Itérations Asynchrones : un Cadre Uniforme pour la Propagation de Contraintes Parallèle et Répartie

Eric Monfroy – Jean-Hugues Réty[1]

Résumé : Nous proposons dans cet article un cadre général pour la propagation de contraintes basé sur la notion d'itération asynchrone. Nous en dérivons deux algorithmes parallèles de propagation de contraintes fondés sur des paradigmes de programmation différents : l'un à mémoire partagée et l'autre à mémoire répartie. Chacun de ces algorithmes permet d'utiliser une architecture matérielle particulière afin de réaliser une implantation bénéficiant de ses spécificités.

Mots-clés : Résolution de Contraintes, Propagation de Contraintes, Itération Asynchrone, Algorithme Parallèle, Algorithme Réparti

1. Introduction

La propagation de contraintes est l'une des techniques les plus utilisées pour résoudre des problèmes de satisfaction de contraintes (CSP). Elle procède généralement par le calcul d'une propriété de consistance locale grâce à l'application répétée de fonctions de réduction, jusqu'à obtention d'un point fixe. De nombreux algorithmes, soit adaptés à des notions particulières de consistances locales [MAC 77], à des stratégies de réduction spécialisées [MOH 86], à des domaines de contraintes particuliers [VAN 97], à des architectures spécifiques [NGU 98], ou proposant des cadres plus généraux [BEN 97], ont été proposés dans la littérature. Plus récemment, un cadre unificateur fondé sur la notion d'itération chaotique a été défini par K.R. Apt [APT 97, APT 98] : la

[1]CWI, Kruislaan 413, 1098 SJ Amsterdam, The Netherlands
Email : {eric,rety}@cwi.nl, Web : http://www.cwi.nl/{~eric, ~rety}

plupart des algorithmes de propagation de contraintes sont des instances d'un algorithme général appelé *general chaotic iteration algorithm* (GCI). Notons que ce cadre se consacre uniquement à la propagation de contraintes proprement dite, c'est-à-dire en isolation de tout autre mécanisme lié à la résolution de contraintes (comme par exemple, le *labelling*, ou la recherche avec retour-arrière).

Les itérations chaotiques ont été introduites dans [CHA 69] par Chazan et Miranker afin de résoudre des systèmes d'équations, puis reformulées par Cousot [COU 78] et Wei [WEI 94]. Cette notion d'itérations chaotiques est le plus souvent accompagnée de deux notions plus générales appelées *itérations asynchrones* et *itérations asynchrones avec mémoires* dont l'objectif est de permettre une implantation parallèle sur système à multi-processeurs [BAU 78].

Nous étudions dans cet article l'application de méthodes d'itérations asynchrones à la propagation de contraintes. Ce qui nous permet ensuite d'étudier deux algorithmes parallèles pour la propagation de contraintes : l'un dédié à des ordinateurs à mémoire partagée (appelé par la suite algorithme parallèle), l'autre dédié à des ordinateurs à mémoire répartie (algorithme réparti). Le cadre de la propagation de contraintes s'avérant être un peu plus complexe que celui considéré traditionnellement par les études concernant les méthodes d'itérations, nous devrons tout d'abord redéfinir la notion d'itération asynchrone de façon plus générale. Ensuite, nous prouverons les théorèmes de convergences liés à cette notion généralisée d'itérations asynchrones. Nous étendons ainsi les résultats de [APT 97, APT 98] concernant les itérations chaotiques aux itérations asynchrones.

Ce cadre pour la propagation de contraintes nous permet ensuite de déduire deux algorithmes basés sur des paradigmes de programmation différents. Le premier, fondé sur la programmation répartie, nécessite les notions de processus indépendants et d'échange de données entre ces processus via des canaux de communication. Le second, un algorithme parallèle, requiert le concept de mémoire partagée et des mécanismes d'exclusion mutuelle. Chacun de ces algorithmes peut donner lieu à une implantation sur une architecture matérielle spécifique : par exemple, réseau de stations de travail pour le premier, et machine de type SMP (Symmetrical Multi-Processor) pour le second. Dans [MON 99], une implantation de l'algorithme réparti que nous présentons dans cet article a été décrite. Ce prototype illustre l'intérêt qu'offre le cadre que nous proposons dans cet article, en matière d'efficacité, de fonctionalités, et de thèmes de recherche futurs.

Nous formalisons en Section 2 le cadre de propagation de contraintes que nous considérons. La Section 3 définit la notion d'itération asynchrone pour la propagation de contraintes. Un algorithme réparti est étudié en Section 4.1, ainsi qu'un algorithme parallèle en Section 4.2. Ces deux algorithmes sont prouvés corrects. Section 5, nous présentons des directions de recherche future.

2. Préliminaires

Notre formalisation de la propagation de contraintes est largement inspirée de celle de [APT 97]. Nous appelons *problème de satisfaction de contraintes* (CSP) un ensemble de contraintes portant sur des variables (conditions requises sur des variables) ainsi que des domaines associés à ces mêmes variables (l'ensemble des valeurs possibles de ces variables). Une contrainte est définie comme étant un sous ensemble du produit cartésien des valeurs des domaines des variables. Les domaines de variables sont des éléments d'une structure algébrique appelée *ordre partiel conjonctif.*

A partir d'un CSP, des *fonctions de réduction* sont générées en association avec les contraintes et les variables qui y apparaissent. Il existe deux types de fonctions de réduction : les fonctions de réduction de domaines dont l'application à pour but de réduire des domaines de variables, et les fonctions de réduction de contraintes qui génèrent de nouvelles contraintes à partir desquelles de nouvelles informations peuvent être réduites. Nous nous limiterons dans cet article aux fonctions de réduction de domaines [2] (drf).

La propagation de contraintes consiste à appliquer sur les CSPs des fonctions de réduction afin de transformer un CSP en un CSP équivalent (même ensemble de solutions), mais "plus simple". Cette notion dépend du domaine de calcul. Ainsi, pour des contraintes entières linéaires, "plus simple" peut correspondre à des domaines contenant moins d'éléments, alors que pour des contraintes sur les nombres réels, "plus simple" peut être associé à des intervalles plus petits. Le processus de propagation de contraintes procède de manière incrémentale et calcule ainsi des limites d'itérations d'ensembles finis de fonctions de réduction. Nous définissons maintenant ces notions de façon plus formelle.

Définition 2.1 *Un ordre partiel $\langle D, \sqsubseteq \rangle$ est un $\sqcup$-po (ordre partiel conjonctif) s'il vérifie :*

- *$\langle D, \sqsubseteq \rangle$ possède un plus petit élément $\perp \in D$,*
- *pour toute suite croissante $d_0 \sqsubseteq d_1 \sqsubseteq d_2 ...$ d'éléments de D, le plus petit majorant $\sqcup_{i \geq 0} d_i$ existe,*
- *et pour tout $a, b \in D$, le plus petit majorant $a \sqcup b$ existe.*

Un ordre partiel $\langle D, \sqsubseteq \rangle$ vérifie la *propriété de chaînes finies* si toutes les suites croissantes $d_0 \sqsubseteq d_1 \sqsubseteq d_2 ...$ d'éléments de D vérifient : $\exists j \geq 0$ tel que $\forall i \geq j, d_i = d_j$. Clairement, le produit cartésien d'une suite finie de $\sqcup$-po est lui-même un $\sqcup$-po, et le produit cartésien d'une suite finie de $\sqcup$-po vérifiant la propriété de chaînes finies est un $\sqcup$-po vérifiant la propriété de chaînes finies.

[2] Les fonctions de réduction de domaines sont appelées fonctions de narrowing dans [BEN 97].

Considérons une suite finie V de variables deux à deux distinctes et une suite de $\sqcup$-po $(\langle D_v, \sqsubseteq_v \rangle)_{v \in V}$. Nous notons $\langle D, \sqsubseteq \rangle$ le produit cartésien des éléments de la suite $(\langle D_v, \sqsubseteq_v \rangle)_{v \in V}$.

Définition 2.2 *Un scheme sur V est une suite d'éléments de V deux à deux distincts.*

Pour des raisons de commodité, nous utiliserons parfois des prédicats ensemblistes sur des schemes. En particulier, si s, s' sont des schemes, $s \subseteq s'$ signifie que tous les éléments de s sont également dans s'.

Soit s un scheme sur V, nous notons $\langle D_s, \sqsubseteq_s \rangle$ le produit cartésien des $\sqcup$-po $(\langle D_v, \sqsubseteq_v \rangle)_{v \in s}$. Nous notons $\perp_s$ le plus petit élément de D_s. Quand cela ne générera pas d'ambiguïté, nous omettrons l'indice s de $\sqsubseteq_s$.

Nous aurons aussi besoin d'un opérateur de projection. Soit $d \in D_s$, s étant un scheme sur V. Pour chaque variable $v \in s$, $d[v] \in D_v$ est la projection de d sur v. Nous étendons les opérateurs de projection aux schemes de la manière suivante : Soient s, s' deux schemes sur V, et soit $d \in D_s$, $d[s']$ est l'élément de $D_{s'}$ vérifiant $\forall v \in s'$, $d[s'][v] = d[v]$ si $v \in s$, et $d[s'][v] = \perp_v$ si $v \notin s$.

Définition 2.3 *Une fonction de réduction de domaines (drf) sur $(V, \langle D, \sqsubseteq \rangle)$ est un triplet (is, os, f) où is et os sont des schemes sur V (respectivement le scheme d'entrée et le scheme de sortie de f) et f est une fonction monotone $f : D_{is} \to D_{os}$.*
Nous noterons parfois (is, os, f) par f. Si $d \notin D_{is}$ alors $f(d)$ dénote $f(d[is])$.

Définition 2.4 *Un problème est un triplet $(V, \langle D, \sqsubseteq \rangle, F)$ où*

- *V est une suite finie de variables deux à deux distinctes,*
- *$\langle D, \sqsubseteq \rangle$ est le produit cartésien d'une suite de $\sqcup$-po $(\langle D_v, \sqsubseteq_v \rangle)_{v \in V}$,*
- *et F est un ensemble fini de drfs sur $(V, \langle D, \sqsubseteq \rangle)$.*

Cette notion de problème formalise la notion standard de CSP à un niveau plus abstrait, dédié à la description de la propagation de contraintes par l'intermédiaire de drfs. Considérons un CSP $P = \langle Dom_1, .., Dom_n; C \rangle$ où C est un ensemble de contraintes, et les Dom_i sont les domaines des variables qui y apparaissent. Pour $i \in [1, n]$, considérons l'ordre partiel $\sqsubseteq_i$ défini par : $\forall X, Y \subseteq Dom_i, X \sqsubseteq_i Y$ ssi $X \supseteq Y$. Alors nous avons : pour tout $i \in [1, n]$, $\langle P(Dom_i), \sqsubseteq_i \rangle$ est un $\sqcup$-po avec $\perp_i = Dom_i$ et $X \sqcup_i Y = X \cap Y$.

Un ensemble de fonctions de réduction de domaines est dérivé de l'ensemble de contraintes en regard des domaines en question et du type de propagation à effectuer. L'ensemble de contraintes lui-même n'est alors plus nécessaire, la propagation de contraintes procédant par application des drfs sur les domaines.

Exemple 2.1 *Considérons par exemple la consistance d'arc [MAC 77] sur une contrainte binaire $c \subseteq Dom_{v1} \times Dom_{v2}$. Les deux drfs suivantes sont suffisantes* *pour obtenir la consistance d'arc :* $drf_1 = (v1.v2, v1, f_1(X,Y) = X')$*, où* $X' = \{a \in X \mid \exists b \in Y, (a,b) \in c\}$*, et* $drf_2 = (v1.v2, v2, f_2(X,Y) = Y')$*, où* $Y' = \{b \in Y \mid \exists a \in X, (a,b) \in c\}$*. Nous avons donc le problème* $(v_1.v_2, \langle P(Dom_{v_1}), \supseteq\rangle \times \langle P(Dom_{v_2}), \supseteq\rangle, \{drf_1, drf_2\})$*. La consistance d'arc est obtenue par application de ces deux drfs jusqu'à ce qu'un point fixe soit atteint.*

3. Itérations Asynchrones

Notre définition de la notion d'itération asynchrone est inspirée de celle de Cousot [COU 78] mais est adaptée au contexte de la propagation de contraintes. Dans [COU 78], le "problème" à résoudre est spécifié par une unique fonction monotone $f : D^n \to D^n$, D étant un treillis complet. Le calcul s'effectue par itération de l'application des projections $f_i : D^n \to D$, pour $i \in [1,n]$. Nous considérons à la place un ensemble fini de drfs monotones sur une structure $(V, \langle D, \sqsubseteq\rangle)$ où $\forall v \in V$, D_v est un $\sqcup$-po. Le cadre de [COU 78] est ainsi une restriction du nôtre dans lequel :

1. les domaines sont homogènes,
2. les schemes de sortie sont réduits à une seule variable et il y a exactement n drfs (autant que de variables), chacune ayant une variable différente pour scheme de sortie.

L'Item 1 signifie que nous étendons le cadre à la propagation de contraintes sur plusieurs domaines. L'Item 2 a pour conséquence de nous obliger à modifier la définition de la notion d'itération asynchrone elle-même. Dans [COU 78], le résultat de convergence d'itérations asynchrones repose sur la propriété qu'une seule projection de la fonction f peut produire de l'information nouvelle sur une variable donnée. Afin d'obtenir malgré cela la propriété de convergence souhaitée, nous allons devoir lors de l'application d'une drf considérer le plus petit majorant de la valeur nouvellement calculée et de l'anciennne valeur (Item 6, ci-dessous), sans quoi l'itération asynchrone pourrait diverger.

Définition 3.1 *Soit* $P = (V, \langle D, \sqsubseteq\rangle, F)$ *un problème.*

Considérons une suite de drfs $f_1, f_2, \ldots$ *vérifiant :*

1. $\forall i \in N, f_i \in F$
2. $\forall i \in N, \forall f \in F, \exists j \geq i, f_j = f$

Nous notons is_i *(respectivement* os_i*) le scheme d'entrée (respectivement le scheme de sortie) de* f_i*.*

Considérons une suite $S_1, S_2, \ldots$ *vérifiant :*

3. $\forall i \in N, S_i$ *est une famille d'entiers naturels indexés par* is_i*. (Nous notons* S_i^v *l'élément de* S_i *associé à la variable* $v \in is_i$*).*

4. $\forall i \in N, \forall v \in is_i, S_i^v < i$

5. $\forall i \in N, \forall v \in V, \exists j \geq i, \forall k \geq j, (v \in is_k \Rightarrow i \leq S_k^v)$

L'itération asynchrone de P définie par $a_0 \in D$ et les suites $f_1, f_2, \ldots,$ et $S_1, S_2, \ldots$ est la suite $a_0, a_1, a_2, \ldots$ vérifiant :

6. $\forall i \geq 1$, *supposons* $is_i = v_1, \ldots, v_p$, $a_i = a_{i-1} \sqcup f_i(a_{S_i^{v_1}}[v_1], .., a_{S_i^{v_p}}[v_p])[V]$

La différence essentielle entre une itération asynchrone et une itération chaotique réside dans le fait que les drfs peuvent ici être calculées en utilisant d'anciennes valeurs des domaines. En effet, les valeurs utilisées peuvent provenir d'itérations précédentes, ceci en fonction des indices d'itération spécifiés par la suite $S_1, S_2, \ldots$ La monotonicité des drfs est ici primordiale : l'information nouvellement calculée peut être transmise à n'importe quel moment, en temps fini, mais aucune synchronisation n'est nécessaire (d'où le nom d'*itération asynchrone*). Nous énonçons maintenant la propriété de convergence des itérations asynchrones.

Proposition 3.1 *Soit $P = (V, \langle D, \sqsubseteq \rangle, F)$ un problème.*
Soit $h : D \to D$ la fonction définie comme suit :

$$h(x) := \bigsqcup_{f \in F} (f(x)[V] \sqcup x)$$

Considérons la suite d'éléments de D définie par :

$$h \uparrow 0 = a_0, \ \forall i \geq 0, h \uparrow (i+1) = h(h \uparrow i)$$

Toute itération asynchrone de P possède une limite, et celle-ci est égale à

$$\bigsqcup_{i \geq 0} h \uparrow i$$

Preuve 3.1 *Remarquons tout d'abord que la fonction h est extensive, i.e., $\forall d \in D$, $d \sqsubseteq h(d)$. $\langle D, \sqsubseteq \rangle$ étant un $\sqcup$-po, la limite $\bigsqcup_{i \geq 0} h \uparrow i$ existe donc.*
De même, toute itération asynchrone $a_0, a_1, \ldots$ est monotone, i.e., $\forall i \geq 0, a_i \sqsubseteq a_{i+1}$ (de par l'Item 6 de la Définition 3.1). Donc, sa limite existe.
Le résultat découle directement des deux lemmes ci-dessous.

Lemme 3.1 $\forall i \geq 0, \exists j \geq 0$ *tel que* $h \uparrow i \sqsubseteq a_j$

La preuve est par induction sur i :

Base : *Immédiat car nous avons :* $h \uparrow 0 = a_0$

Pas d'induction : *Considérons* $i, j \geq 0$ *tels que* $h \uparrow i \sqsubseteq a_j$
De par l'Item 5 de la Définition 3.1, nous avons :
$\exists k \geq j$ *tel que* $\forall v \in V, \forall m \geq k, (v \in is_m \Rightarrow j \leq S_m^v)$
Considérons un tel k.

De par l'Item 2 de la Définition 3.1, nous avons :
$\exists l \geq k$ *tel que* $\forall f \in F, \exists m, k \leq m \leq l$ *et* $f_m = f$
Considérons un tel l.
Soit $f \in F$. *Considérons* m *tel que* $k \leq m \leq l$ *et* $f_m = f$. *Nous avons :*
$a_m = a_{m-1} \sqcup f(d)[V]$ *avec* $\forall v \in is_f, d[v] = a_{S_m^v}[v]$ *et* $\forall v \in is_f, S_m^v \geq j$
Alors, parce que l'itération asynchrone $a_0, a_1, a_2, ...$ *est monotone, nous avons :* $a_j[is_f] \sqsubseteq d$.
D'où, f *étant monotone,* $f(a_j)[V] \sqcup a_j \sqsubseteq a_m$ *et par conséquent :*
$f(a_j)[V] \sqcup a_j \sqsubseteq a_l$.
Ce résultat est vrai pour tout $f \in F$, *donc :* $\bigsqcup_{f \in F} f(a_j)[V] \sqcup a_j \sqsubseteq a_l$
D'autre part, h *étant monotone, nous avons :*

$$h \uparrow (i+1) = h(h \uparrow i) \sqsubseteq h(a_j) = \bigsqcup_{f \in F} f(a_j)[V] \sqcup a_j$$

et finalement : $h \uparrow (i+1) \sqsubseteq a_l$.

Lemme 3.2 $\forall i \geq 0, a_i \sqsubseteq h \uparrow i$

La preuve est par induction sur i *:*

Base : *Immédiat car nous avons :* $h \uparrow 0 = a_0$
Pas d'induction : *Considérons* $i \geq 0$ *tel que* $a_i \sqsubseteq h \uparrow i$
Nous avons : $a_{i+1} = a_i \sqcup f(d)[V]$ *pour un certain* $f \in F$ *avec* $\forall v \in is_f, d[v] = a_{S_{i+1}^v}[v]$ *et* $S_{i+1}^v \leq i$.
Donc, par monotonicité de l'itération asynchrone, $d \sqsubseteq a_i[is_f]$, *et par monotonicité de* f, $f(d) \sqsubseteq f(a_i[is_f]) \sqsubseteq f(h \uparrow i[is_f])$.
D'où, $a_i \sqcup f(d)[V] \sqsubseteq h \uparrow i \sqcup f(h \uparrow i[is_f])$
Et finalement : $a_{i+1} \sqsubseteq h(h \uparrow i)$

Remarquons que si, de plus, $\langle D, \sqsubseteq \rangle$ est un treillis complet et a_0 est un pré-point-fixe de la fonction h, alors la limite des itérations asynchrones pour P et a_0 est le plus petit point fixe de h plus grand ou égal à a_0.

Nous dirons qu'une itération asynchrone $a_0, a_1, a_2, \ldots$ est *stabilisante* si elle vérifie : $\exists k, \forall i \geq k, a_i = a_k$.

Proposition 3.2 *Soit* $P = (V, \langle D, \sqsubseteq \rangle, F)$ *un problème. Si tous les domaines* $\langle D_v, \sqsubseteq_v \rangle$, $v \in V$ *vérifient la propriété de chaîne finie alors toutes les itérations asynchrones de* P *sont stabilisantes.*

Preuve 3.2 *Dans ce cas,* $\langle D, \sqsubseteq \rangle$ *vérifie la propriété de chaîne finie. D'autre part, chaque itération asynchrone forme une chaîne ascendante sur* $\langle D, \sqsubseteq \rangle$ *(Item 6 de la Définition 3.1). Aussi, en supposant qu'il existe une itération asynchrone* $a_0, a_1, a_2, \ldots$ *qui ne soit pas stabilisante, alors* $a_0, a_1, a_2, \ldots$ *est une* *chaîne infinie strictement croissante, ce qui contredit le fait que $\langle D, \sqsubseteq \rangle$ vérifie la propriété de chaîne finie.*

Proposition 3.3 *Soit P un problème. S'il existe une itération asynchrone stabilisante de P alors toutes les itérations asynchrones de P sont stabilisantes.*

Preuve 3.3 *Supposons qu'il existe une itération asynchrone stabilisante $a_0, a_1, a_2, \ldots$ de P. Soit a_k sa limite (k vérifie : $\forall i \geq k, a_i = a_k$). Alors, par le Lemme 3.2 et la Proposition 3.1, nous avons, en conservant les notations du Lemme 3.1 : $h \uparrow k = a_k$.*
Maintenant, considérons une autre itération asynchrone $b_0, b_1, b_2, \ldots$ Par le Lemme 3.1 et la Proposition 3.1, nous avons : $\exists l \geq 0, \forall i \geq l, b_i = h \uparrow k$.
Ainsi, $b_0, b_1, b_2, \ldots$ est stabilisante.

Nous définissons ci-dessous une notion d'*itération quasi-asynchrone*. Cette notion peut-être rapprochée de celle d'itération semi-chaotique dans [APT 98].

Définition 3.2 *La notion d'itération quasi-asynchrone est définie par la Définition 3.1 dans laquelle l'Item 2' ci-dessous remplace l'Item 2.*

2'. $\forall f \in F$, si $\exists i, \forall j \geq i, f_j \neq f$ alors $\exists k \geq i, \forall l \geq k, f(a_l) \sqsubseteq a_l$.

Intuitivement, l'Item $2'$ ci-dessus signifie que les fonctions qui, à partir d'une certaine itération, ne modifieront plus le résultat peuvent n'être appliquées qu'un nombre fini de fois seulement. Comme cela est établi par la proposition suivante, les itérations quasi-asynchrones calculent toujours la même limite.

Proposition 3.4 *A chaque itération quasi-asynchrone correspond une itération asynchrone de même limite.*

Preuve 3.4 *Une itération quasi-asynchrone peut être transformée en une itération asynchrone de même limite en ajoutant (un nombre infini de fois) à certains endroits des applications de drfs qui ne modifient pas la valeur courante.*

Corollaire 3.1 *Les Propositions 3.2 et 3.3 s'étendent aux itérations quasi-asynchrones.*

4. Deux algorithmes fondés sur les itérations asynchrones

Le cadre général des itérations asynchrones appliqué à la propagation de contraintes peut donner lieu à plusieurs algorithmes de propagation de contraintes basés sur des paradigmes de natures différentes. Les implantations en résultant peuvent donc être réalisées sur des architectures spécifiques afin d'améliorer les performances de la propagation de contraintes.

Nous étudions dans les sous-sections suivantes un algorithme réparti et un algorithme parallèle. Le premier nécessite des processus indépendants, la notion de canaux de communication entre ces processus, ainsi que la lecture/écriture de données sur ces canaux. Ainsi, un réseau (local de préférence, afin de ne pas pénaliser l'exécution par des communications externes) de postes de travail constitue une architecture adéquate à la réalisation d'une telle implantation. Le deuxième algorithme quant à lui requiert la notion de processus, et de mémoire partagée par ces processus. De plus, un accès en exclusion mutuelle à cette mémoire est nécessaire. Ainsi, une machine multi-processeur à mémoire partagée (du type SMP, Symmetrical Multi-Processor, par exemple) peut être une bonne solution pour l'implantation d'un tel algorithme.

Remarquons que l'algorithme général d'itération chaotique [APT 97] est un algorithme séquentiel correspondant à notre cadre d'itération asynchrone, les itérations chaotiques étant un cas particulier des itérations asynchrones.

Pour les deux algorithmes que nous allons présenter, nous considérons la terminaison implicite. Des algorithmes standards de détection de terminaison pourraient être utilisés (par exemple [TEL 94] pour les algorithmes répartis).

4.1. *Itérations Chaotiques Réparties*

Nous avons proposé ([MON 98]) un algorithme réparti de propagation de contraintes appelé DCI en extension de l'algorithme GCI proposé par K.R. Apt dans [APT 97]. Ce cadre est basé sur la notion d'itération chaotique et nous avons effectué une preuve "directe" de la correction de l'algorithme DCI, directe dans le sens où nous montrons que la limite calculée est bien celle attendue. Nous précisons dans cette section les bases mathématiques de cet algorithme réparti en montrant que ce qu'il calcule est finalement une itération asynchrone.

L'algorithme DCI est présenté Figure 1. Il s'agit en fait d'une version légèrement simplifiée dans laquelle l'envoi (*send*) d'une valeur est effectué dès qu'une nouvelle valeur est calculée. Dans l'algorithme originel, cet envoi peut être retardé de façon à communiquer éventuellement dans un même message le résultat de plusieurs applications de drfs, ce qui permet de définir des stratégies concernant non seulement l'application des drfs mais aussi la communication des nouvelles valeurs. Cela nous permettra de simplifier la preuve de correction de l'algorithme qui autrement s'en trouverait considérablement alourdie. Pour la même raison, nous supposerons ici que l'ordre des messages est respecté bien que cela ne soit pas requis par l'algorithme lui-même.

Informellement, l'algorithme fonctionne comme suit : localement, chaque agent gère un sous-ensemble des drfs et un sous-ensemble des variables [3], et communique avec les autres agents par passage asynchrone de messages. Tous les agents sont homogènes et exécutent le même algorithme. Chaque drf est gérée par un seul agent. Le problème est découpé selon une partition de l'ensemble de ses drfs et une partition de l'ensemble de ses variables. Cependant, afin d'assurer l'indépendance des calculs, il faut s'assurer que les drfs puissent être calculées localement, c'est-à-dire que si une drf f est gérée par un agent A_i, alors toute les variables dans le scheme d'entrée de f doivent être gérées par A_i. Des variables dans le scheme de sortie peuvent néanmoins être gérées par d'autres agents. Dans ce cas, une nouvelle valeur pour une variable v sera transmise par l'envoi d'un message sur le canal ch_v. Cela motive la notion de n-répartition définie ci-dessous. Considérons un problème $P = (V, \langle D, \sqsubseteq\rangle, F)$.

Définition 4.1 *Une n-répartition de* $P = (V, \langle D, \sqsubseteq\rangle, F)$ *est formée d'un ensemble* $\{V_1, .., V_n\}$ *de schemes sur* V *et d'un ensemble* $\{F_1, ..., F_n\}$ *de drfs vérifiants :*

- $\{V_1, ..., V_n\}$ *est une partition de* V,
- $\{F_1, ..., F_n\}$ *est une partition de* F,
- $\forall i \in [1, n]$, $\forall (is, os, f) \in F_i$, $is \subseteq V_i$.

Considérons une n-répartition $\{V_1, .., V_n\}$, $\{F_1, ..., F_n\}$ d'un problème P. Alors, pour $i \in [1, n]$, une variable v est dite *externe* à i si v n'est pas une variable de V_i et v appartient au scheme de sortie d'une (au moins) drf de F_i. De manière plus formelle :

Définition 4.2 *Pour tout* $i \in [i, n]$, *l'ensemble* Ext_i *des variables externes à* i *est défini par :* $Ext_i = \{v \in V \setminus V_i \mid \exists (is, os, f) \in F_i, v \in os\}$
L'ensemble global des variables externes est : $Ext = \bigcup_{i \in [1,n]} Ext_i$.

Remarquons que pour $i \in [1, n]$, $P_i = (V_i \cup Ext_i, \langle D_{V_i \cup Ext_i}, \sqsubseteq\rangle, F_i)$ est un problème.

Une manière naturelle de répartir un CSP consiste à imposer une partition de ses variables, ou bien de ses contraintes. Etant donné une partition de ses drfs (ou de ses variables), tout problème peut être étendu en une n-répartition "équivalente" par duplication de variables et extention des drfs pour prendre en compte ces nouvelles variables. Nous donnons dans [MON 98], une méthode pour construire des n-répartitions de problèmes.

Les fonctions *select* et *update* définissent la stratégie utilisée par l'algorithme DCI. Nous nous contentons ici de spécifier quelles propriétés doivent vérifier ces fonctions afin d'assurer la correction de l'algorithme. [MON 98] fournit un exemple de définition d'une stratégie standard pour l'algorithme DCI.

[3]Une variable appartenant à plusieurs agents, est dupliquée et porte des noms différents dans chaque agent. Les duplications sont liées par des canaux de communication.

```
funct DCI(P = (V, ⟨D, ⊑⟩, F), V_in, V_out)
  d := ⊥_V ; G := F ;
  while true do
        sel := select(G, V_in)
        if (sel = (is, os, g)) ∈ G
          then G := G \ (is, os, g)
               e := g(d[is])
               d' := d ⊔ e[V]
               foreach v ∈ os ∩ V_out do
                      if d[v] ≠ d'[v] then send(ch_v, d[v]) fi
               od
               G := update(P, G, d, d')
               d := d'
        fi
        if sel ∈ V_in
          then e := read(ch_sel)
               d' := d ⊔ e[V]
               G := update(P, G, d, d')
               d := d'
        fi
  od
  return(d)
```

Figure 1. *L'algorithme DCI*

La fonction *update* doit vérifier :

1. $d = d' \Rightarrow update(P, G, d, d') = G$
2. $\forall (is_f, os_f, f) \in F \setminus update(P, G, d, d'),\ f(d'[is_f]) \sqsubseteq d'[os_f]$

Et la fonction *select* doit vérifier :

1. $select(G, V_{in}, V_{send}) \in G \cup V_{in}$,
2. et si $v = select(G, V_{in}, V_{send}) \in V_{in}$, alors il y a un message en attente de lecture sur le canal ch_v.

La fonction *select* est bloquante : si $G = \emptyset$, et pour tout $v \in V_{in}$ aucun message n'est en attente de lecture sur le canal ch_v, alors $select(G, V_{in}, V_{send})$ bloque jusqu'à ce qu'un message arrive sur au moins un de ses canaux.

Nous appelons *agent DCI* un agent exécutant une fonction DCI. Un *système DCI* pour un n-répartition $V_1, .., V_n$ et $F_1, ..., F_n$ d'un problème $P = (V, \langle D, \sqsubseteq \rangle, F)$ est composée de n agents DCI $a_1, \ldots, a_n$ tels que pour tout $i \in [1, n]$, a_i exécute le problème local $P_i = (V_i \cup Ext_i, \langle D_{V_i \cup Ext_i}, \sqsubseteq \rangle, F_i)$. Ainsi, chaque agent a_i exécute $DCI(P_i, Ext \cap V_i, Ext_i)$.

L'exécution d'un système DCI est dite *équitable* si toute drf présente dans l'ensemble G d'un agent DCI est sélectionnée au bout d'un temps fini et tout message est lu au bout d'un temps fini. Nous prouvons ci-dessous que toute exécution d'un système DCI calcule une itération quasi-asynchrone.

Proposition 4.1 *Soit $V_1, \ldots, V_n$, $F_1, \ldots, F_n$ une n-répartition d'un problème P. Toute exécution équitable d'un système DCI pour cette n-répartition correspond à une itération quasi-asynchrone de P.*

Preuve 4.1 *L'exécution d'un système DCI peut être sériée, et donc être perçue comme une suite d'itérations d'agents DCI. Aussi, nous pouvons représenter l'exécution d'un système DCI par une suite $(d_0, action_0), (d_1, action_1), \ldots$, où d_i est la valeur courante des variables à l'itération i (le plus petit majorant des variables locales d dans les agents DCI) et les actions sont soit :*

- *une drf f, si f est sélectionnée par l'agent DCI en question,*
- *un canal ch_v si le canal de réception ch_v est sélectionné.*

A cette suite correspond une itération quasi-asynchrone $a_0, a_1, \ldots$ définie par les suites $f_1, f_2, \ldots$ et $S_1, S_2, \ldots$ de la manière suivante. Considérons les variables $Send_v$ pour $v \in V$ où chaque $Send_v$ est une file d'entiers qui nous servira à enregistrer les itérations lors desquelles un send a été effectué.

- *A l'initialisation, nous avons $d_0 = \bot_V$, $a_0 = \bot_V$ et $\forall v \in V, Send_v = 0$. A $(d_0, action_0)$, nous associons donc $a_o = \bot_V$*
- *Supposons que $(d_i, action_i)$ soit associé à a_j alors :*
  - *si $action_i = f$ avec $is_f = v_1, \ldots, v_p$ alors $f_{j+1} = f$, $\forall v \in V, S^v_{j+1}$ est égal au premier élément de la file $Send_V$, et $(d_{i+1}, action_{i+1})$ est associé à $a_{j+1} = a_j \sqcup f(a_{S^{v_1}_{j+1}}[v_1], .., a_{S^{v_p}_{j+1}}[v_p])[V]$*
  - *si $action_i = ch_v$ alors j est ajouté à la file $Send_v$, et $(d_{i+1}, action_{i+1})$ est associé à a_j.*

La suite $a_0, a_1, \ldots$ ainsi obtenue est une itération quasi-asynchrone de P définie par $a_0 = \bot_V$ et les suites $f_1, f_2, \ldots$ et $S_1, S_2, \ldots$ construites ci-dessus.
En particulier, les conditions posées sur la fonction update nous assurent que la Condition $2'$ de la Définition 3.2 est vérifiée.

4.2. *Un Algorithme Parallèle*

Nous nous situons ici dans le cadre d'une architecture multiprocesseurs à mémoire partagée. Le cadre des itérations asynchrones peut être implanté de manière parallèle sur ce type d'architecture par l'algorithme décrit Figure 2 [4].

[4]Nous notons $mutex_v()$ l'accès en exclusion mutuelle à la variable partagée v.

Cet algorithme permet la partition quelconque d'un problème selon l'ensemble de ses drfs. Considérons un problème $P = (V, \langle D, \sqsubseteq \rangle, F)$ et une partition $F_1, \ldots, F_n$ de F. Nous appelons agent PCI un agent exécutant la fonction PCI donnée en Figure 2. Un *système de propagation de contraintes parallèle* pour la partition $F_1, \ldots, F_n$ du problème P consiste en n agents PCI gérant respectivement les drfs $F_1, \ldots, F_n$. A chaque élément F_i de la partition, nous associons un ensemble V_i qui contient les variables dont l'agent PCI correspondant devra pouvoir lire les valeurs : $\forall i \in [1, n]$, $V_i = \{v \in V \mid \exists (is, os, f) \in F_i, v \in is\}$.

Les variables sont partagées : plusieurs agents PCI peuvent y avoir accès. Chaque variable $v \in V$ est initialisée avec $v := \bot_v$.

funct $PCI(P = (V, \langle D, \sqsubseteq \rangle, F), F_i, V_i)$
  global var $d := \bot_V$, $d \in D_V$
  local **var** $e := \bot_V$, $e \in D_V$
  local **var** $l, l' := \bot_{V_i}$, $l, l' \in D_{V_i}$
  local **var** $G := F_i$, $G \subseteq F_i$
  while *non terminé* **do**
    $l := d[V_i]$
    $G := update(F_i, G, l', l)$
    $l' := l$
    $(is, os, g) := select(G)$
    $e[os] := g(l[is])$
    for $v \in os$ **do** $\textbf{mutex}_v(d[v] := d[v] \sqcup e[v])$ **od**
  od

Figure 2. *La fonction PCI*

Comme dans le cas de l'algorithme DCI, la fonction *update* doit vérifier :

1. $l = l' \Rightarrow update(F_i, G, l', l) = G$
2. $\forall (is_f, os_f, f) \in F_i \setminus update(F_i, G, l', l)$, $f(l[is_f]) \sqsubseteq f(l'[is_f])$

La fonction *select* doit vérifier $select(G) \in G$.

L'exécution d'un système PCI est dite équitable si toute drf présente dans l'ensemble G d'un agent PCI est sélectionnée au bout d'un temps fini.

Proposition 4.2 *Toute exécution équitable d'un système PCI pour un problème P correspond à une itération quasi-asynchrone de P.*

Esquisse de preuve : Puisque la mise à jour des valeurs s'effectue en exclusion mutuelle, les opérations d'écriture sur les variables partagées peuvent être sériées. Nous construisons ainsi la suite $f_1, f_2, \ldots$ des drfs appliquées. Etant donné les conditions posées sur les arguments de la fonction *update* et l'exécution étant équitable, cette suite vérifie l'Item $2'$ de la Définition 3.2 : si une drf

f n'est pas appliquée un nombre infini de fois, alors l'Item 2 des conditions sur la fonction *update* nous assure qu'appliquer encore (possiblement une infinité de fois) f ne modifierait pas la limite.
D'autre part, chaque drf est appliquée, possiblement sur des valeurs anciennes (si d'autres agents ont changé la valeur de d entre le moment de la lecture et celui de la mise à jour). Cela produit la suite $S_1, S_2, \ldots$
Considérons $a_i \in D_V$, la valeur des variables partagées à l'itération i, la suite $a_0, a_1, a_2, \ldots$ est l'itération asynchrone de P définie par $a_0 = \bot_V$ et les suites $f_1, f_2, \ldots$ et $S_1, S_2, \ldots$.

Corollaire 4.1

1. *Toute exécution terminée* [5] *d'un système PCI calcule sur les variables V la limite de la fonction h sur D définie par :*

$$h(x) := \bigsqcup_{f \in F} (f(x)[V] \sqcup x)$$

2. *Si pour tout $v \in V$, $\langle D_v, \sqsubseteq_v \rangle$ vérifie la propriété de chaînes finies alors toute exécution d'un système PCI termine* [5].
3. *S'il existe une exécution équitable d'un système PCI qui termine alors toutes les exécutions équitables terminent* [5].

Preuve 4.2 *L'Item 1 découle directement de la Proposition 4.2. Les Items 2 et 3 découlent de la Proposition 4.2 et du Corollaire 3.1.*

Remarquons que le Corollaire 4.1 s'applique également au système DCI présenté dans la sous-section précédente.

5. Conclusion

Nous avons présenté dans cet article une méthode d'itération asynchrone appliquée à la propagation de contraintes. Après avoir adapté le cadre général des méthodes d'itérations à la propagation de contraintes, nous avons prouvé les théorèmes de convergence liés à cette notion.

A partir de ce cadre formel, nous avons déduit deux algorithmes (l'un parallèle, l'autre réparti) dont les implantations respectives peuvent bénéficier d'architectures matérielles spécifiques.

Dans [MON 99], nous décrivons les résultats obtenus à partir d'une implantation prototype (spécialisée pour la propagation sur les contraintes polynomiales réelles non-linéaires) de l'algorithme DCI (algorithme réparti). Ces

[5]Rappelons que nous parlons ici de terminaison implicite.

résultats prometteurs montrent des améliorations significatives d'efficacité (accélérations en fonction du nombre de processus utilisés).

Nous avons l'intention de réaliser une implantation de l'algorithm PCI, et de comparer les comportements des versions répartie et parallèle.

Remerciements : nous remercions K.R. Apt pour de nombreuses discussions fructueuses à propos de l'itération chaotique et de la propagation de contraintes.

Références

[APT 97] K.R. Apt. From Chaotic Iteration to Constraint Propagation. In *Proc. of International Colloquium on Automata, Languages and Programming (ICALP '97)*, pages 36–55. Springer-Verlag, LNCS 1256, 1997. Lecture invitée.

[APT 98] K.R. Apt. The Essence of Constraint Propagation. *Theoretical Computer Science*, 1998. A paraître.

[BAU 78] G.M. Baudet. Asynchronous Iterative Methods for Multiprocessors. *Journal of the ACM*, 25(2) :226–244, Avril 1978.

[BEN 97] F. Benhamou et W.J. Older. Applying Interval Arithmetic to Real, Integer and Boolean Constraints. *Journal of Logic Programming*, 1997.

[CHA 69] D. Chazan et W. Miranker. Chaotic Relaxation. *Linear Algebra and Applications*, 2 :199–222, 1969.

[COU 78] P. Cousot. *Méthodes Itératives de Construction et d'Approximation de Points Fixes d'Opérateurs Monotones sur un Treillis, Analyse Sémantique des Programmes.* PhD thesis, Institut National Polytechnique de Grenoble, 1978.

[MAC 77] A. Mackworth. Consistency in networks of relations. *Artificial Intelligence*, 8(1) :99–118, 1977.

[MOH 86] R. Mohr et T. Henderson. Arc and Path Consistency Revisited. *Artificial Intelligence*, 28 :225–233, 1986.

[MON 98] E. Monfroy et J.-H. Réty. A Distributed Chaotic Iteration Algorithm. In *Proc. of ERCIM/COMPULOG Workshop on Constraints*, Amsterdam, Septembre 1998. CWI.

[MON 99] E. Monfroy et J.-H. Réty. Chaotic Iteration for Distributed Constraint Propagation. In *Proc. of ACM Symposium on Applied Computing (SAC'99), San Antonio, Texas, USA*, 1999. (To appear.).

[NGU 98] T. Nguyen et Y. Deville. A Distributed Arc-Consistency Algorithm. *Science of Computer Programming*, 30(1-2) :227–250, 1998.

[TEL 94] G. Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 1994.

[VAN 97] P. Van Hentenryck, D. McAllester et D. Kapur. Solving polynomial systems using a branch and prune approach. *SIAM Journal on Numerical Analysis*, 34(2), 1997.

[WEI 94] J. Wei. *Fixpoint Transformations and Parallel Asynchronous Iterative Algorithms.* GMD report n. 229. R. Olenbourg Verlag, 1994.