Une corrélation en programmation par contraintes distribuée

A Correlation in Distributed Contraints Programming

Agnès Gorge Philippe Morignot
AXLOG Ingéniérie, 19-21, rue du 8 mai 1945, 94110 Arcueil, France.
Phone : +33 (0)1 41 24 31 27 / +33 (0)1 41 24 31 19 Fax : +33 (0)1 41 24 07 36
{agnes.gorge ; philippe.morignot}@axlog.fr

Résumé

Cet article présente une approche pour la programmation par contraintes distribuée : une méthode de décomposition d'un arbre de recherche en sous arbres, basée sur le tronçonnement du domaine de variables, pour résolution séparée asynchrone (par agents). La communication entre agents repose sur des messages « nogood » signifiant à tous les autres agents qu'une branche morte a été trouvée dans le sous-arbre d'un agent, et n'a pas à être explorée par les autres agents. Nous présentons un algorithme de distribution statique (les sous-arbres sont générés une fois pour toutes) et/ou dynamique (les sous-arbres sont générés à la demande en cours de résolution). Les expériences mettent en évidence une bonne corrélation entre l'inverse du nombre d'agents et le temps de recherche de toutes les solutions d'un problème donné.

Mots Clef

Programmation par contraintes, distribution, agents, partage de domaines.

Abstract

This paper presents an approach for distributed constraint programming: a method for decomposing a search tree into sub-trees, based on splitting the domain of variables, for separate asynchronous resolution (one per agent). Communication among agents relies on « nogood » messages, meaning that one agent has found a dead combination (variable, value) and preventing the other agents from exploring it. We present an algorithm in static (the sub-trees are generated once before exploration) and/or dynamic (the sub-trees are generated on the fly when needed) versions for handling this distribution. Experiments exhibit a good correlation between the inverse of the number of agents and the time for computing all the solutions of a given problem.

Keywords

Constraint programming, distribution, agents, domain split.

1 Introduction

La programmation par contraintes a été introduite il y a 30 ans (e.g., [8]), mais reste encore limitée par son temps de résolution --- dans le pire des cas exponentiel en fonction du nombre de variables et de la taille de leur domaine. Dans cet article, nous présentons une approche permettant de distribuer cette résolution sur plusieurs processeurs. Notre but est de diviser le temps de résolution par ce nombre de processeurs, non pas en utilisant la parallélisation au niveau du code binaire, mais en raisonnant sur les variables, domaines et contraintes d'un problème représenté en programmation par contraintes.

Cette étude vise à être utilisée dans une application impliquant une patrouille d'avions de combat pilotés par ordinateurs, qui doivent attaquer une cible au sol en territoire ennemi [18]. Plus précisément, ces avions calculent leur trajectoire (discrète, dans un graphe de couloirs de vol, et continue, trajectoire à l'intérieur d'un couloir de vol) et leurs actions en utilisant la programmation par contraintes. De plus, ils doivent re-calculer dynamiquement leurs actions et trajectoires (re-planification), en raison de menaces inopinées (des radars ennemis, avec leur batterie de missiles associée, qui peuvent éclairer un avion). La programmation par contraintes *distribuée* conduit à un calcul de re-planification (embarqué) à bord d'avions considérés comme *au même niveau*, et non au calcul centralisé sur un avion chef de patrouille --- temps de réaction global plus rapide, meilleur réalisme, etc.

Le domaine de la programmation par contraintes distribuée commence à être abordée par de nombreux chercheurs. Une première approche consiste à étudier les problèmes intrinsèquement distribués, dus à la confidentialité des données ou au temps de communication [15]. Une variable définit un « agent », qui connaît au moins les

contraintes dans lesquelles cette variable prend part. Ces agents sont ordonnés heuristiquement, pour être utilisés dans l'algorithme suivant : le premier agent choisit une valeur pour sa variable et la transmet à l'agent suivant ; le deuxième agent essaie de choisir une valeur pour sa propre variable, qui soit compatible avec la première affectation : si elle existe, le processus se poursuit récursivement vers la variable suivante ; sinon, un message est renvoyé au premier agent, pour qu'il modifie la valeur qu'il avait choisie.

Une deuxième approche éclate le problème global en sous problèmes locaux, un par contrainte, avec les variables y prenant part. Puis, toutes les solutions locales sont générées et testées par paire vis-à-vis du problème global (jointure et test) [9].

Une troisième approche construit un graphe de microstructure [3] : un nœud est une paire (variable, valeur) ; un arc existe entre deux nœuds si et seulement si (i) les variables sont différentes et (ii) les valeurs dans les nœuds sont compatibles avec toutes les contraintes. Chercher une solution revient alors à chercher une k-clique dans ce graphe --- ce qui est aussi un problème NP-difficile.

L'approche que nous proposons se base sur la seconde approche précédente (séparation du problème en plusieurs sous-problèmes appelés « agents »), mais en diffère en ce que la séparation concerne les *valeurs* d'une ou plusieurs variables, et non les contraintes --- chaque sous-problème/agent contient *toutes* les contraintes du problème global. Dans notre approche, une solution de l'un des agents sera une solution du problème global.

Un avantage est que tous les sous-problèmes peuvent être résolus en même temps de façon asynchrone (parallélisme vrai). De plus, une coopération est établie par communication entre agents, pour éviter que la même branche d'un sous-arbre de recherche ne soit explorée par deux agents différents --- en particulier, lorsqu'elle ne contient pas de solution.

Cet article est organisé comme suit : d'abord nous décrivons le modèle de distribution de la recherche de solution ; puis des résultats expérimentaux sont présentés pour valider l'approche. Enfin, nous relions plus finement notre travail aux approches existantes, discutons nos hypothèses et résumons notre contribution.

2 Modèle

2.1. Tronçonnement du domaine de variables

Le modèle est basé sur un partitionnement de l'arbre de recherche en sous-arbres, un par agent, en partageant une ou plusieurs variables sur ces agents. Les variables partagées sont les *N* premières variables triées par ordre décroissant en fonction de la taille de leur domaine. *N*, le nombre de variables partagées, est *le plus petit entier tel que le produit de la taille des domaines des variables partagées est supérieur au nombre d'agents*.

Par exemple, supposons que l'on ait un problème à 8 variables, chacune ayant un domaine entier, un intervalle de 0 à i, où i est le numéro de la variable (i varie de 1 à 8) ; supposons de plus que l'on veuille décomposer ce problème sur 100 agents. Le produit de la taille des **3** premières variables vaut 9 * 8 * 7 = 504, et est donc supérieur au nombre d'agents (100), alors que le produit de la taille des **2** premières variables vaut 9 * 8 = 72, qui est inférieur au nombre d'agents. En partageant les deux premières variables, le problème pourrait donc être réparti sur 72 agents au maximum. Dans cet exemple, N vaut donc 3.

Mais rappelons que le but est de pouvoir répartir *plus* d'une combinaison de valeurs de ces variables par agent --- dans l'exemple précédent, le premier agent obtiendrait la combinaison {V1 = 0, V2 = 0, V3 = 0}, le second {V1 = 0, V2 = 0, V3 = 1}, etc. Comme le nombre de combinaisons possibles est égal au produit de la taille des domaines des variables partagées, la définition de N s'ensuit

En première approche, on supposera que le nombre d'agents est suffisamment petit pour que $N = 1$ --- en général, le nombre d'agents est inférieur au nombre de valeurs du domaine de la variable à partager, ce qui fait qu'un agent obtient au moins une combinaison de valeurs du domaine de la variable à partager. De plus, si $N > 1$, la méthode de parallélisation ci-dessous (voir Figure 1) serait proche (plusieurs variables partagées en fin de sous-arbre).

Aussi, en nommant *P* le produit de la taille des domaines des *N* variables partagées, on obtient ainsi *P* sous-arbres de recherche, chaque agent devant en explorer au moins un.

Le partitionnement en sous-arbres peut être défini au sein d'un algorithme statique ou dynamique. Dans le partitionnement ***statique***, un agent reçoit au début tous les sous-arbres qu'il doit explorer. Chaque agent explore le même nombre de sous-arbres, à 1 près[1] dans le cas où *P* n'est pas un multiple du nombre d'agent. Mais un agent peut

[1] On a P combinaisons à répartir sur Na agents. Soit Q le quotient de la division de P par Na et R son reste. Les R premiers agents auront donc Q+1 sous-arbres à explorer, alors que les (Na – R) agents suivants n'auront que Q sous-arbres à explorer.

avoir fini d'explorer son dernier sous arbre et rester alors à ne rien faire, alors que les autres agents explorent encore leurs sous arbres. Aussi un partitionnement ***dynamique*** ne donne au début qu'un seul sous-arbre à chaque agent, mais dès que l'agent a fini de l'explorer, il en fournit un autre jusqu'à ce qu'il n'y ait plus de sous-arbres à explorer.

La communication entre agents s'effectue de la façon suivante : dès qu'un agent découvre qu'une combinaison de valeurs conduit par propagation à un domaine vide d'une de ses variables, il envoie cette combinaison aux autres agents pour qu'ils n'explorent pas cette branche --- un message « nogood » est une ensemble d'instanciations (variable, valeur), qui n'appartient à aucune solution (il contient toute l'historique des instanciations effectuées par l'agent jusqu'à la propagation amenant à un domaine vide). Un message « nogood » peut également provenir d'un élagage. Un élagage est la suppression de branche telle que la propagation des contraintes sur les instanciations à la base de cette branche rend vide le domaine d'au moins une variable. Le « nogood » est alors constitué des instanciations à la base de cette branche. Un « nogood » peut donc être la conséquence d'un élagage, où l'algorithme n'explore pas la branche vide, ou d'un retour en arrière, où l'algorithme explore la branche vide et ce faisant, se rend compte qu'elle est vide.

Remarquons que si un tel message contient l'une des variables partagées, les autres agents ne peuvent pas par définition contenir l'instanciation de cette variable (la valeur d'une variable partagée appartient par définition à un seul agent), aussi ce message-là ne sera pas communiqué aux autres agents (filtrage des messages).

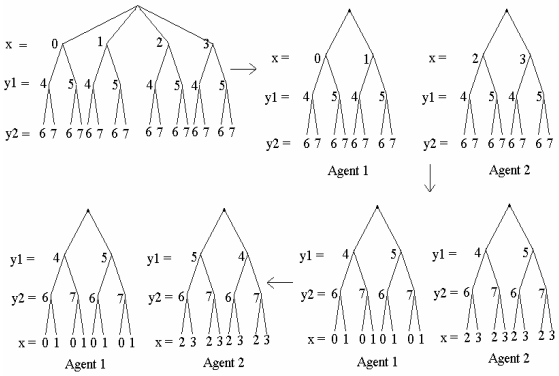

Figure 1 : exemple de décomposition en 2 agents.

L'ordre dans lequel les variables sont considérées par chaque agent importe pour la distribution --- elle est également d'une grande importance pour la rapidité de résolution d'un problème centralisé en général. Aussi une heuristique est utilisée pour ordonner les variables de chaque agent (voir Figure 1) :

1. Déterminer la ou les variables partagées (e.g., x, y_1, y_2 avec x comme variable partagée, ayant un domaine sous forme d'intervalle d'entiers compris entre 0 et 3);
2. Séparer le domaine des variables partagées, et attribuer un tronçon de domaine par agent (e.g., le sous-domaine [0, 1] pour l'agent 1 et le sous-domaine [2, 3] pour l'agent 2) ;
3. Pour chaque sous-agent, placer heuristiquement les variables partagées en dernier --- voir la remarque ci-dessus sur l'instanciation des variables partagées et les messages « nogood ». Par exemple, l'ordre des variables de l'agent 1 est maintenant x_1, y_1 puis x ; et un ordre similaire pour l'agent 2. Les messages « nogood » gardent ainsi une chance de faire des coupes les plus efficaces possibles dans chaque sous-arbre de recherche.
4. Enfin, une heuristique vise à changer l'ordre des valeurs pour les variables de chaque agent, afin d'éviter aux agents de commencer à parcourir le même sous-arbre --- ceci peut être un point d'entrée pour des heuristiques spécifiques à un problème donné, en fonction de la connaissance et de l'intuition qu'en a le programmeur.

2.2. Algorithmes

L'algorithme proposé passe par la communication entre plusieurs agents. Beaucoup d'architecture sont possibles, mais plusieurs arguments militent en faveur d'une architecture client/serveur (i.e., un client est un agent) : (1) un agent communique toujours avec tous les autres, aussi une communication point-à-point serait inutile ; (2) le serveur, indépendant des agents, est bien placé pour partitionner un problème ; (3) le serveur peut vérifier qu'un même message « nogood » n'est pas envoyé deux fois aux agents **---** les variables partagées étant instanciées en dernier (voir Figure 1), il est possible qu'un message « nogood » ne fasse pas intervenir de variables partagées ; dans ce cas, le message « nogood » peut être généré par n'importe quel agent puisqu'il fait référence à la partie commune des sous-problèmes : entre le

temps où un message « nogood » est envoyé par un agent et celui où il est reçu, un autre agent peut avoir généré le même message « nogood ».
Ainsi, le serveur se charge de décomposer le problème en plusieurs sous-problèmes et de stocker cette décomposition. Les clients peuvent alors lui envoyer un message d'initialisation, afin de recevoir la définition de leur nouveau sous-arbre à explorer. Lors de l'exploration d'un sous-arbre par un agent, les messages envoyés au serveur peuvent être soit un message « nogood », soit une nouvelle demande d'initialisation (qui signifie que l'exploration courante est terminée).

Le pseudo-code de l'algorithme du **serveur** est (pour $N = 1$) :

```
1.   Définir le problème (variables,
      domaines, contraintes).
2.  Choisir une variable partagée et
      tronçonner son domaine.
3. WHILE une solution n'a pas été trouvée :
4.    SWITCH message du serveur
5.    CASE initialisation : division du
      problème en sous-problèmes, en
      partageant les domaines des variables
      à partager.
6.    CASE nogood :
7.     SI test(nogood)
       ALORS propager aux autres agents.
8.     CASE solution :
              Envoyer FIN aux autres
             agents.
9.    CASE fin :
10.    SI Card{agents terminés} = total
11.    ALORS FIN_SERVEUR
12.   FIN SWITCH
13. FIN WHILE
```

Dans le cas dynamique, les messages d'initialisation ne sont pas reçus uniquement au début de l'exécution de l'algorithme mais également à chaque fois qu'un agent a terminé d'explorer son sous-arbre de recherche : il sollicite du serveur un nouveau sous-arbre à explorer (cas « initialisation »).
Le cas « initialisation » crée les sous-arbres ; le cas « nogood » teste le message « nogood » et le propage éventuellement aux autres agents ; le cas « fin » est reçu de chaque agent qui a terminé l'exploration de son sous-arbre (si tous les agents envoient le message « fin », le problème est résolu et le serveur peut terminer à son tour, FIN_SERVEUR).
La fonction « test(nogood) » vérifie que (1) le message « nogood » n'a pas déjà été généré par un agent (voir la remarque ci-dessus) et (2) que le message « nogood » est valable pour les autres agents aussi (et pas seulement pour l'agent qui l'a envoyé). En effet, trois cas sont possibles : (i) le cas où la variable partagée intervient explicitement dans le message « nogood » (i.e., « la valeur v_1 ne peut pas être associée à la variable partagée x »). Ce type de message « nogood » n'est pas valable pour les autres agents (qui n'ont pas par définition cette valeur de x puisque son domaine est tronçonné sur les agents) et n'est pas envoyé par l'agent qui est à même de les détecter ; (ii) le cas n'est cette fois pas détectable par l'agent : les domaines des variables partagées interviennent dans les instanciations interdites contenues dans le message « nogood ». Par exemple, dans le problème des 10 reines répartis sur 5 agents, la première variable partagée x_0 ne peut prendre que 2 valeurs, e.g. 0 ou 1. Les instanciations $x_1 = 0$ & $x_2 = 1$, n'appartiennent à aucune solution, puisque x_0 ne peut plus prendre aucune valeur : les contraintes du problème imposent $x_0 != x_1$ et $x_0 != x_2$. Ces deux instanciations feront donc l'objet d'un « nogood », qui ne concerne pas les autres agents puisque le domaine [0, 1] de la variable partagée x_0 n'appartient par définition qu'à cet agent. (iii) le message « nogood » est envoyé à tous les agents, ce qui correspond au cas où le message « nogood » ne dépend pas des variables partagées.
Le pseudo-code de l'algorithme d'un **client** (i.e., un agent) est :

```
1.   Définir le problème (variables,
   domaines, contraintes).
2. Envoyer le message « initialisation » au
   serveur
3.  Attendre le message « InitMsg » du
   serveur, contenant les contraintes
   permettant de définir le sous-problème.
4. WHILE NOT(processus terminé)
5.    Ajouter les nouvelles contraintes
       fournie par « InitMsg ».
6.    RESOLUTION telle que :
7.      Dès qu'un retour en arrière ou un
              élaguage est effectué,
              envoyer un nogood.
8.       Periodiquement, lire les messages
              venant du serveur :
9.       SWITCH message du serveur
10.         CASE Solution : Attendre_fin()
11.         CASE Nogood : Ajouter au
                      problème la
                      contrainte contenue
                      dans le message
12.       FIN SWITCH
13.    FIN RESOLUTION
14.    IF version_dynamique
15.      Envoyer le message «initalisation»
16.      Recevoir un nouveau « InitMsg »
17.      IF « InitMsg » est « Solution »
18.        Achever le processus
19. FIN WHILE
20. Attendre_fin()
```

La définition du problème (ligne 1) est nécessaire pour des questions d'implémentation ---- le moteur de programmation par contraintes utilisé (voir section 4.1.) n'accepte pas des variables supplémentaires en cours de résolution (les couches basses d'Unix que nous utilisons ne permettent pas de passer des instances de classes entre deux

processus, seulement des chaînes de caractères). Par contre, les contraintes supplémentaires (messages « nogood ») peuvent eux être exprimés sous ce format et venir restreindre en cours de résolution le problème initialement posé. Un message d'initialisation contient donc un ensemble de contraintes (ligne 3). Par exemple dans le cas des 10-reines où la variable x_0 est partagée sur 5 agents, les contraintes seront : $0 <= x_0 <= 1$ pour le 1er agent, $2 <= x_0 <= 3$ pour le second, etc.
La fonction « Attendre_fin() » (lignes 10 et 20) consiste à envoyer un message « Terminé» au serveur (i.e., l'agent a terminé d'explorer tous les sous-arbres), et à attendre de recevoir le message « Fin » du serveur (le serveur l'envoie lorsque tous les agents ont terminé, voir ligne 8 de l'algorithme du serveur).

4. Résultats expérimentaux

4.1. Implémentation

Les algorithmes sont implémentés en langage C++, avec l'aide du langage de script TCL[2] et du moteur de programmation par contraintes ILOG SOLVER [16] (une librairie C++). Un agent est un processus Unix contenant une instance de SOLVER, qui explore le sous-arbre qui lui est attribué.
Le serveur est lancé avant les clients. Les clients envoient des messages au serveur, qui les redistribue éventuellement à tous les autres agents.
Le langage de scripts TCL permet aux processus de communiquer via une couche bas niveau (i.e., les *sockets*) et s'interface facilement avec le langage C++ (e.g., un processus client n'a besoin de connaître que le numéro de port du serveur).
Le moteur SOLVER permet de pister des événements lors du fonctionnement du moteur, en définissant des sous-classes de classes prédéfinies, ce qui permet d'implémenter un comportement particulier sur chaque type d'événement. Par exemple, la fonction membre « whenFail() » de la classe « IlcSearchMonitor » permet d'implémenter l'envoi d'un message « nogood » dès qu'un retour-arrière est détecté par le moteur. Ou encore la classe « IloSearchSelector » intervient sur le choix de la variable à instancier, et la classe « IloNodeEvaluator » décide de la valeur à associer à la variable.
Les contraintes sont ajoutées dynamiquement lors d'une résolution par la fonction membre « IloAddConstraint ».

4.2. Domaine

En première approche, les algorithmes statiques et dynamiques sont testés sur un problème, en fonction de la taille des données et du nombre d'agents : le problème des N-reines avec le nombre de reines comme taille des données.
Remarquons que rechercher la première solution revêt un caractère aléatoire en raison de la position de la solution dans l'arbre de recherche --- de par le tronçonnement du domaine des variables partagées et la façon dont les sous-arbres sont construits avec, un agent peut recevoir un sous-arbre dans lequel la première solution peut être tout en haut de l'arbre, donc trouvée très rapidement par l'agent, ce qui est certes un effet de bord intéressant en pratique pour résoudre un problème, mais relève du pur hasard et ne reflète pas à notre sens la qualité de la distribution effectuée. Aussi *toutes* les solutions sont recherchées, pour construire une mesure la plus objective possible des performances.

4.3. Méthodologie

Le temps passé par chaque agent pour résoudre son problème individuel est mesuré --- variable « time » de SOLVER. Comme le temps de résolution du problème est celui de l'agent qui termine en dernier l'exploration de son sous-arbre, le maximum de ces temps de résolution est considéré.
Remarquons que les processus agents n'étaient pas uniques sur la machine de test (nous n'avons pas pu monopoliser une machine pour cette étude), aussi les temps de communication entre agents n'ont pas pu être mesurés ---- il aurait fallu prendre en compte le temps *réel* de résolution pour l'ensemble des agents, ce qui n'était pas représentatif sur notre machine pour la raison mentionnée ci-dessus.
Une moyenne du maximum des temps de résolution est effectuée sur plusieurs lancements.
Etant donné la croissance exponentielle du temps de calcul en fonction de la taille des données, un gain (au sens du domaine de l'Electronique, voir équation 1) est défini, i.e., une valeur relative au temps mis par un seul agent pour résoudre son problème.

$$gain(n) = \frac{t(1)}{n\ t(n)} \qquad (1)$$

Ce gain est le temps mis pour résoudre le problème avec un seul agent $t(1)$, divisé par le nombre n d'agents et par le temps moyen maximum de calcul précédent des agents $t(n)$. Il est donc compris entre 0 et 1, et est d'autant meilleur qu'il est proche de 1.
De plus afin de prendre en compte le surcoût (*overhead*) du à la décomposition en sous-problème, on trace également la courbe du rapport du temps de calcul sans décomposition du problème

[2] Tool Command Langage.

sur le temps mis par un seul agent en ayant quand même fait la décomposition. En effet le fait d'utiliser un seul agent en faisant quand même la décomposition permet de mesurer le surcoût de temps du à la décomposition du problème, sans que le temps de calcul soit réduit par le fait de répartir le calcul sur plusieurs agents. On obtient donc un temps de calcul égal à la somme du temps nécessaire à la décomposition et du temps nécessaire à la résolution sans décomposition. Connaissant ce dernier on en déduit le temps nécessaire à la décomposition.
Pour présenter les résultats expérimentaux, on commencera par calculer le gain en fonction de la taille des données et du nombre d'agents, avec et sans coopération (i.e., avec et sans messages « nogood ») ; puis le rapport entre le temps mis par l'algorithme centralisé (aucun surcoût de distribution) sur le temps mis par un agent (avec surcoût de distribution, même simple). Nous conforterons ensuite nos résultats par une corrélation entre le temps de calcul et le nombre d'agents, puis par une régression linéaire.

4.4. Expériences

Sur la figure 2, on observe une décroissance du gain en fonction du nombre d'agents et une croissance en fonction du nombre de reines.

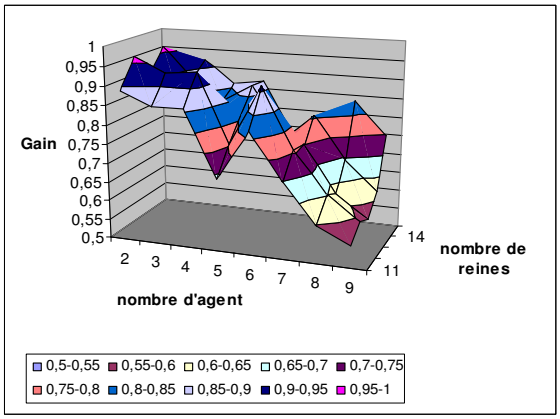

Figure 2 : gain de l'algorithme statique avec coopération.

La figure 3 montre que le rapport du temps de référence (i.e., avec résolution centralisée) sur le temps mis par un agent, c'est-à-dire le coût de la distribution ne dépend pas du nombre de reines (quasi constant autour de la valeur 0.82).

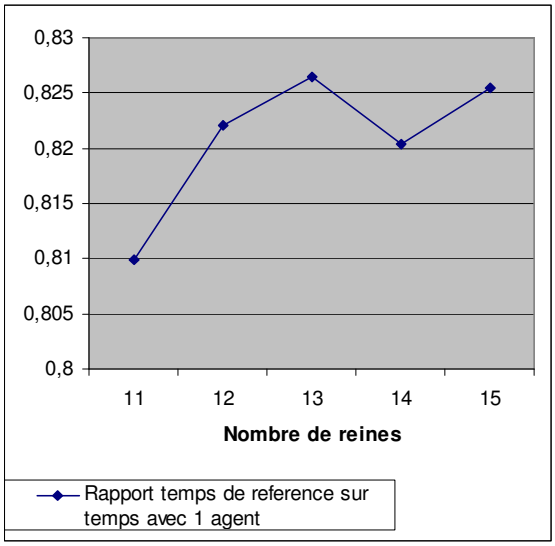

Figure 3 : rapport entre le temps de référence et celui de l'algorithme statique à un agent avec coopération.

Dans la figure 4 on observe que le gain reste cette fois de l'ordre de 1. Malgré quelques pics on retrouve le fait que le gain décroît en fonction du nombre d'agent et croît en fonction du nombre de reines. Cela peut s'expliquer par le fait que le temps du à l'initialisation augmente en fonction du nombre d'agents : plus il y a d'agents, plus le serveur est occupé et mets longtemps à répondre. Par contre ce surcoût ne dépend pas du nombre de reines et est donc relativement plus petit par rapport au temps de calcul que le nombre de reines est grand.

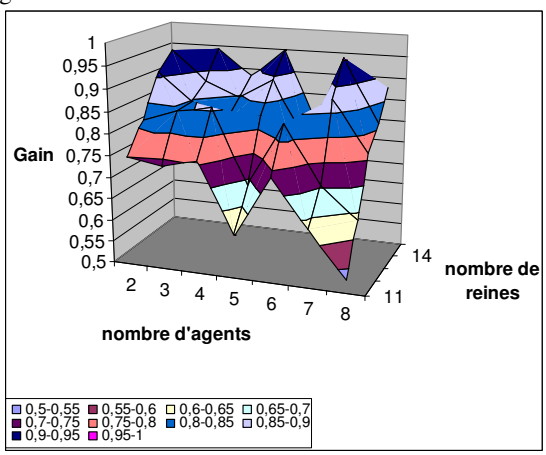

Figure 4 : gain de l'algorithme statique sans coopération.

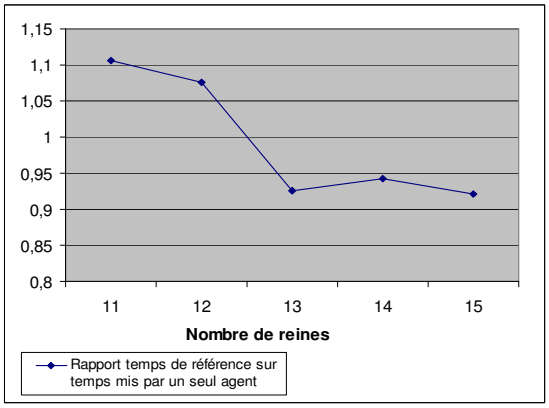

Figure 5 : rapport entre le temps de référence et celui de l'algorithme statique à 1 agent sans coopération.

On observe que sur la figure 5, pour 11 et 12 agents, on obtient avec 1 seul agent un temps de résolution meilleur qu'avec une résolution sans décomposition, ce qui est a priori choquant. Cela peut être expliqué par le fait qu'on modifie les heuristiques gérant l'ordre d'instanciations des variables et des valeurs.

En confrontant les figures 4 (i.e., sans coopération) et 6 (i.e., avec coopération), on observe que les courbes du gain en fonction du nombre de reines ont une pente beaucoup plus élevée que dans la version avec coopération, surtout pour un faible nombre d'agents. Cela peut venir du fait que la coopération fait gagner plus de temps aux gros problèmes qu'aux petits --- ce qui est l'effet souhaité.

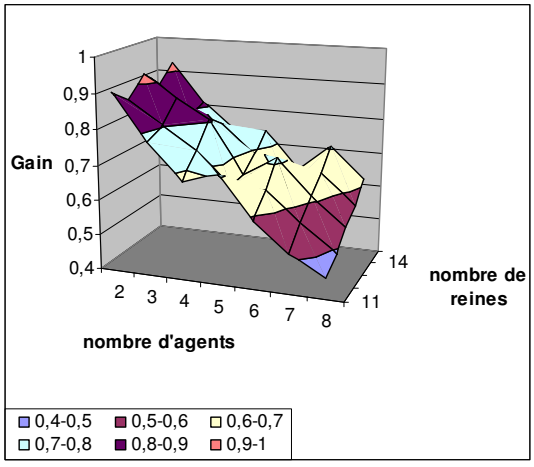

Figure 6 : gain de l'algorithme dynamique avec coopération.

Sur la figure 7, on constate que les écarts entre les valeurs sont très faibles. Ainsi dans ce cas le surcoût du à la répartition ne varie quasiment pas en fonction du nombre de reines, c'est-à-dire en fonction de la taille du problème.

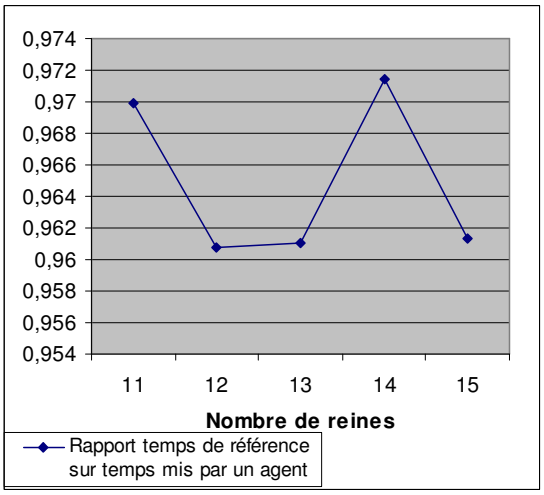

Figure 7 : rapport entre le temps de référence et celui de l'algorithme dynamique à 1 agent avec coopération.

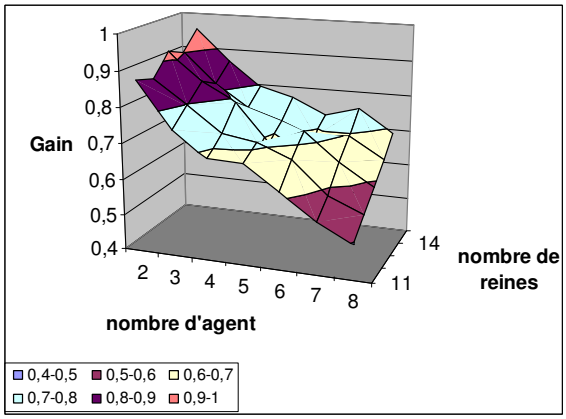

Figure 8 : gain de l'algorithme dynamique sans coopération.

Ces deux dernières surfaces (Figures 6 & 8) se démarquent des autres surfaces de la version dynamique. Elles suivent beaucoup moins les courbes des inverses (i.e., d'équation « $y = 1/x$ » où x est le nombre d'agents).

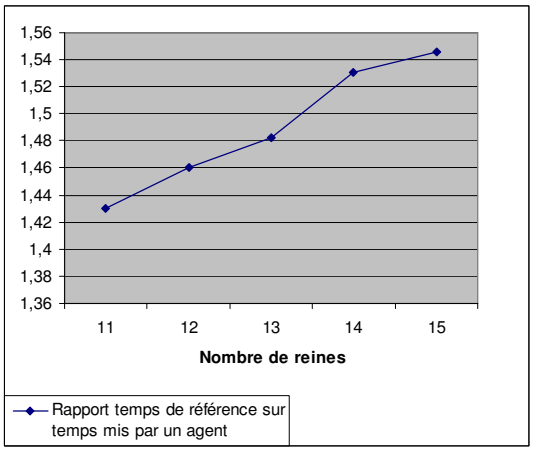

Figure 9 : Rapport entre le temps de référence et celui de l'algorithme dynamique à 1 agent sans coopération.

Sur la figure 9, la courbe du rapport du temps mis par un agent sur le temps de référence est très régulière : Ce rapport augmente quasi-linéairement en fonction du nombre de reines. Ceci montre que les surcoûts sont d'autant plus négligeables que la taille du problème est grande.
La différence entre les résultats obtenus avec et sans coopération est beaucoup plus marquée lorsque le nombre de reines est élevé. Avec coopération, la diminution du gain en fonction du nombre d'agents est beaucoup moins marquée. Ceci s'explique par le fait que la coopération est plus efficace pour les gros problèmes que pour les petits. En effet les nogood y sont plus fréquents et contiennent plus d'informations.

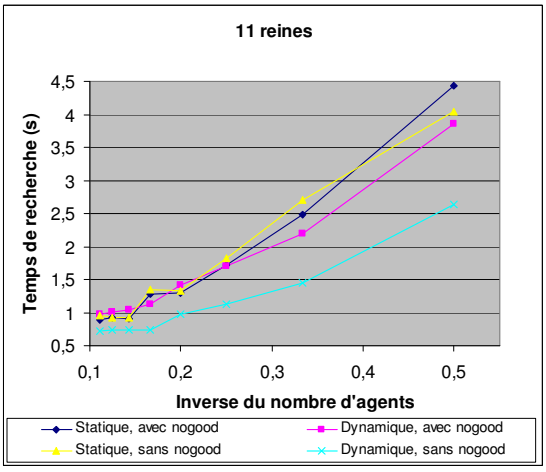

Figure 10 : corrélation entre le temps de calcul et le nombre d'agents (11 reines).

Sur les figures 10 et 11, on observe que la courbe du temps de calcul en fonction de l'inverse du nombre d'agents se rapproche d'une droite, surtout dans le cas statique avec coopération. Ces résultats seront confirmés par la régression linéaire.

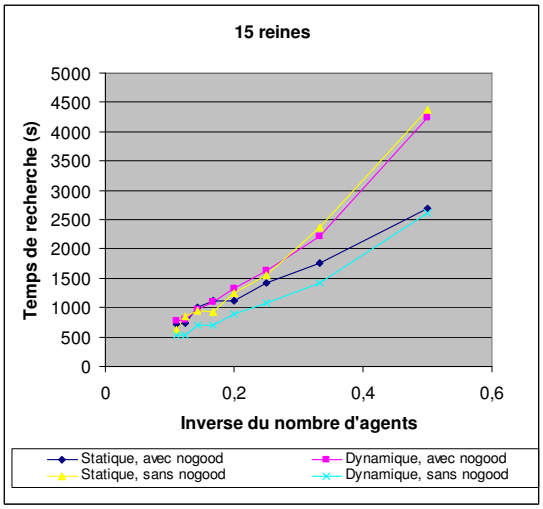

Figure 11 : corrélation entre le temps de calcul et le nombre d'agents (15 reines).

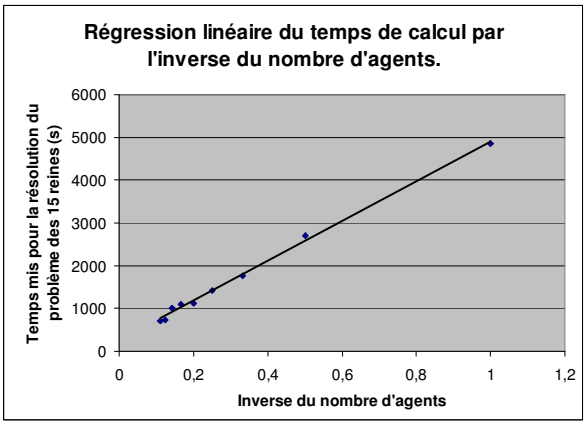

Figure 12 : Régression linéaire du temps de calcul par l'inverse du nombre d'agents.

Sur la figure 12, on effectue une régression linéaire du gain en fonction de l'inverse du nombre d'agent (voir équation 2) :

$$t(n) = t(1)\,(\frac{a'}{n} + b') + e(n,r) \qquad (2)$$

Le temps de calcul *t(n)* de *n* agents dépend de l'inverse du nombre *n* d'agents (coefficients *a'* et *b'*), avec comme coefficient le temps de calcul pour 1 seul agent *t(1)*, et avec une erreur *e* dépendant du nombre *n* d'agents et du nombre *r* de reines. L'erreur *e* peut être expliquée par le fait que l'élagage des branches de l'arbre de recherche est d'autant plus efficace qu'il y a peu de valeurs dans

le domaine des variables. En effet dès que les contraintes interdiront toutes les valeurs sauf une d'une variable, la variable sera affectée à la valeur restante.
Par exemple, on se place dans le cas du problème des 10-reines avec 5 agents. Chaque agent n'aura plus que 2 valeurs, v_1 et v_2, pour l'une des variables qu'on nommera x. Dès qu'une autre variable prendra la valeur v_1 (ou v_2), x sera affecté à v_2 (ou v_1). L'élagage est donc plus efficace que dans le problème non décomposé, ou il aurait fallu que 9 valeurs sur 10 du domaine de x soient utilisées avant de pouvoir en déduire l'instanciation de x.
Dans l'equation (2), pour un nombre r de reines donnés, $t(n)$ est donc égal à la somme de trois termes : (i) $t(1)\ a'/n$, qui correspond au temps d'exploration pour 1 agent, divisé par le nombre d'agents (le temps de recherche est proportionnel au nombre de nœuds dans l'arbre de recherche, qui est divisé par le nombre d'agents lors de la répartition) ; (ii) $t(1)\ b'$, qui correspond aux surcoûts liés à la répartition et à la gestion des messages « nogood » ; (iii) une erreur e indépendante. La régression linéaire de l'équation 1 sur la courbe de la figure 12 est d'excellente qualité --- le coefficient statistique r2 est autour de 0.95.

5. Travaux connexes et discussion

[1] propose de réduire l'espace des solutions, en utilisant les propriétés de k-cohérence. Chaque agent prend en entrée un domaine d'une variable et un ensemble de contraintes sur cette variable, et rend un domaine dans lequel des ensembles de valeurs ont été supprimés. Cette méthode offre une efficacité exponentielle avec un coût polynomial. Le gain exponentiel est donc nettement meilleur que le gain linéaire que nous obtenons, alors que le coût, c'est-à-dire la complexité de la méthode de décomposition, est polynomial, là où nous obtenons un coût linéaire --- meilleur cas d'un coût polynomial. En effet, dans le pire des cas, cette méthode se compose du tri des variables par ordre de taille de domaines (complexité n), de n produits des tailles des domaines des k premières variables (complexité n), et d'une division euclidienne. On reste donc dans une complexité de l'ordre de n (où n est le nombre de variables).
Dans la méthode d'émergence d'une solution à partir de solutions partielles [9], **c**haque agent correspond à une contrainte, et est chargé de trouver toutes les instanciations de son ensemble de variables (i.e., des variables qui interviennent dans sa contrainte) vérifiant sa contrainte. Une fois trouvée, les contraintes sont fusionnées deux à deux jusqu'à obtenir l'ensemble des solutions vérifiant toutes les contraintes. Cet algorithme est certes systématique mais nécessite de grosses capacités de mémoire, afin de stocker tous les ensembles de solutions intermédiaires de toutes les contraintes. De plus, contrairement au nôtre, cet algorithme n'utilise pas les réductions de temps qu'apporte l'élagage des branches mortes.
Dans la méthode de décomposition de l'espace de recherche [2], deux sortes d'agents sont utilisés : un « planificateur » et des « explorateurs ». Les « explorateurs » explorent l'espace de recherche qui leur est attribué par le « planificateur ». Lorsqu'un agent est inoccupé, le planificateur demande à un agent occupé de partager son espace de recherche. Cette approche peut être comparée à la version dynamique de la nôtre, à la différence que dans notre algorithme, le serveur alloue les sous-problèmes de façon dynamique, alors que dans la méthode décrite ci-dessus, les agents doivent répartir le problème en sous-problèmes au fur et à mesure.
Enfin, notre algorithme utilise des heuristiques pour construire les sous-arbres de recherche d'un agent (voir section 2). Ces heuristiques semblent naturelles et fournissent effectivement de bons résultats (voir section 3), mais, comme toute heuristique, peuvent être remises en cause en fonction du problème à résoudre --- à ce titre, le point d'entrée d'heuristiques liées au domaine (point 4 de la décomposition en agents de la section 2) peut être exploité pour tester d'autres types d'heuristiques (e.g., les laisser libres au programmeur en fonction de son intuition sur le domaine).

6. Conclusion et travaux futurs

Dans cet article, nous avons présenté une méthode simple et efficace pour décomposer un arbre de recherche en sous-arbres, basée sur le tronçonnement du domaine de variables partagées. Nous avons proposé une version statique (i.e., sous-arbres fixés au départ) et une version dynamique (i.e., sous-arbres générés dynamiquement) d'un algorithme pour le serveur comme pour les agents/clients utilisant cette décomposition en sous-arbres. Nous avons ensuite présenté un mode de communication entre agents basé sur la notion de combinaison d'instanciations interdites (variable, valeur), appellé messages « nogood », lorsqu'un retour-arrière est détecté par un agent, évitant aux autres agents de parcourir cette branche morte. Nous avons implémenté notre approche, et avons présenté des résultats expérimentaux sur un problème test. Ces résultats mettent en évidence une excellente corrélation entre l'inverse du nombre

d'agents et le temps maximal par agent de recherche de toutes les solutions. Il était intuitif que la présence de messages « nogood » améliore les temps de calcul, puisqu'ils réduisent l'arbre de recherche. Les expériences montrent en plus que cela n'induit pas un surcoût.

Nos directions de recherche futures comprennent :

1. Tester ce modèle de distribution sur le maximum d'exemples possibles (par exemple, la vingtaine d'exemples de la distribution de SOLVER), afin d'encore mieux valider expérimentalement l'approche. L'idéal serait de couvrir toutes les « formes » de domaines possibles, et étudier celles qui se prêtent bien à ce modèle de distribution et les autres (ce qui nous amènerait à modifier notre modèle). Ceci peut s'appréhender par exemple en utilisant toutes[3] les contraintes du moteur de propagation, une à une puis de façon croisée, ou même de façon aléatoire à la façon d'un problème SAT.
2. Tenir compte du temps de communication entre agents. Il y a en effet beaucoup de messages qui transitent entre les agents via le serveur, et mesurer les performances par les temps de calcul individuels de chaque agent n'est probablement pas suffisant. Le temps de résolution individuel décroît en fonction du nombre d'agents, alors que le volume de communication croît en fonction du nombre d'agents : il est vraisemblable qu'en tenant compte du temps de communication entre agents, on constaterait la présence d'un optimum.
3. Enfin, en pratique, embarquer ce mode de distribution sur l'application concernant le calcul de chemin d'avions de chasse autonomes [18].

Bibliographie

[1] K. R. Apt. The essence of constraint propagation. *Theoretical Computer Science*, 221(1-2):179--210, 1999.

[2] F. Arbab, E. Monfroy. Distributed splitting of constraint satisfaction problem. *SEN-R0027*, 2000.

[3] Z. Habbas, M. Krajecki, D. Singer. Domain decomposition for parallel resolution of constraint satisfaction problem with OpenMP. *Proceedings of the second european workshop on OpenMP (EWOMP2000)* , Edinburgh, UK, ppages 1-8, 2000

[4] Y. Hamadi. Interleaved backtracking in distributed constraint network. *International Journal on Artificial Intelligence Tools* Vol 11, No 2 (2002), pages167-188]

[5] Y. Hamadi, C. Bessiere, J. Quinqueton. Backtracking in distributed constraint network. In *Proceedings of the 13th European Conference on Artificial Intelligence*. In ECAI-98, H. Prade ed., 1998, pages 219-223.

[6] T. Hogg, B. A. Hubermann. Better than the best, the power of cooperation. *Lectures in Complex Systems*, 1992, pages 165-184

[7] R. Karlsson. A high performance OR-parallel Prolog system. *PhD thesis, Department of Telecommunication and Computer Systems, The Royal Institute of Technology*, Stockholm, 1992.

[8] J.-L. Laurière. Un langage et un programme pour énoncer et résoudre des problèmes combinatoires. Thèse de Doctorat d'Etat, Université Paris 6, France, mai 1976, 227 pages.

[9] R. Mailler, V. Lesser. A mediation base protocol for distributed constraint satisfaction. *The Fourth International Workshop on Distributed Constraint Reasoning*, 2003, pages 49-58.

[10] E. Monfroy. A coordination-based chaotic iteration algorithm for constraint propagation. *Proceedings of the 2000 ACM symposium on Applied computing,* 2000, pages 262-269.

[11] E. Monfroy, J.-H. Rety. Itération asynchrone, un cadre uniforme pour la propagation de contraintes parallèles et réparties. In *Actes des JFPL*, Hermès, 1999, pages 123-137.

[12] P. Morignot, J.-C. Poncet, J. Baltié, P. Fabiani, E. Bensana, J.-.L Farges, B. Patin. Simulating Uninhabited Combat Aircraft in Hostile Environment (Part II). In *Proceedings of the European Simulation Interoperability Workshop* (EURO SISO SIW'05), Toulouse, France, 27-29 juin 2005, 9 pages, référence 05E-SIW-39 sur le CD-ROM. A paraître.

[13] M. Platzer, B. Rinner. Design and implementation of a parallel constraint satisfaction algorithm. *International Journal of Computers and Their Applications*, 1998, pages 106-116.

[14] M. Yokoo, K. Hirayama. Distributed breakout algorithm for solving distributed constraint satisfaction problems. *Proceedings of the Second International Conference on Multi-agent Systems*, 1996.

[15] M. Yokoo, T. Ishida. Search Algorithm for agents. *Multi-Agents Systems*, G. Weiss ed., MIT Press, 1999, pages 165-199.

[16] SOLVER Reference Manual. ILOG, Gentilly, France, 2004.

[3] SOLVER offre la possibilité d'encoder ses propres contraintes dans le langage d'implémentation (C++), ce qui n'est évidemment pas possible de couvrir.