Validation of the compilation of Data-Parallel C “while” loops for shared memory architectures

Gil Utard

Laboratoire de l'Informatique du Parallélisme

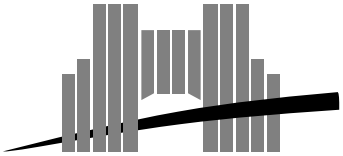
Ecole Normale Supérieure de Lyon
Unité de recherche associée au CNRS n°1398

Validation de la compilation des boucles `while` *Data-Parallel C sur des architectures à mémoire partagée*

Gil Utard

Mars 1994

Research Report N^{O} 94-13



Ecole Normale Supérieure de Lyon
46 Allée d'Italie, 69364 Lyon Cedex 07, France
Téléphone : (+33) 72.72.80.00 Télécopieur : (+33) 72.72.80.80
Adresse électronique : lip@lip.ens−lyon.fr

Validation de la compilation des boucles `while` Data-Parallel C sur des architectures à mémoire partagée

Gil Utard

Mars 1994

Abstract

This report focuses on the compilation of the `while` loops in data-parallel languages for MIMD Shared Memory architectures. An efficient compilation must decrease the number of *global synchronizations* due to dependencies. We validate an optimization suggested by Hatcher and Quinn for the DPC language. It consists in splitting the original loop into two loops: one *computation loop* without any additional control dependencies, and one *waiting loop* to assure termination. Computation loop's body presents a minimal number of global synchronizations. We study informaly its correction proof, and give the methodology leading of its conception. The formal proof is based on the axiomatic semantics of Owiki and Gries. We give an axiomatization of the global synchronization statement, and specify which are the sufficient conditions for a non-deadlocking execution. In Hatcher and Quinn's solution, we observe that the waiting loop is independant of the computation one. The former loop absorbs residual synchronizations of any parallel program. We conclude by presenting a modular method to elaborate parallel programs.

Keywords: Concurrent Programming; Data-Parallel Languages; Compilation; Validating Compilation Scheme; Axiomatic Semantics.

Résumé

Nous étudions la compilation des boucles `while` des langages data-parallèles pour les architectures MIMD à mémoire partagée. Une compilation efficace doit réduire le nombre de *barrières de synchronisation* induites par les dépendances. Nous validons une optimisation proposée par Hatcher et Quinn pour le langage DPC. Elle consiste à dégager *une boucle de calcul* sans dépendances de contrôle supplémentaires et une *boucle d'attente* assurant la terminaison. Le corps de la boucle de calcul contient un nombre minimal de barrières de synchronisation. Nous étudions informellement sa correction et dégageons la méthode qui a permis sa conception. La preuve formelle est basée sur la sémantique axiomatique d'Owiki et Gries. Nous proposons une axiomatisation de la barrière de synchronisation et dérivons des conditions suffisantes de non-blocage. Dans la solution de Hatcher et Quinn, nous observons que la boucle d'attente est indépendante de la boucle de calcul et permet de résoudre un problème plus général : l'*absorption des synchronisations résiduelles* des programmes parallèles. Nous en déduisons une méthode de conception modulaire des programmes parallèles.

Mots-clés: programmation parallèle ; langages data-parallèles ; compilation ; validation des schémas de compilation ; sémantique axiomatique.

Validation de la compilation des boucles `while` Data-Parallel C sur des architectures à mémoire partagée

Gil Utard

Mars 1994

Table des matières

Introduction

On assiste actuellement à un engouement croissant pour les langages à parallélisme de données. Jusqu'alors confiné à la programmation des machines SIMD, ce type de langage se libère de son origine architecturale, et peut être considéré comme un *modèle de programmation* à part entière [3]. La preuve en est le formidable effort du "High Performance Fortran Forum" pour définir "une spécification d'un langage portable de la station de travail à la machine massivement parallèle" [5].

Les deux grandes caractéristiques des langages à parallélisme de données sont le *synchronisme*, qui permet de développer et de mettre au point les applications de manière aisée, et le parallélisme *explicite*, qui permet d'appréhender la complexité d'un programme par le concepteur et le compilateur. Toute la puissance des langages data-parallèles proviendra de compilateurs capables d'exploiter au mieux les architectures cibles. C'est la voie adoptée par Hyper C et C^* qui proposent une compilation sur différentes architectures. Un effort important doit donc être porté sur la génération de code exploitant au mieux le matériel.

Dans les applications scientifiques, l'effort de la compilation doit se porter sur les boucles. C'est là en général que l'application concentre ses calculs. Dans leur présentation du langage Data Parallel *C* (DPC [7]), Hatcher et Quinn proposent une méthode de compilation originale des boucles pour les machines MIMD à mémoire partagée. Elle minimise le nombre de *barrières de synchronisation* dans le corps de la boucle de calculs. L'objet de cette note est de démontrer formellement la correction de cette méthode de compilation.

Dans un premier temps nous présentons DPC ainsi que le processus de compilation sur architecture à mémoire partagée. Ensuite nous étudions différentes transformations possibles de la compilation de boucles DPC en SPMD, et nous décrivons incrémentalement la solution adoptée par Hatcher et Quinn. Ensuite nous rappelons la définition d'un langage parallèle asynchrone à mémoire partagée d'Owiki et Gries, et son système de preuve axiomatique associé. Nous y introduisons une axiomatisation de la barrière de synchronisation et des conditions suffisantes de non-blocage. Nous validons formellement le schéma de compilation de Hatcher et Quinn, et nous montrons que leur solution résoud un problème plus général : l'*absorbtion des synchronisations résiduelles* d'un programme parallèle. Nous en déduisons une méthode de conception modulaire des programmes parallèles.

1 Le langage Data Parallel C et sa compilation

Comme son nom l'indique, Data Parallel *C* est une extension du langage *C* au parallélisme de données, tout comme MPL sur la MASPAR [10], ou HYPER C [9]. Ce langage correspond historiquement aux premières versions de *C** (jusqu'à la version 6.0) sur la Connection Machine [12].

Dans le modèle de programmation data-parallèle, les objets de base sont les tableaux dont les composantes sont accessibles en parallèle. Deux types d'opérations peuvent être appliqués à ces objets :

- les opérations d'*éléments à éléments* telles que l'addition de matrices ;
- les opérations de *réarrangement* telles que la transposition d'une matrice.

Un programme est une composition séquentielle de telles opérations. Chaque opération est associée à un ensemble de composantes où elle sera appliquée : ces composantes sont dites *actives*. L'ensemble des composantes actives est appelé le *contexte d'activité* ou *la portée du parallélisme*. Le parallélisme est ainsi *explicitement* spécifié par les opérations parallèles appliquées aux objets.

Historiquement ce modèle de programmation est issu du modèle d'exécution SIMD. C'est celui mis en œuvre dans l'architecture de la CM2 ou de la Maspar MP-1 : une unité de contrôle centralisée diffuse de manière synchrone les instructions du programme à un ensemble de processeurs, qui les exécute ou non selon leur activité. L'activité se déduit des différentes alternatives du programme.

Dans DPC, les tableaux parallèles sont appelés *domain*. À chaque élément d'un *domain* est associé un *processeur virtuel*. Les opérations *éléments à éléments* sont dénotées par les opérateurs séquentiels classiques tels que l'addition ou la multiplication.

Dans DPC, les blocs data-parallèles et séquentiels sont explicitement séparés contrairement à HYPER C ou MPL. L'activation d'un bloc data-parallèle est spécifiée par une structure de contrôle supplémentaire : la sélection de domain (*domain select statement*). À l'intérieur d'un bloc data-parallèle, les structures de contrôle *C* classiques (`while`, `if`, ;) acquièrent une nouvelle sémantique.

Composition séquentielle : $S_1; S_2$. Les instructions S_1 et S_2 sont exécutées en séquence, de manière synchrone, par l'ensemble des processeurs actifs.

```
domain matrice {
  double A_row[N], B_col[N], C_row[N];
} Mat[N];

main(){
  [domain matrice].{
    int i, j;
    i = 0;
    while (i < N) {
        C_row[i] = 0;
        j = 0;
        while (j < N) {
            C_row[i] += A_row[j] * Mat[i].B_col[j];
            j++; }
        i++; }
  }
}
```

FIG. 1 - *Multiplication de matrices en DPC: $C = A \times B$*

Alternative : `if` (B) S_1 `else` S_2**.** L'ensemble des processeurs actifs évaluent l'expression B. Les processeurs actifs qui ont évalué B à faux deviennent inactifs. S_1 est exécuté dans le nouveau contexte. L'activité initiale est restaurée. Les processeurs actifs qui avaient évalué B à vrai deviennent inactifs. S_2 est exécuté dans ce nouveau contexte. L'activité initiale est restaurée.

Itération : `while` (B) S**.** À chaque itération, les processeurs actifs évaluent la condition B. Les processeurs qui ont évalué B à faux deviennent inactifs pour toutes les itérations suivantes. Le corps S est alors exécuté par les processeurs restant actifs. L'itération termine lorsque tous les processeurs sont inactifs. L'activité initial est alors restaurée. Il faut noter qu'un processeur inactivé au cours de l'itération ne pourra redevenir actif, même si un autre processeur modifie l'état de la machine tel que B s'évaluerait à vrai. Ce comportement est différent de l'itération `whilesomewhere` de HYPER C, où le processeur serait réactivé.

Le code de la figure 1 est un exemple de programme DPC. Ce programme calcule la multiplication de matrices d'ordre N. Ce programme définit le domain *matrice* comme un tableau linéaire de N éléments. Chaque élément est traité par un processeur virtuel. Chaque processeur k possède le k^{e} vecteur ligne de la matrice d'entrée A, le k^{e} vecteur colonne de la matrice d'entrée B, et le k^{e} vecteur ligne de la matrice résultat C. L'instruction [`domain` *matrice*] est la sélection de domain qui active les processeurs virtuels. Les lignes résultats sont calculées en parallèles, et chaque processeur k calcule séquentiellement le k^{e} vecteur ligne de la matrice résultat C. Chaque processeur peut accéder aux éléments d'un autre comme dans l'expression $Mat[i].B\_col[j]$.

La compilation de DPC pour machines MIMD à mémoire partagée telles que le Sequent ou la KSR-1 est décrite dans [7]. Par rapport au modèle d'exécution SIMD du data-parallélisme, il apparaît deux différences majeures :

- *asynchronisme* entre les instructions : chaque processeur possède son propre séquenceur ;
- *nombre de processeurs fixé* : dans DPC, aucune limite sur le nombre de processeurs virtuels n'est fixée.

Les quatre phases de compilation décrites par Hatcher et Quinn sont les suivantes.

Analyse des dépendances du code data-parallèle. La mémoire étant partagée, les interactions entre les processeurs sont effectuées par de simples lectures en mémoire. De par l'asynchronisme de la machine cible, le compilateur doit détecter les points de dépendance de données entre les processeurs.

Factorisation des points de dépendance. Le compilateur génère des points de synchronisation qui regroupent de manière optimale plusieurs dépendances.

Réécriture des structures de contrôle. Les points de synchronisation dégagés auparavant sont générés à partir du graphe de dépendance des données uniquement. Les dépendances de contrôle ne sont pas prises en compte. Par exemple, un point de synchronisation peut être généré dans un `if` et non dans le `else` correspondant. Le code est transformé de telle manière que tous les processeurs exécutent les points de synchronisation.

Virtualisation. Chaque processeur physique va simuler plusieurs processeurs virtuels. Ainsi, entre chaque synchronisation, une boucle exécute le code data-parallèle sur ses différentes instances de processeurs virtuels.

Une compilation efficace doit minimiser le nombre de points de synchronisation. Des méthodes de réduction du nombre de points de synchronisation pour les programmes sans boucle sont connues [8]. Nous nous intéressons ici exclusivement à la réécriture des structures de contrôle pour les boucles `while` DPC. Hatcher et Quinn proposent une méthode optimisée. Nous présentons une étude et une preuve formelle de celle-ci.

2 Compilation des boucles `while` data-parallèles

2.1 Description du langage cible

L'architecture cible est une machine MIMD à mémoire partagée (MIMD-SM). Les auteurs présentent leur schéma de traduction dans le langage C de la machine Sequent (Sequent-C). Pour les besoins de la validation, nous présentons le schéma de traduction dans un langage parallèle simplifié. Un programme est une composition parallèle de processus séquentiels. Chaque processus est exécuté par un processeur, et est spécifié par un langage de programmation séquentielle classique enrichi de primitives de synchronisation. Les communications entre processeurs s'effectuent par l'intermédiaire de la mémoire partagée.

La partie séquentielle du langage de programmation se définit classiquement.

Affectation : $x := E$**.** La variable x est affectée avec la valeur de l'expression E.

Alternative : `if` B `then` S `end`**.** Si l'évaluation de l'expression booléenne B est *vraie*, alors le processeur exécute S.

Composition séquentielle : $S_1; S_2$**.** Le processeur exécute le code de S_1 puis celui de S_2.

Itération : `while` B `do` S `end`**.** Le processeur exécute itérativement le code S tant que la condition B est vérifiée.

La partie parallèle du langage se définit par la composition parallèle et une primitive de synchronisation.

Composition parallèle : $S_1 \parallel \ldots \parallel S_p$**.** Les programmes séquentiels $S_1, \ldots, S_p$ s'exécutent concurremment.

Barrière de synchronisation : `sync`**.** Un processeur qui commence l'exécution de cette instruction ne peut la terminer que quand tous les autres auront aussi commencé l'exécution d'un `sync`. D'autre part, chaque processeur i possède une variable particulière $nsync_i$ qui représente le nombre de synchronisations effectuées.

2.2 Boucles `while` avec un seul point de dépendance

Considérons le cas où la phase d'analyse des dépendances signale qu'il existe une dépendance dans le corps de la boucle entre deux blocs S et T (figure 2). Pour simplifier la description de la compilation, nous considérons que la condition d'arrêt B est une variable booléenne parallèle. Une première traduction pour notre architecture cible est représentée sur la même figure. Pour chaque processeur i, la condition d'arrêt de la boucle porte sur la composante i de la variable parallèle B noté b_i. Les traductions des blocs de codes S et T sont dénotées par S_i et T_i. La variable $Actif_i$ dénote l'activité du processeur. Ce schéma de traduction suit la sémantique du `while` DPC (chaque instruction de la traduction est dénotée par une étiquette).

α : Pour tout processeur actif, si B s'évalue à faux, alors il devient inactif jusqu'à la fin de l'itération. À chaque itération, la nouvelle activité d'un processeur i est déterminée par la conjonction de son activité courante $Actif_i$ avec la condition b_i.

β : Les processeurs exécutent les blocs S et T selon leur activité.

γ : L'itération termine lorsque tous les processeurs sont inactifs : la condition d'arrêt est donc un *ou global* sur l'activité de tous les processeurs.

δ : À la fin de l'itération, l'activité initiale est restaurée. L'activité du processeur i est sauvegardée avant la première itération dans $Sauve_i$, puis restaurée à la fin de l'itération.

Le modèle d'exécution étant asynchrone, deux types de barrières de synchronisation sont introduites :

a : cette synchronisation correspond à la dépendance entre les blocs de calculs S et T ;

b : la condition d'arrêt est un *ou global* sur les variables $Actif_i$ (γ). Une synchronisation est nécessaire après la mise à jour de l'activité (α) pour tenir compte de cette nouvelle dépendance.

Une variante possible et de séparer l'exécution des processeurs actifs et inactifs. Les processeurs actifs exécutent une première boucle dite de *calcul*, les inactifs exécutent une deuxième dite d'*attente* (figure 3). La condition d'arrêt d'un processeur inactif reste un *ou global* sur l'activité de tous les processeurs. La condition d'arrêt d'un processeur actif i porte sur l'activité locale $Actif_i$. Bien que non nécessaire à la dépendance des calculs, la deuxième synchronisation b de la boucle de calcul est maintenue pour assurer la terminaison. La trace des synchronisations et calculs reste identique.

Une propriété remarquable de cette traduction porte sur la parité du nombre de synchronisations effectuées par les processeurs. Comme on a deux synchronisations dans les deux boucles, cette parité est invariante à chaque itération. Pour chaque processeur i, on a les propositions suivantes :

– la parité de $nsync_i$ en entrée de la boucle de calcul reste la même en sortie ;

DPC	MIMD-SM		
	δ	$Sauve_i := Actif_i$;	/* Sauvegarde de l'activité courante */
	α	$Actif_i := Actif_i \wedge b_i$;	/* Activité première itération */
	b	`sync`;	/* Cohérence du *ou global* */
	γ	`while` $(\vee_k Actif_k)$ `do`	/* Tant qu'il reste un processeur actif */
		`if` $(Actif_i)$ `then`	/* Le processeur i est actif */
`while` (B) {	β	S_i	/* Le processeur i exécute le code S */
S;		`end`;	
/* Dépendance */	a	`sync`;	/* Synchronisation de dépendance */
T		`if` $(Actif_i)$ `then`	/* Le processeur i est actif */
}	β	T_i	/* Le processeur i exécute le code T */
		`end`;	
	α	$Actif_i := Actif_i \wedge b_i$;	/* Activité itération suivante */
	b	`sync`;	/* cohérence du *ou global* */
		`end`;	
	b	`sync`;	/* cohérence du *ou global* */
	δ	$Actif_i := Sauve_i$	/* Restauration de l'activité initiale */

FIG. 2 - *Code DPC avec une dépendance et une première traduction*

δ	$Sauve_i := Actif_i$;	/* Sauvegarde de l'activité courante */
α	$Actif_i := Actif_i \wedge b_i$;	/* Activité première itération */
b	`sync`;	/* cohérence du *ou global* */
	`while` $(Actif_i)$ `do`	/* Tant que le processeur i est actif */
β	S_i;	/* Le processeur i exécute le code S */
a	`sync`;	/* Synchronisation de dépendance */
β	T_i;	/* Le processeur i exécute le code T */
α	$Actif_i := Actif_i \wedge b_i$;	/* Activité itération suivante */
b	`sync`;	/* Cohérence du *ou global* */
	`end`;	
γ	`while` $(\vee_k Actif_k)$ `do`	/* Tant qu'il reste un processeur actif */
a	`sync`;	/* Synchronisation de dépendance */
α	$Actif_i := Actif_i \wedge b_i$;	/* Activité itération suivante : toujours `faux` */
b	`sync`	/* Cohérence du *ou global* */
	`end`;	
b	`sync`;	/* Cohérence du *ou global* */
δ	$Actif_i := Sauve_i$	/* Restauration de l'activité initiale */

FIG. 3 - *Séparation des processeurs actifs et inactifs*

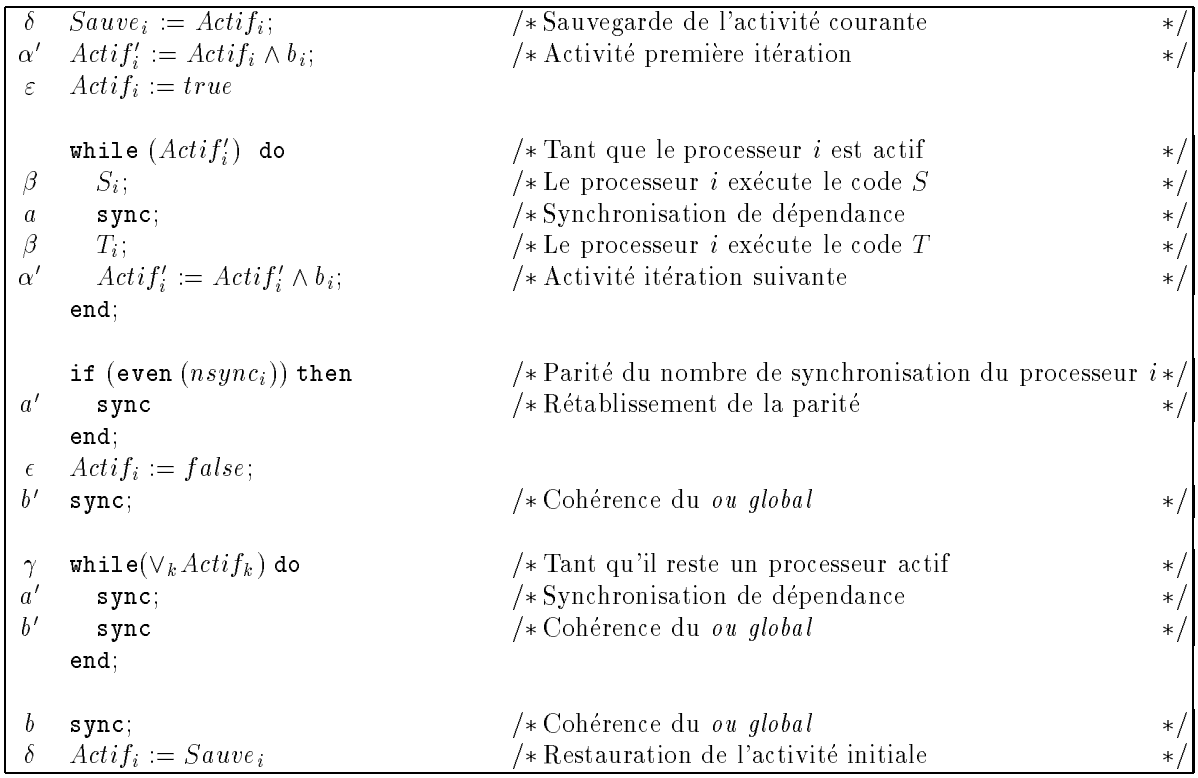


FIG. 4 - *Suppression de la synchronisation redondante dans la boucle de calcul*

– la boucle d'attente effectue le test de terminaison sur une parité fixée ;

– $Actif_i$ ne peut être mis à $faux$ que sur l'autre parité (comme il y a une synchronisation avant la boucle de calcul, ceci est vrai pour la première modification de $Actif_i$).

Une hypothèse suffisante à la correction de cette traduction est que tous les processeurs aient effectué un nombre de synchronisations de parité égale avant d'entamer le code. Dans la traduction globale d'un programme DPC, cette hypothèse est vérifiée car tous les processeurs effectuent un nombre de synchronisations identique pour chaque *phase* du calcul.

Si on supprime la deuxième synchronisation dans la boucle de calcul, alors l'invariance de parité n'est plus vérifiée. La modification de l'activité d'un processeur et l'évaluation du *ou global* par d'autres ne sont plus nécessairement séparées par une synchronisation. Une solution consiste à (figure 4) :

Rétablir une parité fixée à la sortie de la boucle de calcul : chaque processeur exécute ou non une nouvelle synchronisation selon la parité du nombre de synchronisation qu'il a effectuées.

Mettre à $faux$ la variable $Actif_i$ à cette parité seule (ϵ) : une nouvelle variable $Actif'_i$ est introduite et représente l'activité pour la boucle de calcul (α'), et $Actif_i$ est initialisée à $true$ (ε) ($Actif'_i \Rightarrow Actif_i$).

Insérer une synchronisation avant la boucle d'attente : elle sépare la modification d'une variable $Actif_i$ de l'évaluation du *ou global*. La synchronisation avant la boucle de calcul n'est plus nécessaire.

Cette solution assure que tous les processeurs termineront en même temps lorsque tous les processeurs seront inactifs. Par rapport à la précédente version, les processeurs effectueront deux fois moins de synchronisations.

2.3 Boucles `while` avec deux ou plusieurs points de dépendance

Une traduction directe d'une boucle `while` DPC à deux dépendances est présentée sur la figure 5. Le code où les processeurs actifs et inactifs sont séparés est présenté sur la figure 6. Le nombre de synchronisations effectuées par chacune des deux boucles est un multiple de trois. La terminaison est assurée lorsque les processeurs ont effectué le même nombre de synchronisations avant le début du code.

La même transformation est appliquée : on supprime la dernière synchronisation (de type b) de la boucle de calcul, et on insère un mécanisme qui rétablit la multiplicité par trois du nombre de synchronisations après la boucle de calcul (figure 7).

Cette optimisation se généralise quel que soit le nombre n de dépendances dans la boucle de calcul.

– Séparation en deux boucles distinctes des processeurs actifs et inactifs :

  – boucle des actifs avec n `sync` correspondant aux dépendances ;
  – boucle des inactifs avec $n + 1$ `sync`.

– Insertion d'un mécanisme qui rétablit la multiplicité par $n + 1$ du nombre de `sync` en sortie de la boucle des processeurs actifs.

– Modification de la variable représentant l'activité une fois la multiplicité rétablie.

Le schéma de traduction dépend du nombre de dépendances dans la boucle initiale.

DPC	MIMD-SM		
	δ	$Sauve_i := Actif_i$;	/*Sauvegarde de l'activité courante */
	α	$Actif_i := Actif_i \wedge b_i$;	/*Activité première itération */
	b	`sync`;	/*Cohérence du *ou global* */
			|
	γ	`while` $(\vee_k Actif_k)$ `do`	/*Tant qu'il reste un processeur actif*/
		`if` $(Actif_i)$ `then`	/*Le processeur i est actif */
	β	S_i	/*Le processeur i exécute le code S */
`while` (B) {		`end`;	
S;	a	`sync`;	/*Synchronisation de dépendance */
/*Dépendance*/		`if` $(Actif_i)$ `then`	/*Le processeur i est actif */
T;	β	T_i	/*Le processeur i exécute le code T */
/*Dépendance*/		`end`;	
U	a	`sync`;	/*Synchronisation de dépendance */
}		`if` $(Actif_i)$ `then`	/*Le processeur i est actif */
	β	U_i	/*Le processeur i exécute le code U */
		`end`;	
	α	$Actif_i := Actif_i \wedge b_i$;	/*Activité itération suivante */
	b	`sync`;	/*Cohérence du *ou global* */
		`end`;	
			|
	b	`sync`;	/*Cohérence du *ou global* */
	δ	$Actif_i := Sauve_i$	/*Restauration de l'activité initiale */

FIG. 5 - *Code DPC avec deux dépendances et une traduction directe*

```
δ   Sauve_i := Actif_i;                    /* Sauvegarde de l'activité courante */
α   Actif_i := Actif_i ∧ b_i;              /* Activité première itération       */
b   sync;                                  /* Cohérence du ou global            */

    while (Actif_i) do                     /* Tant que le processeur i est actif */
β     S_i;                                 /* Le processeur i exécute le code S  */
a     sync;                                /* Synchronisation de dépendance      */
β     T_i;                                 /* Le processeur i exécute le code T  */
a     sync;                                /* Synchronisation de dépendance      */
β     U_i;                                 /* Le processeur i exécute le code U  */
α     Actif_i := Actif_i ∧ b_i;            /* Activité itération suivante        */
b     sync;                                /* Cohérence du ou global             */
    end;

γ   while(∨_k Actif_k) do                  /* Tant qu'il reste un processeur actif */
a     sync;                                /* Synchronisation de dépendance      */
a     sync;                                /* Synchronisation de dépendance      */
b     sync                                 /* Cohérence du ou global             */
    end;

b   sync;                                  /* Cohérence du ou global             */
δ   Actif_i := Sauve_i                     /* Restauration de l'activité initiale */
```

FIG. 6 - *Séparation des processeurs actifs et inactifs*

2.4 Résumé de la méthode et généralisation

La méthodologie qui a permis de dégager cette méthode de traduction se résume en :

- à partir d'une solution directe, isoler les invariants du programme (i.e. multiplicité par $n+1$ du nombre de synchronisations) et les hypothèses nécessaires à son bon déroulement (i.e. nombre de synchronisations identique au début du code) ;
- effectuer des optimisations *locales* (i.e. suppression de la synchronisation non nécessaire aux dépendances dans la boucle de calcul) ;
- introduire un mécanisme qui rétablit les propriétés initiales (i.e. multiplicité par $n+1$ en sortie de la boucle des processeurs actifs).

Les propriétés implicites du schéma de traduction directe sont explicitées dans le programme même.

La solution dégagée par Hatcher et Quinn a un champ d'application plus large : l'*absorption des synchronisations résiduelles* d'un programme parallèle. Soit p processus séquentiels à deux état :

calcul : tous les processus dans cet état coopèrent et se synchronisent par `sync` ;

attente : les processus qui ont terminé leur calcul, continuent à participer aux synchronisations.

Le programme termine lorsque tous les processus sont dans l'état **attente**.

Un *absorbeur de synchronisations* correspond à la partie attente des processus et assure l'absence de blocage du programme parallèle. La partie attente de la traduction optimisée pour une

```
δ   Sauve_i := Actif_i;                    /* Sauvegarde de l'activité courante                          */
α'  Actif'_i := Actif_i ∧ b_i;             /* Activité première itération                                */
ε   Actif_i := true;

    while (Actif'_i) do                    /* Tant que le processeur i est actif                         */
β     S_i;                                 /* Le processeur i exécute le code S                          */
a     sync;                                /* Synchronisation de dépendance                              */
β     T_i;                                 /* Le processeur i exécute le code T                          */
a     sync;                                /* Synchronisation de dépendance                              */
β     U_i;                                 /* Le processeur i exécute le code U                          */
α'    Actif'_i := Actif'_i ∧ b_i;          /* Activité itération suivante                                */
    end;

    while ((nsync_i mod 3) ≠ 0) do         /* Nombre de synchronisations du processeur i multiple de 3?  */
a'    sync                                 /* Synchronisation supplémentaire                             */
    end;
ε   Actif_i := false;
b'  sync;                                  /* Cohérence du ou global                                     */

γ   while(∨_k Actif_k) do                  /* Tant qu'il reste un processeur actif                       */
a'    sync;                                /* Synchronisation de dépendance                              */
a'    sync;                                /* Synchronisation de dépendance                              */
b'    sync                                 /* Cohérence du ou global                                     */
    end;

b   sync;                                  /* Cohérence du ou global                                     */
δ   Actif_i := Sauve_i                     /* Restauration de l'activité initiale                        */
```

FIG. 7 - *Suppression de la synchronisation redondante dans la boucle de calcul*

boucle `while` à n dépendances vérifie cette propriété. Les absorbeurs obtenus ne font aucune hypothèse sur la partie calcul : la méthode de traduction définit une *classe universelle* d'absorbeurs de synchronisations.Comme nous le verrons dans la dernière partie, le représentant canonique est le cas où n est égal à un.

3 Validation de programmes parallèles

Owiki et Gries ont proposé une méthode basée sur la sémantique axiomatique pour vérifier les propriétés de programmes parallèles (exclusion mutuelle, synchronisation et absence d'interblocage) [11]. Nous employons la même méthode pour valider l'absorbeur de synchronisation extrait de la compilation d'un `while` DPC à une dépendance.

3.1 Vérification de programmes parallèles par une approche axiomatique

Nous rappelons brièvement les principes de la vérification de programmes parallèles basés sur la sémantique axiomatique du langage de programmation. Cette sémantique se présente sous la forme de triplets $\{P\}\ S\ \{Q\}$, où P et Q sont des prédicats décrivant les états possibles de l'exécution du programme S. Plus précisément, une assertion $\{P\}\ S\ \{Q\}$ signifie que si l'état du programme satisfait P avant l'exécution de S, alors il satisfait Q après celle ci. Un ensemble d'axiomes et de règles d'inférences sur la syntaxe des programmes fournit une méthode de dérivation de telles preuves.

Les règles pour la partie séquentielle du langage sont définies classiquement. Dans ce qui suit x représente une variable, E une expression, B une expression booléenne, S et S_i des blocs de programmes séquentiels, P et Q des assertions.

Affectation

$$\{P[E/x]\}\ x\ \texttt{:=}\ E\ \{P\}$$

est l'instruction d'affectation de l'expression E à la variable x. $P[E/x]$ représente l'assertion P où toutes les occurrences libres de x sont remplacées par l'expression E ;

Alternative

$$\frac{\{P \wedge B\}\ S\ \{Q\}, P \wedge \neg B \Rightarrow Q}{\{P\}\ \texttt{if}\ B\ \texttt{then}\ S\ \texttt{end}\ \{Q\}}$$

est l'alternative, c.-à-d. si B est vraie alors l'exécution de S dans un état satisfaisant P mène à un état satisfaisant Q, sinon P satisfait Q ;

Composition séquentielle

$$\frac{\{P\}\ S_1\ \{R\}, \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1\ ;\ S_2\ \{Q\}}$$

est la composition séquentielle. Si l'exécution de S_1 partant d'un état satisfaisant P conduit à un état satisfaisant R, et de même si l'exécution de S_2 partant d'un état satisfaisant R conduit à un état satisfaisant Q, alors l'exécution de la composition séquentielle S_1 ; S_2 partant d'un état satisfaisant P conduit à un état satisfaisant Q.

Itération

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \texttt{end}\ \{I \wedge \neg B\}}$$

est la règle d'itération. Ici I représente ce qu'on appelle un *invariant* de boucle. Si la condition B est vérifiée, alors l'exécution du corps S de la boucle dans un état respectant l'invariant conduit à un autre état qui le respecte encore. L'exécution s'arrête quand la condition B n'est plus vérifiée.

La preuve d'un programme S peut être représentée par le code original agrémenté d'assertions intermédiaires. Il se note par S^* et est appelé l'*annotation du programme* S. Pour toute construction C du programme S, $pre(C)$ est la pré-condition de C dans le programme annoté, et $post(C)$ est la post-condition.

Définition 1 (Validité) *Soit S un programme parallèle. On dit que l'annotation $\{P\}\ S^*\ \{Q\}$ est* valide *si pour toute instruction C de S, sa pré-condition (resp. post-condition) $pre(C)$ (resp. $post(C)$) dans S^* vérifie la propriété suivante :*

> *pour toute exécution de S à partir d'un état σ satisfaisant P, l'état σ' avant (resp. après) exécution de C satisfait $pre(C)$ (resp. $post(C)$).*

La *correction* du système de preuve par rapport à la sémantique opérationnelle du langage de programmation assure que toute annotation correcte d'un programme est valide [11].

La preuve d'un programme parallèle doit tenir compte des interactions possibles entre les processeurs. Owiki et Gries ont proposé une méthode de preuve en deux phases [11]. La phase initiale consiste à vérifier que l'exécution concurrente d'un programme séquentiel n'*interfère* pas sur l'exécution des autres. En sémantique axiomatique, cela signifie que toute instruction d'un programme séquentiel, s'exécutant dans un état satisfaisant sa pré-condition, n'affecte aucune condition des autres programmes. Plus formellement, cela peut s'écrire :

Définition 2 *Soit une instruction C d'un programme séquentiel S, et $pre(C)$ sa pré-condition dans $\{P\}\ S^*\ \{Q\}$, soit un programme S', on dit que C n'interfère pas avec $\{P'\}\ S'^*\ \{Q'\}$ si :*

1. $\{Q' \wedge pre(C)\}\ C\ \{Q'\}$;
2. *quelle que soit l'instruction C' de S' et sa pré-condition $pre(C')$ dans $\{P'\}\ S'^*\ \{Q'\}$, on a $\{pre(C') \wedge pre(C)\}\ C\ \{pre(C')\}$.*

Ceci concerne une une instruction isolée, pour le cas général on a :

Définition 3 *On dit que les programmes séquentiels annotés*

$$\{P_1\}\ S_1^*\ \{Q_1\}, \{P_2\}\ S_2^*\ \{Q_2\}, ..., \{P_p\}\ S_p^*\ \{Q_p\}$$

sont libres d'interférences si quelque soit l'instruction d'affection C d'un programme S_i, alors pour tout $j \neq i$, C n'interfère pas avec $\{P_j\}\ S_j^*\ \{Q_j\}$.

On en déduit la règle suivante :

Composition parallèle

$$\frac{\{P_1\}\ S_1\ \{Q_1\}, \{P_2\}\ S_2\ \{Q_2\}, ..., \{P_p\}\ S_p\ \{Q_p\} \text{ sont libres d'interfèrences}}{\{P_1 \wedge P_2 \wedge ... \wedge P_p\}\ S_1 \parallel S_2 \parallel ... \parallel S_p\ \{Q_1 \wedge Q_2 \wedge ... \wedge Q_p\}}$$

Il est à noter que la notion de non-interférence est associée aux programmes annotés par leur preuve, c.-à-d. qu'un même programme sera libre ou non d'interférences selon ce que l'on veut prouver. Par exemple :

$$\{x \neq 0\}\ x := 1/x\ \{true\} \parallel \{true\}\ x := x - 1\ \{true\}$$

n'est pas libre d'interférences. Par contre :

$$\{x \neq 0\}\ x := 1/x\ \{true\} \parallel \{x \neq 1\}\ x := x - 1\ \{true\}$$

est libre d'interférences.

Il reste à définir une axiomatisation de la primitive de synchronisation `sync`. On note par $n\vec{s}\vec{y}nc$ le vecteur d'ordre p $(nsync_1, \ldots, nsync_p)$, $\vec{k}$ un vecteur d'ordre p et $\vec{1}$ le vecteur unité d'ordre p.

Synchronisation

$$\{n\vec{s}\vec{y}nc = \vec{k}\}\ \texttt{sync}\ \{n\vec{s}\vec{y}nc = \vec{k} + \vec{1}\}$$

tous les processeurs passent ensemble la barrière de synchronisation et le compteur local est incrémenté.

On en déduit les deux propriétés suivantes :

Propriété 3.1 (Locale) *le passage d'une barrière de synchronisation incrémente le compteur local, i.e. pour tout processeur i on a :*

$$\{nsync_i = k\}\ \texttt{sync}\ \{nsync_i = k + 1\}$$

Propriété 3.2 (Invariant de synchronisation) *les processeurs ne peuvent passer une barrière de synchronisation qu'ensemble, i.e. l'instruction* `sync` *vérifie l'invariant suivant :*

$$\{\exists m : \forall k : nsync_k = m\}\ \texttt{sync}\ \{\exists m : \forall k : nsync_k = m\}$$

3.2 Interblocage et preuve de non interblocage

Le système de preuve décrit précédemment permet de vérifier la *correction partielle* : la preuve ne s'intéresse pas aux propriétés de terminaison des programmes parallèles. La correction est dite *totale* quand le système de preuve permet de plus de vérifier la terminaison des programmes. Une exécution de programme parallèle peut ne pas terminer pour deux raisons principales :

- par *divergence*, une des composantes séquentielles est dans une boucle infinie ;
- par *blocage*, une des composantes reste bloquée sur une primitive de synchronisation.

Nous ne nous intéressons ici qu'à la preuve de non blocage d'un programme parallèle.

Définition 4 (Blocage) *Soit $S \equiv S_1 \parallel \ldots \parallel S_p$ un programme parallèle sur p processeurs et σ un état. On dit que S bloque à partir de σ, s'il existe une exécution de S partant de l'état σ, telle qu'il existe deux processeurs i et j $(i \neq j)$ avec :*

- *le processeur i exécute une instruction* `sync` *de S_i ;*
- *le processeur j a terminé l'exécution de S_j.*

Théorème 1 *Soit $S \equiv S_1 \| ... \| S_p$ un programme parallèle sur p processeurs. On note par S_j^k les instructions* `sync` *de S_j. Pour tout j, on note $pre(S_j^k)$ et $post(S_j)$ les assertions dérivées d'une annotation valide $\{P\}\ S^*\ \{Q\}$. Si pour tout i et j, $i \neq j$ on a :*

$$(\vee_k pre(S_j^k) \wedge post(S_i)) \Rightarrow false$$

alors le programme S ne bloque pas à partir de tout état initial σ satisfaisant P.

Preuve soit σ un état tel que σ satisfait P et S bloque sur σ. Par définition, cela signifie qu'il existe une exécution de S partant de σ telle qu'un processeur i ait terminé l'exécution de S_i, et qu'il existe un k tel qu'un processeur $j \neq i$ exécute l'instruction $S_j^k \equiv$`sync`. Soit σ' l'état courant à ce point de l'exécution. De par la définition de la validité 1, cela signifie que $\sigma' \models post(S_i)$ et $\sigma' \models pre(S_j^k)$. D'où contradiction. □

3.3 Absorption des synchronisations

Définition 5 (Absorption) *Soit un programme parallèle sur p processeurs $C = C_1 \| ... \| C_p$. Soit $\mathcal{E}$ un ensemble d'états. Un ensemble de p programmes séquentiels $\mathcal{A} = (A_i)_{1 \leqslant i \leqslant p}$ est un absorbeur de synchronisations pour C selon $\mathcal{E}$ si le programme parallèle*

$$(C_1; A_1) \| \ldots \| (C_p; A_p)$$

ne bloque pas à partir de tout état $\sigma \in \mathcal{E}$.

La traduction d'un `while` DPC à une dépendance est une illustration de l'absorption des synchronisations. La traduction annotée est représentée sur la figure 8. Pour un processeur i la partie calcul est représentée par le bloc de code C_i et la partie absorption par le bloc de code A_i. Soit $\mathcal{E}$ l'ensemble des états tel que quelque soit $\sigma \in \mathcal{E}$, $\sigma \models \exists m : \forall k : nsync_k = m$. L'ensemble des programmes séquentiels $\mathcal{A} = (A_i)_{1 \leqslant i \leqslant p}$ est un absorbeur de synchronisations pour le programme parallèle $C = C_1 \| ... \| C_p$ et $\mathcal{E}$. En effet, pour tout i et j, $i \neq j$, on a :

- $Post(A_i) \wedge Pre(C_j^1) \Rightarrow false$ car $Post(A_i) \Rightarrow Actif_j = false$ et $Pre(C_j^1) \Rightarrow Actif_j = true$;
- $Post(A_i) \wedge Pre(A_j^1) \Rightarrow false$ car $Post(A_i) \Rightarrow Actif_j = false$ et $Pre(A_j^1) \Rightarrow Actif_j = true$;
- $Post(A_i) \wedge Pre(A_j^2) \Rightarrow false$ car l'invariant sur le nombre de synchronisations $\exists m : \forall k : nsync_k = m$ est conservé, et $Post(A_i) \Rightarrow Even(nsync_j)$ et $Pre(A_j^2) \Rightarrow Odd(nsync_j)$;
- $Post(A_i) \wedge Pre(A_j^3) \Rightarrow false$ car $Post(A_i) \Rightarrow \forall k : Actif_k = false$ et $Pre(A_j^3) \Rightarrow \exists k : Actif_k = true$;
- $Post(A_i) \wedge Pre(A_j^4) \Rightarrow false$ car l'invariant sur le nombre de synchronisations $\exists m : \forall k : nsync_k = m$ est conservé, et $Post(A_i) \Rightarrow Even(nsync_j)$ et $Pre(A_j^4) \Rightarrow Odd(nsync_j)$;

D'après le théorème 1, le programme parallèle $(C_1; A_1) \| ... \| (C_p; A_p)$ est exempt de blocage. La seule hypothèse qui est faite sur la partie calcul, est que l'assertion $\{Actif_i = true\}$ est invariante pour C_i. L'ensemble des programmes séquentiels $\mathcal{A}$ peut servir de squelette à un ensemble d'absorbeurs de synchronisations. Soit un ensemble de p variables $X = (X_i)_{1 \leqslant i \leqslant p}$, et l'ensemble de p programmes séquentiels $\mathcal{A}[X] = (A_i[X])_{1 \leqslant i \leqslant p}$ défini comme suit :

$A_i[X] \equiv$ `if` $Even(nsync_i)$ `then sync end;`
$X_i := false;$
`sync;`
`while` $(\vee_k X_k)$ `do sync; sync end`

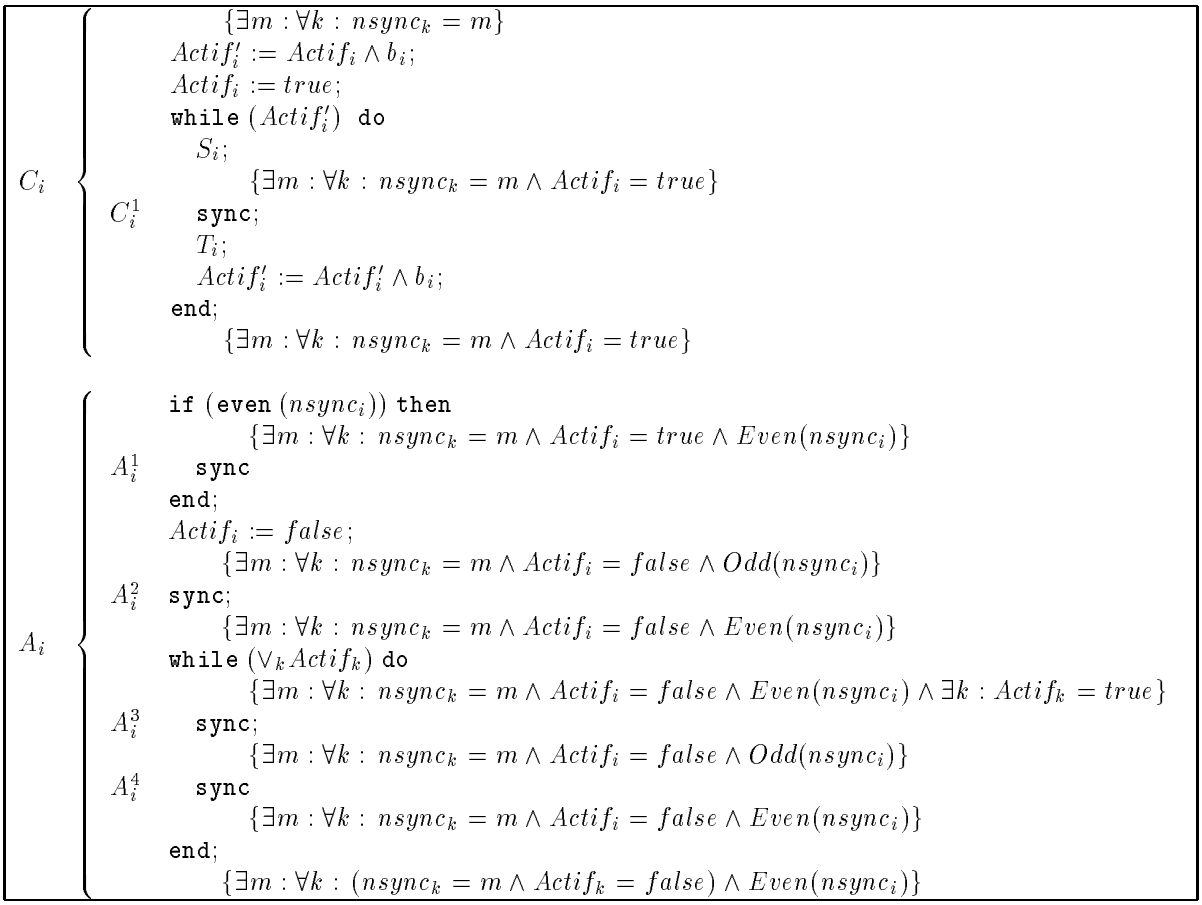


FIG. 8 - *Absorption de synchronisations*

Propriété 3.3 *Soit un programme parallèle sur p processeurs* $C = C_1 \| ... \| C_p$. *Soit p variables* X_i *de* C. *Soit un ensemble d'états initiaux* $\mathcal{E}$. *Si* $\mathcal{E} \models \exists m : \forall k : nsync_k = m$, *et s'il existe une annotation valide du programme parallèle* $\{P\}\ C^*\ \{Q\}$ *telle que :*

- $\mathcal{E} \models P$;
- *l'assertion* $\{X_i = true\}$ *est invariante pour* C_i ;

alors, l'ensemble des programmes $\mathcal{A}[X] = (A_i[X])_{1 \leqslant i \leqslant p}$ *est un absorbeur de synchronisations pour* C *et* $\mathcal{E}$.

La preuve est une application du théorème 1 comme l'exemple précédent.

Conséquence 1 *Tout programme parallèle* $C = C_1 \| ... \| C_p$ *peut être libéré de risque de blocage. On introduit un ensemble de nouvelles variables* X_i, *et le programme*

$$C' = (X_1 := true; \texttt{sync}; C_1) \| ... \| (X_p := true; \texttt{sync}; C_p)$$

a comme absorbeur de synchronisations $\mathcal{A}[X]$ *pour tout ensemble d'états initiaux* $\mathcal{E}$ *tel que* $\mathcal{E} \models \exists m : \forall k : nsync_k = m$.

4 Compilation de DPC et modularité de programmes parallèles

4.1 Retour à la compilation de boucles `while` DPC

Soit $Actif = (Actif_i)_{1 \leqslant i \leqslant p}$. Pour toute boucle `while` DPC à n dépendances, il existe une traduction qui ne bloque pas. Le schéma de traduction est présenté sur la figure 9. Si aucun bloc de code S_i^j ne modifie une variable de $Actif$, l'assertion $\{Actif_i = true\}$ est invariante dans la boucle de calcul du processeur i. D'après la propriété 3.3, $\mathcal{A}[Actif]$ est un absorbeur de synchronisations pour tout ensemble d'états initiaux $\mathcal{E}$ tel que $\mathcal{E} \models \exists m : \forall k : nsync_k = m$. L'absorbeur de synchronisations est indépendant du nombre de dépendances.

Il est à noter que cette traduction n'est possible que de par la sémantique du `while` DPC. Les processeurs devenus inactifs dans la boucle ne pouvant être réactivés par des effets de bords des processeurs actifs, il est possible de dégager au cours de l'exécution deux boucles successives calcul et attente. Cette optimisation est donc impossible dans le cas du `whilesomewhere` de HYPER C qui permet la réactivation de processeurs inactifs par des effets de bords des actifs. Cela souligne l'importance du choix de la sémantique des structures de contrôle pour une compilation efficace.

4.2 Compositionnalité de programmes parallèles

La primitive de synchronisation `sync` permet d'introduire une propriété de *compositionnalité* des programmes parallèles qui existe dans le cas du parallélisme disjoint [1].

Théorème 2 *Soient deux programmes parallèles sur p processeurs*

$$S = S_1 \| \ldots \| S_p$$

et

$$T = T_1 \| ... \| T_p$$

δ	$Sauve_i := Actif_i;$	/* Sauvegarde de l'activité courante	*/
α'	$Actif'_i := Actif_i \wedge b_i;$	/* Activité première itération	*/
ε	$Actif_i := true;$		
	`while` $(Actif'_i)$ `do`	/* Tant que le processeur i est actif	*/
β	$S_i^1;$	/* Le processeur i exécute le code S^1	*/
a	`sync`;	/* Synchronisation de dépendance	*/
β	$S_i^2;$	/* Le processeur i exécute le code S^2	*/
a	`sync`;	/* Synchronisation de dépendance	*/
	$\vdots$		
a	`sync`;	/* Synchronisation de dépendance	*/
β	$S_i^n;$	/* Le processeur i exécute le code S^n	*/
α'	$Actif'_i := Actif'_i \wedge b_i;$	/* Activité itération suivante	*/
	`end`;		
	`if even` $(nsync_i)$ `then`	/* Nombre de synchronisations du processeur i pair?	*/
a'	`sync`	/* Synchronisation supplémentaire	*/
	`end`;		
ϵ	$Actif_i := false;$		
b'	`sync`;	/* Cohérence du *ou global*	*/
γ	`while`$(\vee_k Actif_k)$ `do`	/* Tant qu'il reste un processeur actif	*/
a'	`sync`;	/* Synchronisation de dépendance	*/
b'	`sync`	/* Cohérence du *ou global*	*/
	`end`;		
b	`sync`;	/* Cohérence du *ou global*	*/
δ	$Actif_i := Sauve_i$	/* Restauration de l'activité initiale	*/

FIG. 9 - *Schéma de traduction sans blocage d'une boucle à n dépendances*

qui vérifient les spécifications $\{P\}\ S\ \{R\}$ *et* $\{R\}\ T\ \{Q\}$. *Supposons que pour tout état initial vérifiant* P, S *est exempt de blocage. Alors le programme parallèle*

$$U = (S_1; \texttt{sync}; T_1) \parallel \ldots \parallel (S_p; \texttt{sync}; T_p)$$

vérifie la spécification $\{P\}\ U\ \{Q\}$.

Preuve De par la sémantique de l'instruction `sync`, et de par l'hypothèse de non blocage de S, aucun processeur i ne pourra entamer le bloc de code T_i alors qu'il reste un processeur j exécutant une partie de S_j. D'autre part, comme S est spécifié par $\{P\}\ S\ \{R\}$, les processeurs ne débuterons l'exécution du bloc de T qu'à partir d'un état satisfaisant R. Comme T est spécifié par $\{R\}\ T\ \{Q\}$, alors l'état final satisfera Q. □

Il existe donc un moyen pratique de composer séquentiellement deux programmes parallèles sur p processeurs S et T :

- si S n'est pas libre de blocage, lui adjoindre un absorbeur de synchronisations (par exemple $\mathcal{A}[X]$ comme le suggère la consequence 1) ;
- effectuer *la composition séquentielle répartie* ([2]) de S, `sync` et T, i.e.

$$U = (S_1; \texttt{sync}; T_1) \parallel \ldots \parallel (S_p; \texttt{sync}; T_p)$$

Cette technique permet une approche modulaire de la conception des programmes parallèles. L'application est décomposée en couches successives. Le développement de chaque couche est effectué en faisant abstraction des problèmes de blocages, l'existence d'absorbeurs universels permettant d'assurer le non-blocage de chaque couche.

5 Conclusion

Dans cette étude de la compilation des boucles DPC proposée par Hatcher et Quinn, nous avons dégagé une méthode de raffinement des schémas de compilation d'un langage data-parallèle. Celle-ci consiste à :

- rechercher des propriétés implicites d'un schéma de traduction directe ;
- isoler les causes de perte de performance et proposer des optimisations locales ;
- réintroduire explicitement les propriétés perdues lors de l'amélioration.

La propriété que nous avons cherché à maintenir est celle de l'absence de blocage de la traduction. Le problème posé rentre dans un cadre plus large : l'absorption des synchronisations d'une itération répartie et la dérivation d'absorbeurs corrects.

La sémantique axiomatique d'Owiki et Gries a servi de cadre formel à la validation du schéma de traduction. Nous y avons ajouté une axiomatisation de la barrière de synchronisation disponible sur toute architecture à mémoire partagée. Nous avons défini la notion de blocage induite par cette nouvelle primitive, et dérivé des conditions suffisantes de non blocage.

Ensuite, nous avons défini la notion d'absorbeur de synchronisations d'un programme parallèle. Nous avons observé que le schéma de traduction d'une boucle à une dépendance contenait le schéma d'un tel absorbeur.

D'autre part, nous avons observé que cet absorbeur pouvait être employé pour *toutes* les boucles DPC ayant un nombre de dépendances quelconque, comme préconisé par Hatcher et Quinn. De même, on observait que ce schéma d'absorption des synchronisations pouvait être employé pour *tout* programme parallèle ayant subi une transformation simple.

Enfin, nous avons montré qu'un programme parallèle exempt de blocage et qu'un programme parallèle quelconque pouvaient être composés séquentiellement par l'intermédiaire d'une barrière de synchronisation. Avec le schéma d'absorbeur de synchronisations « universel » observé auparavant, nous avons donc dégagé une méthode de conception modulaire des programmes parallèles.

Dans le cas des programmes distribués (modèle CSP), cette méthode de décomposition en couches successives a été émise par Elrad et Francez : aucune communication ne peut intervenir entre deux couches distinctes [4]. La notion de couche close en communication dévelopée par Gerth et Shrira [6], qui est équivalente à l'absence de blocage, permet de développer des modules indépendants de programmes distribués. Mais, en l'absence de canal de communication particulier, la preuve de « fermeture » doit être faite pour chaque module. Ici, grâce à l'existence d'absorbeurs simples, il est possible de développer chaque module sans s'inquiéter de sa « fermeture ».

Dans cette étude nous avons employé une méthode de preuve axiomatique. Comme le principal sujet d'étude est la trace des synchronisations, on peut envisager une approche par une sémantique observationnelle de type CCS. La synchronisation serait une action particulière dont le mécanisme résiderait dans le masquage. Un absorbeur de synchonisations serait un opérateur particulier sur l'algèbre des processus, modélisant une barrière de synchronisation « d'ordre supérieur ».

L'analyse des propriétés algébriques permettrait peut-être de formaliser de manière plus aisée le processus de transformation des structures de contrôle de la compilation d'un programme data-parallèle. Par exemple, le cas des boucles DPC imbriquées que nous n'avons pas traité dans cette étude.

Remerciements. Je remercie Luc Bougé et Joachim Gabarró pour leurs commentaires sur ce travail.

Références

[1] K.R. Apt and E.R. Olderog. *Verification of Sequential and Concurrent Programs.* Text and Monographs in Computer Science. Springer Verlag, 1990.

[2] L. Bougé. Modularité et Symétrie pour les Systèmes Répartis : Application au Langage CSP. Laboratoire d'Informatique de L'Ecole Normale Supérieure, 1987. Thèse d'Etat.

[3] L. Bougé. Modèle de programmation à parallélisme de données : une perspective sémantique. *Technique et science informatiques*, 12(5):541–562, 1993.

[4] T. Elrad and N. Francez. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming*, 2:155–173, 1982.

[5] High Performance Fortran Forum. *High Performance Fortran language specification (draft version).* CITI/CRPC, Rice Univ., Houston, January 1993. Version 1.0 Draft.

[6] R. Gerth and L. Shrira. On proving communication closedness of distributed layers. In K. V. Nori, editor, *Sixth conf. of FST/TCS*, number 241 in LNCS, pages 330–343, New Delhi, India, December 1986. Springer Verlag.

[7] P.J. Hatcher and M.J. Quinn. *Data-Parallel Programming on MIMD Computers.* Scientific and Engineering Computation. The MIT Press, 1991.

[8] E.A. Heinz and M. Philippsen. Synchronization Barrier Elimination in Synchronous FORALLs. Technical Report 13/93, Universität Karlsruhe, Fakultät für Informatik, 1993.

[9] Hyperparallel Technologies, Ecole Polytechnique Projet X-Pôle 91128 Palaiseau Cedex France. *Hyper C Documentation*, June 1993.

[10] MasPar Computer Corporation, Sunnyvale CA. *Maspar Parallel Application Language Reference Manual*, 1990.

[11] S. Owiki and D. Gries. Verifying Properties of Parallel Programs: An Axiomatic Approach. *Communications of the ACM*, 19(5):279–285, May 1976.

[12] Thinking Machine Corporation, Cambridge MA. *C* programming guide*, 1990.