stealing*. La comparaison entre EPS et les méthodes de *portfolio* est présentée et commentée en dernier.

4.3.1 Machine multi-cœurs (*fourmis*)

Nous utilisons seulement la technologie *threads* pour résoudre les instances `xcsp 1`.

4.3.1.1 Efficacité et *hyperthreading* ($w = 40, 80$)

Nous testons l'apport de l'*hyperthreading* et nous montrons son efficacité pour la méthode EPS.

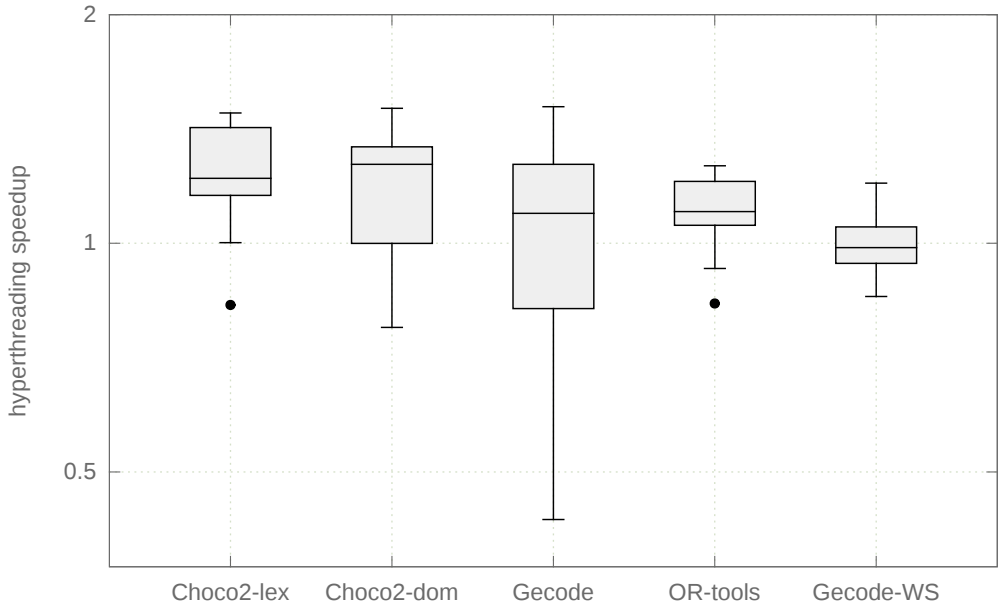

FIGURE 4.8 – Facteurs de gain améliorés grâce à *l'hyperthreading* ($w = 40, 80$).

La figure 4.8 est un diagramme en boîtes des différents gains fournis par la technologie *hyperthreading* pour chaque solveur parallèle `Choco2`, `Gecode` et `OR-tools`. `Choco2` est testé avec les stratégies de recherche `lex` et `dom` alors que `Gecode` et `OR-tools` n'utilisent que la stratégie `lex`.

Nous comparons également avec l'approche du *work stealing* proposée par [Schulte 2000] que l'on nomme `Gecode`-WS.

Les facteurs de gains indiquent le nombre de fois qu'un solveur parallèle utilisant 80 *workers* est plus rapide que lorsqu'il utilise 40 *workers* (au maximum 2 fois). L'*hyperthreading* améliore nettement l'efficacité d'EPS malgré la demande de ressources élevée en temps de calcul alors que les performances avec le *work stealing* restent inchangées.
Mis à part les instances `lemma-100-9-mod` et `squares-9-9`, `Choco2` et `OR-tools` sont plus rapides avec 80 *workers*. Pour l'instance `lemma-100-9-mod`, la décomposition dans `Choco2` avec 80 *workers* prend plus de temps et génère plus de sous-problèmes qu'avec 40 *workers*. Avec `OR-tools`, la décomposition prend seulement quelques secondes et il est donc difficile d'améliorer encore plus son efficacité. Pour l'instance `squares-9-9`, les décompositions changent également, mais prennent plus de temps.
Avec le solveur `Gecode`, l'efficacité est également réduite pour d'autres instances et l'intérêt d'utiliser *l'hyperthreading* est moins évident.

Le tableau 4.4 donne les temps de résolution et les facteurs de gain des solveurs parallèles avec 80 *workers* pour les instances de `xcsp 1`. Premièrement, les implémentations d'EPS sont plus rapides et plus efficaces que le *work stealing* `Gecode`-WS. EPS atteint souvent des facteurs de gain linéaires alors que `Gecode`-WS n'y arrive jamais. Et ce qui est le plus regrettable c'est que trois instances ne sont pas résolues dans la limite de temps (12h) en utilisant le *work stealing* de `Gecode` alors qu'elles utilisent le solveur en séquentiel.
Pour `Choco2`, la stratégie de recherche `dom` est plus efficace que la stratégie `lex`. La décomposition est responsable de cette mauvaise performance pour les instances `knights-80-5` et `lemma-100-9-mod`. La décomposition de l'instance `knights-80-5` prend plus de 1100 secondes, empêchant ainsi toute performance possible, et génère trop de sous-problèmes (environ 50 000). En effet, `knights-80-5` n'a seulement que quelques variables avec de très grands domaines et l'arbre de recherche résultant est très étendu. Ce problème est mieux géré par la décomposition séquentielle par contrainte de table en séquentiel de `OR-tools` et également avec la décomposition parallèle de `Gecode`. On note que leurs temps de résolution en séquentiel (t_0) sont respectivement 20 et 40 fois plus élevés que les temps de résolution avec `Choco2`. De même, la longue décomposition de l'instance `lemma-100-9-mod` conduit à un faible facteur de gain avec `Choco2`.
Nous observons que `Gecode` et `OR-tools` sont souvent plus efficaces et plus rapides que `Choco2`. `Choco2`, `Gecode` et `OR-tools` ont différents comportements lorsqu'ils utilisent la stratégie de recherche `lex`, car les méthodes de décomposition diffèrent d'un solveur à un autre. Particulièrement, la décomposition parallèle par contrainte de table ne préserve pas l'ordre des feuilles de l'arbre de recherche.

	Choco2-lex		Choco2-dom	
	t	su	t	su
costasArray-14	191,2	31,4	240,0	**38,8**
knights-80-5	1138,3	1,2	1133,1	1,5
latinSquare-dg-8_all	479,4	39,0	328,1	39,2
lemma-100-9-mod	109,7	4,0	123,4	4,1
ortholatin-5	248,7	30,0	249,9	36,0
pigeons-14	1003,8	13,8	899,1	15,5
quasigroup5-10	182,2	30,7	123,5	**32,5**
queenAttacking-6	872,4	23,4	**622,5**	**28,5**
series-14	39,3	29,9	39,3	32,9
squares-9-9	126,8	19,0	1213,0	16,1
M.G (su) ou M.A (t)	439,2	15,9	497,2	17,4

	Gecode		OR-tools		Gecode-WS	
	t	su	t	su	t	su
costasArray-14	62,3	19,1	**50,9**	33,4	594,0	2,0
knights-80-5	**548,7**	**37,6**	2173,9	18,5	–	–
latinSquare-dg-8_all	251,7	**42,0**	**166,6**	35,2	4488,5	2,4
lemma-100-9-mod	6,7	10,1	**1,8**	**22,9**	3,0	22,3
ortholatin-5	421,7	13,5	**167,7**	**38,1**	2044,6	2,8
pigeons-14	**211,8**	**39,1**	730,3	18,5	–	–
quasigroup5-10	18,6	26,4	**17,0**	**36,9**	22,8	21,5
queenAttacking-6	15899,1	?	–	–	–	–
series-14	**11,3**	**34,2**	16,2	28,7	552,3	0,7
squares-9-9	**17,9**	18,4	81,4	**35,0**	427,8	0,8
M.G (su) ou M.A (t)	1745,0	24,0	378,4	28,7	1161,9	3,3

TABLE 4.4 – Temps de résolution et facteurs de gain sur une machine multi-cœurs ($w = 80$).

4.3.1.2 Variations par rapport au problème du *n-queens*

Ici, nous vérifions l'efficacité de la décomposition dynamique en la testant avec quatre modèles pour le problème du *n-queens* (ici $n = 17$). Le problème du *n-queens* est de placer n reines sur un échiquier de taille $n \times n$ de sorte que deux reines ne se menacent pas

mutuellement. Ici, nous énumérons toutes les solutions et les stratégies de recherche `lex` et `dom` sont des choix raisonnables en terme de temps de résolution. Les modèles utilisés sont :

- (AC) : utilisation des contraintes globales `alldiff` [Régin 1994] réalisant la consistance d'arc ;
- (BC) : utilisation des contraintes globales `alldiff` réalisant la consistance des bornes ;
- (NEQ) : utilisation des contraintes arithmétiques de différence ($x < y$) ;
- (JC) : utilisation d'une contrainte globale dédiée [Rossi 2006].

Le tableau 4.5 donne les temps de résolution et les facteurs de gain de `Choco2` avec 80 *workers* où la profondeur de l'arbre de recherche qu'atteint la décomposition est de 3 ou 4. Nous remarquons que contrairement à ce que nous attendons que notre méthode de décomposition donne d'excellents résultats, avec un facteur de gain linéaire jusqu'à 40 *workers*, à l'exception du modèle JC. Ce qui est dommage car le modèle JC est clairement le meilleur modèle avec la résolution séquentielle et qu'en l'utilisant dans la résolution parallèle, nous obtenons des résultats similaires au modèle NEQ en parallèle. On note également que la stratégie de recherche `dom` est toujours un meilleur choix que `lex`. De façon surprenante, les nombres de sous-problèmes générés avec `dom` ($p = 2756$ si $d = 3$ et $p = 25660$ si $d = 4$) sont les mêmes, quel que soit le modèle alors que les nombres totaux de nœuds sont différents.

	`lex`	$d=3$		$d=4$		`dom`	$d=3$		$d=4$	
Modèle	t_0	t	su	t	su	t_0	t	su	t	su
BC	32155,5	838,8	38,3	835,1	38,5	24808,2	640,4	38,7	**635,5**	**39,0**
AC	119277,1	3070,2	38,8	3038,9	**39,3**	90761,2	2336,2	38,8	**2314,7**	39,2
NEQ	8930,0	280,7	31,8	241,9	36,9	6870,5	188,8	36,4	**181,1**	**37,9**
JC	4132,3	202,4	20,4	196,9	21,0	3409,3	**140,6**	**24,2**	148,8	22,9

TABLE 4.5 – Variations par rapport au *n-queens* de taille 17 avec une machine multi-cœurs (`Choco2`, $w = 80$).

Il est intéressant de noter que plusieurs travaux ont reporté des résultats similaires (par exemple avec *n-queens* de taille 15 [Zoeteweij 2004]), où le nombre de *workers* est de 16 ; les résultats de [Pedro 2010] ont donné des facteurs de gain avoisinant 20 pour 24 *workers*. Le *framework* Bobpp utilisé dans [Menouer 2014a] donne des facteurs de gain de 7.

Avec le problème du *n-queens* de taille 16, les résultats de [Menouer 2014a] donnent un facteur de gain de 8,52 avec 12 *workers* et ceux de [Pedro 2010] obtiennent un facteur de gain de 21,5 avec 24 *workers*.

Quant au problème du *n-queens* de taille 17, les résultats dans [Bordeaux 2009] que nous avons comparées avec la décomposition statique simple dans la première analyse de la décomposition, donnent un facteur de gain linéaire jusqu'à 30 cœurs, mais entre

30 et 64 cœurs, le facteur de gain reste à 30. Pour le même problème, les résultats de [Machado 2013] donnent un gain linéaire jusqu'à 512 *workers* en utilisant la méthode du *work stealing* de manière à respecter la hiérarchie des différents nœuds du *cluster* avec l'approche du *work stealing* local et distant.

Le cadre expérimental précédent est évidemment en faveur d'EPS parce que nous explorons l'espace de recherche de manière exhaustive, et le problème est très symétrique. La figure 4.9 montre la part de sous-problèmes résolus exprimés en pourcentage durant un temps donné pour les différents modèles lorsque la profondeur de l'arbre de recherche qu'atteint la décomposition est à 3. Il peut être interprété comme une fonction de répartition empirique de la variable aléatoire qui représente le temps de résolution des sous-problèmes. Cela confirme que l'espace de recherche est très symétrique, car les pentes sont élevées et les fonctions convergent rapidement. En outre, plus le niveau de consistance est élevé, à savoir les modèles BC et AC, plus la variance est réduite. Mais, dans ce cas, on augmente la moyenne en comparaison avec les modèles NEQ et JC. Il montre également que les faibles facteurs de gain du modèle JC ne sont probablement pas causés par des problèmes d'équilibrage de charge parce le modèle NEQ présente des résultats où il y a une plus grande moyenne et une variance élevée.
Enfin, on retrouve des résultats similaires avec les *n-queens* de taille 15 et 16.

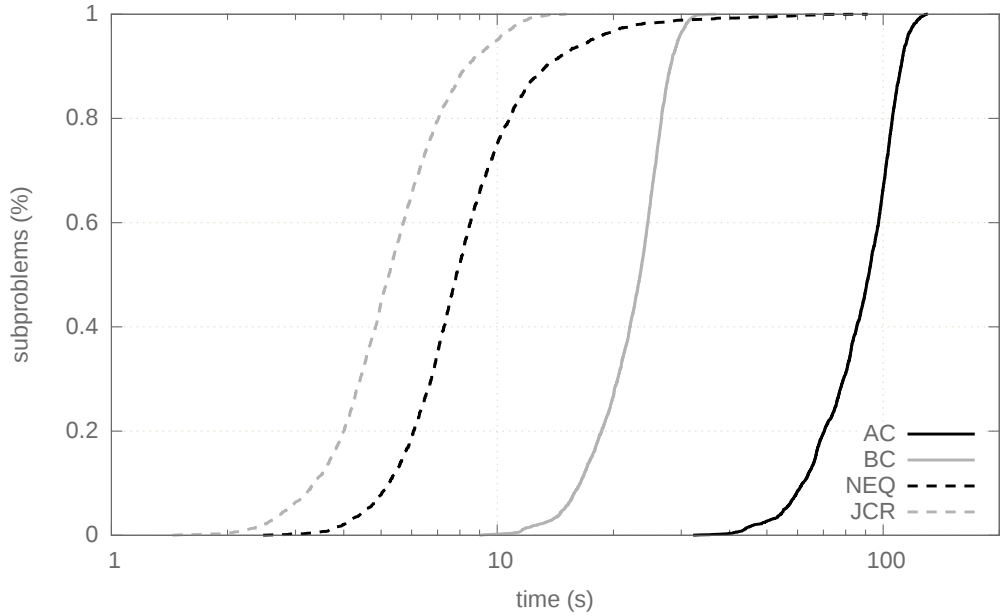

FIGURE 4.9 – Les différents temps de résolution des sous-problèmes de *n-queens* 17 (`Choco2`, $w = 80$, $d = 3$).

4.3.2 Centre de calcul (*cicada*)

Dans cette section, nous utilisons `Gecode` avec MPI et `Choco2` avec les *threads*. L'*hyperthreading* est désactivé dans le centre de calcul, car les performances sont généralement dégradées avec deux *workers* par cœur. Mais, ce n'est pas toujours le cas lorsqu'on change la décomposition. Nous utilisons ainsi seulement un *worker* par cœur ($c = w$).

4.3.2.1 `Choco2` ($w = 16$)

Nous étudions les performances de `Choco2` utilisant 16 *workers* avec les instances `xcsp` en utilisant toutes les stratégies de recherche présentées dans le chapitre 2.

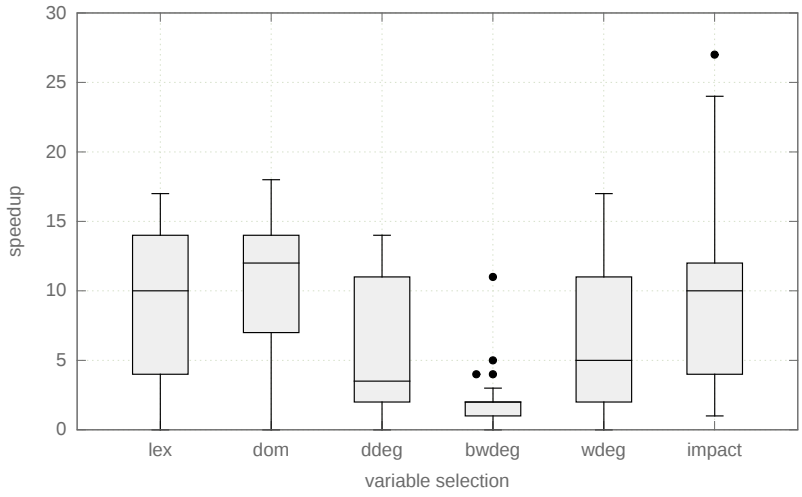

FIGURE 4.10 – Facteurs de gain et temps de résolution de `Choco2` en utilisant différentes stratégies de recherche.

La figure 4.10 est un diagramme en boîtes représentant la répartition des facteurs de gain pour chaque stratégie de recherche.

En premier lieu, les facteurs de gain sont très bas pour la stratégie de recherche `dom/bwdeg`, car la décomposition n'est pas efficace. La stratégie de recherche `dom/bwdeg` implique un branchement binaire qui établit la contrainte $x = a$ dans la branche gauche et $x \neq a$ dans les autres branches. Ainsi, la charge de travail entre la branche gauche et les autres branches est souvent mal répartie. En second lieu, les stratégies de recherche `lex` et `dom` qui n'utilisent pas de règles d'apprentissage permettent à l'algorithme parallèle d'être plus efficace et plus robuste en termes de facteur de gain. Pour les autres stratégies de recherche qui utilisent des règles d'apprentissage, l'algorithme parallèle peut explorer un arbre de recherche différent de l'algorithme séquentiel. En effet, la décomposition explore seulement le haut de l'arbre de recherche. Son exploration peut changer avec des règles d'apprentissage et ainsi peut momentanément changer les décisions dans le parcours des branches de l'arbre de recherche. Le *worker* peut apprendre des règles d'apprentissage seulement à partir de ses sous-problèmes (niveau local), et non celles venant de tout l'arbre de recherche (niveau global).

Ici, on remarque qu'il y a souvent une surcharge dans l'exploration des nœuds de l'arbre de recherche pour résoudre l'instance `queensKnights-20-5-mul` (12 fois plus de nœuds) et on peut avoir parfois des facteurs de gain linéaires avec l'instance `quasigroup5-10` (3 fois moins de nœuds). Enfin, les faibles facteurs de gain sont obtenus avec toutes les stratégies de recherche.

Le tableau 4.6 donne tous les temps de résolution ainsi que les facteurs de gain. Un tiret dans le tableau indique que l'instance n'a pas pu être résolue dans le temps imparti.

4.3.2.2 `Gecode` ($w = 16, 512$)

Dans cette section, nous comparons l'approche EPS et le *work stealing* avec les instances `xcsp` sur 16 *worker* et 512 *workers*. Nous avons rallongé le temps imparti à 24 heures.

Le tableau 4.7 montre une comparaison entre EPS et une implémentation du *work stealing* sur le centre de calcul *cicada* avec les instances `xcsp` utilisant soit 16 ou 512 *workers*. Le *work stealing* est une implémentation du MPI basée par les travaux de [Nielsen 2006]. Avec 16 *workers*, EPS est meilleur que le *work stealing*, c'est-à-dire son facteur de gain moyen est de 13,5 alors que celui du *work stealing* est de 7,4. Lorsqu'on augmente le nombre de *workers* à 512, EPS surpasse le *work stealing* ; le facteur de gain moyen d'EPS est de 246,2 alors que celui *work stealing* est de 33,4. On note qu'EPS et le *work stealing* résolvent plusieurs instances qui ne sont pas résolues en séquentiel comme *langford-3-17* et *knights-20-9*. Ainsi, nous ne pouvons pas comparer les facteurs de gain, car le temps de résolution en séquentiel n'existe pas, mais nous pouvons comparer les temps de résolution en parallèle et nous remarquons qu'EPS est encore meilleure que le *work stealing* ; EPS résout l'instance *langfort-3-7* en 713,5 secondes alors que le *work stealing* la résout en 7443,5 secondes (environ 10 fois plus long).

Dans le même cadre, on présente la même comparaison que précédemment avec les instances `fzn`. Le tableau 4.8 présente donc une comparaison entre EPS et le *work stealing* de `Gecode` avec 512 *workers* avec les instances `fzn`. Le gain de performance moyen du *work stealing* est de 5,4. On remarque que plusieurs instances ne sont pas résolues avec 512 *workers*. Pire encore, certaines ne sont pas résolues en parallèle alors qu'elles étaient résolues en séquentiel. L'utilisation de la communication à outrance par le *work stealing* explique ce problème. Les *workers* prennent plus de temps pour communiquer que de résoudre les sous-problèmes. En revanche, EPS atteint un gain de performance moyen de 223,9. La méthode EPS est meilleure que le *work stealing* sur toutes les instances sélectionnées. EPS arrive à obtenir des gains de performance linéaires avec 7 instances. Les autres instances étant booléennes, l'estimation de la profondeur d'EPS pour générer les sous-problèmes est mauvaise, car elle s'appuie sur le produit des puissances de 2 (domaines booléens).

4.3.2.3 Analyse des facteurs de gain avec `Gecode` ($w = 16, 32, 64, 128, 256, 512$)

La figure 4.11 est un diagramme en boîtes répartissant des facteurs de gain pour différents nombres de *workers* ($w = c = 16, 32, 64, 128, 256, 512$) pour les instances `fzn`. Les résultats montrent que les facteurs de gain d'EPS suivent un schéma de progression

Instances	lex		dom		dom/ddeg	
	su	*t*	*su*	*t*	*su*	*t*
cc-15-15-2	**5,2**	1947,1	**?**	25701,7	–	–
costasArray-14	**12,4**	**500,4**	12,1	641,9	8,6	895,3
crossword-m1-words-05-06	4,4	506,1	–	–	–	–
crossword-m1c-words-vg7-7_ext	0,6	2376,9	0,6	**1173,9**	0,5	1316,2
fapp07-0600-7	–	–	–	–	–	–
knights-20-9	17,2	359,3	17,3	353,9	14,9	357,4
knights-25-9	17,8	855,3	**18,0**	840,6	13,3	986,1
knights-80-5	2,0	**708,5**	2,0	726,9	2,1	716,4
langford-3-17	**?**	38462,7	**12,5**	**708,3**	2,5	5701,6
langford-4-18	**?**	40465,2	**14,4**	**148,2**	2,2	1541,9
langford-4-19	–	–	**16,9**	**747,2**	**0,1**	–
latinSquare-dg-8_all	14,3	1161,7	12,2	903,2	**14,4**	812,0
lemma-100-9-mod	**4,1**	**110,5**	3,7	117,6	3,7	180,4
ortholatin-5	13,5	572,6	13,5	558,9	11,5	475,5
pigeons-14	9,8	**1330,1**	8,3	1492,6	**11,8**	1471,6
quasigroup5-10	12,6	397,2	12,9	**277,3**	3,6	1156,6
queenAttacking-6	7,3	2596,7	**10,6**	1411,8	4,2	4789,7
queensKnights-20-5-mul	–	–	–	–	–	–
ruler-70-12-a3	16,8	137,4	**17,5**	2410,6	–	–
ruler-70-12-a4	3,9	6832,0	4,7	4021,1	2,2	7549,2
scen11-f5	–	–	–	–	–	–
series-14	**14,8**	**77,8**	12,4	89,1	3,4	9828,6
squares-9-9	10,5	220,7	7,2	1987,4	9,4	129,7
squaresUnsat-19-19	–	–	–	–	–	–
M.G (*su*) ou M.A (*t*)	7,5	5243,1	**8,6**	2332,2	4,8	2369,3

Instances	dom/bwdeg		dom/wdeg		impact	
	su	*t*	*su*	*t*	*su*	*t*
cc-15-15-2	4,6	**1524,9**	2,1	2192,1	**?**	31596,1
costasArray-14	2,4	4445,0	11,4	649,9	10,5	652,2
crossword-m1-words-05-06	1,9	492,0	5,1	204,6	2,9	**179,5**
crossword-m1c-words-vg7-7_ext	0,7	1471,3	0,6	1611,9	**1,9**	4689,6
fapp07-0600-7	2,1	**1069,5**	1,8	2295,7	–	–
knights-20-9	1,3	5337,5	**17,5**	491,3	16,5	**215,4**
knights-25-9	1,3	13264,8	14,1	1645,2	16,9	**550,8**
knights-80-5	0,9	1829,5	**3,4**	1395,6	2,5	896,3
langford-3-17	1,9	6397,5	3,7	3062,2	**?**	5995,6
langford-4-18	2,1	1307,1	4,8	538,3	10,0	1041,5
langford-4-19	2,3	7280,1	5,6	2735,3	**?**	4778,9
latinSquare-dg-8_all	4,2	416,9	11,3	294,8	5,0	**28,7**
lemma-100-9-mod	3,5	154,4	3,5	145,3	2,5	226,8
ortholatin-5	11,6	453,1	**13,7**	**362,4**	10,6	641,7
pigeons-14	2,6	6331,1	5,1	2993,3	4,2	3637,2
quasigroup5-10	5,2	733,5	7,9	451,5	**27,8**	308,4
queenAttacking-6	1,9	2891,0	5,4	706,4	6,2	**427,1**
queensKnights-20-5-mul	0,2	**1517,8**	**1,0**	5209,5	–	–
ruler-70-12-a3	2,4	51,5	6,7	42,8	12,1	**24,8**
ruler-70-12-a4	0,9	1412,0	2,3	1331,3	**24,4**	**102,9**
scen11-f5	**0,1**	38698,7	**0,1**	–	–	–
series-14	2,6	1232,2	9,9	338,9	8,5	346,5
squares-9-9	2,2	697,2	**11,0**	**115,9**	10,8	138,9
squaresUnsat-19-19	1,2	3766,1	**2,9**	**3039,8**	–	–
M.G (*su*) ou M.A (*t*)	1,6	4282,3	4,9	**1385,0**	7,7	2823,9

TABLE 4.6 – Détails des facteurs de gain et temps de résolution de Choco2 en utilisant différentes stratégies de recherche.

Instances		$w = 16$				$w = 512$			
		EPS		WS		EPS		WS	
	t_0	t	su	t	su	t	su	t	su
costasArray-14	879,4	**64,4**	**13,6**	69,3	12,7	**3,6**	**243,8**	17,7	49,8
crossword-m1c-words-vg7-7_ext	3159,8	**240,6**	**13,1**	482,1	6,6	**18,7**	**168,6**	83,1	38,0
crossword-m1-words-05-06	2486,0	**171,7**	**14,5**	178,5	13,9	**13,3**	**187,3**	57,8	43,0
knights-20-9	–	**5190,7**	?	38347,4	?	**153,4**	?	3312,4	?
knights-25-9	–	**7462,3**	?	–	–	**214,9**	?	–	–
knights-80-5	16253,9	**1413,7**	**11,5**	8329,2	2,0	**49,3**	**329,8**	282,6	57,5
langford-3-17	–	24351,5	?	**21252,3**	?	**713,5**	?	7443,5	?
langford-4-18	–	**3203,2**	?	25721,2	?	**94,6**	?	5643,1	?
langford-4-19	–	**26871,2**	?	–	–	**782,5**	?	–	–
latinSquare-dg-8_all	8047,5	**613,5**	**13,1**	621,2	13,0	**23,6**	**341,7**	124,4	64,7
lemma-100-9-mod	50,1	**3,4**	**14,7**	5,8	8,6	**1,0**	**51,4**	2,5	19,7
ortholatin-5	4371,0	**309,5**	**14,1**	335,8	13,0	**10,4**	**422,0**	71,7	61,0
pigeons-14	5564,5	**383,3**	**14,5**	6128,9	0,9	**15,3**	**363,1**	2320,2	2,4
quasigroup5-10	364,3	**27,1**	**13,5**	33,7	10,8	**1,7**	**211,7**	9,8	37,3
queenAttacking-6	–	42514,8	?	**37446,1**	?	**1283,9**	?	9151,5	?
ruler-70-12-a3	1455,7	**96,6**	**15,1**	105,5	13,8	**3,7**	**389,3**	67,7	21,5
ruler-70-12-a4	2571,0	**178,9**	**14,4**	185,2	13,9	**6,0**	**429,5**	34,1	75,5
series-14	302,1	**22,5**	**13,4**	56,9	5,3	**1,1**	**264,0**	8,2	36,9
squares-9-9	254,3	**22,8**	**11,1**	44,3	5,7	**1,3**	**191,7**	7,6	33,7
M.G (su) ou M.A (t)	45759,6	**113141,7**	**13,5**	139343,4	7,4	**3391,8**	**246,2**	28637,9	33,5

TABLE 4.7 – Facteurs de gain et temps de résolution de Gecode avec les instances xcsp.

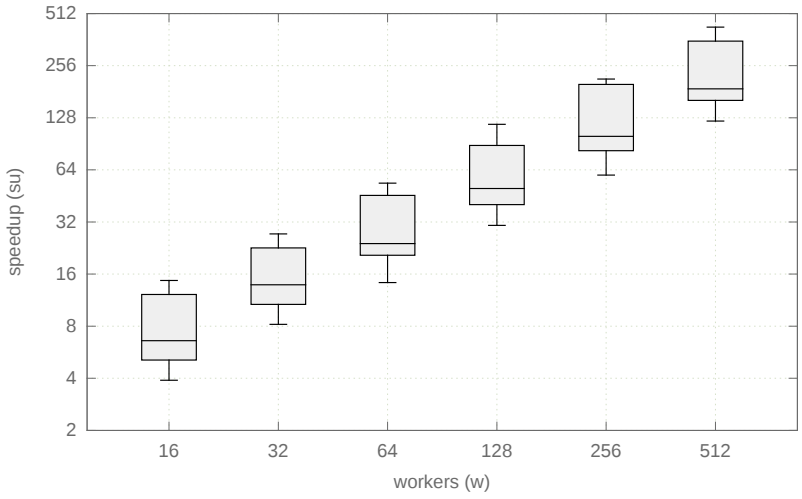

FIGURE 4.11 – Facteurs de gain de Gecode avec les instances fzn.

quasi-linéaire lorsque nous augmentons le nombre de *workers*. Nous remarquons également qu'EPS donne des facteurs de gain proche de $w/2$ lorsque $w = c$. Les détails des facteurs de gain sont décrits dans le tableau 4.9. On remarque que la résolution du *Golomb* et des instances du *market-split* (environ 400 for 512 *workers*) montre qu'EPS obtient de très bonnes performances avec des gains linéaires.

Instances		$w = 512$			
		WS		EPS	
	t_0	t	su	t	su
market_split_s5-02	3314,4	-	-	8,2	405,9
market_split_u5-09	3266,6	-	-	7,9	411,8
market_split_s5-06	3183,9	-	-	8,3	384,0
prop_stress_0600	2729,2	1426,4	1,9	14,1	193,1
nmseq_400	2505,8	-	-	10,4	240,4
prop_stress_0500	1350,6	670,0	2,0	8,4	161,6
fillomino_18	763,9	-	-	5,1	150,7
steiner-triples_09	604,9	79,0	7,7	1,8	332,0
nmseq_300	555,3	-	-	4,2	131,7
golombruler_13	1303,9	15,5	83,9	3,0	427,9
cc_base_mzn_rnd_test.11	3279,5	-	-	26,8	122,6
ghoulomb_3-7-20	2993,8	575,4	5,2	22,8	131,1
pattern_set_mining_k1_yeast	2871,3	299,8	9,6	15,7	183,2
still_life_free_8x8	2808,9	1672,8	1,7	16,8	166,9
bacp-6	2763,3	330,1	8,4	7,3	378,9
depot_placement_st70_6	2665,1	1902,9	1,4	11,3	235,1
open_stacks_01_wbp_20_20_1	1523,2	153,9	9,9	9,5	160,8
bacp-27	1499,7	579,6	2,6	4,6	326,5
still_life_still_life_9	1145,1	140,1	8,2	6,3	182,9
talent_scheduling_alt_film117	566,1	95,5	5,9	3,2	175,8
Temps total (**t**) ou M.G (**su**)	41694,5	7941	5,4	195,7	223,9

TABLE 4.8 – Comparaison entre le *work stealing* et EPS avec 512 *workers* avec les instances fzn.

4.3.3 Cloud Computing (*azure*)

Sur le *cloud* de Windows Azure, nous expérimentons avec les instances fzn. Il y a 32 cœurs qui sont disponibles. Sur ces 32 cœurs, il faut réserver 1 nœud maître qui gère les nœuds et réserver également des nœuds de communication (appelés nœuds *proxy*) qui facilitent l'équilibrage de charge et la communication entre les nœuds du *cluster*. En effet, les nœuds du *cluster* peuvent être éloignés les uns des autres, ce qui entraîne des ralentissements dans la communication. Selon la politique de *cloud* Azure, il faut minimum 2 nœuds *proxy* pour gérer la communication jusqu'à 400 nœuds de *cluster* et 1 nœud *proxy* est nécessaire pour 200 nœuds supplémentaires. Ces nœuds *proxy* sont gérés par le *cloud* (Azure) et nous n'avons aucun contrôle sur eux. Chaque nœud *proxy* prend deux cœurs ainsi il nous reste 27 cœurs. Par conséquent, nous pouvons réserver 27 nœuds pour la résolution de problème, avec 1 nœud pour 1 cœur. Cependant, chacun de ces nœuds est limité à 1,75 Go de mémoire RAM, ce qui nous permet de calculer seulement quelques instances

Instance	nombre de *workers*								
	$t_0(s)$	su							
	1w	8w	16w	32w	64w	96w	128w	256w	512w
market_split_s5-02	3314,4	7,3	14,2	25,4	50,7	69,7	101,5	201,7	405,9
market_split_u5-09	3266,6	7,3	14,3	25,7	51,5	68,6	103,0	207,4	411,8
market_split_s5-06	3183,9	6,4	12,7	24,0	48,0	64,0	96,0	197,5	384,0
prop_stress_0600	2729,2	3,8	6,7	16,1	24,1	32,2	48,3	104,2	193,1
nmseq_400	2505,8	4,1	7,2	15,0	30,1	40,1	60,1	117,7	240,4
prop_stress_0500	1350,6	2,5	4,4	13,1	20,2	26,9	40,4	81,8	161,6
fillomino_18	763,9	2,4	5,1	11,4	18,8	25,1	37,7	72,4	150,7
steiner-triples_09	604,9	5,7	12,3	21,7	41,5	55,3	83,0	143,2	332,0
nmseq_300	555,3	2,4	5,1	8,2	16,5	21,9	32,9	69,3	131,7
golombruler_13	1303,9	7,3	14,7	27,3	53,7	89,5	117,4	213,1	427,9
cc_base_mzn_rnd_test.11	3279,5	1,7	5,1	8,9	14,3	20,4	30,6	59,7	122,6
ghoulomb_3-7-20	2993,8	2,3	3,9	8,2	17,4	21,8	32,8	76,3	131,1
pattern_set_mining_k1_yeast	2871,3	2,9	5,8	11,5	23,9	30,5	45,8	91,6	183,2
still_life_free_8x8	2808,9	2,6	6,4	10,4	20,9	27,8	41,7	83,5	166,9
bacp-6	2763,3	6,7	12,1	23,7	47,4	63,1	94,7	212,4	378,9
depot_placement_st70_6	2665,1	3,4	7,3	14,7	29,4	39,2	58,8	147,6	235,1
open_stacks_01_wbp_20_20_1	1523,2	3,1	6,5	10,0	23,1	26,8	40,2	95,4	160,8
bacp-27	1499,7	5,6	11,2	20,4	43,8	54,4	81,6	214,3	326,5
still_life_still_life_9	1145,1	3,1	6,1	11,4	22,9	30,5	45,7	89,4	182,9
talent_scheduling_alt_film117	566,1	2,4	4,3	11,0	22,0	31,3	51,7	95,9	175,8
Temps total (t_0) ou M.G (su)	41694,5	3,7	7,5	14,7	28,3	37,9	56,7	117,3	223,9

TABLE 4.9 – Facteurs de gain détaillés pour chaque instance et pour chaque nombre de *workers* avec EPS.

de `fzn`. On peur réserver autrement en prenant des nœuds de huit cœurs, car chacun de ces nœuds possède 56 Go de mémoire RAM, ce qui est largement suffisant pour nos expériences. Nous avons donc réservé 3 nœuds (24 cœurs au total). Le tableau 4.10 montre une comparaison entre EPS et le *work stealing* dans le *cloud* de Microsoft Azure avec les instances `fzn` en utilisant 24 *workers*. Nous avons augmenté le nombre de sous-problèmes, car nous avons eu de meilleurs résultats ($\#sspw = 100$ au lieu de $\#sspw = 30$). Avec 24 *workers*, EPS a de meilleures performances que le *work stealing*, son facteur de gain moyen est de 21,8 alors que celui du *work stealing* est de 10,6. Le problème majeur du *work stealing* est l'utilisation d'un grand nombre de mécanismes de communication pour traiter les sous-problèmes. Un ralentissement de la communication vient également du besoin d'un grand nombre de messages échangés entre les nœuds du *cluster* et comme nous n'avons aucun contrôle sur les nœuds *proxy*, nous ne pouvons pas améliorer les échanges de messages de la meilleure façon.

4.3.4 Comparaison avec les méthodes du *portfolio*

Nous comparons EPS et plusieurs méthodes de *portfolio* sur différents solveurs. Les approches *portfolio* exploitent la variabilité des performances délivrées par plusieurs et différents solveurs, ou par plusieurs et différents paramètres de configuration d'un même solveur (par exemple l'utilisation de plusieurs stratégies de recherche). Nous comparons notre

Instance		EPS		Work Stealing	
	t_0	t	su	t	su
market_split_s5-02	11367,4	**467,1**	**24,3**	658,6	17,3
market_split_s5-06	11039,8	**452,7**	**24,4**	650,7	17,0
market_split_u5-09	11421,6	**468,1**	**24,4**	609,2	18,7
pop_stress_0600	9437,5	**874,8**	**10,8**	2195,7	4,3
nmseq_400	2924,2	**342,4**	**8,5**	943,2	3,1
pop_stress_0500	4384,8	**433,2**	**10,1**	811,0	5,4
fillomino_18	2227,1	**160,2**	**13,9**	184,6	12,1
steiner-triples_09	1870,9	**108,8**	**17,2**	242,4	7,7
nmseq_300	751,2	**114,5**	**6,6**	313,1	2,4
golombruler_13	3167,3	**154,0**	**20,6**	210,4	15,1
cc_base_mzn_rnd_test.11	8306,9	**1143,6**	**7,3**	2261,3	3,7
ghoulomb_3-7-20	4184,2	**618,2**	**6,8**	3366,0	1,2
still_life_free_8x8	8973,4	**931,2**	**9,6**	1199,4	7,5
bacp-6	6571,7	**400,8**	**16,4**	831,0	7,9
depot_placement_st70_6	7929,0	**433,9**	**18,3**	1172,5	6,8
open_stacks_01_wbp_20_20_1	5338,7	**302,7**	**17,6**	374,1	14,3
bacp-27	4256,8	**260,2**	**16,4**	548,4	7,8
still_life_still_life_9	3187,4	**189,0**	**16,9**	196,8	16,2
talent_scheduling_alt_film117	1677,8	**22,7**	**74,0**	110,5	15,2
M.G (su) ou M.A (t)	109017,6	**7878,1**	**21,8**	16878,9	10,6

TABLE 4.10 – Temps de résolution et facteurs de gain de Gecode sur le *cloud* de Microsoft Azure.

approche avec CPHydra [O'Mahony 2008], un *portfolio* de stratégies de recherche présentes dans OR-tools dont impact, dom, lex, et random, et un *portfolio* à taille fixe utilisant les solveurs Choco2, Gecode et AbsCon (CAG).

L'objectif de CPHydra est de savoir si une instance possède au moins une solution ou non en utilisant 3 solveurs (Choco2, Mistral, AbsCon) et arrête la résolution après 30 minutes. Utilisant le raisonnement sur les caractéristiques des instances (en anglais *features reasoning*) comme le nombre de variables ou la taille des domaines , CPHydra a pour objectif de prendre la meilleure stratégie de recherche de chaque solveur afin d'obtenir le plus rapidement possible le statut de la résolution. Nous l'avons intégré dans nos résultats à titre indicatif, mais comme il ne fait que de la recherche de la première solution, nous pouvons seulement comparer CPHydra avec les autres approches sur les instances qui ne possèdent aucune solution.

Le *portfolio* d'OR-tools proposé par [Perron 2012] est un *portfolio* des stratégies de recherche disponibles comme dom, impact, lex et random. Lorsque le nombre de *workers* est plus grand que le nombre de stratégies de recherche disponibles, chaque *worker* en excès prend une stratégie de recherche random avec des graines aléatoires différentes. Ces

Instances	EPS			
	Choco2 ($w = 16$)		Gecode	
	dom	dom/wdeg	$w = 16$	$w = 512$
cc-15-15-2	25701,7	2192,1	–	–
costasArray-14	641,9	649,9	69,3	**3,6**
crossword-m1-words-05-06	–	204,6	482,1	**18,7**
crossword-m1c-words-vg7-7_ext	1173,9	1611,9	178,5	**13,3**
fapp07-0600-7	–	2295,7	–	–
knights-20-9	353,9	491,3	38347,4	153,4
knights-25-9	840,6	1645,2	–	214,9
knights-80-5	726,9	1395,6	8329,2	**49,3**
langford-3-17	**708,3**	3062,2	21252,3	713,5
langford-4-18	148,2	538,3	25721,2	**94,6**
langford-4-19	**747,2**	2735,3	–	782,5
latinSquare-dg-8_all	903,2	294,8	621,2	**23,6**
lemma-100-9-mod	117,6	145,3	5,8	**1,0**
ortholatin-5	558,9	362,4	335,8	**10,4**
pigeons-14	1492,6	2993,3	6128,9	**15,3**
quasigroup5-10	277,3	451,5	33,7	**1,7**
queenAttacking-6	1411,8	**706,4**	37446,1	1283,9
queensKnights-20-5-mul	–	5209,5	–	–
ruler-70-12-a3	2410,6	42,8	105,5	**3,7**
ruler-70-12-a4	4021,1	1331,3	185,2	**6,0**
scen11-f5	–	–	–	–
series-14	89,1	338,9	56,9	**1,1**
squares-9-9	1987,4	115,9	44,3	**1,3**
squaresUnsat-19-19	–	**3039,8**	–	–
M.A	2332,2	1385,0	8196,7	**178,5**

Instances	Portfolio			
	Choco2	CAG	OR-tools	CPHydra
		$w = 16$	$w = 16$	
cc-15-15-2	1102,6	**3,5**	1070,0	1,3
costasArray-14	6180,8	879,4	1368,8	0,6
crossword-m1-words-05-06	512,3	512,3	22678,1	–
crossword-m1c-words-vg7-7_ext	721,2	721,2	13157,2	0,4
fapp07-0600-7	37,9	**3,2**	–	–
knights-20-9	3553,9	**0,8**	–	–
knights-25-9	9324,8	**1,1**	–	–
knights-80-5	1451,5	301,6	32602,6	–
langford-3-17	8884,7	8884,7	–	0,6
langford-4-18	2126,0	2126,0	–	–
langford-4-19	12640,2	12640,2	–	–
latinSquare-dg-8_all	65,1	36,4	4599,8	–
lemma-100-9-mod	435,3	50,1	38,2	0,2
ortholatin-5	4881,2	4371,0	4438,7	0,1
pigeons-14	12336,9	5564,5	12279,6	0,1
quasigroup5-10	3545,8	364,3	546,0	–
queenAttacking-6	2644,5	2644,5	–	–
queensKnights-20-5-mul	235,3	**1,0**	–	–
ruler-70-12-a3	123,5	123,5	8763,1	–
ruler-70-12-a4	1250,2	1250,2	–	–
scen11-f5	45,3	**8,5**	–	–
series-14	1108,3	302,1	416,2	–
squares-9-9	1223,7	254,3	138,3	0,1
squaresUnsat-19-19	4621,1	4621,1	–	–
M.A	3293,8	1902,7	7853,6	0,4

TABLE 4.11 – EPS et Portfolio. Temps de résolution utilisant le centre de calcul *cicada* ($c = w$)

graines aléatoires permettent d'avoir différentes stratégies de recherche de type `random`.

Le tableau 4.11 permet de comparer les temps de résolution de notre approche et des méthodes de *portfolio*. En comparant les approches avec 16 *workers*, le *portfolio* d'`OR-tools` obtient les plus mauvais résultats en observant le temps moyen qui est à 7853,6 secondes. Notre approche sur `Gecode` est meilleure que les autres avec 16 *workers* et 512 *workers* en observant les temps de résolution ainsi que la résolution d'instances non résolues en séquentiel.

4.4 Embarrassingly Distributed Search

Dans cette section, nous étudions une variante de notre méthode, nommée *Embarrassingly Distributed Search* (EDS), en utilisant l'outil de planification *OAR* fourni par le centre de calcul *cicada*. Dans cette variante, nous modifions le solveur `Choco2` parallèle afin que les *workers* écrivent les sous-problèmes dans des fichiers au lieu de les résoudre. Ensuite, nous lançons un script qui soumet les différents travaux contenant les sous-problèmes à *OAR*, puis nous attendons leur terminaison et enfin nous rassemblons les résultats. *OAR* planifie les travaux dans le centre de calcul en utilisant une file de priorité basée sur des critères de fréquences des travaux soumis et du temps pris par les travaux. Chaque *worker* est créé par *OAR* qui lui attribue des ressources prédéfinies. Un *worker* peut être soit un solveur séquentiel soit parallèle. Cette approche offre un avantage pratique pour la réservation des ressources sur un centre de calcul. En effet, lorsque nous utilisons l'approche MPI, *OAR* va attendre qu'il y a suffisamment de ressources disponibles pour lancer les travaux. Ici, les ressources (cœurs ou nœuds du centre de calcul) sont mises à disposition dès qu'elles deviennent disponibles, ce qui peut réduire considérablement le temps d'attente. Cependant, il augmente clairement une surcharge dans le traitement des sous- problèmes, car à chaque travail lancé par *OAR*, le modèle est lu par le solveur et la propagation initiale s'effectue pour chaque sous-problème. Alors que dans l'approche MPI, la lecture du modèle et la propagation initiale s'effectuent seulement pour chaque *worker*. Il faut également comptabiliser le temps de soumission de l'ensemble des travaux à *OAR* qui est non négligeable.
Cette approche permet également de contourner d'une autre manière les limites de la technologie *threads* et de montrer la flexibilité d'EPS.

4.4.0.1 Variations par rapport à l'instance `crossword-m1c-words-vg7-7_ext`

Dans cette section, nous nous intéressons aux mauvais résultats obtenus avec l'instance `crossword-m1c-words-vg7-7` sur le centre de calcul *cicada* quelle que soit la stratégie de recherche (voir tableau 4.6). En effet, à l'exception de la stratégie de recherche `impact`, les algorithmes parallèles sont plus longs que les algorithmes séquentiels.

Ainsi, nous comparons les résultats de l'algorithme parallèle ($c = w = 16$) et celui du distribué (la profondeur de la décomposition des *workers* d_w est fixée à 0) avec des *workers* qui résolvent les sous-problèmes en séquentiel pour une décomposition avec différentes profondeurs. Le tableau 4.12 montre les temps de résolution et les facteurs de gain

		$d_w = 0$		$c = w = 16$	
d	p	t	su	t	su
2	186	73,0	10,2	1069,9	0,7
3	827	229,0	3,3	1074,2	0,7
4	2935	797,0	1,1	1091,8	0,7

TABLE 4.12 – Facteurs de gain d'EDS et d'EPS pour l'instance `crossword-m1c-words-vg7-7_ext` utilisant `Choco2` en variant les profondeurs de décomposition d pour le *master* et les profondeurs de décomposition des *workers* d_w.

obtenus. Le nombre de ressources distinctes utilisées par EDS est une mauvaise estimation du nombre de *workers* utilisés, car quelques *workers* sont utilisés seulement pour une courte période de temps. Par conséquent, le nombre de *workers* w d'EDS est estimé par le rapport entre le temps total de calcul de tous les *workers* et le temps d'utilisation du centre de calcul. Le facteur de gain est calculé à partir de cette estimation de w.
En premier lieu, l'algorithme parallèle est toujours plus lent que l'algorithme séquentiel, quel que soit le niveau de décomposition. Cependant, les facteurs de gain obtenus par les algorithmes distribués sont importants même s'ils diminuent rapidement. La chute des facteurs de gain montre qu'EDS n'est pas extensible avec les *workers* en séquentiel. En effet, la surcharge dans le traitement des sous-problèmes, et en particulier le temps de soumission des travaux deviennent importants lorsque le nombre de sous-problèmes augmente.
En second lieu, nous pouvons voir que les mauvaises performances de l'algorithme parallèle ne sont pas causées par une décomposition statistiquement déséquilibrée, car nous aurions des performances similaires pour l'algorithme distribué. L'investigation de l'algorithme parallèle sur ce cas particulier suggère que les mauvaises performances viennent du solveur sous-jacent lui-même (`Choco2`). En effet, les algorithmes parallèles ont passé plus de la moitié de leur temps dans certaines méthodes internes de contraintes de table. De plus, le nombre d'instructions reste le même pour les algorithmes parallèles et séquentiels, mais au niveau du cycle de vie des processus, le nombre de changements de contexte entre les processus, le nombre de mises à jour des caches et celui des défauts de cache ont considérablement augmenté. Nous constatons que ce phénomène se produit sur toute l'infrastructure et pour différentes machines virtuelles Java. Ainsi, ces expérimentations montrent clairement que MPI est mieux adapté que les *threads*, car la mémoire n'est pas partagée. On note que certaines instances utilisent des contraintes de table, mais le problème est moins visible.

4.4.0.2 Variations par rapport au problème de la règle de *Golomb*

Une règle de *Golomb* est un ensemble de marques à des positions entières (graduations), telles que les distances entre les paires de marques soient toutes différentes. Le nombre de marques sur la règle définit son ordre, et la plus grande distance entre une paire de ses marques est sa longueur. Ici, nous énumérons les règles optimales (minimum n pour la valeur spécifique de m) avec un modèle de simple contrainte inspiré de [Galinier 2003] et les heuristiques `lex` ou `dom` sont un choix raisonnable. Le tableau 4.13 donne le temps de résolution, les facteurs de gain de l'algorithme parallèle ($c = w = 16$), l'algorithme distribué avec les *workers* en séquentiel (la profondeur de la décomposition des *workers* d_w est fixée à 0), et les algorithmes distribués avec les *workers* parallèles ($w = 16$ et la profondeur de la décomposition des *workers* d_w est fixée à 2) à différentes profondeurs de décomposition.

		$d_w = 0$		$d_w = 2$		$w = 16$	
d	p	t	su	t	su	t	su
		`lex`					
1	20	–	–	572,0	89,7	11141,7	4,6
2	575	769,0	66,8	497,0	103,3	4084,2	12,6
3	14223	17880,0	2,9	–	–	3502,6	14,7
		`dom`					
1	20	–	–	1538,0	78,6	28299,9	4,3
2	222	2394,0	50,5	366,0	330,2	9703,6	12,4
3	5333	3018,0	40,0	–	–	8266,6	14,6

TABLE 4.13 – Facteurs de gain d'EDS et d'EPS pour la règle de *Golomb* d'ordre 14 en utilisant `Choco2` ($c = w$).

Au premier plan, les algorithmes parallèles ($w = 16$) obtiennent des facteurs de gain presque linéaires lorsque la profondeur de l'arbre de recherche qu'atteint la décomposition est assez grande. Nous rappelons que les facteurs de gains sont faibles lorsqu'il n'y a pas assez de sous-problèmes. Dans un second plan, les algorithmes distribués avec des *workers* qui exécutent en séquentiel ($d_w = 0$) sont efficaces dans le cas où le nombre de sous-problèmes reste faible. Dans le cas contraire, on peut encore obtenir des améliorations avec `dom`, mais nous gaspillons des ressources. En effet, soumettre plusieurs travaux avec la stratégie de recherche `lex` à *OAR* ajoute une surcharge de calcul non négligeable (environ 13 minutes) et dégrade globalement la performance du centre de calcul. Notez que la combinaison $d = 1$ et $d_w = 0$ n'a pas été testée, car elle ne peut pas donner de meilleures accélérations que l'algorithme parallèle. Enfin, les algorithmes distribués avec des *workers*

qui exécutent la résolution des sous-problèmes en parallèle offrent un bon compromis entre accélération et efficacité, car ils permettent d'utiliser beaucoup de ressources tout en soumettant seulement quelques travaux, réduisant ainsi le temps de soumission et le temps de surcharge de calcul. De même, la combinaison $d = 3$ et $d_w = 16$ n'a pas été testée, car elle serait encore moins efficace que $d = 3$ et $d_w = 0$.

La plupart des autres approches parallèles donnent de bonnes performances pour le problème de la règle de *Golomb*. Par exemple, les expérimentations dans [Michel 2009] et [Chu 2009] ont respectivement montré des facteurs de gain linéaires pour 4 et 8 *workers*. EDS est plus efficace que [Menouer 2014a] en utilisant 48 *workers* pour la règle de *Golomb* d'ordre 13. EDS est également efficace autant que [Fischetti 2014] en utilisant 64 *workers* pour la règle de *Golomb* d'ordre 14 tout en ayant des facteurs de gain très grands.

CHAPITRE 5

Élaboration d'une API pour la méthode EPS

Sommaire

5.1 Introduction

La méthode EPS permet de décomposer un problème PPC en un grand nombre de sous-problèmes qui sont ensuite résolus par les *workers*. La réalisation d'une interface de programmation applicative (API) permettant une mise en œuvre aisée de la méthode EPS sur différentes architectures est un point critique pour nos expérimentations. En effet, nous avons implémenté la méthode EPS sur différents supports, à savoir :

- sur une machine multi-cœurs (**fourmis**),
- sur un centre de calcul (**cicada**),
- sur le *cloud* de Microsoft Windows Azure (**azure**).

Ces différentes architectures imposent différents modes d'exécution des programmes en parallèle, différents modes de communication et utilisent différentes bibliothèques de fonctions. L'exécution d'un programme parallèle implique la création d'un ou plusieurs processus qui se chargent de son exécution. Chaque processus gère la mémoire qui lui est allouée de manière indépendante. Nous distinguons deux types de programmes qui sont caractérisés par leur contexte d'exécution : programme s'exécutant localement, c'est-à-dire au même endroit sur la même machine et le programme s'exécutant dans des endroits identiques ou différents. Dans le premier cas, nous avons le choix entre deux types de communication : la communication inter-processus et la communication par mémoire partagée. La communication inter-processus permet d'échanger différentes informations entre le *master* et les *workers* ; dans la méthode EPS, nous avons par exemple les sous-problèmes, les résultats, etc. Comme la mémoire est non partagée entre les processus, la communication s'effectue *via* des fichiers. Quant à la communication par mémoire partagée, nous utilisons les *threads* qui sont les fils d'exécution présents au sein d'un même processus. Dans le cas où le programme est exécuté sur des machines identiques ou différentes, nous utilisons le standard MPI ou la programmation réseau par TCP/IP. En MPI, le processus est dupliqué sur les différentes machines *via* des accès entre la machine depuis laquelle le programme est lancé et les autres machines. Il faut installer un agent MPI dans chaque machine qui est responsable de la création du processus au niveau local et de la communication avec les autres machines. Il faut noter que MPI fonctionne très bien lorsque les machines ont des caractéristiques identiques comme posséder la même puissance de calcul. Dans le cas contraire, il faut gérer les problèmes de synchronisation d'horloge dus à des vitesses de calcul et de réponse entre les agents différents. En programmation réseau par TCP/IP, chaque processus d'une machine peut communiquer avec un processus d'une autre machine *via* une abstraction de la communication de l'utilisation des interfaces réseau (carte réseau avec le câble Ethernet, carte réseau Wi-Fi) que l'on nomme les sockets. La figure 5.1 montre les différents cas d'utilisation d'une exécution d'un programme parallèle. Avec ces différents cas d'utilisation, écrire un programme pour chaque architecture n'est pas pensable tant au niveau de la maintenance de code qu'à l'ajout de nouvelles fonctionnalités. Par conséquent, il faut écrire un programme fonctionnant sur différentes architectures avec une partie commune qui concerne le déroulement de la méthode EPS en définissant un protocole de communication des différents acteurs (*master* et *workers*) et une partie spécifique qui gère le mode de communication et le démarrage des acteurs. Pour cela, il est nécessaire d'établir une interface de programmation commune pour les différentes implémentations sur les différents modes de communication. Dans ce contexte, nous proposons la création d'un protocole de communication et l'élaboration d'une API pour permettre aux différents *workers* d'échanger des informations en utilisant ce protocole de communication. Cette API doit être générique pour permettre l'implémentation sur différentes architectures. Nous utilisons le modèle de communication adapté à chaque type

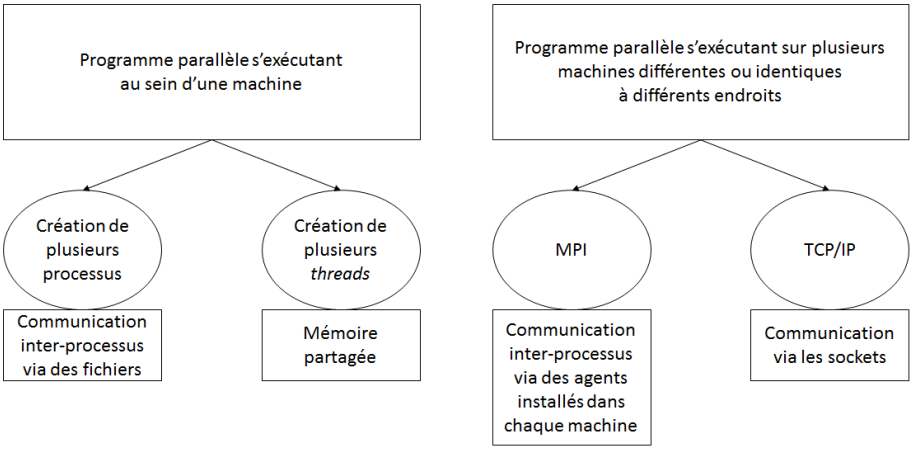

FIGURE 5.1 – Les différents types d'implémentation d'un programme parallèle.

d'architecture :

- modèle avec mémoire partagée : utilisé principalement dans les architectures multi-cœurs. Nous utilisons les *threads* (processus légers) pour l'implémentation.
- modèle par passage de message : utilisé dans les architectures multi-processeurs organisées sur plusieurs nœuds ou les architectures réseau. Nous utilisons MPI (Message Programming Interface) dans notre implémentation.

L'approche EPS suit le modèle client-serveur [Berson 1992] avec le *master* qui est le serveur de sous-problèmes et les *workers* qui sont les clients demandant des sous-problèmes à résoudre. Dans le modèle client-serveur, le serveur est fournisseur de services et le client est consommateur de services. Un serveur traite plusieurs clients en même temps et contrôle leurs accès aux ressources. C'est toujours le client qui déclenche la demande de service *via* une requête. Le serveur attend passivement les requêtes des clients puis une fois les requêtes traitées, il envoie une ou plusieurs réponses aux clients.

5.2 Architecture d'un système d'exploitation

Le système d'exploitation, décrit dans [Peterson 1985] est un ensemble de programmes qui contrôle l'ensemble des ressources d'un ordinateur afin de faire fonctionner des programmes utilisateurs que l'on nomme applications (traitement de textes, navigateur Internet, etc.). Le système d'exploitation est l'intermédiaire entre les applications et le matériel.

Un système d'exploitation multi-tâches permet de partager le temps du processus pour plusieurs tâches (également appelées processus), ainsi nous avons l'illusion que les programmes s'exécutent en même temps : il n'est pas nécessaire d'attendre que celui qui est en train de s'exécuter ait terminé pour commencer à en exécuter d'autres. Dans le cas où on n'a qu'une seule unité de calcul, celle-ci doit être donnée à tour de rôle aux diverses tâches qui s'exécutent. Le rôle d'un système d'exploitation multi-tâches est de mettre à

disposition les unités de calcul disponibles aux différentes applications ; ces applications sont dites concurrentes, car elles sont en concurrence pour l'accès aux unités de calcul.

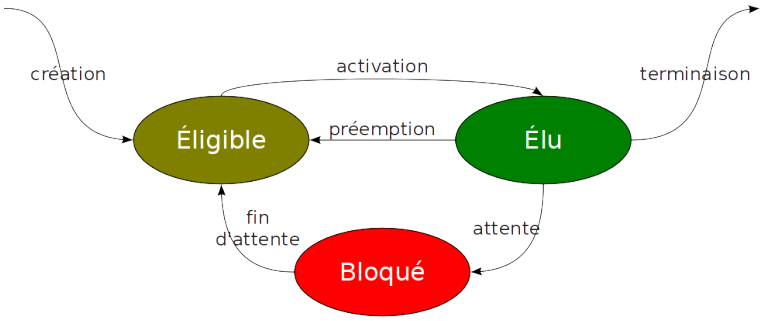

FIGURE 5.2 – Le cycle de vie d'un processus dans un système préemptif.

La partie du système d'exploitation qui effectue la distribution des processus sur une ou plusieurs unités de calcul est appelée ordonnanceur (appelé également planificateur). La présence d'un ordonnanceur permet au système d'exploitation d'être préemptif, c'est-à-dire qu'il répartit, selon des critères de priorité, le temps machine entre les différents processus qui en font la demande. Le système peut ainsi arrêter à tout moment un processus et démarrer l'exécution d'un autre en fonction du contexte comme l'interruption par une touche clavier déclenchée par l'utilisateur qui met en priorité le processus lié à l'application en premier plan. La figure 5.2 illustre les différentes phases de la vie d'un processus. Au départ, lors du lancement d'une application, le système d'exploitation crée un processus. Il lui alloue une zone mémoire qui lui est propre et il s'assure qu'aucun processus n'accède aux données propres à un autre processus (protection des données). Le système d'exploitation évite la propagation de cette erreur sur les autres processus (isolation) et ceux-ci peuvent donc continuer à s'exécuter sans problème. Puis, l'ordonnanceur l'ajoute dans sa file de priorité (état éligible). Lorsque c'est au tour du processus de s'exécuter, l'ordonnanceur lance son activation (état élu) et lui donne un temps imparti pour s'exécuter. Lorsque le temps imparti est écoulé, l'ordonnanceur l'arrête (état bloqué) et donne la main à un autre processus qui est en état d'éligibilité. Le processus en état bloqué retourne dans la file de priorité de l'ordonnanceur. Si un événement est déclenché comme une interruption clavier, l'ordonnanceur arrête le processus en cours d'exécution (on parle de préemption) en le mettant dans un état d'éligibilité et donne la main au processus qui gère l'interruption clavier, car cet événement possède la priorité la plus haute et doit être immédiatement traité.

5.2.1 Communication inter-processus sur une même machine

Couplé à la notion du multi-tâches du système d'exploitation et à l'utilisation de plusieurs unités de calcul, le système d'exploitation permet déjà de faire du parallélisme. Le point manquant est la communication entre les processus. Dans le cas d'une programmation concurrente disjointe, le lancement de plusieurs processus en parallèle est simplifié, car l'ordonnanceur du système d'exploitation sait gérer le cycle de vie des processus ainsi que

l'affectation des processus aux unités de calcul. On parle alors de gestion automatique des affinités (tel processus est traité par telle unité de calcul).

Dans le cas d'une programmation concurrente de type compétitive ou coopérative, les processus doivent communiquer pour s'échanger des informations. Ce type de communication inter-processus est appelé *Inter Process Communication* (IPC) [Rashid 1980]. Il existe plusieurs moyens de communication inter-processus. Nous pouvons citer, entre autres, les variables communes, les fichiers communs, les signaux, les messages et les tubes de communication. Les tubes de communication sont des fichiers spéciaux gérés par le système d'exploitation permettant de créer des communications inter-processus. La figure 5.3 illustre l'utilisation d'un tube de communication par deux processus.

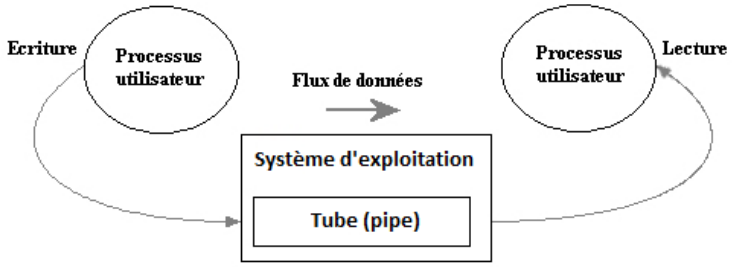

FIGURE 5.3 – Communication inter-processus par le biais d'un tube de communication (fichier spécial) sur une même machine.

5.2.2 Communication intra-processus

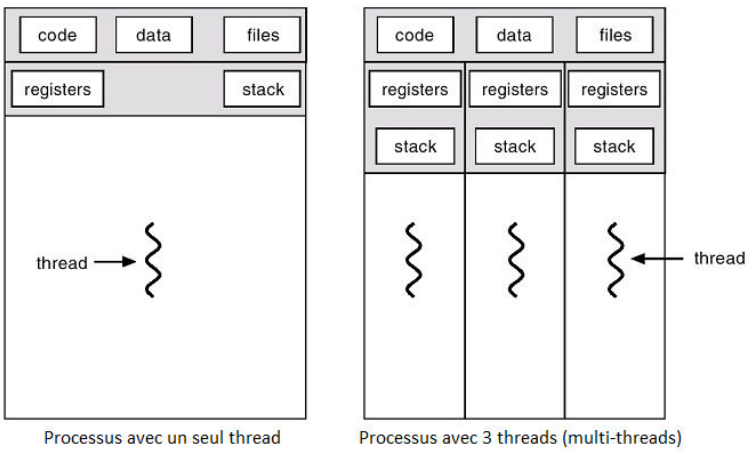

FIGURE 5.4 – Les différences entre un processus mono-*thread* et un processus multi-*threads*.

Les processus sont composés d'entités concurrentes plus élémentaires appelées fils d'exécution, *threads* en anglais [Mueller 1993, Kleiman 1996]. Les *threads* appartenant au même processus se partagent son contexte mémoire, mais chacun possède sa propre

pile d'exécution. La figure 5.4 illustre ce partage de mémoire en montrant également la différence entre un processus avec un seul *thread* (mono-*thread*) et un processus avec plusieurs *threads* (multi-*threads*). On dit que ce dernier est *multi-threads*. Les *threads* peuvent être manipulés par des bibliothèques comme Pthreads (pour Posix Threads) [Mueller 1993, Kleiman 1996], largement utilisées dans les systèmes d'exploitation Linux et Solaris, ou encore Threads Win32 [Pat 1999], utilisées sur des plate-formes Windows. Ces bibliothèques définissent les opérations de manipulation des *threads* comme la création ou la destruction d'un *thread*, la synchronisation (verrou, sémaphore, barrière de synchronisation, exclusion mutuelle, etc.) entre *threads*.

5.3 Architecture à mémoire distribuée

Lors du passage aux architectures multi-nœuds où chaque nœud représente une machine, la communication inter-processus se limite seulement à l'utilisation du mécanisme de passage de messages, car la mémoire entre machines n'est pas partagée. Il faut donc utiliser des technologies supportant la mémoire distribuée. Deux technologies permettent de faire du passage de messages :

- la bibliothèque MPI ;
- la communication réseau avec la programmation par socket.

5.3.1 Bibliothèque MPI

MPI [Lester 1993, Gropp 1993] est une bibliothèque de fonctions permettant la programmation d'applications parallèles sur des machines à mémoire distribuée. Un programme MPI s'exécute de la manière suivante : une copie du programme s'exécute, une fonction MPI renvoie le numéro du processus ce qui permet d'exécuter un code spécialisé sur chaque processus selon la valeur de son numéro. À l'initialisation, un nombre fixé de processus est créé et généralement chaque processus exécute un même code unique. Ce modèle d'exécution, appelé SPMD (pour *Single Program Multiple Data*) [Atallah 2002] permet l'exécution du même programme sur plusieurs unités de calcul. Les processus communiquent entre eux par échange de messages. La communication entre deux processus peut être :

- de type point à point (envoi et réception d'une donnée d'un émetteur vers un destinataire) ;
- de type collectif (diffusion d'un message à un groupe de processus, opération de réduction, distribution ou redistribution des données envoyées).

Dans un programme MPI, un commutateur appelé *Probe* permet de connaître l'ensemble des processus actifs. MPI définit des fonctions qui à tout moment permettent de connaître le nombre de processus gérés dans un commutateur et le rang d'un processus dans le commutateur. La bibliothèque met à la disposition de nombreuses fonctions permettant au programmeur d'effectuer différents types d'envoi et de réception de messages (point à point

bloquant (synchrone), point à point non bloquant (asynchrone) et des opérations globales (barrière, réduction, diffusion)).

Il y a plusieurs implémentations du standard MPI comme OpenMPI [Gabriel 2004], Intel MPI [IntelMPI 2015], MPI-CH [MPICH 2015] et celui de Microsoft nommé MS-MPI [Krishna 2010, Lantz 2008]. MPI est en effet un standard et non un système. Ainsi, les caractéristiques d'une machine et les différentes optimisations possibles ne peuvent pas être prises en compte dans le standard MPI. En général, les fournisseurs de machines comme Bull, IBM et Intel donnent leur propre implémentation de MPI selon les spécifications propres des machines.

On peut noter que l'implémentation en MPI est vraiment différente de l'implémentation avec les *threads*, car toute interaction entre les différents objets doit passer par des messages. Avec les *threads*, l'existence d'une section critique est un passage obligé, car la mémoire est partagée par les *workers*. Les *threads* doivent s'exclure mutuellement pour récupérer un sous-problème. En MPI, comme chaque processus possède une mémoire indépendante, la section critique n'existe pas. Le mécanisme d'auto-exclusion se fait automatiquement, car MPI permet le traitement synchrone des messages.

5.3.2 Communication réseau

Un réseau de communication se définit par un ensemble des ressources matérielles et logicielles liées à la transmission et l'échange d'information entre différentes entités. Les entités peuvent être des processus, des *threads* ou des machines. Suivant leur organisation, ou architecture, les distances, les vitesses de transmission et la nature des informations transmises, les réseaux de communication font l'objet d'un certain nombre de spécifications et de normes. D'un point de vue typologique, l'architecture réseau est respectivement qualifiée de LAN (*Local Area Network*), MAN (*Metropolitan Area Network*) et WAN (*Wide Area Network*) lorsque le réseau s'étend sur un périmètre local (< 1 km), métropolitain (< 100 km) et longue distance.

Dans notre étude, nous nous limitons à l'architecture TCP/IP [Postel 1981] qui est largement utilisée sur Internet. TCP/IP est un acronyme désignant 2 protocoles étroitement liés : un protocole de transport, TCP (Transmission Control Protocol) et un protocole IP (Internet Protocol). Le protocole IP permet d'identifier une machine dans un réseau et le protocole TCP fournit un service sécurisé de remise des paquets en mode connecté, c'est-à-dire qu'il garantisse l'ordre et la remise des paquets de données et il vérifie leur intégrité. Le protocole TCP/IP est adapté aux applications client-serveur et aux services critiques tels que le transfert de fichiers.

Il y a également le protocole de transport UDP qui est un complément du protocole TCP. Il offre un service permettant d'envoyer des paquets de données (appelés datagrammes) sans connexion qui ne garantit ni la remise ni l'ordre de ces paquets délivrés. Les sommes de contrôle des données sont facultatives dans le protocole UDP. Ceci permet d'échanger des données sur des réseaux à fiabilité élevée sans utiliser inutilement des ressources réseau ou du temps de traitement. Les messages (ou paquets UDP) sont transmis de manière autonome sans garantie de livraison. Le protocole UDP prend également en charge l'envoi de données d'un unique expéditeur vers plusieurs destinataires. Cependant,

l'approche EPS a besoin d'un contrôle d'intégrité des messages envoyés : seul le protocole des transports TCP est donc utilisé.

Afin d'utiliser aisément le protocole TCP/IP et communiquer facilement entre les machines, on utilise les sockets. Introduites dans les distributions de Berkeley [Miller 1986], les sockets représentent la couche logicielle qui permet d'utiliser les services d'un protocole réseau et masquer la gestion du réseau prise en charge par le système d'exploitation. Ainsi, les sockets peuvent aisément recevoir et expédier des données entre les machines. Précisément, une socket est simplement un moyen de désigner l'extrémité d'une connexion, côté émetteur ou récepteur, en l'associant à un port. Une fois la connexion bidirectionnelle établie *via* des sockets entre un processus client et un processus serveur, ceux-ci peuvent communiquer en utilisant les mêmes primitives `read` (pour la lecture) et `write` (pour l'écriture) pour l'accès aux fichiers.

5.4 Interface de programmation

Nous décrivons l'interface de programmation commune pour les différentes implémentations de la méthode EPS. Nous utilisons la programmation orientée-objet pour décrire notre interface de programmation, car elle permet de décrire simplement les objets et leurs interactions. Pour cela, nous l'illustrons avec un diagramme de classes qui permet de décrire les objets utilisés et un diagramme de séquence qui détaille le cas d'utilisation. La communication est effectuée entre deux types d'entités qui sont le *master* et le *worker*. Nous considérons que le *worker* est responsable des déclenchements des événements, et le *master* envoie les actions que le *worker* doit exécuter. En partant de ce constat, nous décrirons les différentes implémentations (*threads*, MPI et utilisation des sockets) et leurs particularités dans le traitement de la méthode EPS.

5.4.1 Diagramme de classes

Nous détaillons le diagramme de classes de l'API. Il décrit l'ensemble des interfaces et classes qui seront utilisées dans le code. Un diagramme de classe représente le cœur de la conception d'un système et décrit le type des objets ou données du système ainsi que les différentes formes de relation statique qui les relient entre eux. La figure 5.5 donne respectivement les diagrammes de classes du *master* et du *worker* ainsi que les méthodes utilisées. Nous donnons une brève description de chaque classe et de son rôle :

FIGURE 5.5 – Diagramme des classes de l'API.

- MASTER : cette classe représente le *master*.
- MASTERTASKSOLVERI : cette classe générique permet d'effectuer le travail du *master*. Par exemple, en spécialisant cette classe pour EPS (MASTERTASKSOLVEREPS), le *master* pourra initialiser la décomposition d'un problème et contrôler la liste des sous-problèmes.
- MASTERTRANSACTIONI : cette interface permet l'implémentation de la communication entre le *master* et les *workers*. Plusieurs implémentations sont possibles : MASTERTRANSACTIONMPI pour MPI, MASTERTRANSACTIONTHREAD pour les *threads* et MASTERTRANSACTIONSOCKETTCP pour les sockets.
- WORKER : cette classe représente un *worker*.
- WORKERTASKSOLVERI : cette classe générique permet au *worker* d'exécuter une tâche. Par exemple, en spécialisant cette classe pour EPS (WORKERTASKSOLVEREPS), le *worker* pourra résoudre un sous-problème.
- WORKERTRANSACTIONI : cette interface permet l'implémentation de la communication entre le *worker* et son *master*. Les différentes implémentations sont les suivantes : WORKERTRANSACTIONMPI pour MPI, WORKERTRANSACTIONTHREAD pour les *threads* et WORKERTRANSACTIONSOCKETTCP pour les sockets.

5.4.2 Diagramme de séquences

Pour comprendre pas à pas les différentes interactions entre les différents objets utilisés, on utilise les diagrammes de séquences. Les diagrammes de séquences mettent en valeur les échanges de messages (déclenchant des événements) entre objets de manière chronologique, l'évolution du temps se lisant de haut en bas.

Nous détaillons le diagramme de séquences 5.6 de la méthode EPS. Il faut noter que les interactions entre le *master* et les *workers* sont représentées par des méthodes de requête de haut niveau qui utilisent les différentes méthodes de communication des classes TRANSACTIONMASTERI et de TRANSACTIONWORKERI. Le traitement des données est géré par les classes dérivant de MASTERTASKSOLVERI (côté *master*) et WORKERTASKSOLVERI (côté *worker*). Soit P une instance du problème à résoudre. P peut être chargée à partir d'un fichier (écrit en XCSP ou en FlatZinc) ou écrit à partir de l'API du solveur. Nous commençons par l'initialisation qui correspond à l'allocation des ressources ainsi que la décomposition du problème. Tout d'abord, on crée le *master* qui charge le problème P. Puis le *master* crée les *workers* puis donne une copie du problème P à chaque *worker*. Chaque *worker* charge le problème P fourni par le *master*. Ensuite, le *master* décompose le problème P qui va générer la liste des sous-problèmes.

Nous décrivons maintenant les interactions entre le *master* et le *worker*. Lorsqu'un *worker* demande un sous-problème, le *master* modifie la liste des sous-problèmes en prenant un sous-problème et fournit celui-ci au *worker*. Dans le cas d'un problème d'optimisation, le *master* fournit en plus du sous-problème la meilleure borne courante de la valeur de l'objectif.

FIGURE 5.6 – Diagramme de séquence montrant les interactions entre le *master* et les *workers*.

À la réception des résultats, le *master* enregistre les résultats. Dans le cas d'un problème d'optimisation, le *master* met à jour la borne supérieure de l'objectif afin de la transmettre au prochain sous-problème envoyé à un *worker*.

Enfin, lors de la fin de la résolution, on procède à la désallocation des ressources. Le *master* envoie un message de terminaison (message DESTROY) chaque fois qu'un *worker* demande un sous-problème (message ASK). Lorsque tous les *workers* ont terminé, le *master* affiche les résultats de la résolution et se termine.

5.5 Implémentation *threads*

Dans l'implémentation *threads*, la mémoire est partagée, c'est-à-dire que la liste des sous-problèmes créée par la décomposition est partagée entre le *master* et les *workers*. Comme on a des accès concurrents à la liste des sous-problèmes, on utilise les routines de syn-

chronisation. Pour récupérer un sous-problème (message GIVE), un *worker* doit demander un accès exclusif à la liste des sous-problèmes. Cet accès exclusif permettra de prendre un sous-problème dans la liste des sous-problèmes puis de l'enlever de la liste des sous-problèmes (accès en lecture et en écriture). Cette opération n'est pas atomique, il faut donc mettre en place un mécanisme de verrou comme l'utilisation du mutex pour "protéger"' cet accès. La "zone protégée" par le mutex est nommée section critique. Le code suivant montre pas à pas l'utilisation du mutex qui permet au *worker* de récupérer un sous-problème.

```
1  //Demande de verrou
2  //Opération bloquante tant qu'on a pas obtenu le verrou
3  mutex->acquire();
4
5  //Section Critique
6
7  //Prendre un sous-problème dans la liste des sous-problèmes Q
8  sous-probleme = Q->pop()
9
10 //Libération du verrou
11 mutex->release()
```

5.6 Implémentation par passage de messages

5.6.1 Routines de communication

Les échanges de messages utilisent les routines de communication qui sont au nombre de deux :

- SEND(MESSAGE, DESTINATAIRE) permet d'envoyer un message à un destinataire.
- MESSAGE = RECV(EXPÉDITEUR) permet de recevoir un message d'un expéditeur.

Nous considérons que ces routines de communication sont bloquantes. Les routines de communication utilisent une mémoire tampon afin d'allouer la taille correspondante du message et de le copier. On peut noter que les routines de communication (envoyer et recevoir) pour l'échange des sous-problèmes entre le *master* et les *workers* n'existent pas dans l'implémentation *threads*, car la mémoire est partagée. Ainsi, les routines de communication conviennent seulement aux implémentations au passage de messages (MPI et Socket).

En MPI, ces routines de communication sont représentées par MPI_SEND pour l'envoi et MPI_RECV pour la réception des données. En programmation réseau à l'aide des sockets, elles sont représentées par les primitives SEND et RECV.

5.6.2 Implémentation MPI

Le code MPI est un code exécuté par plusieurs processus ressemblant à l'utilisation de la fonction FORK dans les systèmes Unix qui permet de faire la duplication de processus.

Pour que chaque processus puisse exécuter un code différent, on différencie le code par le test du rang du processus exécuté :

- rang 0 : *master*
- rang 1 : *worker* 1
- rang 2 : *worker* 2
- rang N : *worker* N

```
// Exemple de code
if MPI_Comm_rank() == 0
  //code Master
else
  //code Worker
endif
```

En MPI, l'émetteur et le récepteur sont identifiés par leur rang dans le communicateur. L'enveloppe d'un message est constituée :

- du rang (identifiant) de l'émetteur ;
- du rang du récepteur ;
- de l'étiquette (*tag* en anglais) du message ;
- du nom du communicateur qui définit le contexte de communication.

Les messages envoyés sont typés (INT, DOUBLE, CHAR, etc.). Pour encoder les sous-problèmes et les résultats, on envoie des messages en chaînes de caractères (CHAR[]). Cependant, il faut spécifier la taille du message au destinataire afin que ce dernier puisse allouer une mémoire tampon (*buffer* en anglais) ayant la même taille du message. Le code ci-dessous présente l'envoi d'un message et la réception d'un message. Pour récupérer la taille du message, il faut interroger le commutateur MPI (*Probe*), qui est responsable de l'état des messages réceptionnés avec la fonction MPI_PROBE. Cette fonction permet de récupérer le statut (MPI_STATUT) contenant l'identifiant de l'émetteur. Puis, avec MPI_GET_COUNT, on récupère la taille du message afin qu'on puisse allouer le *buffer*. Enfin, on appelle MPI_RECV qui va écrire le message dans le *buffer* avec le type correspondant au message envoyé par l'émetteur avec la fonction MPI_SEND. Les codes ci-dessous présentent l'utilisation des fonctions pour envoyer (code émetteur) et recevoir (code receveur) un message en MPI.

```
// Code émetteur
Buffer message("Bonjour");

// Envoi du message
MPI_Send(&message[0], message->size(), MPI_CHAR, ...);
```

```
// Code receveur
size_message;
Buffer message;
MPI_Status status;

// Interroger le Probe
// de l'état d'un message en attente de réception
MPI_Probe(id_emetteur, ..., status);
MPI_Get_count(status_, MPI_CHAR, size_message);

// Allouer la bonne taille pour le message qu'on va recevoir
message->resize(size_message);

// Réception du message
MPI_Recv(message, size_message, MPI_CHAR, ...);
```

5.6.3 Implémentation à l'aide de socket

L'initialisation de la communication à l'aide de socket entre le *master* et le *worker* est complexe. Tout d'abord, il y a une distinction entre l'initialisation d'une socket pour le *master* qui est une socket serveur et une socket pour le *worker* qui est une socket client. Le schéma 5.7 illustre les interactions entre un serveur et un client avec l'aide des sockets.

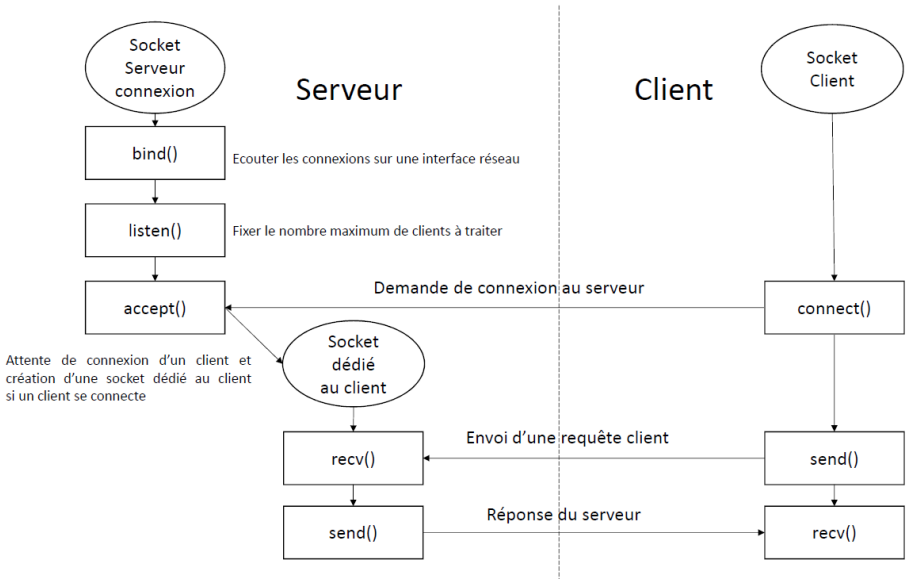

FIGURE 5.7 – Schéma du modèle client-serveur en utilisant les sockets.

Plus précisément, du côté serveur, nous avons deux types de socket : la socket connexion, qui permet d'écouter les demandes de connexions et la socket qui est dédiée à la communication avec un seul client. Tout d'abord, la socket connexion doit être liée à une interface réseau (par exemple une carte réseau Wi-Fi) en se déclarant sur une adresse IP et un port (numéro) afin qu'il puisse écouter les demandes de connexion des clients. On

utilise pour cela la fonction *bind* (pour spécifier l'interface réseau, l'adresse IP et le port) puis *listen* (pour le nombre maximum de clients à traiter). Une fois le serveur en écoute, il est prêt à accepter les demandes de connexion avec la fonction *accept* qui est bloquante. Après acceptation d'une demande, la fonction *accept* crée une socket. On peut noter qu'un serveur possède une seule socket connexion et autant de sockets que de clients connectés. La socket du côté du serveur communique avec la socket du côté client en lisant les requêtes du client avec la primitive *recv* et envoie les réponses avec la primitive *send*.

Du côté client, nous n'avons besoin que d'une seule socket. La socket se connecte à l'adresse IP et au port que le serveur écoute. Puis, une fois la connexion faite, il n'y a plus qu'à envoyer des requêtes avec la primitive *send* et recevoir les réponses avec la primitive *recv*.

5.7 Implémentation d'EPS avec MPI

5.7.1 Initialisation

Nous expliquons l'utilisation de la technologie MPI avec la méthode EPS. Nous rappelons que P est une instance d'un problème. Par exemple, avec l'API de `Gecode`, on utilise l'objet SPACE pour créer une instance d'un problème en programmation par contraintes. Soit k, le nombre de sous-problèmes que nous souhaitons générer. MPI suit le modèle *SPMD* (pour *Single Program Multiple Data*), nous avons donc un seul code pour tous les processus. Nous initialisons l'environnement MPI afin de créer les processus. Si le rang est égal à 0, nous initialisons le *master*. Dans le cas contraire, nous initialisons un *worker* en lui spécifiant le rang du *master* afin de communiquer avec lui. Au niveau du *master*, nous créons l'objet MASTERTRANSACTIONMPI, responsable de la communication avec les *workers* et l'objet MASTERTASKSOLVEREPS, qui décrit les actions du *master*. Ces objets sont référencés dans l'objet *master*. Enfin, nous lançons la méthode START_COMMUNICATION afin de démarrer la communication. De manière analogue, nous appliquons le même procédé pour le *worker* avec l'objet de communication WORKERTRANSACTIONMPI et l'objet WORKERTASKSOLVEREPS. Nous considérons que les actions des *master* et *worker* sont assumées par MASTERTASKSOLVEREPS et WORKERTASKSOLVEREPS.

Le code suivant montre pas à pas cette initialisation.

```
begin
  // Initialiser l'environnement MPI
  MPI_init()
  // Rang du processus
  r = MPI_Comm_rank()
  // Nombre de workers
  w = MPI_Comm_size() - 1

  // Fixer l'idMaster à zéro
  idMaster = 0
  if r == idMaster
    // Processus master
    MasterTaskEPS taskM(P, k, w)
    MasterTransactionMPI transM()
    Master master(taskM, transM, idMaster)
    master->start_communication()
  else
    // Processus worker
    WorkerTask taskW(P, idMaster)
    WorkerTransactionMPI transW()
    Worker worker(taskW, transW, r)
    worker->start_communication()
  endif

  // Fermer l'environnement MPI
  MPI_Finalize()
end
```

Si le processus exécuté est celui du *master*, on lance la méthode START_COMMUNICATION pour initialiser l'objet MASTERTASKSOLVEREPS avec la méthode INIT puis tant que la tâche du *master* n'est pas terminée (dans notre cas tant que la liste de sous-problèmes Q n'est pas vide) alors on fait les actions suivantes : on attend la réception d'un message d'un *worker*. Une fois le message reçu, on le traite dans la méthode RUN du MASTERTASKSOLVEREPS puis on envoie une réponse au *worker*.

```
Master::start_communication()
begin
  task_->init()
  while not task_->done()
  do
    trans_->receive_from_worker()
    task_->run()
    trans_->send_response_to_worker()
  done
end
```

5.7.2 Gestion des sous-problèmes avec la décomposition séquentielle

Dans cette section, nous étudions l'implémentation de la décomposition séquentielle vue dans le chapitre 3. Nous avons d'un côté un traitement effectué du côté du *master* qui va

prendre en charge la décomposition afin de générer les sous-problèmes et d'un autre côté nous avons le traitement des messages envoyés par les *workers* afin de leur envoyer les sous-problèmes et gérer la fin de la résolution.

5.7.2.1 Côté *master*

L'initialisation du MASTERTASKSOLVEREPS permet d'appeler la méthode de décomposition séquentielle vue dans le chapitre 3 afin de créer notre liste de sous-problèmes Q.

```
1  MasterTaskSolverEPS::init()
2  begin
3    Q = SequentialDecomposition(P, k)
4  end
5
6  MasterTaskSolverEPS::SequentialDecomposition(P, k)
7  begin
8    d = 0
9    Q = createEmptyList()
10   while Q->size() >= k
11   do
12     d = computeDepth(P, Q->size(), d)
13     Q = solveDBDFS(p, d)
14     if Q->isEmpty()
15       break
16     endif
17   done
18   return Q
19 end
```

Dans la méthode RECEIVE_FROM_WORKER, on utilise le code vu précédemment dans la section *Implémentation MPI* pour recevoir un message de n'importe quel *worker* (MPI_ANYSOURCE). Nous utilisons un *buffer* interne du *master* pour faire une copie du message. Nous avons plusieurs types de messages qui impliquent différentes actions que doivent faire le *master* et un *worker*. Nous considérons que les types de messages sont mis dans l'en-tête des messages et on utilise la méthode STARTWITH d'un message pour connaitre son type.

```
1  MasterTransactionMPI::receive_from_worker()
2  begin
3    // Interroger le Probe pour sélectionner un message prêt à être reçu
4    MPI_Probe(MPI_ANY_SOURCE, ..., status_)
5    size_message = 0
6    MPI_Get_count(status_, MPI_CHAR, size_message)
7
8    // Changer la taille du buffer interne du master
9    // afin de correspondre à la taille du message
10   master_->buffer()->resize(size_message)
11   MPI_Recv(master_->buffer(), size_message, MPI_CHAR, ...)
12 end
```

Ensuite, au niveau du *master* on appelle la méthode RUN de l'objet MASTERTASKSOLVEREPS. Dans cette méthode, on décode le message reçu puis on le traite. Plusieurs

types de messages existent :

- message ASK : le *worker* demande un sous-problème à résoudre ;
- message RESULTSSP : le *worker* a résolu un sous-problème et envoie les résultats de la résolution de ce sous-problème, puis demande à nouveau un sous-problème à résoudre. Du côté du *master*, on appelle la méthode ADDRESULTS pour enregistrer les résultats.

À chaque message reçu, on enlève un sous-problème de la liste des sous-problèmes, on l'encode dans un message puis on l'envoie au *worker*. Lorsque la liste de sous-problèmes Q est vide, le *master* envoie un message au *worker* pour qu'il s'arrête et on décrémente le nombre de *workers* disponibles ($w = w - 1$).

Les messages d'envoi sont donc de deux types :

- message SSP : le *worker* décode ce message qui contient le problème à résoudre
- message QUIT : le *worker* quitte la communication avec le *master* (à la fin de la résolution du problème par exemple)

Le code ci-dessus décrit la méthode RUN du MASTERTASK qui appelle la méthode RESOLUTION. Nous mettons de côté pour le moment la résolution et nous la verrons plus tard, car elle nous sera utile lorsque nous aborderons la décomposition parallèle.

```
MasterTask::run()
begin
  resolution()
end
```

```
MasterTask::resolution()
begin
  if w == 0
    return
  endif

  messageReceived = master()->buffer()
  if messageReceived->startWith("RESULTSSP")
    result = decode(messageReceived)
    master()->addResults(idWorker, result)
  endif

  messageToSend = ""
  if Q->isNotEmpty()
    ssp = Q->pop()
    messageToSend = encode("SSP", ssp)
  else
    w = w - 1
    messageToSend = "QUIT"
  endif
  master()->buffer() = messageToSend
end
```

Enfin, lorsque le message d'envoi est initialisé en l'affectant au *buffer* interne du *master*, on l'envoie au *worker* dont son identifiant a été récupéré lors de la réception du message avec l'objet MPI_STATUS par la méthode SEND_RESPONSE_TO_WORKER de l'objet MASTERTRANSACTIONMPI.

```
MasterTransactionMPI::send_response_to_worker()
begin
  message = master()->buffer()
  MPI_Send(message, message->size(), MPI_CHAR, status_->MPI_SOURCE, ...)
end
```

Lorsque la liste de sous-problèmes Q est vide et que tous les *workers* ont reçu le message QUIT, le *master* peut arrêter la communication avec la méthode STOP_COMMUNICATION et le processus se termine en arrêtant l'environnement MPI avec la fonction MPI_FINALIZE.

5.7.2.2 Côté *worker*

Du côté du *worker*, on lance la méthode START_COMMUNICATION pour envoyer une demande au *master* d'un nouveau sous-problème à résoudre en passant par l'instance de la classe WORKERTRANSACTIONMPI avec la méthode SEND_AND_RECEIVE_TO_MASTER. Cette méthode prend en paramètre le message à envoyer et attend la réponse du *master*. Une fois le message reçu, on l'affecte au *buffer* interne du *worker*. Tant que la tâche du *worker* n'est pas terminée par le biais de la méthode DONE, *a.k.a* tant que le *worker* ne reçoit pas de message QUIT, on traite le message reçu dans la méthode RUN du WORKERTASKSOLVEREPS puis on envoie une réponse au *master*.

```
Worker::start_communication() {
  trans_->send_and_receive_to_master("ASK")
  while task_->done()
  do
    task_->run()
    trans_->send_and_receive_to_master(this->buffer())
  done
}
```

```
WorkerTransactionMPI::send_and_receive_to_master(message)
begin
  // Envoi du message au master
  idMaster = worker_->master()->id()
  MPI_Send(message, message->size(), MPI_CHAR, idMaster, ...)

  // Interroger le Probe pour sélectionner un message prêt à être reçu
  MPI_Probe(MPI_ANY_SOURCE, ..., status_)
  size_message = 0
  MPI_Get_count(status_, MPI_CHAR, size_message)

  // Changer la taille du buffer interne du worker
  worker_->buffer()->resize(size_message)
  MPI_Recv(worker_->buffer(), size_message, MPI_CHAR, ...)
end
```

Lors du traitement du message, nous vérifions le type de message en regardant les premiers caractères du message. Si le message est de type SSP, alors nous décodons le message pour recréer le sous-problème correspondant puis on le résout avec SOLVE grâce au solveur. Une fois la résolution terminée, on met les résultats dans un *buffer* interne du *worker* afin de pouvoir l'envoyer avec la méthode SEND_AND_RECEIVE_TO_MASTER en lui mettant en paramètre le *buffer*. Dans le cas où le *master* envoie au *worker* un message de type QUIT, la tâche du worker signale par le biais du drapeau _DONE que le *worker* a terminé.

```
1  WorkerTask::run()
2  begin
3    if _done == true
4      return
5
6    messageReceived = worker()->buffer()
7    if messageReceived.startWith("SSP")
8      ssp = decode(messageReceived)
9      results = solve(ssp)
10     messageToSend = encode("RESULTSSP", results)
11     worker()->buffer() = messageToSend
12   else if messageReceived->startWith("QUIT")
13     _done = true
14   endif
15 end
```

5.7.3 Gestion des sous-problèmes avec la décomposition parallèle

Dans cette section, nous décrivons l'implémentation de la décomposition parallèle vue dans le chapitre 3. La gestion des sous-problèmes avec la décomposition parallèle change légèrement les tâches du *master* et des *workers*, car elle fait intervenir les *workers* dans le processus de décomposition. Pour cela, nous ajoutons trois types de messages SSPD, RESULTSSPD et NOP. Le premier est ajouté à la liste des messages envoyés par le *master* au *worker* qui est de type SSPD. Ce message contient un sous-problème avec une profondeur donnée. Le *worker* décompose ce sous-problème et génère les sous-problèmes qui seront ensuite envoyés au *master*. Ces sous-problèmes seront encodés dans un message avec le type RESULTSSPD. En se référant à la description de la décomposition parallèle, il existe des périodes de synchronisation qui se traduisent par des attentes de *workers*, car il n'y a plus de sous-problèmes à décomposer. Par conséquent, durant ces périodes de synchronisation, le *master* doit envoyer des messages de type NOP pour signaler aux *workers* de le solliciter à nouveau plus tard avec le message de type ASK. Le *master* n'est donc pas bloqué et il pourra traiter les sous-problèmes générés par les *workers* encore actifs. Dans la

décomposition parallèle, nous avons le tableau $tabK$ qui contient les différents nombres de sous-problèmes à atteindre pour chaque étape de la décomposition. La première étape est une décomposition statique simple pour atteindre w sous-problèmes dont on ne vérifie pas leur consistance par propagation. Cette décomposition initialise la liste de sous-problèmes Q et la profondeur d que les *workers* vont devoir atteindre afin de générer les nouveaux sous-problèmes. Du côté de la tâche du *master*, on modifie la méthode INIT pour lancer la première étape de la décomposition parallèle. On initialise la variable $currentIndexTabK$ qui nous permettra d'avancer dans les étapes de la décomposition. On utilise également le drapeau $decompositionDone$ qui nous permettra de démarrer la résolution lorsque la décomposition est terminée. Nous notons que la méthode de résolution du *master* n'a pas à être modifiée, ce qui donne moins de changements à faire au niveau du code.

```
1 MasterTaskSolverEPS::init()
2 begin
3   currentIndexTabK = 0
4   (Q, d) = splitDomains(P, tabK[currentIndexTabK])
5   currentIndexTabK = currentIndexTabK + 1
6   decompositionDone = false
7 end
```

```
1 MasterTask::run()
2 begin
3   if not decompositionDone
4     parallelDecomposition()
5   else
6     resolution()
7   endif
8 end
```

En premier lieu, chaque étape de la décomposition est jalonnée par une synchronisation afin de rassembler tous les sous-problèmes générés par les *workers*. La fin d'une synchronisation est caractérisée lorsque tous les *workers* ont terminé la décomposition de tous les sous-problèmes contenus dans Q ; nous pouvons le savoir lorsque le nombre de *workers* disponibles est nul ($w = 0$). Pour récupérer les sous-problèmes générés par les *workers*, nous utilisons une autre liste de sous-problèmes $Q1$ qui nous permettra de les stocker. Ces sous-problèmes seront ensuite remis dans Q pour la prochaine étape de la décomposition. La décomposition se termine lorsqu'on atteint le nombre de sous-problèmes souhaité k, ou qu'on a atteint la dernière étape de la décomposition ou bien la liste des sous-problèmes est vide. Nous pouvons donc faire appel à la méthode RESOLUTION pour traiter le premier *worker* disponible pour la résolution.

```
MasterTask::parallelDecomposition()
begin
  // Fin de synchronisation
  if w == 0
    // Vider Q1 pour remettre les sous-problèmes dans Q
    while not Q1->isEmpty()
    do
        Q->add(Q1->pop())
    done
    // Remettre le nombre de workers afin de les rendre tous disponibles
    w = MPI_Comm_size() - 1

    // Scruter si la décomposition est terminée
    if Q->isEmpty() or Q->size() >= k or currentIndexTabK >= tabK->size()
      decompositionDone = true
      // Appel de la methode resolution
      // pour traiter le premier worker disponible pour la résolution
      resolution()
      return
    endif
    // Nouveau nombre de sous-problèmes à atteindre
    currentIndexTabK = currentIndexTabK + 1
    newK = tabK[currentIndexTabK]
    d = computeDepth(P, Q->size(), d, newK)
  endif

  messageReceived = master()->buffer()
  if messageReceived->startWith("RESULTSSPD")
    newQ = decode(messageReceived)
    Q1->addAll(newQ)
  endif

  messageToSend = ""
  if Q->isNotEmpty()
    ssp = Q->pop()
    messageToSend = encode("SSPD", ssp, d)
  else
    if messageReceived->startWith("RESULTSSPD")
      w = w - 1
    endif
    messageToSend = "NOP"
  endif
  master()->buffer() = messageToSend
end
```

Enfin, au niveau du *worker*, il faut simplement ajouter le traitement des messages SSPD et NOP. Lors du traitement du message SSPD, le *worker* décode le message afin de récupérer le sous-problème à décomposer à la profondeur indiquée dans le message. Les résultats de cette décomposition sont représentés par une liste de sous-problèmes nommée $listSSP$ qui sera ensuite encodée dans un message de type RESULTSSPD. Pour le message de type NOP, le *worker* crée un message de type ASK pour demander à nouveau un sous-problème.

```
1  WorkerTask::run()
2  begin
3    if _done == true
4      return
5
6    messageReceived = worker()->buffer()
7    // Reçoit un message pour décomposer un problème : étape décomposition
8    if messageReceived->startWith("SSPD")
9      (ssp, d) = decode(messageReceived)
10     listSSP = solve(ssp, d)
11     messageToSend = encode("RESULTSSPD", listSSP)
12     worker()->buffer() = messageToSend
13   // Reçoit un message pour résoudre un problème : étape résolution
14   else if messageReceived->startWith("SSP")
15     ssp = decode(messageReceived)
16     results = solve(ssp)
17     messageToSend = encode("RESULTSSP", results)
18     worker()->buffer() = messageToSend
19   // Ne rien faire et demander à nouveau
20   // dans le cas de l'attente qu'une synchronisation se termine
21   else if messageReceived->startWith("NOP")
22     messageToSend = "ASK"
23     worker()->buffer() = messageToSend
24   else if messageReceived->startWith("QUIT")
25     _done = true
26   endif
27 end
```

CHAPITRE 6

Conclusion et perspectives

Dans cette thèse, nous avons proposé une nouvelle méthode de parallélisation qui permet d'améliorer la résolution de problème de l'espace de recherche en programmation par contraintes (PPC). Cette méthode, nommée *Embarrassingly Parallel Search* (EPS), obtient de très bonnes performances non seulement lors de la recherche de toutes les solutions d'un problème ou dans la preuve qu'un problème n'a pas de solution, mais également lors de la résolution de problèmes d'optimisation. En effet, les résultats montrent que sur différentes architectures (machine multi-cœurs, centre de calcul et *cloud computing*), la méthode obtient des gains linéaires en fonction du nombre d'unités de calcul.

Le principe d'EPS est d'arriver statistiquement à un équilibrage des temps de résolution de chaque unité de calcul afin d'obtenir une bonne répartition de la charge de travail aux unités de calcul. EPS s'appuie sur la propriété suivante : la somme des temps de résolution de chacun des sous-problèmes est comparable au temps de résolution du problème en entier. Cette propriété est vérifiée en PPC, ce qui nous permet de disposer d'une méthode simple et efficace en pratique.

EPS exploite la décomposition du problème en le partitionnant en un grand nombre de sous-problèmes disjoints. Ces sous-problèmes sont mis dans une file d'attente qui sera à la disposition des unités de calcul. Chaque unité de calcul prend tour à tour un sous-problème de la file d'attente afin de le résoudre.

Nous avons proposé quatre algorithmes de décomposition permettant de générer les sous-problèmes. Les trois premiers algorithmes permettent une décomposition exacte et statique, c'est-à-dire qu'ils génèrent le nombre de sous-problèmes fixé au départ. Le premier algorithme proposé est séquentiel et est basé sur la répétition d'un parcours en profondeur bornée. Nous montrons ses limites avec des centaines d'unités de calcul car engendrer un grand nombre de sous-problèmes prend de plus en plus de temps. En proposant deux versions parallèles de cet algorithme, nous arrivons à remédier à ce problème. Le dernier algorithme proposé permet d'effectuer une décomposition dynamique séquentielle. Il ne génère pas exactement le nombre de sous-problèmes souhaité afin de les servir plus rapidement aux unités de calcul dès qu'un sous-problème est généré.

Les analyses sur la décomposition montrent que celle-ci doit générer au moins 30 sous-problèmes par unité de calcul et que ces sous-problèmes doivent être consistants avec la propagation pour obtenir des charges de travail par unité de calcul équivalentes. Nous avons également comparé les résultats avec différentes approches comme le *work stealing* et les méthodes de *portfolio* existantes et nous avons conclu qu'EPS donne les meilleures performances dans l'énumération des solutions d'un problème et dans la recherche de la solution optimale d'un problème d'optimisation. Nous avons souligné les limites de l'approche du

work stealing qui présente des difficultés avec des centaines d'unités de calcul dues entre autres à l'utilisation massive de la communication. Quant aux méthodes de *portfolio*, EPS donne de meilleures performances lorsque le problème ne possède aucune solution.

Enfin, nous avons proposé au lecteur une interface de programmation (API) générique afin qu'il puisse implémenter facilement la méthode EPS sur n'importe quelle architecture (architecture à mémoire partagée avec les *threads* ou architecture distribuée avec MPI et programmation réseau).

Plusieurs perspectives se dégagent de ce travail.

Pour la recherche de la première des solutions d'un problèmes, nous pouvons combiner EPS en combinant avec les approches du *portfolio*. Des contributions utilisant EPS dans ce domaine pourraient être bénéfiques.

Une autre piste intéressante pourrait être explorée dans le portage de la méthode EPS sur d'autres types de modélisation de problème comme les approches SAT et MIP (*Mixed-Integer Programming*). Cependant, il est difficile de vérifier que la somme des temps de résolution des sous-problèmes soit comparable au temps de résolution du problème. En SAT, des méthodes proches d'EPS comme *CubeAndConquer* [Heule 2012, Van Der Tak 2012] ont été proposées. Les résultats montrent que leurs méthodes surpassent les méthodes parallèles existantes comme la méthode *CDCL* [Biere 2009].

Enfin, de nos jours, le développement du *cloud computing* pose une problématique. Comment trouver un compromis entre le coût des ressources et les performances souhaitées ?
Nous répondons en partie à cette problématique dans [Rezgui 2014b]. À partir des gains de performance constatés d'EPS et des coûts horaires d'utilisation des ressources avec le *cloud* de Microsoft Azure, nous estimons le coût des ressources en fonction des performances souhaitées. Il serait intéressant de continuer dans cette voie en estimant le temps de résolution d'un problème avec l'utilisation des méthodes d'échantillonnage sur les sous-problèmes générés par EPS.

ANNEXE A

Annexe : Notations

Afin de faciliter la lecture des différents paramètres utilisés dans l'interprétation de nos résultats, nous utilisons une grille de notation.

notation	**description**	**valeur par défaut**
c	nombre de cœurs	
w	nombre de *workers*	$w = 2 \times c$
$\#sspw$	nombre de sous-problèmes par *worker*	$\#sspw = 30$
$p(w)$	nombre de sous-problèmes qu'on souhaite générer	$p(w) = \#sspw \times w$
p	nombre réel de sous-problèmes générés	
d	profondeur de l'arbre de recherche atteinte par la décomposition	
d_m	profondeur de l'arbre de recherche atteinte par la décomposition (niveau *master*) (EDS/EDPS)	
p_m	nombre de sous-problèmes générés au niveau du *master* (EDS/EDPS)	
d_w	profondeur de l'arbre de recherche qu'atteint la décomposition (niveau *worker*) (EDS/EDPS)	
t_0	temps de résolution en séquentiel	
t	temps de résolution en parallèle	
t_e	temps estimé	
t_d	temps de décomposition	
t_w	temps de résolution par les *workers*	
su	facteur de gain	$su = t_0 \div t$
su_w	facteur de gain	$su_w = t_0 \div t_w$
M.G	moyenne géométrique	
M.A	moyenne arithmétique	

TABLE A.1 – Notations

ANNEXE B

Annexe : Instances

Voici la liste des instances utilisées dans les expérimentations.

- Queens 17/18 [Rossi 2006].
- Golomb ruler 13/14 [Galinier 2003, Smith 1999, Dollas 1998]
- `xcsp 1` : 5 unsat ; 5 enum ; solvable with `lex` and `dom`

```
xcsp1
|-- enum
|   |-- costasArray-14.xml
|   |-- ortholatin-5.xml
|   |-- queenAttacking-6.xml
|   |-- series-14.xml
|   `-- squares-9-9.xml
`-- unsat
    |-- knights-80-5.xml
    |-- latinSquare-dg-8_all.xml
    |-- lemma-100-9-mod.xml
    |-- pigeons-14.xml
    `-- quasigroup5-10.xml

2 répertoires, 10 fichiers
```

- `xcsp 2` :11 unsat ; 3 enum ;

```
xcsp2
|-- enum
|   |-- crossword-m1c-words-vg7-7_ext.xml
|   |-- crossword-m1-words-05-06.xml
|   `-- langford-3-17.xml
`-- unsat
    |-- cc-15-15-2.xml
    |-- fapp07-0600-7.xml
    |-- knights-20-9.xml
    |-- knights-25-9.xml
    |-- langford-4-18.xml
```

```
    |-- langford-4-19.xml
    |-- queensKnights-20-5-mul.xml
    |-- ruler-70-12-a3.xml
    |-- ruler-70-12-a4.xml
    |-- scen11-f5.xml
    `-- squaresUnsat-19-19.xml

2 répertoires, 14 fichiers
```

- `fzn 1` : 6 enum ; 12 optim

```
fzn1
|-- enum
|   |-- allinterval_15.fzn
|   |-- magicsequence_40000.fzn
|   |-- sportsleague_10.fzn
|   |-- sb_sb_13_13_6_4.fzn
|   |-- quasigroup7_10.fzn
|   `-- non_non_fast_6.fzn
`-- optim
    |-- golombruler_13
    |-- warehouses.fzn
    |-- setcovering.fzn
    |-- depot_placement_att48_5.fzn
    |-- depot_placement_rat99_5.fzn
    |-- fastfood_ff58.fzn
    |-- open_stacks_01_problem_15_15.fzn
    |-- open_stacks_01_wbp_30_15_1.fzn
    |-- sugiyama2_g5_7_7_7_7_2.fzn
    |-- patternSetMiningGerman.fzn
    |-- radiation_03.fzn
    `-- talent_scheduling_alt_film116.fzn

2 répertoires, 18 fichiers
```

- `fzn 2` : 1 unsat ; 6 enum ; 11 optim

```
fzn2
|-- enum
|   |-- fillomino_18.fzn
|   |-- market_split_s5-02.fzn
|   |-- market_split_s5-06.fzn
|   |-- nmseq_300.fzn
|   |-- nmseq_400.fzn
|   `-- steiner-triples_09.fzn
|-- optim
|   |-- golombruler_13
|   |-- bacp-27.fzn
|   |-- bacp-6.fzn
|   |-- cc_base_mzn_rnd_test.11.fzn
|   |-- depot_placement_st70_6.fzn
|   |-- ghoulomb_3-7-20.fzn
|   |-- open_stacks_01_wbp_20_20_1.fzn
|   |-- pattern_set_mining_k1_yeast.fzn
|   |-- still_life_free_8x8.fzn
|   |-- still_life_still_life_9.fzn
|   `-- talent_scheduling_alt_film117.fzn
`-- unsat
    `-- market_split_u5-09.fzn

3 répertoires, 18 fichiers
```

Bibliography

[Aggoun 1993] Abderrahmane Aggoun et Nicolas Beldiceanu. *Extending CHIP in order to solve complex scheduling and placement problems*. Mathematical and Computer Modelling, vol. 17, no. 7, pages 57–73, 1993. (Cité en page 19.)

[Almasi 1989] G. S. Almasi et A. Gottlieb. Highly Parallel Computing. Benjamin-Cummings Publishing Co., Inc., Redwood City, CA, USA, 1989. (Cité en page 5.)

[Amadini 2012] Roberto Amadini, Maurizio Gabbrielli et Jacopo Mauro. *An Empirical Evaluation of Portfolios Approaches for solving CSPs*. CoRR, 2012. (Cité en page 9.)

[Amdahl 1967] Gene Amdahl. *Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities*. In Proceedings of the April 18-20, 1967, Spring Joint Computer Conference, AFIPS '67, pages 483–485, New York, NY, USA, 1967. ACM. (Cité en page 22.)

[Atallah 2002] Mikhail J Atallah. Algorithms and theory of computation handbook. CRC press, 2002. (Cité en page 82.)

[Baptiste 1996] Philippe Baptiste et Claude Le Pape. *Edge-® finding constraint propagation algorithms*. In Proc. of the ECAI workshop on Intelligent Scheduling of Production Processes, 1996. (Cité en page 19.)

[Beck 2005] Christopher Beck, Patrick Prosser et Richard Wallace. *Trying Again to Fail-First*. In Recent Advances in Constraints, pages 41–55. Springer Berlin Heidelberg, 2005. (Cité en page 21.)

[Beldiceanu 1994] Nicolas Beldiceanu et Evelyne Contejean. *Introducing global constraints in CHIP*. Mathematical and computer Modelling, vol. 20, no. 12, pages 97–123, 1994. (Cité en page 20.)

[Beldiceanu 2005] Nicolas Beldiceanu, Mats Carlsson et Jean-Xavier Rampon. *Global Constraint Catalog*, 2005. Research Report SICS T2005-08. (Cité en pages 19 et 20.)

[Berge 1983] Claude Berge. Graphes. Gauthier-Villars, 1983. (Cité en page 12.)

[Berson 1992] Alex Berson*et al.* Client/server architecture, volume 2. McGraw-Hill New York, 1992. (Cité en page 79.)

[Bessiere 1994] Christian Bessiere. *Arc-consistency and arc-consistency again*. Artificial intelligence, vol. 65, no. 1, pages 179–190, 1994. (Cité en pages 3 et 18.)

[Bessiere 1996] Christian Bessiere et Jean-Charles Régin. *MAC and combined heuristics: Two reasons to forsake FC (and CBJ?) on hard problems*. In Principles and Practice of Constraint Programming—CP96, pages 61–75. Springer Berlin Heidelberg, 1996. (Cité en page 21.)

[Bessière 1997] Christian Bessière et Jean-Charles Régin. *Arc Consistency for General Constraint Networks: Preliminary Results*. In Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence, IJCAI 97, Nagoya, Japan, August 23-29, 1997, 2 Volumes, pages 398–404. Morgan Kaufmann, 1997. (Cité en page 18.)

[Bessière 2001] C. Bessière et J-C. Régin. *Refining the Basic Constraint Propagation Algorithm*. In Proceedings of IJCAI'01, pages 309–315, Seattle, WA, USA, 2001. (Cité en pages 3 et 8.)

[Biere 2009] Armin Biere, Marijn Heule, Hans van Maaren et Toby Walsh. *Conflict-driven clause learning SAT solvers*. Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications, pages 131–153, 2009. (Cité en page 102.)

[Blelloch 1998] Guy E. Blelloch, Charles E. Leiserson, Bruce M Maggs, C Greg Plaxton, Stephen J Smith et Marco Zagha. *An experimental analysis of parallel sorting algorithms*. Theory of Computing Systems, vol. 31, no. 2, pages 135–167, 1998. (Cité en page 6.)

[Bordeaux 2009] Lucas Bordeaux, Youssef Hamadi et Horst Samulowitz. *Experiments with Massively Parallel Constraint Solving*. In Boutilier [Boutilier 2009], pages 443–448. (Cité en pages 8, 9, 27, 50, 51 et 63.)

[Boussemart 2004] Frédéric Boussemart, Fred Hemery, Christophe Lecoutre et Lakhdar Sais. *Boosting Systematic Search by Weighting Constraints*. In Proceedings of the 16th Eureopean Conference on Artificial Intelligence, ECAI'2004, including Prestigious Applicants of Intelligent Systems, PAIS, pages 146–150, 2004. (Cité en page 21.)

[Boutilier 2009] Craig Boutilier, editeur. IJCAI 2009, proceedings of the 21st international joint conference on artificial intelligence, pasadena, california, usa, july 11-17, 2009. (Cité en pages 110 et 114.)

[Burton 1981] F. Warren Burton et M. Ronan Sleep. *Executing Functional Programs on a Virtual Tree of Processors*. In Proceedings of the 1981 Conference on Functional Programming Languages and Computer Architecture, FPCA '81, pages 187–194, New York, NY, USA, 1981. ACM. (Cité en page 27.)

[Capit 2005] Nicolas Capit, Georges Da Costa, Yiannis Georgiou, Guillaume Huard, Cyrille Martin, Grégory Mounié, Pierre Neyron et Olivier Richard. *A batch scheduler with high level components*. In Cluster Computing and the Grid, 2005. CCGrid 2005. IEEE International Symposium, volume 2, pages 776–783. IEEE, 2005. (Cité en page 48.)

[Cérin 2006] Christophe Cérin, Jean-Christophe Dubacq et Jean-Louis Roch. *Methods for partitioning data to improve parallel execution time for sorting on heterogeneous clusters*. In Advances in Grid and Pervasive Computing, pages 175–186. Springer, 2006. (Cité en page 6.)

[Cheng 2010] Kenil C. K. Cheng et Roland H. C. Yap. *An MDD-based generalized arc consistency algorithm for positive and negative table constraints and some global constraints*. Constraints, vol. 15, no. 2, pages 265–304, 2010. (Cité en page 18.)

[Choco 2010] Team Choco. *Choco: an open source java constraint programming library*. Ecole des Mines de Nantes, Research report, vol. 1, pages 10–02, 2010. (Cité en page 4.)

[Chong 2006] Yek Loong Chong et Youssef Hamadi. *Distributed Log-Based Reconciliation*. In ECAI, volume 141, pages 108–112, 2006. (Cité en pages 8 et 24.)

[Chu 2009] Geoffrey Chu, Christian Schulte et Peter J. Stuckey. *Confidence-Based Work Stealing in Parallel Constraint Programming*. In Ian P. Gent, editeur, CP, volume 5732 of *Lecture Notes in Computer Science*, pages 226–241. Springer, 2009. (Cité en pages 8, 29 et 76.)

[Colmerauer 1990] A. Colmerauer. *An introduction to PROLOG III*. Communications of the ACM, vol. 33, pages 69–90, 1990. (Cité en page 1.)

[Cornuéjols 2006] Gérard Cornuéjols, Miroslav Karamanov et Yanjun Li. *Early Estimates of the Size of Branch-and-Bound Trees*. INFORMS Journal on Computing, vol. 18, pages 86–96, 2006. (Cité en pages 27, 32 et 44.)

[Dechter 1992] R. Dechter. *From local to global consistency*. JAI, vol. 55, pages 87–107, 1992. (Cité en page 13.)

[Dollas 1998] Apostolos Dollas, William T Rankin et David McCracken. *A new algorithm for Golomb ruler derivation and proof of the 19 mark ruler*. IEEE Transactions on Information Theory, vol. 44, pages 379–382, 1998. (Cité en page 105.)

[Ezzahir 2007] Redouane Ezzahir, Christian Bessiere, Mustapha Belaissaoui et El Houssine Bouyakhf. *DisChoco: A platform for distributed constraint programming*. Proceedings of the IJCAI, vol. 7, pages 16–21, 2007. (Cité en page 24.)

[Fischetti 2014] Matteo Fischetti, Michele Monaci et Domenico Salvagnin. *Self-splitting of workload in parallel computation*. In CPAIOR'14, 2014. (Cité en page 76.)

[Freuder 1978] E.C. Freuder. *Synthesizing constraint expressions*. CACM, vol. 21, no. 11, pages 958–966, 1978. (Cité en page 1.)

[Gabriel 2004] Edgar Gabriel, Graham Fagg, George Bosilca, Thara Angskun, Jack Dongarra, Jeffrey Squyres, Vishal Sahay, Prabhanjan Kambadur, Brian Barrett, Andrew Lumsdaine*et al. Open MPI: Goals, Concept, and Design of a next generation MPI*

implementation. In Recent Advances in Parallel Virtual Machine and Message Passing Interface, pages 97–104. Springer, 2004. (Cité en pages 48 et 83.)

[Galinier 2003] Philippe Galinier. A constraint-based approach to the golomb ruler problem. Montréal: Center for Research on Transportation (CRT), 2003. (Cité en pages 75 et 105.)

[Gent 1999] Ian Gent et Toby Walsh. *CSPLIB: A Benchmark Library for Constraints*. In Proceedings of the 5th International Conference on Principles and Practice of Constraint Programming, CP '99, pages 480–481, 1999. (Cité en page 46.)

[Gomes 1997] Carla Gomes et Bart Selman. *Algorithm Portfolio Design: Theory vs. Practice*. In Proceedings of the Thirteenth conference on Uncertainty in artificial intelligence, pages 190–197, 1997. (Cité en page 9.)

[Gomes 1999] Carla Gomes et Bart Selman. *Search strategies for hybrid search spaces*. In Tools with Artificial Intelligence, 1999. Proceedings. 11th IEEE International Conference, pages 359–364. IEEE, 1999. (Cité en page 9.)

[Gomes 2000] Carla Gomes et Bart Selman. *Hybrid Search Strategies For Heterogeneous Search Spaces*. International Journal on Artificial Intelligence Tools, vol. 09, pages 45–57, 2000. (Cité en pages 8 et 25.)

[Gomes 2001] Carla Gomes et Bart Selman. *Algorithm Portfolios*. Artificial Intelligence, vol. 126, pages 43–62, 2001. (Cité en page 9.)

[Gondran 2009] Michel Gondran et Michel Minoux. Graphes et algorithmes. Tec & Doc Lavoisier, 2009. (Cité en page 12.)

[Gropp 1993] William Gropp et Ewing Lusk. *The MPI communication library: its design and a portable implementation*. In Scalable Parallel Libraries Conference, 1993., Proceedings of the, pages 160–165. IEEE, 1993. (Cité en pages 48, 49 et 82.)

[Halstead 1984] Robert Halstead. *Implementation of Multilisp: Lisp on a Multiprocessor*. In Proceedings of the 1984 ACM Symposium on LISP and Functional Programming, LFP '84, pages 9–17, New York, NY, USA, 1984. ACM. (Cité en page 27.)

[Hamadi 1998] Youssef Hamadi, Christian Bessière et Joël Quinqueton. *Backtracking in Distributed Constraint Networks*. International Journal on Artificial Intelligence Tools, pages 219–223, 1998. (Cité en page 8.)

[Hamadi 2002] Youssef Hamadi. *Optimal Distributed Arc-Consistency*. Constraints, vol. 7, pages 367–385, 2002. (Cité en page 8.)

[Hamadi 2009] Youssef Hamadi, Said Jabbour et Lakhdar Sais. *ManySAT: a Parallel SAT Solver.* JSAT, vol. 6, no. 4, pages 245–262, 2009. (Cité en pages 9 et 25.)

[Haralick 1980] Robert Haralick et Gordon Elliott. *Increasing Tree Search Efficiency for Constraint Satisfaction Problems*. Artificial intelligence, vol. 14, no. 3, pages 263–313, 1980. (Cité en page 21.)

[Harary 1969] Frank Harary. Graph theory. Addison-Wesley, 1969. (Cité en page 12.)

[Hentenryck 1992] Pascal Van Hentenryck, Yves Deville et Choh man Teng. *A Generic Arc-Consistency Algorithm and its Specializations*. Artificial Intelligence, vol. 57, pages 291–321, 1992. (Cité en pages 3 et 18.)

[Heule 2012] Marijn JH Heule, Oliver Kullmann, Siert Wieringa et Armin Biere. *Cube and conquer: Guiding CDCL SAT solvers by lookaheads*. In Hardware and Software: Verification and Testing, pages 50–65. Springer, 2012. (Cité en page 102.)

[Hirayama 1997] Katsutoshi Hirayama et Makoto Yokoo. *Distributed Partial Constraint Satisfaction Problem*. In Principles and Practice of Constraint Programming-CP97, pages 222–236. Springer, 1997. (Cité en pages 8 et 24.)

[Hyde 1999] Paul Hyde. Java thread programming, volume 1. Sams, 1999. (Cité en pages 48 et 49.)

[IntelMPI 2015] IntelMPI. *Intel MPI Library*. `https://software.intel.com/en-us/intel-mpi-library`, 2015. (Cité en pages 48, 49 et 83.)

[Jaffar 2004] Joxan Jaffar, Andrew E. Santosa, Roland H. C. Yap et Kenny Qili Zhu. *Scalable Distributed Depth-First Search with Greedy Work Stealing*. In ICTAI, pages 98–103. IEEE Computer Society, 2004. (Cité en pages 8, 29 et 50.)

[Jeon 2002] Minsoo Jeon et Dongseung Kim. *Parallelizing merge sort onto distributed memory parallel computers*. In High Performance Computing, pages 25–34. Springer, 2002. (Cité en page 6.)

[Kautz 2002] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes et Bart Selman. *Dynamic Restart Policies*. AAAI/IAAI, vol. 97, pages 674–681, 2002. (Cité en page 9.)

[Kilby 2006] Philip Kilby, John K. Slaney, Sylvie Thiébaux et Toby Walsh. *Estimating Search Tree Size*. In AAAI, pages 1014–1019, 2006. (Cité en page 27.)

[Kjellerstrand 2014] Hå kan Kjellerstrand. *Håkan Kjellerstrand's Blog*. `http://www.hakank.org/`, 2014. (Cité en page 46.)

[Kleiman 1996] Steve Kleiman, Devang Shah et Bart Smaalders. Programming with threads. Sun Soft Press, 1996. (Cité en pages 48, 81 et 82.)

[Knuth 1975] Donald E. Knuth. *Estimating the efficiency of backtrack program*. Mathematics of Computation, vol. 29, pages 121–136, 1975. (Cité en page 27.)

[Kowalski 1979] Robert Kowalski. *Algorithm= logic+ control*. Communications of the ACM, vol. 22, no. 7, pages 424–436, 1979. (Cité en page 2.)

[Krishna 2010] Jayesh Krishna, Pavan Balaji, Ewing Lusk, Rajeev Thakur et Fabian Tiller. *Implementing MPI on Windows: Comparison with Common Approaches on Unix*. In Recent Advances in the Message Passing Interface, volume 6305 of *Lecture Notes in Computer Science*, pages 160–169. Springer Berlin Heidelberg, 2010. (Cité en pages 48, 49 et 83.)

[Lantz 2008] Eric Lantz. *Using Microsoft Message Passing Interface (MS-MPI)*. Windows HPC Server, 2008. (Cité en pages 48, 49 et 83.)

[Laurière 1976] J.-L. Laurière. *Un langage et un programme pour énoncer et résoudre des problèmes combinatoires*. PhD thesis, Université de Paris VI, 1976. (Cité en page 1.)

[Laurière 1978] J.-L. Laurière. *A Language and a Program for Stating and Solving Combinatorial Problems*. Artificial Intelligence, vol. 10, pages 29–127, 1978. (Cité en page 1.)

[Le Cun 2007] Bertrand Le Cun, Tarek Menouer et Pascal Vander-Swalmen. *Bobpp*. `http://forge.prism.uvsq.fr/projects/bobpp`, 2007. (Cité en page 27.)

[Léauté 2009] Thomas Léauté, Brammert Ottens et Radoslaw Szymanek. *FRODO 2.0: An open-source framework for distributed constraint optimization*. In Boutilier [Boutilier 2009], pages 160–164. (Cité en pages 8 et 24.)

[Lecoutre 2011] Christophe Lecoutre. *STR2: optimized simple tabular reduction for table constraints*. Constraints, vol. 16, no. 4, pages 341–371, 2011. (Cité en page 18.)

[Lecoutre 2012] Christophe Lecoutre, Chavalit Likitvivatanavong et Roland Yap. *A path-optimal GAC algorithm for table constraints*. In 20th European Conference on Artificial Intelligence (ECAI'12), pages 510–515, 2012. (Cité en page 18.)

[Lester 1993] Bruce Lester. The art of parallel programming. Prentice Hall Englewood Cliffs, NJ, 1993. (Cité en pages 48, 49 et 82.)

[Li 2009] Henry Li. Introduction to windows azure. Springer, 2009. (Cité en page 48.)

[Luby 1993] Michael Luby, Alistair Sinclair et David Zuckerman. *Optimal Speedup of Las Vegas Algorithms*. Inf. Process. Lett., vol. 47, pages 173–180, 1993. (Cité en pages 9 et 25.)

[Machado 2013] Rui Machado, Vasco Pedro et Salvador Abreu. *On the Scalability of Constraint Programming on Hierarchical Multiprocessor Systems*. In ICPP, pages 530–535. IEEE, 2013. (Cité en page 64.)

[Mackworth 1977] Alan K Mackworth. *Consistency in networks of relations*. Artificial intelligence, vol. 8, no. 1, pages 99–118, 1977. (Cité en pages 3 et 18.)

[Malapert 2014] Arnaud Malapert et Christophe Lecoutre. *À propos de la bibliothèque de modèles XCSP*. In 10èmes Journées Francophones de Programmation par Contraintes(JFPC'15), Angers, France, 2014. (Cité en page 46.)

[McCarthy 1977] John McCarthy. *Epistemological problems of artificial intelligence*. Proceedings of the Fifth International Joint Conference on Artificial Intelligence, vol. 2, pages 1038–1044, 1977. (Cité en page 1.)

[McCarthy 1987] John McCarthy. *Generality in artificial intelligence*. Communications of the ACM, vol. 30, no. 12, pages 1030–1035, 1987. (Cité en page 1.)

[Menouer 2014a] Tarek Menouer et Bertrand Le Cun. *Adaptive N To P Portfolio for Solving Constraint Programming Problems on Top of the Parallel Bobpp Framework*. In 2014 IEEE 28th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum, 2014. (Cité en pages 63 et 76.)

[Menouer 2014b] Tarek Menouer, Mohamed Rezgui, Bertrand Le Cun et Jean-Charles Régin. *Mixing Static and Dynamic Partitionning to Parallelize a Constraint Programming Solver*. In High-level Parallel Programming and Applications (HLPP), Amsterdam, Netherlands, jun 2014. (Cité en page 9.)

[Menouer 2015] Tarek Menouer, Mohamed Rezgui, Bertrand Le Cun et Jean-Charles Régin. *Mixing Static and Dynamic Partitionning to Parallelize a Constraint Programming Solver*. In International Journal of Parallel Programming (IJPP), 2015. (Cité en page 9.)

[Michel 2009] Laurent Michel, Andrew See et Pascal Van Hentenryck. *Transparent Parallelization of Constraint Programming*. INFORMS Journal on Computing, vol. 21, pages 363–382, 2009. (Cité en pages 8, 29 et 76.)

[Michel 2012] Laurent Michel et Pascal Van Hentenryck. *Activity-based search for black-box constraint programming solvers*. In Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems, pages 228–243. Springer, 2012. (Cité en page 21.)

[Miller 1986] Barton Miller, Cathryn Macrander et Stuart Sechrest. *A distributed programs monitor for Berkeley UNIX*. Software: Practice and Experience, vol. 16, no. 2, pages 183–200, 1986. (Cité en page 84.)

[MiniZinc 2012] MiniZinc. *MiniZinc and FlatZinc*. `http://www.g12.csse.unimelb.edu.au/minizinc/`, 2012. (Cité en pages 46 et 47.)

[Mohr 1986] Roger Mohr et Thomas C Henderson. *Arc and path consistency revisited*. Artificial intelligence, vol. 28, no. 2, pages 225–233, 1986. (Cité en pages 3 et 18.)

[Mohr 1988] Roger Mohr et Gérald Masini. *Good Old Discrete Relaxation*. In ECAI, pages 651–656, 1988. (Cité en page 18.)

[Montanari 1974] U. Montanari. *Networks of constraints : Fundamental Properties and applications to Picture Processing*. Information Science, vol. 7, pages 95–132, 1974. (Cité en page 1.)

[MPICH 2015] MPICH. *MPICH Library*. http://www.mpich.org/, 2015. (Cité en pages 48 et 83.)

[MS-MPI 2015] MS-MPI. *Microsoft HPC Pack 2012 R2 and HPC Pack 2012*. http://technet.microsoft.com/en-us/library/jj899572.aspx, 2015. (Cité en page 48.)

[Mueller 1993] Frank Muller*et al.* *A Library Implementation of POSIX Threads under UNIX*. In USENIX Winter, pages 29–42, 1993. (Cité en pages 48, 81 et 82.)

[Nethercote 2007] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck et Guido Tack. *MiniZinc: Towards a Standard CP Modelling Language*. In Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP'07, pages 529–543, Berlin, Heidelberg, 2007. Springer-Verlag. (Cité en pages 17 et 46.)

[Nielsen 2006] Morten Nielsen. Parallel Search in Gecode. Master's thesis, KTH Royal Institute of Technology, 2006. (Cité en page 66.)

[O'Mahony 2008] Eoin O'Mahony, Emmanuel Hebrard, Alan Holland, Conor Nugent et Barry O'Sullivan. *Using case-based reasoning in an algorithm portfolio for constraint solving*. Irish Conference on Artificial Intelligence and Cognitive Science, pages 210–216, 2008. (Cité en page 71.)

[Pat 1999] Villani Pat. Advanced win32 programming: Files, threads, and process synchronization. Harpercollins Publishers, ISBN 0-87930-563-0, 1999. (Cité en page 82.)

[Pedro 2010] Vasco Pedro et Salvador Abreu. *Distributed Work Stealing for Constraint Solving*. CoRR, pages 1–18, 2010. (Cité en page 63.)

[Perez 2014] Guillaume Perez et Jean-Charles Régin. *Improving GAC-4 for Table and MDD Constraints*. In Barry O'Sullivan, editeur, Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings, volume 8656 of *Lecture Notes in Computer Science*, pages 606–621. Springer, 2014. (Cité en page 18.)

[Perron 1999] Laurent Perron. *Search Procedures and Parallelism in Constraint Programming*. In CP, volume 1713 of *Lecture Notes in Computer Science*, pages 346–360, 1999. (Cité en pages 8, 25 et 29.)

[Perron 2012] Laurent Perron, Van Omme Nikolaj et Furnon Vincent. *Or-Tools*. Rapport technique, Google, 2012. (Cité en pages 4 et 71.)

[Pesant 2004] Gilles Pesant. *A Regular Language Membership Constraint for Finite Sequences of Variables*. In Mark Wallace, editeur, Principles and Practice of Constraint Programming – CP 2004, volume 3258 of *Lecture Notes in Computer Science*, pages 482–495. Springer Berlin Heidelberg, 2004. (Cité en page 20.)

[Peterson 1985] James Lyle Peterson et Abraham Silberschatz. Operating system concepts, volume 2. Addison-Wesley Reading, MA, 1985. (Cité en page 79.)

[Postel 1981] Jon Postel. *Internet protocol*, 1981. (Cité en page 83.)

[Puget 1994] Jean-François Puget. *ILOG CPLEX CP Optimizer : A C++ implementation of CLP*. `http://www.ilog.com/`, 1994. (Cité en page 4.)

[Rashid 1980] Richard Rashid. *An inter-process communication facility for UNIX*, 1980. (Cité en page 81.)

[Refalo 2004] Philippe Refalo. *Impact-Based Search Strategies for Constraint Programming*. In Mark Wallace, editeur, Principles and Practice of Constraint Programming, 10th International Conference, CP 2004, Toronto, Canada, volume 3258 of *Lecture Notes in Computer Science*, pages 557–571. Springer, 2004. (Cité en page 21.)

[Régin 1994] J-C. Régin. *A filtering algorithm for constraints of difference in CSPs*. In AAAI94, pages 362–367, Seattle, Washington, 1994. (Cité en pages 4, 17, 19, 38 et 63.)

[Régin 1996] Jean-Charles Régin. *Generalized arc consistency for global cardinality constraint*. In Proceedings of the thirteenth national conference on Artificial intelligence-Volume 1, pages 209–215. AAAI Press, 1996. (Cité en page 19.)

[Régin 1997] Jean-Charles Régin et Jean-François Puget. *A filtering algorithm for global sequencing constraints*. In Principles and Practice of Constraint Programming-CP97, pages 32–46. Springer Berlin Heidelberg, 1997. (Cité en page 19.)

[Régin 2004] Jean-Charles Régin. *Modélisation et contraintes globales en programmation par contraintes*. Habilitation à diriger des recherches, Université Nice Sophia Antipolis, Novembre 2004. (Cité en page 4.)

[Régin 2005] Jean-Charles Régin. *AC-*: A configurable, generic and adaptive arc consistency algorithm*. In Principles and Practice of Constraint Programming-CP 2005, pages 505–519. Springer, 2005. (Cité en page 18.)

[Régin 2013] Jean-Charles Régin, Mohamed Rezgui et Arnaud Malapert. *Embarrassingly Parallel Search*. In Christian Schulte, editeur, Principles and Practice of Constraint Programming, volume 8124, pages 596–610. Springer Berlin Heidelberg, 2013. (Cité en page 9.)

[Régin 2014] Jean-Charles Régin, Mohamed Rezgui et Arnaud Malapert. *Improvement of the Embarrassingly Parallel Search for Data Centers*. In Barry O'Sullivan, editeur, Principles and Practice of Constraint Programming, volume 8656 of *Lecture Notes in Computer Science*, pages 622–635. Springer International Publishing, 2014. (Cité en page 9.)

[Rezgui 2013] Mohamed Rezgui, Jean-Charles Régin et Arnaud Malapert. *Une adaptation simple et efficace du modèle MapReduce pour la programmation par contraintes*. In 9èmes Journées Francophones de Programmation par Contraintes(JFPC'14), Aix en Provence, France, jun 2013. (Cité en page 9.)

[Rezgui 2014a] Mohamed Rezgui, Jean-Charles Régin et Arnaud Malapert. *Adaptation de la méthode Embarrassingly Parallel Search pour un centre de calcul*. In 10èmes Journées Francophones de Programmation par Contraintes(JFPC'14), Angers, France, jun 2014. (Cité en page 9.)

[Rezgui 2014b] Mohamed Rezgui, Jean-Charles Régin et Arnaud Malapert. *Using Cloud Computing for Solving Constraint Programming Problems*. In First Workshop on Cloud Computing and Optimization, Lyon, France, 2014. (Cité en pages 9 et 102.)

[Rossi 1990] Francesca Rossi, Charles J Petrie et Vasant Dhar. *On the Equivalence of Constraint Satisfaction Problems*. ECAI, vol. 90, pages 550–556, 1990. (Cité en page 3.)

[Rossi 2006] Francesca Rossi, Peter Van Beek et Toby Walsh, editeurs. Handbook of Constraint Programming. Elsevier, 2006. (Cité en pages 63 et 105.)

[Sanders 2004] Peter Sanders et Sebastian Winkel. *Super scalar sample sort*. In Algorithms–ESA 2004, pages 784–796. Springer, 2004. (Cité en page 6.)

[Schulte 2000] Christian Schulte. *Parallel Search Made Simple*. In "Proceedings of TRICS: Techniques foR Implementing Constraint programming Systems, a post-conference workshop of CP 2000, pages 41–57, Singapore, 2000. (Cité en pages 8, 29 et 60.)

[Schulte 2006] Christian Schulte. *Gecode: Generic Constraint Development Environment*. `http://www.gecode.org/`, 2006. (Cité en page 4.)

[Smith 1999] Barbara M. Smith, Kostas Stergiou et Toby Walsh. *Modelling the Golomb Ruler Problem*. In IJCAI-99 Workshop on Non-Binary Constraints. International Joint Conference on Artificial Intelligence, 1999. (Cité en page 105.)

[Ullmann 2007] Julian R Ullmann. *Partition search for non-binary constraint satisfaction*. Information Sciences, vol. 177, no. 18, pages 3639–3678, 2007. (Cité en page 18.)

[Van Der Tak 2012] Peter Van Der Tak, Marijn JH Heule et Armin Biere. *Concurrent cube-and-conquer*. In Theory and Applications of Satisfiability Testing–SAT 2012, pages 475–476. Springer, 2012. (Cité en page 102.)

[Van Hentenryck 1989] P. Van Hentenryck. Constraint satisfaction in logic programming. M.I.T. Press, 1989. (Cité en page 1.)

[Wahbi 2011] Mohamed Wahbi, Redouane Ezzahir, Christian Bessiere et El-Houssine Bouyakhf. *DisChoco 2: A platform for distributed constraint reasoning*. Proceedings of DCR, vol. 11, pages 112–121, 2011. (Cité en pages 8 et 24.)

[Waltz 1975] D. L. Waltz. *Understanding Line Drawings of Scenes with Shadows*. In The Psychology of Computer Vision, pages 19–91. McGraw Hill, 1975. d'abord paru dans Tech. Rep AI271, MIT MA, 1972. (Cité en page 1.)

[Wilkinson 2005] B. Wilkinson et M. Allen. Parallel Programming: Techniques and Application Using Networked Workstations and Parallel Computers. Prentice-Hall Inc., 2nd édition, 2005. (Cité en pages 31, 32 et 33.)

[XCSP 2008] XCSP. *XML representation of constraint networks format*. `http://www.cril.univ-artois.fr/CPAI08/XCSP2_1Competition.pdf`, 2008. (Cité en page 46.)

[Xie 2010] Feng Xie et Andrew Davenport. *Massively Parallel Constraint Programming for Supercomputers: Challenges and Initial Results*. In CPAIOR, volume 6140 of *Lecture Notes in Computer Science*, pages 334–338. Springer, 2010. (Cité en page 29.)

[Xu 2008] Lin Xu, Frank Hutter, Holger Hoos et Kevin Leyton-Brown. *SATzilla: Portfolio-based Algorithm Selection for SAT*. J. Artif. Int. Res., vol. 32, pages 565–606, Juin 2008. (Cité en page 8.)

[Yokoo 1990] Makoto Yokoo, Toru Ishida et Kazuhiro Kuwabara. *Distributed Constraint Satisfaction for DAI Problems*. In Proceedings of the 1990 Distributed AI Workshop, Bandara, TX, Octobre 1990. (Cité en pages 8 et 24.)

[Yokoo 1998] Makoto Yokoo, Edmund H. Durfee, Toru Ishida et Kazuhiro Kuwabara. *The Distributed Constraint Satisfaction Problem: Formalization and Algorithms*. IEEE Transactions on Knowledge and Data Engineering, vol. 10, pages 673–685, 1998. (Cité en page 8.)

[Zhang 2001] Yuanlin Zhang et Roland HC Yap. *Making AC-3 an optimal algorithm*. In IJCAI, volume 1, pages 316–321, 2001. (Cité en page 18.)

[Zoeteweij 2004] Peter Zoeteweij et Farhad Arbab. *A Component-Based Parallel Constraint Solver*. In Rocco De Nicola, Gian Luigi Ferrari et Greg Meredith, editeurs, COORDINATION, volume 2949 of *Lecture Notes in Computer Science*, pages 307–322. Springer, 2004. (Cité en pages 8, 29 et 63.)

Parallélisme en programmation par contraintes

Mots-clés programmation par contraintes, parallélisme, décomposition, résolution, embarrassingly parallel, work stealing, portfolio

Nous étudions la parallélisation de la procédure de recherche de solution d'un problème en Programmation Par Contraintes (PPC). Après une étude de l'état de l'art, nous présentons une nouvelle méthode, nommée *Embarrassingly Parallel Search* (EPS). Cette méthode est basée sur la décomposition d'un problème en un très grand nombre de sous-problèmes disjoints qui sont ensuite résolus en parallèle par des unités de calcul avec très peu, voire aucune communication. Le principe d'EPS est d'arriver statistiquement à un équilibrage des temps de résolution de chaque unité de calcul afin d'obtenir une bonne répartition de la charge de travail. EPS s'appuie sur la propriété suivante : la somme des temps de résolution de chacun des sous-problèmes est comparable au temps de résolution du problème en entier. Cette propriété est vérifiée en PPC, ce qui nous permet de disposer d'une méthode simple et efficace en pratique. Dans nos expérimentations, nous nous intéressons à la recherche de toutes les solutions d'un problème en PPC, à prouver qu'un problème n'a pas de solution et à la recherche d'une solution optimale d'un problème d'optimisation. Les résultats montrent que la décomposition doit générer au moins 30 sous-problèmes par unité de calcul pour obtenir des charges de travail par unité de calcul équivalentes. Nous évaluons notre approche sur différentes architectures (machine multi-cœurs, centre de calcul et *cloud computing*) et montrons qu'elle obtient un gain pratiquement linéaire en fonction du nombre d'unités de calcul. Une comparaison avec les méthodes actuelles telles que le *work stealing* ou le *portfolio* montre qu'EPS obtient de meilleurs résultats.

Parallelism in Constraint Programming

Keywords constraint programming, parallelism, decomposition, resolution, embarrassingly parallel, work stealing, portfolio

We study the search procedure parallelization in Constraint Programming (CP). After giving an overview on various existing methods of the state-of-the-art, we present a new method, named Embarrassinqly Parallel Search (EPS). This method is based on the decomposition of a problem into many disjoint subproblems which are then solved in parallel by computing units with little or without communication. The principle of EPS is to have a resolution times balancing for each computing unit in a statistical sense to obtain a good well-balanced workload. We assume that the amount of resolution times of all subproblems is comparable to the resolution time of the entire problem. This property is checked with CP and allows us to have a simple and efficient method in practice. In our experiments, we are interested in enumerating all solutions of a problem, and proving that a problem has no solution and finding an optimal solution of an optimization problem. We observe that the decomposition has to generate at least 30 subproblems per computing unit to get equivalent workloads per computing unit. Then, we evaluate our approach on different architectures (multicore machine, cluster and cloud computing) and we observe a substantially linear speedup. A comparison with current methods such as *work stealing* or *portfolio* shows that EPS gets better results.