Traitement de l'irrégularité dans la parallélisation de code séquentiel par distribution des données

Thèse

présentée à l'Université d'Orléans
pour obtenir le grade de

Docteur de l'Université d'Orléans

spécialité Informatique

par

Emmanuel Melin

Soutenue le 17 février 1998

Composition du jury

Président :	Gérard Ferrand	Professeur - Université d'Orléans
Rapporteurs :	Thomas Brandes	Docteur - SCAI - GMD - Sankt Augustin
	Paul Feautrier	Professeur - Université de Versailles
	Claude Jard	Directeur de recherche CNRS - Irisa/Inria Rennes
Examinateurs :	Gaétan Hains	Professeur - Université d'Orléans
	Henri Thuillier	Professeur - Université d'Orléans
	Bernard Virot	Maître de Conférence - Université d'Orléans

Nous nous sommes nourris de la magie des sables, d'autres peut-être y creuseront leurs puits de pétrole, et s'enrichiront de leurs marchandises. Mais ils seront venus trop tard. Car les palmeraies interdites, ou la poudre vierge des coquillages, nous ont livré leur part la plus précieuse: elles n'offraient qu'une heure de ferveur, et c'est nous qui l'avons vécue.

Antoine de Saint-Exupéry "Terre des hommes" (1939).

à Alexandre, Joseph, Louise et mes parents.

Remerciements

Je voudrais remercier Gérard Ferrand, professeur à l'Université d'Orléans, qui me fait l'honneur de présider mon jury et de m'avoir prodigué conseils et encouragements depuis le DEUG jusqu'au Doctorat.

Je remercie également Paul Feautrier d'avoir accepté de rapporter cette thèse. J'ai eu l'occasion de le rencontrer à plusieurs reprises et je lui suis très reconnaissant pour ces précieux commentaires.

Je tiens à remercier Thomas Brandes ainsi que Claude Jard qui ont accepté la même tâche. Leurs remarques m'ont permis de clarifier certains points de mon travail.

Merci à Henri Thuillier, professeur à l'Université d'Orléans, d'avoir été un directeur de thèse attentif et disponible et d'avoir bien voulu participer à mon jury.
Je remercie Gaétan Hains, professeur à l'Université d'Orléans, d'avoir accepté de participer à mon jury. Je lui suis reconnaissant de l'intérêt qu'il porte à ce travail.

Bernard Virot, maître de conférence à l'Université d'Orléans, m'a encadré tout au long de ces années de travail. Je veux lui témoigner toute ma gratitude à la fois pour sa compétence scientifique et sa disponibilité, mais aussi pour ses qualités humaines qui ont largement contribuées à l'aboutissement de ce travail. Il a su me communiquer enthousiasme et exigence vis-à-vis de la recherche.

Un grand merci à Xavier Rebeuf dont la passion est communicative, il a été attentif en tant qu'ami et précieux comme collaborateur. Merci à Bruno Raffin avec qui j'ai aussi collaboré sur plusieurs articles, son amitié a été particulièrement appréciable pendant ces années de thèse. Je salue également Yann Le Guyadec qui m'a initié à son approche de la désynchronisation.

Je remercie Vincent Lefebvre pour sa disponibilité sans faille, ses précieux conseils et pour le soutien qu'il m'a apporté pour la réalisation des tests inclus dans ce travail.

Je remercie l'Institut du Développement et des Ressources en Informatique Scientifique (IDRIS, Orsay) pour avoir mis à ma disposition l'environnement matériel et logiciel nécessaire aux tests inclus dans ce travail *(Projet numéro 970829)*.

Je remercie l'ensemble des membres du Laboratoire d'Informatique Fondamentale d'Orléans pour leur accueil et leur soutien.

En grand merci à mon frère Alexandre qui a toujours été présent lorsqu'il le fallait. Enfin, je remercie tous les amis qui m'ont soutenu durant ces dernières années.

Table des matières

Chapitre 1

Introduction

Animer une machine générique de différentes façons pour des besoins multiples: telle fut l'idée majeure qui allait conduire l'informatique vers des succès croissants. La proposition n'est pas récente, Charles Babbage l'avait pressentie dès 1833. Cependant, ce n'est qu'en 1945 que Von Neumann, poursuivant l'idée de Babbage, suggère d'enregistrer dans la mémoire des ordinateurs, encore balbutiants, des instructions de traitement en même temps que les données. Von Neumann introduit un modèle très simple d'ordinateur séquentiel: une unité de calcul reliée à une mémoire et commandée par un programme stocké dans cette même mémoire. En 1948 apparaissent les premiers assembleurs. L'ordinateur s'affranchit progressivement, mais dans le même élan, de la mécanographie et surtout du fonctionnement câblé. En séparant le matériel du logiciel, on obtient un instrument universel et facilement adaptable aux besoins des utilisateurs. Bien plus, l'unicité du modèle permet d'envisager facilement le portage des applications d'une machine vers une autre, et ainsi, de bénéficier simultanément des gains de performances du matériel et du logiciel.

S'appuyant sur un modèle unique d'architecture, l'évolution du matériel est ponctuée par les sauts technologiques de l'électronique. A la base de cette évolution vers la puissance se situe le passage de la mécanographie à l'ère électronique. Viennent ensuite les progrès réalisés sur les semi-conducteurs. Un saut technologique comparable, rendant caducs les microprocesseurs actuels, parait difficilement envisageable à moyen terme. Tout en ne remettant nullement en cause l'intérêt des progrès sur les architectures séquentielles, l'informatique parallèle propose de s'éloigner de l'architecture des ordinateurs séquentiels dans le but d'atteindre de meilleures performances. Elle propose de coupler plusieurs éléments de calcul et de les faire coopérer à une même tâche, afin de réduire le temps de calcul. En l'état actuel, cette approche n'a pas (encore?) su inventer un modèle qui puisse concurrencer le modèle de Von Neumann en terme d'universalité. Cependant l'objectif reste le même: cumuler indépendamment les gains de performances du matériel et du logiciel. Pour cela la programmation doit s'affranchir du matériel. La tâche du programmeur ne doit pas s'avérer plus complexe que dans le cadre séquentiel sous peine de décourager programmeurs et investisseurs s'ils sont confrontés à un effort disproportionné quant au gain attendu.

1.1 Programmer les machines parallèles actuelles

Dans les machines MIMD (Mutiple Instruction stream, Mutiple Data stream), chaque noeud peut être assimilé à une machine séquentielle autonome, ce qui les rend aisément modulaires. Cela permet en outre de bénéficier directement des apports technologique du séquentiel par intégration des mêmes composants. Ces caractéristiques font qu'elles sont de loin les plus répandues actuellement.

Chaque noeud possède sa propre mémoire et sa propre unité de contrôle et de calcul. Les données sont réparties sur les processeurs. Les noeuds sont reliés par un réseau d'interconnexion dont la topologie varie d'une machine à l'autre et, de plus en plus souvent, au sein même de la machine. Pour peu que la topologie du réseau le permette, ces machines sont aisément extensibles.

Le schéma de coordination entre les tâches est très peu contraint. La maîtrise des multiples flots de contrôle passe par la gestion manuelle, par le programmeur, ou automatisée de l'ordonnancement des différentes tâches à accomplir. Le programmeur doit imaginer l'ensemble des exécutions et leurs interactions. Il doit éviter les interblocages. Son labeur est rendu plus ardu par l'impossibilité de reproduire l'entrelacs des exécutions d'une session à l'autre (indéterminisme), qui complique singulièrement le travail de déboguage. Au final, rien ne garantit que le programme résultant offre les meilleurs performances.

1.2 Décharger le programmeur du parallélisme

Devant les difficultés de la programmation parallèle explicite des machines MIMD naît naturellement l'idée de dégager au maximum le programmeur des tâches de gestion du parallélisme. Le programmeur crée ses programmes dans le modèle de programmation qui lui est le plus familier: le modèle séquentiel. Son programme est ensuite automatiquement transformé en programme parallèle.

La parallélisation automatique d'un programme séquentiel comporte la plupart du temps plusieurs étapes successives. Il faut détecter les dépendances de données, transformer le code pour augmenter le parallélisme potentiel, exprimer le parallélisme en ordonnançant les tâches et en distribuant à la fois tâches et données, puis enfin générer le code parallèle. Chaque méthode est généralement bien adaptée à une classe plus ou moins restreinte de programmes séquentiels et à un langage cible. Il est plus délicat de proposer une méthode à la fois générale, prouvable et ouvrant la voie à de bonnes performances.

Décharger totalement le programmeur de la gestion du parallélisme revient à proposer un compilateur qui prendrait en entrée un code totalement séquentiel, sans indication de distribution des données et fournirait en retour un code parallèle efficace. L'équipe de recherche de P. Feautrier au laboratoire PRiSM de l'université de Versailles a montré qu'il est possible d'apporter une solution à ce problème pour les programmes à contrôle statique constitués de boucles DO et d'affectations [21, 22]. Les seules structures de données autorisées sont les tableaux de dimensions arbitraires. Les indices des tableaux ainsi que les bornes des boucles sont des fonctions affines des compteurs des boucles. L'approche

est basée sur l'analyse exacte des domaines d'itérations des nids de boucles, des ensembles d'opérations, et des dépendances de données entre les opérations. De cette analyse, on tire un ordonnancement respectant le flot de données. Afin de simplifier l'expression de l'ordonnancement, une partie du parallélisme est sacrifié lors de cette phase [21]. Une phase supplémentaire de mise en assignation unique permet de supprimer les dépendances qui ne sont pas dues au calcul mais à son implantation (*fausses dépendances*). Des techniques ont été développées pour réduire le coût en mémoire de l'assignation unique [66]. Un fois l'ordonnancement construit, on génère un code proche du modèle dataparallèle, conduisant à l'exécution de «fronts» d'opérations indépendantes, correspondant aux opérations ayant la même date dans l'ordonnancement. L'approche est séduisante car elle fournit une réponse optimale pour une large classe de programmes à schémas de dépendances prévisibles.

Lorsque le contrôle séquentiel est dynamique, le schéma de dépendances ne peut être totalement déterminé statiquement. Collard [16] a proposé une extension des concepts de la parallélisation des programmes à contrôle statique vers une classe de programmes à contrôle dynamique. Une analyse floue du flot de données (FADA) permet de déterminer la réunion de toutes les écritures dont peut dépendre une lecture, pour toutes les exécutions possibles. Cette analyse est ensuite exploitée via des méthodes de construction d'ordonnancement et de génération de code adaptées au contrôle dynamique. Collard propose d'exploiter sa méthode dans le cadre d'exécutions spéculatives de boucles while. L'approche reste cependant limitée aux programmes dont les indices de tableaux sont des fonctions affines des compteurs de boucles. Elle ne peut être appliquée ni dans le cadre non affine ni dans le cas de communications imprévisibles.

D'autre part, les méthodes précédemment citées génèrent un code fortement synchronisé proche du modèle dataparallèle. Leur efficacité est subordonnée à la qualité du compilateur prenant en charge le code issu de la parallélisation. Une partie du parallélisme est dédiée à des compilateurs tels que ceux de Fortran HPF [15, 35, 40] ou encore ADAPTOR [9, 28]. La désynchronisation du code dataparallèle produit reste souvent extérieure au processus de parallélisation automatique, rendant par là même difficile une approche modulaire fondée sur un modèle théorique unique.

1.3 Accès imprévisibles ou irréguliers aux données

1.3.1 Analyse à l'exécution

La modélisation de la dynamique des fluides ou de la dynamique des molécules sont des exemples importants d'applications scientifiques qui nécessitent des accès non affines aux données. Elles utilisent typiquement des tableaux d'index (ou tableaux d'indirections) qui introduisent un niveau d'indirection dans l'accès aux tableaux. Dans ce cas, les méthodes de parallélisation précédemment décrites ne suffisent plus puisqu'il est impossible de prédire à la compilation l'ordonnancement entre les opérations ainsi que les transferts de données rendus nécessaires par la distribution de ces opérations sur les processeurs.

Plusieurs approches différentes ont été proposées. Elles sont basées sur des techniques d'analyse à l'exécution utilisant le mécanisme inspecteur/exécuteur. La technique de l'inspecteur/exécuteur [15, 24, 52, 53, 56, 58, 63, 68] décompose la parallélisation des boucles en deux phases distinctes.

Lors de la première phase, l'inspecteur, chaque processeur recense les références qu'il doit effectuer et détermine celles qui portent sur des données distantes. Simultanément il construit une table d'accès aux données dans laquelle est assignée une adresse locale pour chaque donnée distante. C'est à ces adresses que seront écrites les données communiquées. Cette information est échangée entre les processeurs lors d'une phase de communication globale. Chaque processeur peut alors construire son propre schéma de communication et de synchronisation, compatible avec celui des autres.

La seconde phase de la méthode, l'exécuteur, exécute la boucle en parallèle en appliquant le schéma de communication et de synchronisation fourni par la phase inspecteur. Chaque donnée transmise est écrite dans la table d'accès aux données. L'exécuteur remplace chaque référence à distance par une référence dans cette table.

La méthode inspecteur/exécuteur peut être appliquée à des boucles ne présentant pas de dépendance inter itération. Dans ce cas son utilité se borne à la gestion de schémas de communications imprévisibles. Les communications peuvent être rassemblées au début de la phase exécuteur [68]. Les projets de compilateurs HPF tels que PREPARE, Vienna Fortran Compilation System, Fortran D intègrent largement cette approche pour la compilation de boucle dataparallèle présentant des accès irréguliers [10, 11, 15, 18, 19, 65, 67].

Si la boucle séquentielle comporte des dépendances inter itérations (boucles DOACROSS) alors l'exécuteur permet de générer un code à passage de message. L'ordonnancement des tâches est alors lié aux communications [18]. Dans le cas d'une mémoire partagée, l'ordonnancement dégagé lors de la phase inspecteur peut aussi être utilisé par la phase exécuteur de façon à parcourir les itérations en respectant des fronts d'ondes séparés par des barrières de synchronisation globales. Il est possible d'affiner cette approche en remplaçant les synchronisations par des attentes actives sur des signaux décrivant la progression des accès aux variables [38, 69].

L'efficacité de la méthode est nécessairement limitée par le coût de la phase *inspecteur*. Elle est particulièrement intéressante lorsque le schéma de communication est répétitif (*static irregular problems*). Dans ce cas il est possible de réutiliser la même phase inspecteur pour plusieurs phases exécuteur. Kennedy a exposé l'intérêt de cette approche qui a été implantée dans la librairie PARTI [18]. Il montre qu'il est possible d'étendre l'approche à des schémas de communication non répétitifs lorsque, d'une exécution à l'autre, l'ordonnancement ne change que partiellement et de façon incrémentale. Il est alors possible d'économiser la communication des données qui ont été transmises lors de phases de calcul antérieures et qui n'ont pas été modifiées depuis.

Lorsque le schéma de communication change à chaque exécution (*adaptive problems*), la phase inspecteur ne peut être réutilisée. Saltz suggère alors de tirer parti de certaines spécificités sémantiques de l'algorithme à paralléliser afin d'optimiser l'inspecteur en allégeant la génération de l'ordonnancement et en proposant une méthode rapide de migration de données (méthodes de cache, vectorisation des communications) [58]. Cette approche

a été implantée dans la librairie CHAOS qui est un surensemble de PARTI. Ces librairies étendent le compilateur Fortran D [24] à la gestion des boucles à accès irréguliers.

1.3.2 Utilisation de signaux

Si le code séquentiel conduit à des communications totalement irrégulières, aucune des méthodes précédemment décrites ne permet de le paralléliser efficacement. Il est alors possible d'utiliser des communications initialisées unilatéralement par les processeurs qui ont besoin de valeurs distantes. Ce schéma conduit à insérer au sein de la boucle des barrières de synchronisations globales qui permettent de respecter les antidépendances ainsi que les dépendances inter et intra itérations. Prakash [54] a montré qu'il est possible de supprimer ces barrières au prix d'une expansion mémoire proportionnelle au gain en termes d'asynchronisme. Lorsqu'un processeur doit lire une donnée distante, il émet une requête vers le processeur calculant cette donnée. Celui-ci réceptionne la requête, puis diffère la réponse jusqu'à ce que la donnée soit disponible. Lorsque la donnée est calculée, le processeur émetteur peut alors répondre aux requêtes courantes et à venir concernant cette donnée. La résolution d'une dépendance nécessite deux communications.

Un autre mécanisme de synchronisation de boucles DOACROSS par envoi de signal est décrit par Hwang [37]. Contrairement à l'approche de Prakash, ici l'analyse est statique. Une instruction d'envoi de signal $Send\_signal(S)$ suit chaque instruction source d'une dépendance S. Symétriquement une attente $Wait\_signal(S)$ est insérée avant le puits de la dépendance S. Les synchronisations sont de type point à point unilatérales. Hwang propose de faire migrer les instructions de synchronisation afin d'optimiser leur recouvrement par le calcul. La méthode suppose qu'il est possible de connaître statiquement la position dans le code de chaque source et puits de dépendance. Elle peut difficilement être appliquée dans les cas où le schéma de dépendance varie dynamiquement.

1.3.3 La résolution dynamique des accès

Lorsque l'accès aux données est très irrégulier et que le contrôle devient trop complexe on utilise de manière standard la distribution de programmes dirigée par les données (*data-driven distribution*). Ses principes de base ont été définis par D.Callahan et K.Kennedy [12]. Depuis, cette approche a été largement étudiée à l'université de Rennes dans le cadre du projet Pandore [14, 2, 26, 43]. Nous allons en exposer les grands principes.

La distribution de programmes dirigée par les données s'appuie sur la répartition par l'utilisateur des données dans un espace de nommage global. Le code parallèle généré est de type SPMD. Il conserve la structure du code séquentiel. Chaque processeur exécute des portions différentes dépendant des données qui lui sont propres. Le choix est effectué suivant le *principe des écritures locales*(Owner Compute Rule), par le biais de gardes. Si un processeur doit modifier une variable en fonction de variables distantes, la mise à jour de ces dernières est assurée par le processus de compilation qui produit les communications ad hoc.

Le schéma de compilation varie selon le type de communication visé: unilatérale ou bilatérale.

- **Communications unilatérales.** C'est le type de communication de base pour les machines à mémoire partagée [43]. Chaque processeur lit dans la mémoire d'un processeur distant (*get*). Synchronisations et communications sont totalement découplées. Le respect des dépendances est garanti par le biais de synchronisations apparaissant explicitement dans le code de l'émetteur et celui du récepteur.

  L'exemple suivant illustre la méthode de compilation. On considère le programme séquentiel:

$$\begin{aligned} x &:= 2 * y \\ z &:= x + 1 \\ y &:= 3 \end{aligned}$$

  Pour chaque variable V, la fonction $Owner(V)$ renvoie le processeur propriétaire de V.

$$\begin{aligned} &Owner(x) = 1 \\ &Owner(y) = 2 \quad \{Les\ variables\ sont\ distribuées\ sur\ les\ processeurs\} \\ &Owner(z) = 3 \\ &Owner(w) = 4 \end{aligned}$$

  Le processeur 1 doit lire la valeur de y dans la mémoire de 2, puis effectuer la première instruction. Le processeur 3 doit lire la valeur de x dans la mémoire de 1, puis effectuer la deuxième instruction. Le processeur 2 n'a rien à lire en mémoire distante pour pouvoir exécuter la troisième instruction. Le tableau qui suit résume les pas de calcul selon le modèle PRAM.

Code séquentiel	*processeur* 1	*processeur* 2	*processeur* 3	*processeur* 4
$x := 2 * y$	*lecture de* y *depuis* 2 $x := 2 * y$			
$z := x + 1$			*lecture de* x *depuis* 1 $z := x + 1$	
$y := 3$		$y := 3$		

Remarques

▷ Nous constatons que le processeur 3 ne peut pas lire la variable x avant que celle-ci ait été affectée par le processeur 1. Ces deux instructions génèrent une dépendance de flot. Si l'exécution est asynchrone, une synchronisation est indispensable entre les deux instructions.

▷ Le processeur 2 ne doit pas modifier la valeur de y avant que la première instruction ne l'ait lue. Si l'exécution est asynchrone, cette dépendance en arrière (ou antidépendance) entre la première instruction et la troisième impose une synchronisation.

Le respect des dépendances peut être imposé par des synchronisations point à point insérées avant et après chaque instruction et ne concernant que les processeurs qui possèdent des variables lues ou affectées par l'instruction concernée.

Code séquentiel	*processeur* 1	*processeur* 2	*processeur* 3	*processeur* 4
	synchronisation entre (1, 2)	*synchronisation entre* (1, 2)		
	lecture de y *depuis* 2			
$x := 2 * y$	$x := 2 * y$			
	synchronisation entre (1, 2)	*synchronisation entre* (1, 2)		
	synchronisation entre (1, 3)		*synchronisation entre* (1, 3)	
			lecture de x *depuis* 1	
$z := x + 1$			$z := x + 1$	
	synchronisation entre (1, 3)		*synchronisation entre* (1, 3)	
$y := 3$		$y := 3$		

Si les communications sont irrégulières, il n'est pas possible de connaître statiquement l'ensemble des synchronisations nécessaires. Pour conserver alors la cohérence des données, des synchronisations globales sont utilisées. Lorsque le contrôle est dynamique chaque processeur doit emprunter le même chemin dans les structures de contrôle afin de participer à toutes les synchronisations globales. Notons que Utard et Hains [30] ont proposé un mécanisme d'absorption des barrières de synchronisations permettant de réduire le nombre de synchronisations globales dans les boucles dataparallèles.

Le code généré est structuré en phases séparées par des synchronisations globales. Cette structure est proche du modèle BSP [46, 64]. Ainsi, ce dernier apparaît comme un candidat possible en tant que modèle cible d'une traduction de code à schéma de dépendance irrégulier. Par contre, les exécutions demeurent fortement synchrones, limitant par là même l'intérêt de la méthode.

- **Communications bilatérales.** Les communications passent par l'envoi de messages qui transitent par des canaux unidirectionnels FIFO. La réception est bloquante alors que l'envoi ne l'est pas. Les valeurs envoyées sont disponibles dans le tampon de réception. Si la distribution n'est pas connue, ou si le schéma de communication ne peut être prévu, on doit alors contrôler l'émission et la réception de messages à l'exécution par le biais de gardes.

  L'exemple suivant illustre cette technique avec un programme comportant une unique

affectation, où les communications échappent à une analyse statique: $A[f(i)] := B[g(i)]$:

Code séquentiel	*Code SPMD*

$A[f(i)] := B[g(i)]$	*Si leprocesseur courant possède* $B[g(i)]$ *alors*
	  envoyer $B[g(i)]$ *au* $Owner(A[f(i)])$
	Si leprocesseur courant possède $A[f(i)]$ *alors*
	  begin
	  recevoir$B[g(i)]$ *depuis* $Owner(B[g(i)])$
	  $A[f(i)] := B[g(i)]$
	  end

L'inconvénient principal avec cette approche est l'évaluation obligatoire par tous les processeurs de toutes les gardes apparaissant dans le flot de contrôle. Lorsque le contrôle est dynamique chaque processeur risque ainsi de parcourir de nombreuses structures de contrôles "inutiles", grevant ainsi le coût de l'exécution. Des optimisations ont toutefois été proposées [26].

Un cadre formel pour la preuve de correction des techniques de parallélisation dirigées par les données a été mis en place dans le cadre du projet Pandore. Les exécutions sont modélisées par le biais d'automates communicants. La preuve de la correction est basée sur la notion de confluence. Claude Jard et Cyrille Bareau ont montré qu'il est possible d'en déduire certaines propriétés de régularité sur le graphe d'états des programmes générés [2, 4]. En particulier, ils montrent que toutes les exécutions possibles du programme parallèle, à partir d'un état initial donné, possèdent la même relation de causalité au sens de Lamport [41]. De plus cette relation de causalité est identique à celle du programme séquentiel source. Claude Jard et Cyrille Bareau ont montré qu'il est possible de calculer les dépendances de données au sein d'une exécution distribuée à partir de la sémantique d'une exécution séquentielle en lui greffant un mécanisme d'estampillage [4]. Les estampilles ainsi crées sont des horloges vectorielles « à la Fidge-Mattern » [23, 45] qui sont celles qui seraient générées lors d'une exécution du programme parallélisé. Cette information est particulièrement utile pour effectuer des mesures de concurrence et du déboguage pour la performance.

1.4 Vers un modèle cible structuré en couches faiblement synchronisées

En programmation séquentielle, la structure d'un programme reflète directement sa sémantique. Au contraire, dans le cadre du parallélisme de tâches, la correspondance entre émissions et réceptions ne suit pas nécessairement la structure syntaxique du programme. La multiplicité des flots de contrôle peut conduire à une explosion combinatoire compliquant la maîtrise des programmes et leur validation formelle [25]. Le déterminisme et l'absence de blocage ne sont pas intrinsèquement garantis, compliquant par là même la preuve de correction de la traduction de code séquentiel en code parallèle.

Le modèle de programmation dataparallèle, quant à lui, autorise une lecture séquentielle des programmes, guidée par la syntaxe. L'expression de la concurrence est restreinte à des accès parallèles sur des tableaux. Ces actions sont exécutées en séquence («`SEQ` de `PAR`») à l'image du flot de contrôle unique des programmes séquentiels [5]. L'unicité du flot de contrôle permet de garantir l'absence de blocage, le déterminisme et facilite la validation formelle des schémas de compilation [30] et les preuves de programmes [6, 7, 31]. L'inconvénient principal de cette approche réside dans l'impossibilité d'exprimer des schémas fins de synchronisation.

Nous montrons qu'il est possible de proposer un cadre unifié, appelé $\mathcal{SCP}$ (Structural Clock Programming), pour la traduction de code séquentiel en code parallèle. Notre approche consiste à structurer les programmes de telle sorte que les informations de dépendance et d'indépendance entre les instructions soient représentées par la syntaxe. La notion intuitive de précédence entre les instructions est formalisée par un ordre logique fondé sur leur position dans la structure du programme. L'ordre logique est encodé par des compteurs multi niveaux, les *horloges structurelles*. L'empilement des compteurs correspond au niveau d'imbrication courant de chaque processeur dans les structures de contrôle. Il autorise le masquage temporaire de l'exécution des structures de contrôle dynamiques. Pour garantir le déterminisme des calculs, nous introduisons une gestion dynamique de la mémoire, associant des horloges à l'empilement de la valeur d'une variable après chaque affectation.

Comme dans le modèle BSP, le programme est structuré en couches séparées par des étapes de mise à disposition des dernières valeurs calculées. Toutefois, à la différence de BSP, aucune synchronisation globale n'est nécessaire. Chaque couche du programme correspond à un groupe d'instructions exécuté *indépendamment*, de manière asynchrone, par chaque processeur. La machine abstraite sous-jacente permet des exécutions faiblement synchronisées. Il est possible de définir un modèle théorique simple fondé sur des sémantiques opérationnelle et dénotationnelle dirigées par la syntaxe, facilitant ainsi les preuves fondamentales de correction, de déterminisme et d'absence de blocage.

La traduction de code séquentiel en code parallèle est fondée sur la distribution des données. Les programmes agissent sur un ensemble de variables réparties sur les processeurs. A chaque processeur est associé un ensemble de variables qui lui appartient en propre. La traduction du code est basée sur le principe des écritures locales (Owner compute rule). Le processeur responsable d'une écriture est celui qui possède la variable écrite. L'instruction primitive est l'affectation gardée, garantissant ainsi l'indépendance entre le code et la distribution des données. Les communications sont implicites et correspondent à des lectures en mémoire distante, facilitant par là même la résolution dynamique des accès dans le cas de dépendances imprévisibles. L'empilement des mémoires permet de supprimer automatiquement les anti-dépendances par un mécanisme dynamique d'assignation unique. La gestion des synchronisations par les horloges structurelles permet une expression fine des dépendances irrégulières. Grâce au schéma de communication dirigé par la syntaxe, la traduction de code séquentiel en code $\mathcal{SCP}$ est modulaire. Elle utilise les lectures distantes pour résoudre les dépendances à l'exécution, permettant ainsi de traiter des problèmes où une analyse statique des dépendances se révèle impossible.

L'inconvénient principal de cette approche réside dans le coût en espace mémoire dû à l'empilement des valeurs intermédiaires. Nous montrons qu'il est possible d'y remédier en remplaçant l'empilement systématique par un empilement *ad hoc*. Ceci nous conduit, tout en conservant le principe des communications dirigées par la structure du programme, à remplacer les lectures en mémoire distantes par le mécanisme de communications explicites par passage de message du langage $\mathcal{SCL}-\mathcal{C}han$ (*Structural Clock Language Channel*). Comme en $\mathcal{SCP}$, la précédence entre les instructions à l'exécution respecte un ordre logique fondé sur leurs positions dans la syntaxe du programme. La machine abstraite de $\mathcal{SCL}-\mathcal{C}han$ reprend le même mécanisme de synchronisation et de masquage de l'exécution d'une structure de contrôle pour un processeur distant, basé sur les horloges structurelles.

Les différences majeures entre $\mathcal{SCP}$ et $\mathcal{SCL}-\mathcal{C}han$ résident dans le mode de communication et dans la structure des programmes.

Les communications de $\mathcal{SCL}-\mathcal{C}han$ sont explicites, asynchrones et par passage de messages à travers des canaux unidirectionnels point à point. Nous définissons un canal par variable et par paire de processeurs. L'échange de données entre les processeurs repose sur un mécanisme d'envois et réceptions explicites. Ainsi, une donnée attribuée à un processeur devient disponible pour un processeur distant seulement si cette donnée est explicitement envoyée à ce processeur, évitant ainsi l'empilement de données inutiles. Le récepteur attend que l'émetteur ait exécuté toutes les instructions d'envoi *précédant* l'instruction de réception. Cette gestion évite ainsi les blocages causés par l'absence d'envoi correspondant à la réception. Les envois asynchrones accumulent les données en transit dans les canaux qui sont gérés en mode LIFO. Les anciennes valeurs envoyées sont absorbées automatiquement pour ne recevoir que la plus récente au sens de l'ordre logique de façon à respecter le schéma de communication de $\mathcal{SCP}$. Dans ce modèle, il n'y a pas de correspondance obligatoire entre les émissions et les réceptions.

Comme avec le modèle $\mathcal{SCP}$, la traduction de code séquentiel en code parallèle est fondée sur la distribution des données. Elle suit le principe des écritures locales. L'exécution des affectations est explicitement gardée. Des gardes sont insérées pour gérer dynamiquement les communications. Le gain en place mémoire risque alors d'être payé au prix d'un coût supplémentaire induit par l'évaluation de toutes les gardes de communications par tous les processeurs. Toutefois si une analyse statique, même partielle, de dépendance est praticable, il est possible de tirer partie de la non correspondance entre les émissions et les réceptions pour réduire le nombre de gardes. Le parcours de structures de contrôle «inutiles» est alors épargné aux processeurs qui n'ont aucun calcul à y réaliser.

En conclusion, nous discutons l'intérêt comparé des approches $\mathcal{SCP}$ et $\mathcal{SCL}-\mathcal{C}han$ du point de vue de l'optimisation de la traduction automatique de code séquentiel en code parallèle. Le modèle $\mathcal{SCL}-\mathcal{C}han$ paraît bien convenir aux programmes permettant l'optimisation des gardes d'émissions grâce à une analyse statique des dépendances. Au contraire, le mode d'accès du langage $\mathcal{SCP}$ le rend bien adapté lorsque cette optimisation échoue, au prix toutefois d'une augmentation du coût de la gestion mémoire.

1.5 Structure de la Thèse

Le chapitre 2 introduit le modèle $\mathcal{SCP}$, les notions d'ordonnancement logique entre les groupes d'instructions, les horloges structurelles, l'ordre sur les horloges et la gestion dynamique de la mémoire. Nous présentons le schéma de traduction de code séquentiel en code $\mathcal{SCP}$. Nous terminons ce chapitre en montrant qu'il est possible d'utiliser la machine abstraite de $\mathcal{SCP}$ pour fournir à BSP un modèle d'exécution faiblement synchronisé.

La construction d'un cadre formel et les preuves relatives au langage $\mathcal{SCP}$ sont abordées dans les chapitres 3 et 4. Nous présentons une sémantique opérationnelle (modèle d'exécution) et une sémantique dénotationnelle (modèle de programmation). La sémantique opérationnelle permet de prouver le déterminisme et l'absence de blocage. L'équivalence entre les deux sémantiques valide le modèle de programmation et sa machine abstraite.

Le chapitre 5 introduit $\mathcal{SCL}-\mathcal{C}han$ et son modèle de communication à passage de messages structuré par la syntaxe. Nous présentons les propriétés fondamentales de ce langage avant de donner le schéma de traduction de code séquentiel en code $\mathcal{SCL}-\mathcal{C}han$. Nous introduisons des exemples de traductions et nous proposons des optimisations en terme d'augmentation de la granularité des gardes, de réduction des espaces d'itérations et de réduction du volume des communications.

Dans le chapitre 6, nous présentons des tests de performance sur Cray T3D basés sur des comparaisons avec le code généré par le compilateur PAF et traduit ensuite en CRAFT.

Les conclusions et perspectives constituent le dernier chapitre.

Chapitre 2

Le modèle $\mathcal{SCP}$

La section 2.2.1 de ce chapitre a fait l'objet de deux publications cosignées avec Yann Le Guyadec, Xavier Rebeuf, Bruno Raffin et Bernard Virot [32, 34].

Le but de ce chapitre est la présentation du modèle $\mathcal{SCP}$ et la construction de la fonction de traduction de code séquentiel en code $\mathcal{SCP}$. Il est divisé en six parties.

La première partie présente informellement le langage $\mathcal{SCP}$ (*Structural Clock Programming*). Le code est de type SPMD (Single Program Multiple Data) et la distribution des calculs est basée sur le principe des écritures locales (Owner compute rule). Les variables étant réparties sur les processeurs, le processeur responsable d'une écriture est celui qui possède la variable écrite. L'instruction primitive est l'affectation gardée.

Dans la deuxième partie nous formalisons l'idée intuitive de précédence entre les instructions. A cette fin nous définissons un ordre logique entre les instructions, fondé sur leurs positions dans la structure syntaxique du programme. Il permet de comparer, de manière cohérente, les positions relatives des processeurs au cours de l'exécution de structures de contrôle dynamiques, facilitant par là même la synchronisation dans les cas d'accès rendus imprévisibles par l'existence de ces structures dynamiques.

La troisième partie décrit l'accès aux données. Les communications sont implicites et correspondent à des lectures en mémoire distante, facilitant par là même la résolution dynamique des accès. La sémantique de l'évaluation des expressions est dirigée par l'ordre logique. On garantie ainsi le déterminisme des exécutions faiblement synchrones des programmes $\mathcal{SCP}$.

La quatrième partie présente la machine abstraite de $\mathcal{SCP}$. Elle est basée sur un codage dynamique de l'ordre logique par des compteurs multi niveaux, les *horloges structurelles*. L'empilement des niveaux correspond au niveau d'imbrication courant de chaque processeur dans les structures de contrôle. Il autorise le masquage temporaire, pour un processeur distant, de l'exécution d'une structure dynamique. Le mécanisme d'évaluation dirigé par l'ordre logique est implanté par une gestion dynamique de la mémoire. Les valeurs intermédiaires de chaque variable sont datées par des horloges structurelles puis empilées.

Dans la cinquième partie nous montrons qu'il est possible d'utiliser $\mathcal{SCP}$ en tant que

langage cible d'une traduction de code séquentiel en code parallèle. Cette traduction est modulaire puisque basée sur un schéma d'accès distants dirigé par la syntaxe. Elle utilise les lectures distantes pour résoudre à l'exécution les dépendances, évitant ainsi une analyse de dépendance. Le principe des écritures locales est mis en application par le biais des gardes implicites de $\mathcal{SCP}$. Le programme distribué est faiblement synchronisé. Nous montrons sur des exemples que $\mathcal{SCP}$ peut être associé à un modèle élémentaire de coût, que nous appelons diagramme de dépendance, tenant compte de la distribution des données. Il permet de choisir le placement de façon à minimiser les attentes dues au mécanisme de synchronisation. L'idée intuitive est de recouvrir autant que possible ces attentes par celles indispensables au respect du flot de données.

Dans la dernière partie, nous montrons qu'il est possible de réutiliser la machine abstraite de $\mathcal{SCP}$ pour offrir à BSP un modèle d'exécution faiblement synchronisé.

2.1 Structure d'un programme $\mathcal{SCP}$

Dans cette section nous présentons le langage $\mathcal{SCP}$et sa sémantique informelle. La machine abstraite sous-jacente est parallèle et asynchrone. Elle comporte Pe_Nb processeurs. Les processeurs, appelés indices, sont notés u, v,.... Chaque indice possède une mémoire locale. L'ensemble Var des variables scalaires est réparti sur les mémoires locales des indices. L'espace de nommage est global. La distribution des données sur les indices est exprimée par le biais de la fonction $Owner$. L'expression $Owner(V)$ note l'indice possédant la variable V. Afin de manipuler les indices de boucle de façon uniforme sur l'ensemble des indices, nous introduisons un ensemble de variables vectorielles Ind : chaque indice possède une composante de chaque variable de Ind. Si $X \in Ind$, $X|_u$ note la composante de X à l'indice u. Nous introduisons la constante vectorielle $This$ telle que $This|_u = u$.

Le langage $\mathcal{SCP}$ est un langage de coordination de programmes SPMD écrits dans un langage impératif hôte: $\mathcal{LH}$. Les programmes écrits en $\mathcal{LH}$ respectent la règle des écritures locales: les indices exécutent une affectation de $\mathcal{LH}$ uniquement si la variable dans le membre gauche leur appartient.

Un programme $\mathcal{SCP}$ est formé de groupes d'instructions $\mathcal{LH}$. La structure de contrôle step permet de rassembler plusieurs instructions $\mathcal{LH}$ en un seul groupe. Le groupe d'instructions constitue la structure la plus fine de $\mathcal{SCP}$. Tout groupe d'instructions est un programme $\mathcal{SCP}$. Les groupes d'instructions du langage $\mathcal{LH}$ et les structures de contrôles constituent les instructions du langage $\mathcal{SCP}$. On construit inductivement les programmes $\mathcal{SCP}$ à partir des groupes et des structures de contrôle suivantes.

Séquence: $S;T$**.** Un indice exécute le programme T après avoir terminé l'exécution de S.

La conditionnelle exprime la notion d'indépendance entre des groupes d'indices.

Conditionnelle : where B do S elsewhere Q end**.** Un indice qui évalue la condition B à $vrai$ exécute le programme S, sinon il exécute Q.

La quatrième structure de contrôle, la boucle, permet d'exprimer la notion d'itération dynamique d'un bloc de code $\mathcal{SCP}$.

La boucle : loopwhere B do S end. : Tant qu'un indice évalue la condition B à *vrai*, il exécute le programme S. Dès qu'il évalue B à $faux$ aucune instruction n'est exécutée et il termine la boucle.

On notera qu'un indice peut sortir de la boucle et poursuivre l'exécution du programme sans attendre les indices qui sont encore dans la boucle.

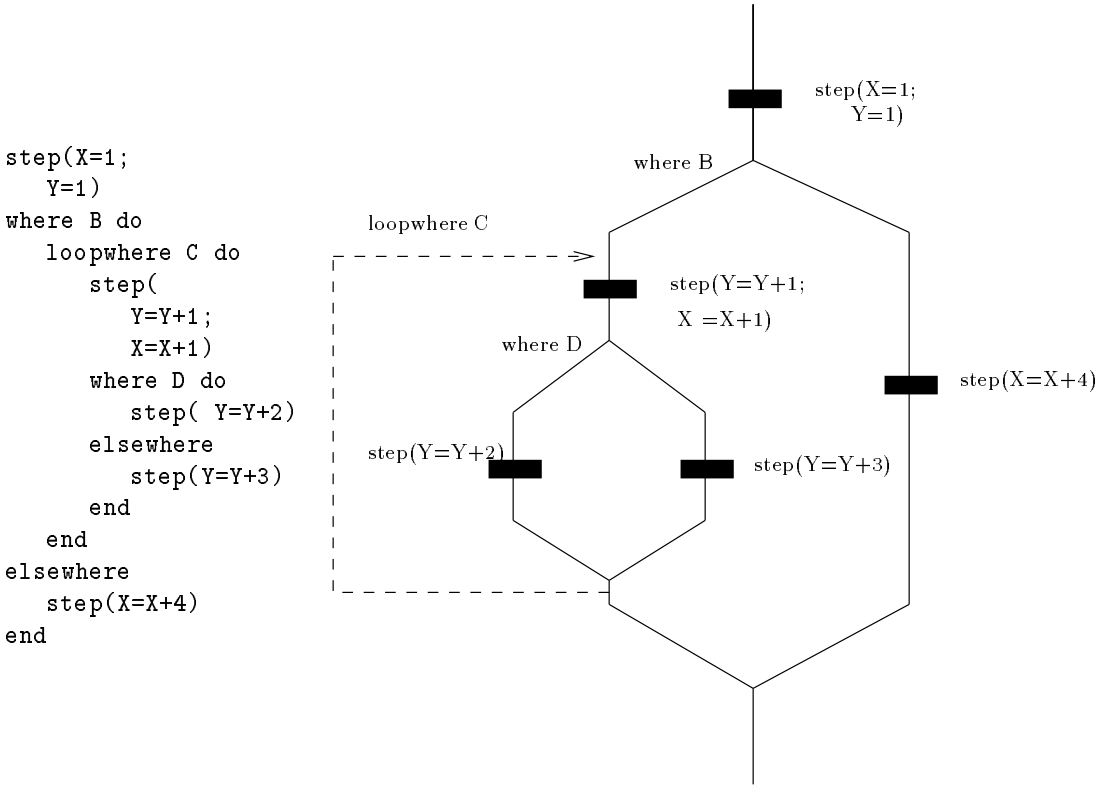


FIG. 2.1 - *Illustration de la structure d'un programme $\mathcal{SCP}$. La partie droite figure le diagramme de Hasse associé au programme. La flèche en pointillés figure le repliement de la boucle*

Exemple 2.1.1 La figure 2.1 illustre la structure d'un programme $\mathcal{SCP}$. Les structures where/elsewhere et loopwhere sont imbriquées les unes dans les autres.

Remarques:

- La structure de groupe ne contient jamais de structures de contrôle $\mathcal{SCP}$.

- Si la condition B d'une boucle loopwhere ne contient que des variables scalaires, tous les indices exécutent toujours le même nombre d'itérations. Par contre si elle contient une variable vectorielle dont les composantes ne sont pas toutes égales, alors il est possible que certain indices exécutent moins d'itérations que d'autres.

2.1.1 Ordonnancement logique des instructions

Nous introduisons dans cette section un ordre logique sur les groupes d'instructions de $\mathcal{LH}$ et les structures de contrôles. Il est basé sur l'ordre d'exécution local à chaque indice. Il formalise l'idée intuitive de précédence entre les instances d'instructions de $\mathcal{SCP}$, les boucles étant supposées déroulées. La séquence entre les instructions de $\mathcal{SCP}$ exprime la précédence entre les instructions de $\mathcal{LH}$. On définit comme incomparables les instructions des deux branches d'un where/elsewhere afin d'exprimer leur indépendance de deux blocs d'instructions.

Définition 2.1.1 (Ordre logique sur les instructions de $\mathcal{SCP}$) *L'ordre logique sur les instructions de $\mathcal{SCP}$ d'un programme P est défini par induction sur la structure du programme.*

Séquence : $S;T$. *Les instructions de S sont* **avant** *celles de T.*

Conditionnelle : where B do S elsewhere T end. *Les instructions de S et T sont* **incomparables**. *Au cours de l'exécution de la conditionnelle, l'évaluation de la condition B est* **avant** *les instructions de S et T.*

Boucle : loopwhere B do S end. *Toute instance d'une instruction du programme S à la i^e itération est* **avant** *toute instance d'une instruction du programme S à la $i{+}1^e$ itération. L'évaluation de B à la première itération est* **avant** *les instances d'instructions de S des autres itérations.*

Extension de l'ordre aux instructions du langage $\mathcal{LH}$. La précédence entre les instances d'instructions du langage hôte est déduite de celle entre les groupes. Il y a absence de précédence entre les instructions qui constituent un même groupe d'instructions. Deux instructions qui appartiennent à deux groupes différents d'instructions ont la même précédence que celle de leurs groupes.

Exemple 2.1.2 Dans le programme de la figure 2.2, le premier groupe d'instructions $\mathcal{LH}$ est **avant** le where/elsewhere. Le premier groupe d'instructions $\mathcal{LH}$ du programme est **avant** les groupes d'instructions $\mathcal{LH}$ des deux branches du where/elsewhere. Les boucles étant supposées déroulées, le where/elsewhere de l'itération $(i, j-1)$ est **avant** le premier groupe d'instructions $\mathcal{LH}$ de l'itération (i, j). Les groupes d'instructions $\mathcal{LH}$ des deux branches du where/elsewhere sont **incomparables**. Les instructions $\mathcal{LH}$ du premier groupe sont **avant** celles des groupes des deux branches du where/elsewhere. Par contre, les instructions $\mathcal{LH}$ des groupes des deux branches du where/elsewhere sont **incomparables**.

```
real :: A(0:N,0:N)
real :: B(0:N,0:N)
step(i=1;
   j=1)
loopwhere i ≤ N do
   loopwhere j≤N do
      step(
         A(i,j)=A(i-1,j)*A(i,j-1);
         X=1)
      where B(i-1,j-1) ≠ 0 do
         step(
            B(i,j)= 1 / SQRT(B(i-1,j-1));
            Y=1 )
      elsewhere
         step(Z=1)
      end
      step(j=j+1)
   end
   step(i=i+1)
end
```

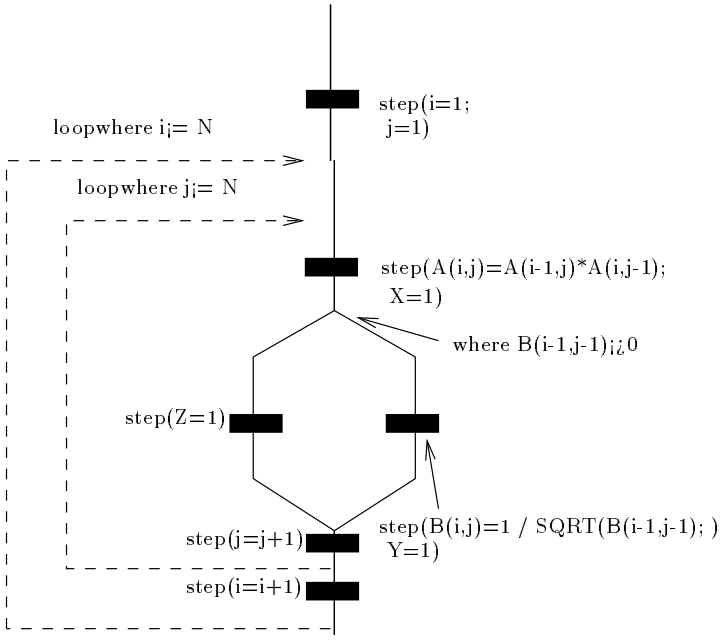


FIG. 2.2 - *Illustration de l'ordre logique sur la structure d'un programme $\mathcal{SCP}$. La partie droite figure le diagramme de Hasse associé au programme. Les flèches en pointillés figurent le repliement*

En identifiant les différentes instances d'instructions on obtient un "ordre logique replié". Il se représente aisément par un diagramme de Hasse calqué sur la structure syntaxique du programme.

2.1.2 L'accès aux données

Dans cette section nous décrivons l'accès aux données dans $\mathcal{SCP}$.

L'ensemble de la mémoire n'est pas directement accessible par un indice distant. Suivant le principe des écritures locales, un indice ne peut affecter une donnée dans une variable que si cette variable est stockée dans sa mémoire locale. De façon similaire, chaque indice peut accéder directement à ses données locales.

L'accès aux données distantes est conditionné par la structure du programme. Intuitivement, une lecture pratiquée au sein d'un groupe d'instructions fait référence à la dernière valeur calculée dans le groupe antérieur le plus récent au sens de l'ordre logique. Ainsi, les écritures effectuées dans un groupe d'instructions ne deviennent visibles pour les autres indices que lors de lectures effectuées dans un groupe ultérieur. Les instructions à l'intérieur d'un même groupe effectuées par deux indices distants sont rendues indépendantes aussi

bien en écriture qu'en lecture par masquage local (au groupe) des modifications.

L'accès à la valeur d'un vecteur est toujours local. L'affectation d'un vecteur dans un scalaire n'induit aucune communication: elle correspond à l'affectation dans le scalaire de la composante du vecteur sur l'indice possédant le scalaire. L'affectation d'un vecteur dans un vecteur n'induit aucune communication: elle correspond à l'affectation composante par composante des deux vecteurs. Au contraire, l'affectation d'un scalaire dans un vecteur induit des communications: elle correspond à l'affectation de toutes les composantes du vecteur par la valeur du scalaire.

Exemple 2.1.3

```
real :: a
real :: b
step( a=1 );
step(
   a=2;
   b=a
)
```

$\neq$

```
real :: a
real :: b
step( a=1 );
step( a=2 );
step( b=a )
```

Les deux programmes ci-dessus ne diffèrent que par le regroupement des instructions.

- Dans le programme de gauche, selon la distribution des données, l'indice effectuant l'instruction `b=a` reçoit, ou bien la valeur de `a` avant le second groupe d'instructions, ou bien celle après l'exécution de `a=2`. Si `a` et `b` sont placés sur le même indice, la valeur de `a` lue par l'indice exécutant l'instruction `b=a` est 2, sinon c'est 1.
- Dans le programme de droite, l'instruction `a=2` est **avant** l'instruction `b=a`. La distribution ne modifie pas le résultat calculé.

Remarques.

1. Si l'on désire éviter que la sémantique du programme dépende de la distribution on ne doit placer au sein d'un même groupe que des instructions qui ne sont pas en dépendance. C'est le cas pour le programme de droite.
2. La notion de groupe sert à exprimer l'indépendance d'un ensemble d'instructions du langage hôte placées sur des indices différents.

   Un groupe constitué de deux instructions placées sur un même indice est équivalent à une séquence de deux groupes.
3. Noter que l'indépendance de deux instructions d'un même groupe n'est garantie que si elles sont placées sur des indices différents.

4. Un autre choix aurait pu être de garantir l'indépendance de deux instructions d'un même groupe en imposant que les lectures en mémoire locale portent sur les valeurs calculées avant le groupe courant. Mais cette solution interdit de fusionner les groupes en tenant compte de la distribution des données. Nous verrons que cette possibilité est un atout important pour optimiser les synchronisations (Voir exemple 2.2.3).

Exemple 2.1.4 Reprenons l'exemple de la figure 2.2. Les différentes instances de l'instruction `A(i,j)=A(i-1,j)*A(i,j-1)` sont les seules à modifier les valeurs du tableau A. Désignons chaque instance de cette instruction par le vecteur d'itération lui correspondant (liste des numéros associés à chaque boucle englobante). L'instance (i, j) lit les valeurs des variables $A(i-1, j)$ et $A(i, j-1)$. Ces variables sont affectées par les instances $(i-1, j)$ et $(i, j-1)$. Or ces instances sont **avant** l'instance (i, j). Les valeurs lues par (i, j) sont donc celles affectées par les instances $(i-1, j)$ et $(i, j-1)$.

Remarques.

- Chaque groupe d'instructions de $\mathcal{LH}$ peut être envisagé comme une opération atomique modifiant globalement la mémoire. Les valeurs accédées sont celles calculées dans le dernier groupe précédent au sens de l'ordre logique, ou bien celles calculées localement. En ce sens il généralise une affectation dataparallèle.

- L'obligation de conserver une trace des calculs constitue la contrepartie principale de ce schéma de communication. Par contre, ces sauvegardes de contexte peuvent n'être effectuées qu'à la fin des groupes d'instructions.

2.2 La machine abstraite: codage de l'ordre logique

Dans cette section nous décrivons la machine abstraite $\mathcal{SCP}$. Puisque l'accès aux données distantes est conditionné par la structure du programme, un mécanisme de synchronisation est nécessaire pour garantir que l'indice distant a terminé l'exécution de toutes les instructions $\mathcal{SCP}$ précédentes au sens de l'ordre logique.

2.2.1 Construction des horloges structurelles

L'idée intuitive est la suivante. La position d'un indice dans la structure du programme dépend du nombre d'instructions $\mathcal{SCP}$ franchis. L'existence de structures de contrôle dynamiques (boucles, conditionnelles) ne permet pas de le connaître statiquement. De plus, il est nécessaire de distinguer les branches d'un même where/elsewhere. Nous empilons un compteur à chaque exécution du corps d'une structure de contrôle de $\mathcal{SCP}$. L'ajout d'un label permet de distinguer les branches d'un même where/elsewhere. La fin de l'exécution de la structure de contrôle, est codé dans l'horloge en dépilant le compteur correspondant et en incrémentant le compteur du programme $\mathcal{SCP}$ englobant.

Nous codons la position d'un indice par un compteur multi niveau appelé horloge structurelle. Une horloge structurelle est une liste de paires. Chaque élément de la liste correspond à un niveau d'imbrication. Une paire (l, c) est formée par un label l et un compteur c. Le compteur c représente le nombre d'instructions $\mathcal{SCP}$ déjà exécutées dans le niveau d'imbrication correspondant. Le label l permet de distinguer les deux branches d'une structure where/elsewhere. Les listes sont construites en empilant et en dépilant par la droite. Voici les règles de construction.

Initialement chaque indice positionne son horloge à $(0, 0)$.

step() Lorsqu'un indice a terminé l'exécution d'un groupe d'instructions, son horloge structurelle courante $t_u = t(l, c)$ devient $t_u = t(l, c + 1)$.

where/elsewhere Après évaluation par l'indice courant u, de la condition B d'une structure de contrôle where/elsewhere, si B est vrai (resp. faux) l'horloge $t_u = t$ devient $t_u = t(1, 0)$ (resp. $t_u = t(2, 0)$). Lorsque l'indice u termine l'instruction where elsewhere, son l'horloge $t_u = t(l, c)(m, d)$ devient $t_u = t(l, c + 1)$.

loopwhere Un indice qui n'exécute aucune itération sort directement. Par conséquent, nous n'avons pas à considérer chacune des itérations comme un nouveau niveau d'imbrication dans la liste de l'horloge. Nous nous contentons d'empiler un nouveau couple à la première itération. Ensuite nous comptons les instructions précédemment exécutées dans le corps du loopwhere.

Lorsque l'indice u entre pour la première fois dans la boucle loopwhere , si la condition B s'évalue à vrai, alors son horloge $t_u = t$ devient $t_u = t(1, 0)$. Dans le cas contraire, son horloge $t_u = t(l, c)$ devient $t_u = t(l, c + 1)$. S'il a déjà exécuté au moins une itération, alors si l'indice u évalue B à *vrai*, son horloge $t_u = t$ reste inchangée, autrement $t_u = t(l, c)(1, d)$ devient $t_u = t(l, c + 1)$.

L'ordre structurel. *L'ordre structurel*, noté $\prec$, permet d'ordonner deux horloges structurelles par comparaison des compteurs des couples correspondants au niveau d'imbrication commun le plus interne. C'est un ordre lexicographique sur les listes de paires.

Définition 2.2.1 *Une horloge t_u est dite* **en retard** *par rapport à une horloge t_v (noté par $t_u \prec t_v$) si une des deux conditions suivantes est vérifiée:*

- *Il existe t non vide telle que*
  $t_v = t_u\ t$;
- *Il existe t^1, t^2, t^3, c_u, c_v et l tels que*
  $t_u = t^1(l, c_u)t^2$, $t_v = t^1(l, c_v)t^3$ *et* $c_u < c_v$.

Supposons que t_u et t_v soient respectivement les horloges courantes des indices u et v. La première condition formalise l'idée intuitive que u n'a pas encore fini l'évaluation de la condition rattachée à une conditionnelle ou à la première itération d'une boucle, alors que

v est à l'intérieur de cette structure. La seconde condition formalise l'idée intuitive que les deux indices sont dans la même branche d'une structure de contrôle et que l'indice v a exécuté plus d'instructions que u.

Exemple 2.2.1

Programme	Horloges
real :: A(0:N,0:N)	
real :: B(0:N,0:N)	
step(i=1;	
j=1)	$(0,1)$
loopwhere i $\leq$ N	$(0,1)(1,2i-2)$
loopwhere j$\leq$N	$(0,1)(1,2i-2)(1,3j-3)$
step(	
A(i,j)=A(i-1,j)*A(i,j-1);	
X=1)	$(0,1)(1,2i-2)(1,3j-2)$
where B(i-1,j-1) $\neq$ 0 do	*début de la* 1^{re} *branche:* $(0,1)(1,2i-2)(1,3j-2)(1,0)$,
	début de la 2^{e} *branche:* $(0,1)(1,2i-2)(1,3j-2)(2,0)$
step(	
B(i,j)= 1 / SQRT(B(i-1,j-1));	
Y=1)	$(0,1)(1,2i-2)(1,3j-2)(1,1)$
elsewhere	
step(Z=1)	$(0,1)(1,2i-2)(1,3j-2)(2,1)$
end	$(0,1)(1,2i-2)(1,3j-1)$
step(j=j+1)	$(0,1)(1,2i-2)(1,3j)$
end	$(0,1)(1,2i-1)$
step(i=i+1)	$(0,1)(1,2i)$
end	$(0,2)$

Nous reprenons l'exemple de la figure 2.2 page 19. Nous associons à chaque instruction l'horloge courante après que la structure ait été exécutée.

Le théorème suivant prouve la correspondance entre l'ordre logique et son codage.

Théorème 2.2.1 *L'ordre structurel est isomorphe à l'ordre logique sur les instructions de* $\mathcal{SCP}$.

Preuve

Nous commençons par prouver la condition nécessaire. Nous montrons que si $t_u \prec t_v$ *alors l'indice* u *est avant* v. *Dans un second temps nous prouvons la condition suffisante en montrant que si l'indice* u *est avant* v, *alors* $t_u \prec t_v$.

Condition nécessaire *Supposons que* $t_u \prec t_v$. *D'après la définition 2.2.1 nous avons:*

- *Il existe* t *non vide telle que* $t_2 = t_1\ t$,
  ou bien
- *Il existe* $t^1,\ t^2,\ t^3,\ c_1,\ c_2$ *et* l *tels que*
  $t_1 = t^1(l,c_1)t^2,\ t_2 = t^1(l,c_2)t^3$ *et* $c_1 < c_2$.

Distinguons les deux cas. S'il existe t non vide telle que $t_2 = t_1\ t$, alors, l'indice u évalue la condition rattachée à une conditionnelle ou à la première itération d'une boucle, alors que l'indice v se trouve à l'intérieur de cette structure. D'après la définition 2.1.1 la condition exécutée par u est avant l'instruction exécutée par v. S'il existe $t^1,\ t^2,\ t^3,\ c_1,\ c_2$ et l tels que $t_1 = t^1(l, c_1)t^2,\ t_2 = t^1(l, c_2)t^3$ et $c_1 < c_2$., alors les deux indices ont atteint la même structure de contrôle, mais leurs positions ne sont pas les mêmes:

1. *Cette structure est une boucle et u exécute une itération qui précède celle exécutée par v.*
2. *Les deux indices u et v exécutent deux instructions en séquence dans la même structure de contrôle.*

Condition suffisante *Si u est avant v, alors deux cas sont possibles:*

- *u et v exécutent des instructions en séquence et l'instruction de u apparaît avant celle de v.*

  Ou bien,

- *u évalue la condition d'une conditionnelle ou de la première itération d'une boucle alors que v exécute une instruction interne à la conditionnelle ou à la boucle.*

Dans les deux cas, d'après la méthode de construction des horloges (voir 2.2.1), on voit que les horloges de u et v sont de la forme: $t_u = t^1(l, c_u)t^2$ et $t_v = t^1(l, c_v)t^3$ avec $c_u < c_v$. ■

La proposition suivante démontre que l'horloge de chaque indice est croissante.

Proposition 2.2.1 (Croissance des horloges) *Au cours de l'exécution d'un programme $\mathcal{SCP}$ l'horloge t_u de chaque indice u est croissante.*

Preuve

Soit un indice u avec une horloge t_u. Après l'exécution de chaque instruction, l'indice u met à jour son horloge. Il incrémente le compteur terminal, empile une nouvelle paire ou dépile un niveau et incrémente son compteur terminal. Dans tous les cas la définition 2.2.1 montre que sa nouvelle horloge t'_u vérifie $t'_u \succ t_u$.

■

Synchroniser avec les horloges structurelles Les horloges structurelles permettent de créer un mécanisme d'attente entre les indices garantissant que l'exécution du programme respecte l'ordre logique. Intuitivement, chaque indice attend que tous les indices sur lesquels il effectue un accès distant aient terminé l'exécution de tous les groupes d'instructions antérieurs selon l'ordre logique.

Condition d'attente

Considérons un indice u avec une horloge structurelle t_u qui doit effectuer un accès distant sur l'indice v. Cet indice peut effectuer l'accès si et seulement si l'horloge t_v de l'indice v satisfait la condition $\neg(t_v \prec t_u)$.

Exemple 2.2.2

Programme	Horloges après exécution
`real :: a`	
`real :: b`	
`step( a=1 );`	$(0,1)$
`step( a=2 );`	$(0,2)$
`step( b=a )`	$(0,3)$

Reprenons l'exemple 2.1.3 page 20. Supposons que `a` et `b` ne sont pas placés sur le même indice. L'indice u qui possède `b` exécute l'instruction `b=a` alors que l'indice v qui possède `a` exécute les instructions `a=1` et `a=2`. L'indice u effectue un accès distant sur la mémoire de v. Avant cette lecture il teste l'horloge de t_v de v et la compare à la sienne $t_u = (0,2)$. Il effectue la lecture dès que la valeur de t_v n'est plus inférieure à celle de t_u, c'est-à-dire lorsque celle-ci est égale à $(0,2)$.

La proposition suivante garantit que si la condition d'attente $\neg(t_v \prec t_u)$ est vérifiée, alors elle restera toujours valide. Elle permet de tester de manière asynchrone la condition d'attente alors que les horloges des indices distants peuvent être mises à jour.

Proposition 2.2.2 (Stabilité de la condition d'attente) *Au cours de l'exécution d'un programme S, si l'horloge t_u d'un indice u vérifie $\neg(t_u \prec t)$, et si par la suite de l'exécution l'indice u possède une nouvelle horloge t'_u alors: $\neg(t'_u \prec t)$.*

Preuve

D'après la proposition 2.2.1, l'horloge de l'indice u est croissante. Nous avons donc $t'_u \succeq t_u$, et par suite $\neg(t'_u \prec t)$. ∎

2.2.2 Gestion de la mémoire

Dans le modèle $\mathcal{SCP}$, la sémantique des accès distants est guidée par la syntaxe. Un indice effectuant une lecture sur une variable distante reçoit la valeur de cette variable à une position antérieure à celle de la lecture au sens de l'ordre logique. Le mécanisme de synchronisation de $\mathcal{SCP}$ impose uniquement les attentes unilatérales nécessaires au respect du flot de données. Il n'existe pas de mécanisme de synchronisation permettant d'éviter l'écrasement des données avant qu'elles ne soient lues. Nous introduisons dans la machine abstraite de $\mathcal{SCP}$ une gestion de la mémoire permettant d'éviter cette source potentielle d'indéterminisme.

A la fin de l'exécution de chaque groupe d'instructions, chaque indice empile dans son environnement les dernières valeurs affectées au cours de l'exécution du groupe. Les horloges structurelles fournissent un mécanisme général pour estampiller les valeurs empilées. Lorsque nous conservons les valeurs des variables à la fin de l'exécution d'un groupe d'instructions, nous les estampillons avec l'horloge courante. De cette façon il est possible de référencer la valeur d'une variable correspondant à un point donné dans la structure du programme. Ce mécanisme permet d'implanter les accès aux données dirigés par l'ordre logique tout en garantissant des exécutions faiblement synchrones.

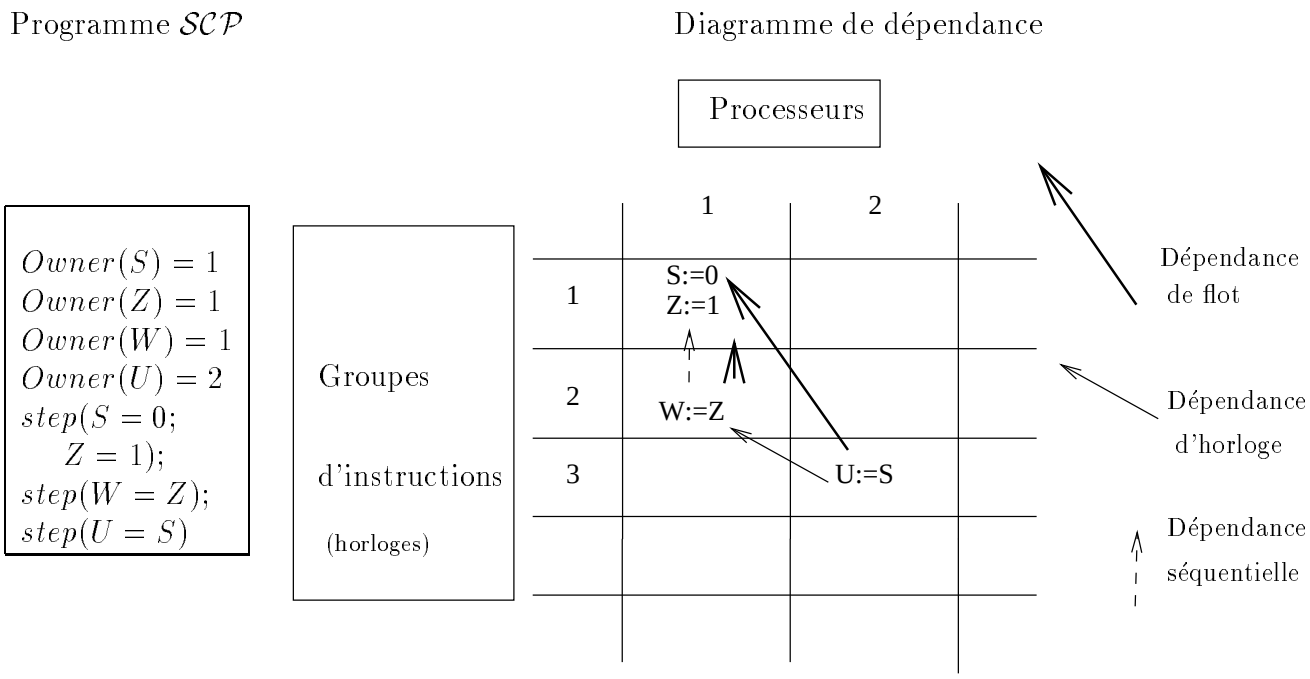
2.2.3 Diagrammes de dépendance

Dans cette section, nous montrons que $\mathcal{SCP}$ peut être doté d'un modèle élémentaire de coût, permettant d'estimer le coût induit par le mécanisme de synchronisation.

Afin de quantifier le coût des synchronisations, nous introduisons dans un tableau à double entrée une représentation des dépendances induites par le flot, par les horloges et par l'ordre séquentiel d'exécution sur chaque indice. Les indices sont représentés en abscisse, et les groupes d'instructions $\mathcal{LH}$ (éventuellement représentés par leurs horloges respectives) en ordonnée. Le diagramme obtenu est appelé diagramme de dépendances.

Le coût de l'exécution est évalué en traçant un diagramme de coût. Le diagramme de coût est un tableau à double entrée où les indices sont représentés en abscisse et le temps d'exécution parallèle en ordonnée. Chaque ligne correspond au temps d'exécution d'un groupe d'instructions au cours duquel est effectué un calcul. On considère que ce temps est uniforme. Dans le diagramme de coût, les groupes d'instructions apparaissent au plus tôt dans le temps d'exécution mais en respectant les dépendances d'horloge et les dépendances séquentielles du diagramme de dépendance. Le coût de l'exécution correspond à l'ordonnée du dernier groupe d'instructions dans le diagramme de coût.

Exemple 2.2.3 Nous présentons un exemple de code $\mathcal{SCP}$, et nous construisons le diagramme de dépendance lui correspondant.

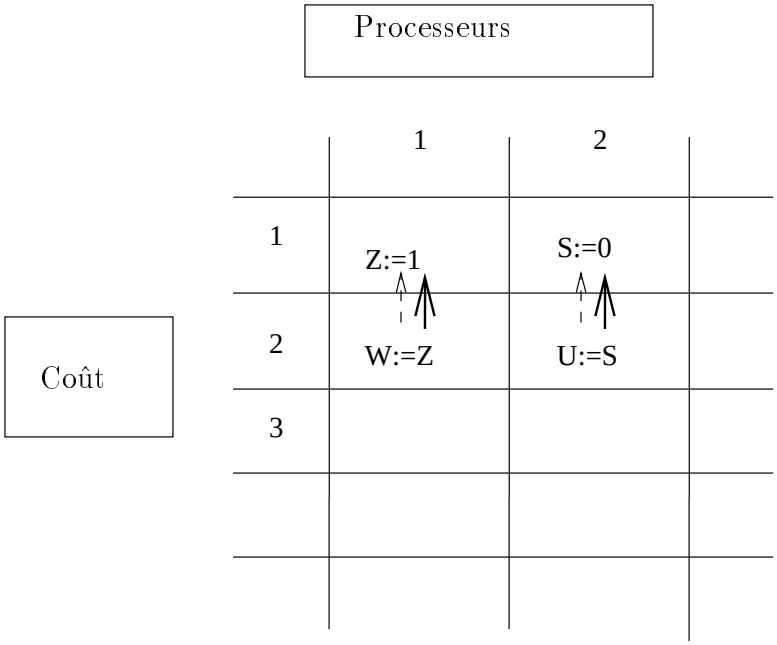


Les flèches continues indiquent les dépendances de flot, les flèches en pointillés indiquent les dépendances dues à la séquentialisation des instructions sur un même indice (*dépendances séquentielles*) et les flèches en gras indiquent les dépendances induites par les accès distants. Les flèches normales figurent les dépendances d'horloge. Soit u l'indice possédant l'affectation puits d'une dépendance. Soit v l'indice possédant l'affectation source d'une dépendance. La dépendance entre u et v part de u pour aller vers la dernière exécution placée entre u et v selon l'ordre logique. Si elle n'existe pas alors elle porte sur la source de la dépendance. Dans notre exemple, nous constatons que la dépendance de flot de 2 vers 1 induit une dépendance d'horloge sur la dernière affectation effectuée par l'indice 1: l'exécution est séquentielle.

Il est possible de résoudre le problème de deux façons, en modifiant le placement, ou en modifiant le code. L'idée intuitive est de recouvrir les dépendances d'horloges par les dépendances de flot.

Modification du placement Si nous plaçons Z sur l'indice 2, il n'y a plus de dépendance inter indices. La dépendance de flot est recouverte par une dépendance séquentielle.

Le diagramme de coût correspondant est le suivant:

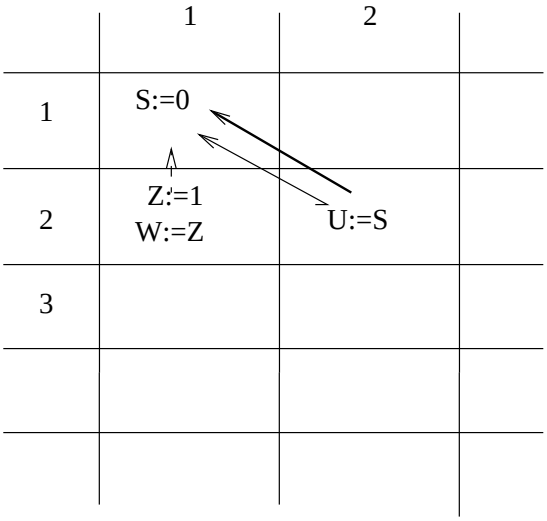


Modification du code Il est possible de conserver la même distribution des données et de supprimer la dépendance d'horloge inutile par modification du contenu des groupes. Si nous déplaçons la seconde affectation du groupe 1 dans le groupe 2 et l'affectation du groupe 3 dans le groupe 2 nous obtenons les codes et diagrammes de coût suivants:

Programme $\mathcal{SCP}$

$Owner(S) = 1$
$Owner(Z) = 1$
$Owner(W) = 1$
$Owner(U) = 2$
$step(S = 0);$
$step(Z = 1;$
$\quad W = Z;$
$\quad U = S)$

Diagramme de coût



Cette fois, la dépendance de flot est recouverte par la dépendance d'horloge. L'exécution est parallèle. Notons toutefois que dans cet exemple les instructions $Z = 1$ et $W = Z$ ne sont pas indépendantes, la «descente» de $Z = 1$ dans le groupe de $W = Z$ n'est possible que parce que les deux instructions sont placées sur le même indice. Par contre les instructions $W = Z$ et $U = S$ sont indépendantes, la «remontée» de $U = S$ dans le groupe de $W = Z$ est possible quelque soit le placement.

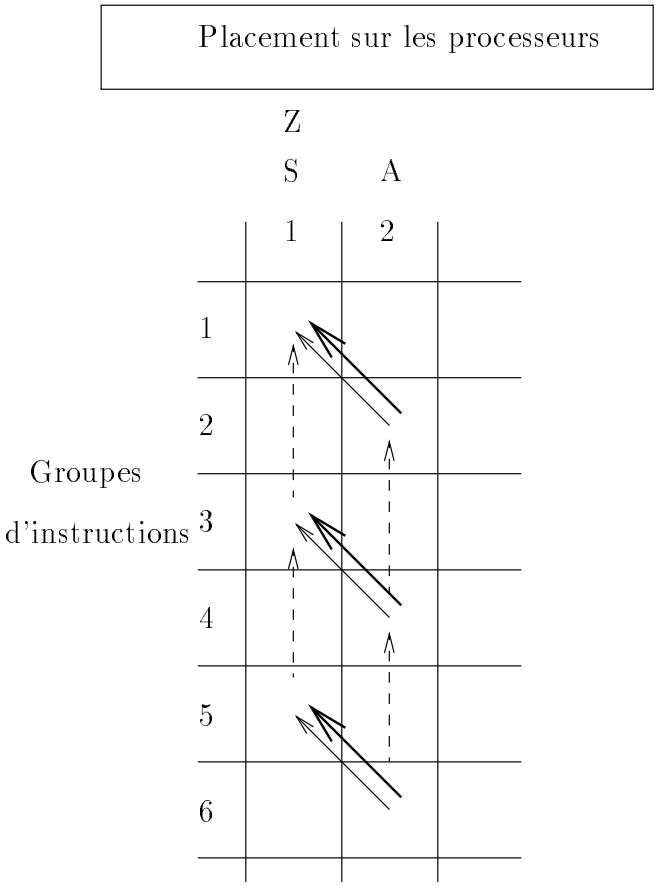
Exemple 2.2.4 Dans cet exemple, nous appliquons la méthode des diagrammes de dépendance sur un exemple très simple de programme parallèle implémentant un pipeline entre deux indices numérotés 1 et 2. A la droite de chaque instruction nous donnons l'horloge de l'indice quand il a fini de l'exécuter. La valeur i correspond au numéro de l'itération dans la boucle. Le premier indice exécute l'affectation $S = i * Z[i]$ alors que le second exécute $A[i] := A[i] + S$. Chaque tâche est répétée 100 fois. Le tableau Z fournit les données en entrée.

L'indice 1 ne communique avec aucun autre indice. Par conséquent il ne génère aucune attente. Pour chaque itération i il empile dans sa mémoire locale le résultat de l'affectation sur S en l'estampillant par son horloge courante: $(0,1)(1,3i)$. Le second indice communique avec le premier par lecture distante de la variable S dans la mémoire locale de 1. Il commence par attendre tant que 1 a une horloge strictement inférieure à la sienne. Une fois cette condition satisfaite, il lit la dernière valeur de S dans la mémoire locale de 1 au regard de sa propre horloge et de l'ordre structurel. Si l'horloge courante de 2 est (0,1)(1,3i+2) alors cette valeur correspond à celle estampillée par l'horloge $(0,1)(1,3i+1)$.

Programme $\mathcal{SCP}$

Programme	Horloges
$i \in Ind$	
$Owner(S) = 1$	
$Owner(Z) = 1$	
$Owner(A) = 2$	
$step(i = 0)$	$(0,1)$
loopwhere $i \leq 100$ do	$(0,1)(1,3i)$
step$(S = i * Z[i])$	$(0,1)(1,3i+1)$
step$(A[i] := A[i] + S)$	$(0,1)(1,3i+2)$
step$(i := i + 1)$	$(0,1)(1,3i+3)$
end;	$(0,2)$

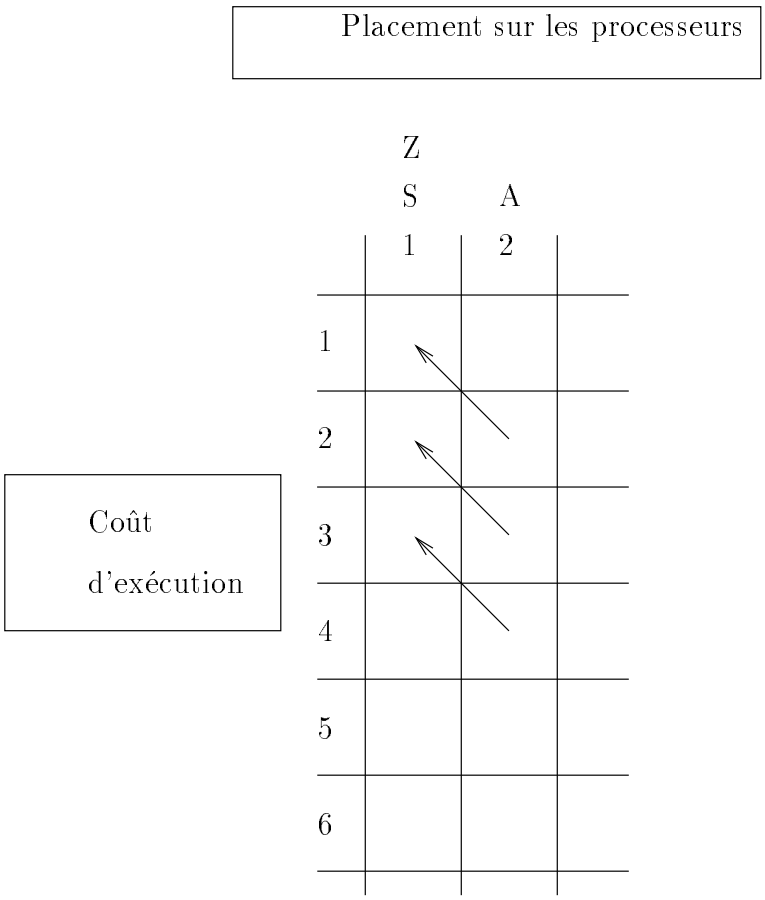
Diagramme de dépendance



Il n'existe pas de dépendance d'horloge inutile: dépendances d'horloges et dépendances de flot sont confondues.

Le second diagramme donne le coût de l'exécution du programme. Il se déduit aisément du diagramme de dépendance en plaçant au plus tôt les exécutions en tenant compte des dépendances d'horloges et séquentielles. Le schéma montre que les exécutions sont entrelacées. Nous constatons que le coût de l'exécution est divisé par deux par rapport au

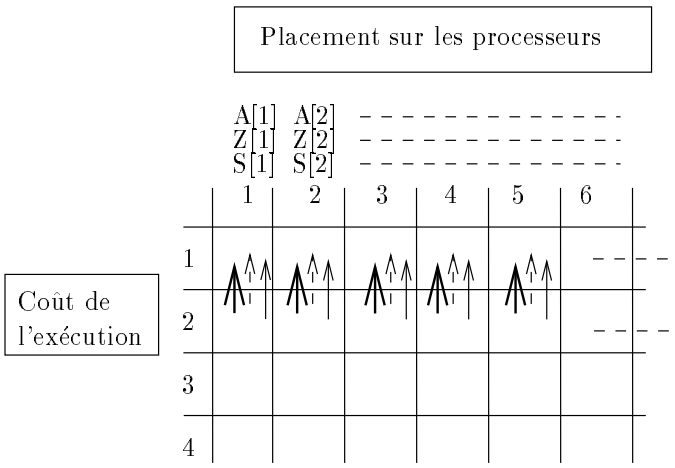
programme séquentiel.



Remarque. Les anti-dépendances ne sont pas représentées dans le diagramme de dépendance. Elles sont automatiquement supprimées par le mécanisme de gestion dynamique de la mémoire. Les dépendances de sorties sont gérées par le principe des écritures locales. La variable S ainsi que ses copies en mémoire sont placées sur l'indice 1. Cette contrainte induit une séquentialisation importante du programme. Il est possible de tirer parti d'une mise en assignation unique, à condition de placer différemment les données. Voici le code mis en assignation unique ainsi que son diagramme de coût.

Programme $\mathcal{SCP}$ Diagramme de coût

Programme	Horloges
$i \in Ind$	
$Owner(S) = 1$	
$Owner(Z) = 1$	
$Owner(A) = 2$	
$step(i = 0)$	$(0,1)$
loopwhere $i \leq 100$ do	$(0,1)(1,3i)$
step($S[i] = i * Z[i]$)	$(0,1)(1,3i+1)$
step($A[i] := A[i] + S[i]$)	$(0,1)(1,3i+2)$
step($i := i + 1$)	$(0,1)(1,3i+3)$
end;	$(0,2)$



Les dépendances d'horloges et de séquences sont confondues, les exécutions sont concurrentes et asynchrones. Le programme termine en un pas de calcul.

L'évaluation du coût prend pour unité de temps l'exécution d'un groupe d'instructions en supposant que celui-ci est uniforme. Il ne tient compte ni du coût des communications, ni de celui du parcours des groupes dans lesquels un indice n'a rien à exécuter. De plus, nous supposons que le coût de l'exécution des groupes d'instructions est uniforme. Ces hypothèses simplificatrices sont du même type que celles retenues pour l'algorithmique PRAM. Notre approche permet de comparer différents programmes et distributions de données. Elle ne permet par de prévoir la performance d'un programme sur une machine à l'instar du modèle de coût de BSP [46, 64]. Toutefois, il serait possible d'affiner la méthode en ne considérant non plus un temps uniforme pour tout groupe, mais en attribuant à chaque couple indice et groupe, un coût déterminé en fonction des instructions du groupe exécutées par l'indice.

2.3 Comparaison avec l'approche Bulk Synchronous Programming

Dans cette section nous proposons une comparaison entre les approches $\mathcal{SCP}$ et BSP.

Le modèle BSP (Bulk Synchronous Programming) introduit par Valiant [46, 64] exprime la concurrence entre des groupes d'instructions répartis sur les processeurs (dataparallélisme de tâches). Le programme est structuré en phases de calculs suivies de phases de communications. Les phases de calculs sont appelées *supersteps*. Les modèles d'exécution sous-jacents imposent des synchronisations globales entre les supersteps. L'indéterminisme n'est pas exclu dans ce modèle puisqu'il autorise des communications unilatérales pouvant engendrer des conflits écriture/écriture ou lecture/écriture. Par contre l'utilisation simultanée des synchronisations globales ainsi qu'un mécanisme de masquage temporaire des variables au sein d'une superstep permet de rendre le modèle déterministe [29]. Le modèle

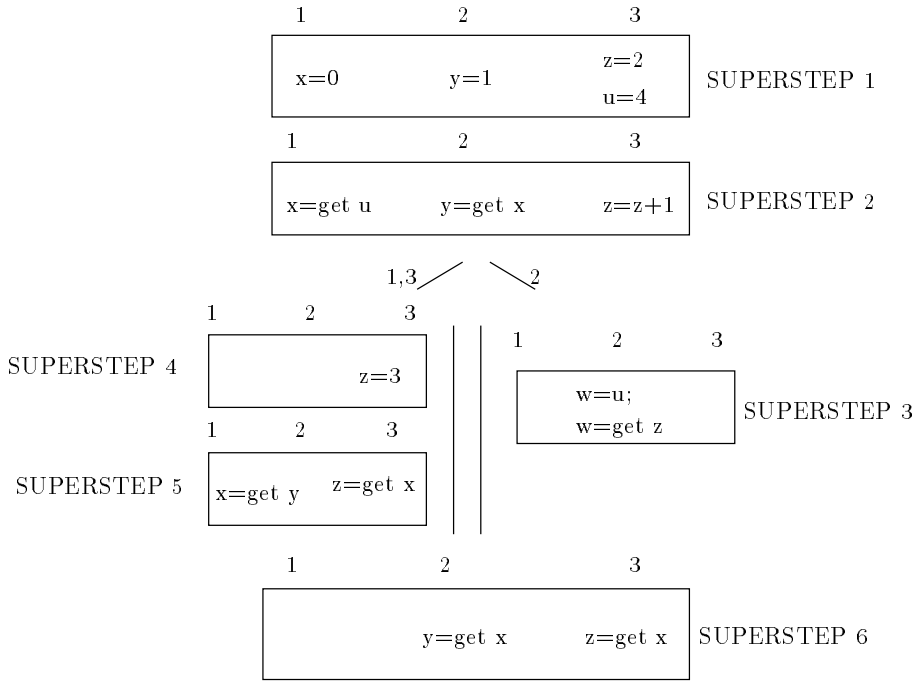
BSP est libre de blocage. La structure en couches des programmes et le modèle d'exécution favorisent la définition d'un modèle modulaire de calcul de coût [27], intégrant des paramètres spécifiques à la machine cible. Ce modèle ouvre la voie vers la prévision fine du comportement d'un programme à la fois en terme de coût mais aussi, dans une moindre mesure, en terme de validation [62].

Comme en $\mathcal{SCP}$, les communications en BSP sont unilatérales. Toutefois ce choix n'est rendu possible que par l'utilisation de synchronisations globales, contrairement à $\mathcal{SCP}$ où les synchronisation sont réduites à de simples attentes unilatérales. Ces barrières de synchronisations constituent une limitation importante à l'efficacité du code produit. Ce modèle ne prévoit pas l'expression de synchronisations n'impliquant que des sous ensembles de processeurs (synchronisations partielles). C'est un frein certain si l'on veut parvenir à gérer des groupes de processeurs concurrents (tâches dataparallèles).

Le langage $MINIBSP$ [60] est inspiré du modèle BSP. Il est adapté à des architectures qui n'offrent pas de mécanisme matériel de barrière de synchronisation. $MINIBSP$ étend BSP d'un opérateur de composition parallèle équivalent au where/elsewhere de $\mathcal{SCP}$. Cet opérateur permet de diviser récursivement l'ensemble des indices en deux sous-ensembles. Il est alors possible d'appliquer le modèle de décomposition en supersteps sur ces sous-ensembles. Les données traitées par chaque indice sont locales. Les communications sont explicites. Comme en $\mathcal{SCP}$, les communications entre des indices appartenant à des sous-ensembles différent sont masquées. Les communications inter sous-ensembles ne sont possibles qu'à la fin des exécutions locales.

Nous donnons ci-dessous un exemple de code $MINIBSP$ sur trois processeurs en mettant en évidence la structure syntaxique du programme. Sa traduction en $\mathcal{SCP}$ est immédiate. Il suffit d'associer un groupe d'instructions à chaque superstep, et de traduire l'opérateur de composition parallèle par un where/elsewhere.

Structure du code $MINIBSP$ Code $\mathcal{SCP}$

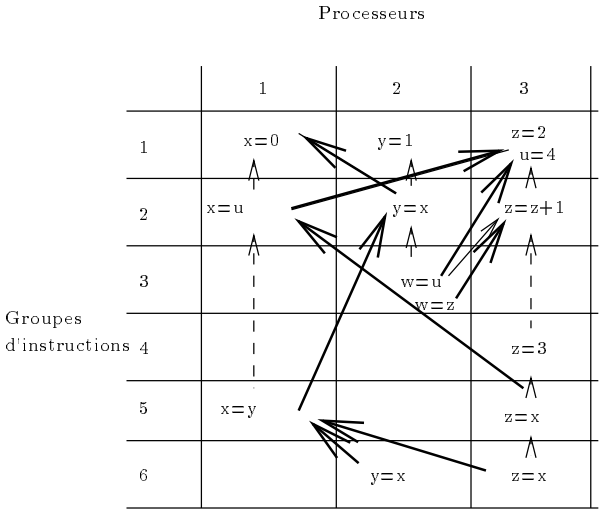


$Owner(X) = 1$
$Owner(Y) = 2$
$Owner(W) = 2$
$Owner(Z) = 3$
$Owner(U) = 3$
$step(X = 0;$
$Y = 1;$
$Z = 2;$
$U = 4);$
$step(Y = X;$
$Z = Z + 1;$
$X = U);$
where $This \neq 2$ do
$step(Z = 3);$
$step(X = Y;$
$Z = X);$
elsewhere
$step(W = U;$
$W = Z);$
end;
$step(Y = X;$
$Z = X);$

Le programme est composé de trois supersteps globaux et trois supersteps portant sur les sous-ensembles d'indices $\{1,3\}$ et $\{2\}$. Les variables locales sont distribuées de la façon suivante: $Owner(X) = 1$, $Owner(Y) = 2$, $Owner(W) = 2$, $Owner(Z) = 3$ et $Owner(U) = 3$. Les accès à des variables distantes sont explicites. Ils sont réalisés par l'instruction *get*. Un indice effectuant un *get* a reçoit la valeur de a à la fin du superstep précédent. Par exemple, au second superstep, l'indice 1 reçoit la valeur de u à la fin du superstep 1. L'indice 2 reçoit la valeur de x à la fin du superstep 1 et non la valeur calculée par l'indice 1 au superstep 2. De même, au superstep 3, l'indice 2 reçoit la valeur de z à la fin du superstep 2, et au superstep 5, l'indice 3 reçoit la valeur de x à la fin du superstep 4. Par contre, au superstep 5, l'indice 1 reçoit la valeur de y à la fin du superstep 2 et non la valeur calculée par l'indice 2 au superstep 3.

Si l'évaluation des sous-ensembles d'indices est dynamique, une synchronisation intervient à l'entrée de l'opération de composition parallèle. Dans notre exemple une synchronisation globale a lieu entre les supersteps 2 d'une part et 4 et 3 d'autre part. Une autre synchronisation globale a lieu entre les supersteps 3, 5 d'une part et 6 d'autre part. Entre les supersteps 4 et 5 intervient une synchronisation partielle ne mettant en jeu que les indices 1 et 3. Les synchronisations globales en entrée et en sortie de l'opération de composition parallèle constituent l'inconvénient majeur de cette approche. Elles sont nécessaires pour connaître les indices concernés par les synchronisations partielles des exécutions locales. Si elles sont fondamentales pour le fonctionnement du modèle, elles ne sont pas nécessairement justifiées par les dépendances de données.

En $\mathcal{SCP}$, il n'y a aucune synchronisation en entrée ou en sortie du where/elsewhere. Les seules synchronisations point à point sont générées par les dépendances de données. Il reste à vérifier que les dépendances d'horloges n'induisent pas de surcoût en terme de dépendances injustifiées par le flot de données. Pour cela, nous traçons le diagramme de dépendance correspondant:



Nous constatons que les dépendances de données recouvrent presque totalement les dépendances d'horloges. Il n'y a qu'une dépendance ajoutée entre les indices 2 et 3 et les supersteps 3 et 2. L'indice 2 attend que l'indice 3 ait terminé l'exécution du supersteps 2 avant de faire un accès à la variable u qui est disponible dès la fin du superstep 1. Hormis cette dépendance ajoutée, il ne reste dans l'exécution $\mathcal{SCP}$ que les synchronisations point à point nécessaires au flot de données.

Notons que si le where/elsewhere n'existe pas dans le code, et que tous les supersteps sont séquentialisés dans un seul flot de contrôle (le superstep 3 puis 4 puis 5), alors la dépendance de flot entre l'indice 1 sur le superstep 5 et l'indice 2 sur le superstep 2 peut induire une dépendance d'horloge sur le superstep 3. L'indice 1 devant attendre que l'indice 2 ait exécuté le superstep 3 avant d'accéder la valeur de y calculée au superstep 2. Cette remarque justifie l'intérêt de la structure where/elsewhere qui ici interdit la formation d'une dépendance d'horloge injustifiée. Pour les supersteps 4 et 5, le superstep 3 est masqué.

Le modèle de programmation de $MINIBSP$ rassemble les caractéristiques de celui de $\mathcal{SCP}$. Par contre le modèle d'exécution de ce dernier est beaucoup moins synchrone car il n'impose que des attentes unilatérales à la place des synchronisations partielles de $MINIBSP$: la machine abstraite de $\mathcal{SCP}$ offre un modèle d'exécution asynchrone pour $MINIBSP$, et par là même pour BSP.

2.4 Application de $\mathcal{SCP}$ à la parallélisation

Dans cette section nous montrons qu'il est possible d'utiliser $\mathcal{SCP}$ en tant que langage cible d'une traduction de code séquentiel en code parallèle. Les lectures distantes

permettent la résolution à l'exécution des dépendances évitant ainsi une analyse de dépendance. Le principe des écritures locales est mis en application par le biais des gardes implicites de $\mathcal{SCP}$. Nous obtenons des programmes parallèles faiblement synchronisés. Des exemples illustrent la traduction et permettent d'introduire quelques optimisations simples.

L'idée intuitive de la traduction consiste à garder autant que possible la structure des programmes séquentiels. Les structures des boucles et conditionnelles des programmes traduits correspondent exactement à celles de leurs sources séquentielles. Pour les indices distants, les données sont rendues disponibles dès qu'elles sont calculées: un groupe est associé à chaque affectation. Les compteurs de boucles sont dupliqués sur tous les indices de telle sorte que chaque indice possède ses propres compteurs. Ainsi, un compteur de boucle i appartient à l'ensemble des variables vectorielles Ind. La distribution des données sur les indices est exprimée par le biais de la fonction $Owner$. Notre traduction suit le principe des écritures locales. L'affectation $A = B$ est réalisée par l'indice $Owner(A)$ alors que la valeur de B est récupérée dans la mémoire de l'indice $Owner(B)$.

2.4.1 Gestion de la mémoire

Dans le cadre des méthodes classiques de parallélisation automatique, la suppression des fausses dépendances (anti-dépendances et dépendances de sortie) est classiquement résolue par une phase préliminaire de mise en assignation unique. Elle consiste à attribuer une cellule mémoire à chaque écriture. Les algorithmes de conversion automatique en assignation unique [20] nécessitent l'énumération de toutes les instances de toutes les instructions, puis l'adaptation des lectures aux écritures par le biais d'une analyse du flot de données. Cette approche statique est s'étend difficilement aux structures de contrôle dynamiques et aux dépendances imprévisibles.

Dans le modèle $\mathcal{SCP}$, afin de garantir le déterminisme des exécutions tout en ne respectant que les synchronisations liées aux dépendances de flot, nous avons introduit une gestion de la mémoire empilant les dernières valeurs affectées à la fin de l'exécution de chaque groupe d'instructions (voir 2.2.2. Cette gestion de la mémoire peut être considérée comme un mécanisme d'*assignation unique dynamique*. En $\mathcal{SCP}$, les fausses dépendances sont respectées.

Le principe des écritures locales (Owner compute rule) séquentialise les accès à une même variable. Notons toutefois que si il est possible d'effectuer une assignation unique statique en amont du placement, nous pouvons alors distribuer les cellules mémoires correspondant aux écritures d'une même variable. Cette approche permet d'augmenter le parallélisme potentiel du code. Cet avantage échappe à l'assignation unique dynamique. Cette dernière reste néanmoins bien adaptée aux schémas de dépendances imprévisibles. Nous donnons un exemple où l'assignation unique statique permet de mieux répartir les exécutions:

Exemple 2.4.1

Code séquentiel	Code mis en assignation unique, distribué et parallélisé

`real :: A`
 	
`for i:=1 to N do`
`  A:=i;`
`end`
	`real :: A(0:N)`
	$Owner(A[i]) = i$
	
	`forall i:=1 to N do`
	`  A[i]:=i;`
	`end`

Code $\mathcal{SCP}$ sans assignation unique	Code $\mathcal{SCP}$ avec assignation unique

`real :: A`
$Owner(A) = 1$
 	
`step(i=1);`
`loopwhere i≤ N do`
`  step(A:=i);`
`  step(i=i+1)`
`end`
	`real :: A(0:N)`
	$Owner(A[i]) = i$
	
	`step(i=1);`
	`loopwhere i≤ N do`
	`  step(A[i]:=i);`
	`  step(i=i+1)`
	`end`

Dans cet exemple il est possible de faire une assignation unique statique. Nous commençons en donnant le code séquentiel et le code parallèle tirant parti de l'assignation unique. Toutes les affectations de A peuvent être effectuées en parallèle dans un forall. Nous donnons le code $\mathcal{SCP}$ traduit à partir du code séquentiel et celui traduit après la mise en assignation unique. Nous constatons que le mécanisme d'assignation unique dynamique ne conduit pas à un code parallèle puisque toutes les itérations de la boucles sont effectuées par l'indice possédant A. Par contre il est possible de tirer parti de l'assignation unique statique en distribuant les cellules mémoires correspondant aux écritures dans la variable A.

2.4.2 La fonction de traduction en $\mathcal{SCP}$

La fonction de traduction, notée $Trans$, est définie par induction sur la structure du programme.

La traduction d'une affectation

Nous explicitons la traduction d'une affectation dans un élément de tableau $X[Exp_1] = Exp_2$, le cas d'une variable simple peut en être aisément déduit. En accord avec la règle des écritures locales, l'affectation est réalisée par l'indice $u = Owner(X[Exp_1])$. Chaque indice exécutant le groupe d'instructions contenant cette affectation évalue l'expression Exp_1. Si elle est locale, l'évaluation n'engendre pas d'attente ni de communication. Au contraire si Exp_1 n'est pas locale, la sémantique de l'évaluation des expressions dans $\mathcal{SCP}$ impose les attentes nécessaires avant les communications. Après cette étape, chaque indice exécutant

le groupe courant est à même de déterminer s'il doit ou non exécuter l'affectation. Un seul indice l'exécute. La fin du groupe d'instructions permet de rendre accessible aux autres indices la valeur de la variable affectée:

$Trans($ $X[Exp_1] = Exp_2$ $)$	$step($ $X[Exp_1] = Exp_2$ $)$
Traduction d'une affectation	

La traduction d'une conditionnelle

Le principe est analogue à celui de l'affectation. Chaque indice exécutant le bloc de code $\mathcal{SCP}$ contenant la conditionnelle évalue la condition Exp. Si l'expression Exp est locale, cette évaluation n'engendre pas d'attente ni de communication. Au contraire si Exp n'est pas pure, la sémantique de l'évaluation des expressions dans $\mathcal{SCP}$ impose les attentes nécessaires avant les communications.

Pour simplifier, nous commençons par introduire la structure de contrôle à une seule branche: where. Nous la définissons par le biais de sa traduction avec un where/elsewhere.

where Exp do S end	where Exp do S elsewhere $step()$ end
Traduction du where *en* where/elsewhere	

Nous passons à présent à la traduction de la conditionnelle:

$Trans($ if Exp then S end $)$	where Exp do $Trans(S)$ end
Traduction d'une conditionnelle	

La traduction d'une boucle

Chaque indice exécutant le bloc de code $\mathcal{SCP}$ contenant la boucle évalue la condition Exp. Si elle est locale, l'évaluation n'engendre pas d'attente ni de communication: par exemple si la condition porte sur des compteurs de boucle. Au contraire si Exp n'est pas locale, la sémantique de l'évaluation des expressions dans $\mathcal{SCP}$ impose les attentes nécessaires avant les communications. Après cette étape, chaque indice est à même de déterminer s'il doit ou non exécuter le code conditionné. Ce processus est réitéré pour chaque tour de boucle. Si un indice évalue Exp à faux, il arrête l'exécution de la boucle et commence immédiatement à exécuter le code qui la suit.

$Trans($ while Exp do S end $)$	loopwhere Exp do $Trans(S)$ end
Traduction d'une boucle while	

La traduction d'un nid de boucles de type for suit le même principe. Ici, le nombre d'itérations est connu avant la première itération grâce à l'introduction de variables temporaires $Tmp1$ et $Tmp2$ ($Tmp1 \in Ind$ et $Tmp1 \in Ind$). Notre traduction évalue une seule fois les bornes du for. Ainsi, les communications induites par ces évaluations ne sont pas répétées à chaque itération.

Nous introduisons dans un premier temps la structure de contrôle forwhere. Cette structure de contrôle utilise le compteur de boucle $I \in Ind$. Chaque composante de cette variable varie indépendamment des autres entre B_{inf} et B_{sup} qui représentent tous les deux des vecteurs ou des scalaires.

Boucle forwhere	*Boucle équivalente avec* loopwhere
forwhere $I = B_{inf}, B_{sup}$ do $\{Code\ \mathcal{SCP}\}$ end;	step$(I = B_{inf})$ loopwhere$I \leq B_{sup}$ do $\{Code\ \mathcal{SCP}\}$ step $(I = I + 1)$ end;

Nous introduisons à présent la traduction du for:

$Trans($ for $i = B_{inf} to B_{sup}$ do S end $)$	step$(Tmp1 = B_{inf};$ $Tmp2 = B_{sup})$ forwhere $i = Tmp1$ to $Tmp2$ do $Trans(S)$ end
Traduction d'une boucle for	

Exemple 2.4.2

Code séquentiel

```
for i = 1 to N − 2 do
   A[i] = A[i − 1];
   if A[i + 1] ≠ 0 then
      A[i] = A[i − 1] ∗ A[i + 1];
   end
end
```

Code parallèle $\mathcal{SCP}$

```
forwhere i = 1, N − 2 do
   step(A[i] = A[i − 1])                          {step 1}
   where A[i + 1] ≠ 0 do
      step(A[i] = A[i − 1] ∗ A[i + 1])            {step 2}
   end
end
```

Les éléments du tableau A sont répartis sur les processeurs. Voici un exemple de distribution: $Owner(A[i]) = i$. Tous les indices exécutent l'ensemble des itérations. Les bornes de la boucle sont évaluables localement: la phase préliminaire de communication des bornes est inutile. Nous notons qu'il n'y a pas d'indirection dans le calcul des indices du tableau A, de sorte que chaque indice peut déterminer localement s'il est propriétaire de la partie gauche d'une affectation. Tous les indices entrent dans le groupe *step 1* mais un seul exécute l'affectation. Les autres poursuivent sans attendre.

Exemple 2.4.3

Code séquentiel

```
for i = 1 to N − 1 do
A[i] = A[i − 1];
B[i] = B[i − 1];
end
```

Code parallèle $\mathcal{SCP}$

```
forwhere i = 1, N − 1 do
   step(A[i] = A[i − 1])          {step i, 1}
   step(B[i] = B[i − 1])          {step i, 2}
end
```

Dans cet exemple, tous les indices exécutent l'ensemble des itérations. Il n'y a pas d'indirection dans le calcul des indices des tableaux A et B , de sorte que chaque indice peut déterminer localement s'il est propriétaire de l'élément gauche d'une affectation. Tous les indices exécutent les groupes d'instructions $step\ i, 1$ et $step\ i, 2$ mais un seul pour chaque itération exécute chaque affectation, les autres poursuivent sans attendre.

Nous allons considérer plusieurs placements. Dans chaque cas nous discuterons, à l'aide de diagrammes de dépendance, la qualité de la parallélisation.

Placement 1. Choisissons le placement suivant: $Owner(A[i]) = i$ et $Owner(B[i]) = i$.

Construisons le diagramme de dépendance correspondant.

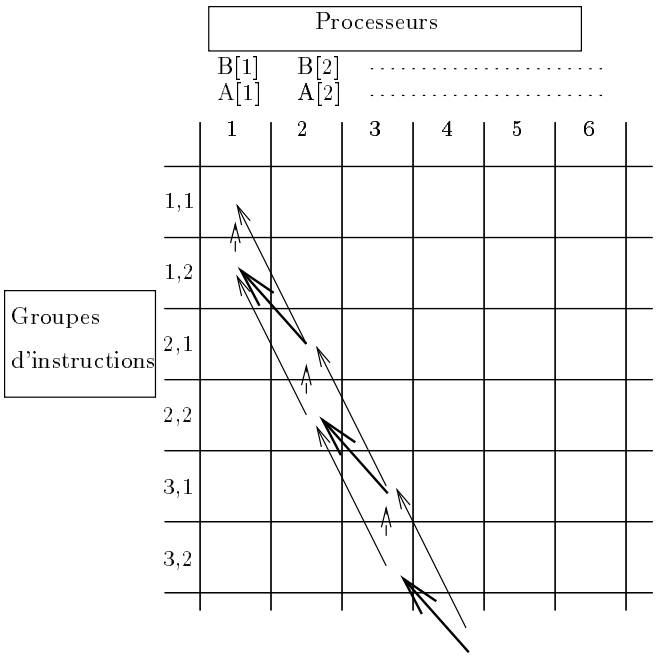
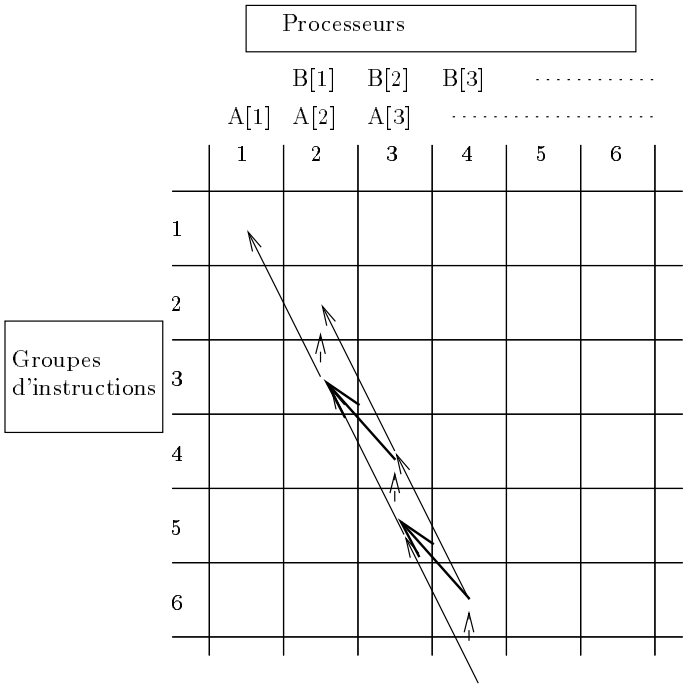


Diagramme de dépendance.

L'indice I exécute les affectations des groupes d'instructions $\{stepI, 1\}$ et $\{stepI, 2\}$. Dans ce cas l'indice $I+1$ possédant $A[I+1]$ doit attendre que I ait terminé le groupe $\{stepI, 2\}$ alors que les dépendances de données n'imposent qu'une attente sur le groupe $\{stepI, 1\}$. Cette attente est liée au placement. Les éléments appartenant à un même front sont placés sur le même indice. Le code $\mathcal{SCP}$ n'est pas plus performant que le code séquentiel. Si l'on veut éviter cela il faut choisir un placement différent où les éléments du front sont répartis sur des indices différents.

Placement 2. Le placement suivant répartit les éléments du front sur des indices différents: $Owner(A[i]) = i$ et $Owner(B[i]) = i - 1$. Voici le diagramme de dépendance

correspondant:



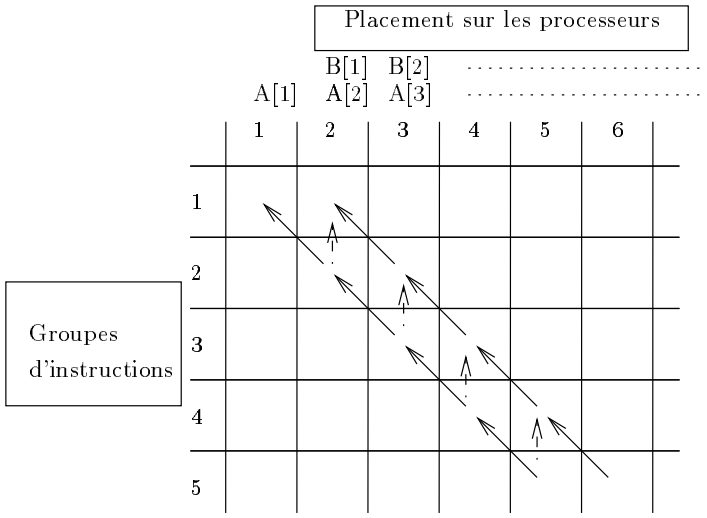
Chaque indice, exécute une seule affectation par itération. L'indice I exécute l'affectation du groupe $\{stepI-1,2\}$ (itération $I-1$) et celle du groupe $\{stepI,1\}$ (itération I). L'indice $I+1$ attend le possesseur de $B[I-1]$ (l'indice I) avant d'exécuter l'affectation du groupe $\{stepI,2\}$. Or l'indice I doit exécuter l'affectation du groupe $\{stepI,1\}$ donc l'indice $I+1$ attend que I ait exécuté l'affectation du groupe $\{stepI,1\}$ avant d'exécuter celle du groupe $\{stepI,2\}$. Le code $\mathcal{SCP}$ n'est pas plus performant que le code séquentiel. Il est possible d'améliorer notre traduction en exprimant l'indépendance entre $\{step\ I,1\}$ et $\{step\ I,2\}$. $\mathcal{SCP}$ permet d'exprimer cette indépendance en regroupant les affectations dans un seul groupe d'instructions. Nous donnons ci-dessous le code modifié ainsi que le diagramme de dépendance lui correspondant.

Programme $\mathcal{SCP}$

Code parallèle $\mathcal{SCP}$

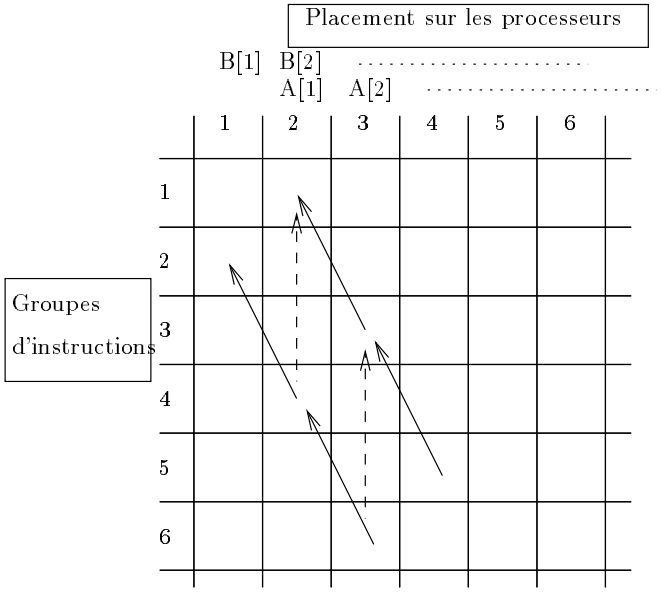
```
forwhere i = 1, 100  do
  step(                          {step i}
    A[i] = A[i − 1];
    B[i] = B[i − 1]
  )
end
```

Diagramme de dépendance



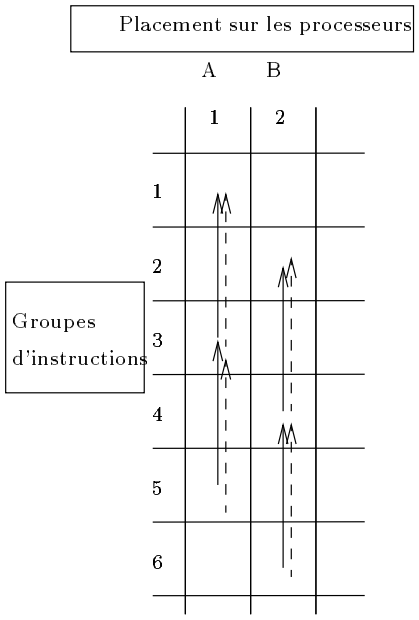
Dans ce cas, l' indice I exécute les affectations des $\{step\ I-1,2\}$ et $\{step\ I,1\}$ et l' indice $I+1$ exécute l' affectation du $\{step\ I,2\}$. Par contre, l' indice $I+1$ n'attend plus que I ait terminé le $\{step\ I,1\}$ avant d'exécuter l'affectation du $\{step\ I,2$, il commence dès qu'il a reçu l'horloge de I mise à jour après le $\{step\ I-1,2\}$.

Placement 3. Le modèle d'exécution de $\mathcal{SCP}$ sait tirer parti d'une expression plus fine des indépendances, mais aussi d'un meilleur placement. Prenons un troisième exemple de placement: $Owner(A[i]) = i-1$ et $Owner(B[i]) = i$. Voici le diagramme de dépendance correspondant:



Comme pour le second placement, chaque indice exécute une seule affectation par itération. L'indice I exécute l'affectation du $\{step\ I-1,1\}$ (itération $I-1$) et celle du $\{step\ I,2\}$ (itération I). L'indice $I+1$ attend le possesseur de $B[I]$ (l'indice I) avant d'exécuter l'affectation du $\{step\ I+1,2\}$. L' indice $I+1$ attend que le $\{step\ I+1,1\}$ soit terminé sur I avant d'exécuter le $\{step\ I+1,2\}$. L'indice responsable de l'exécution de l'affectation du $\{step\ I+1,1\}$ n'est ni I ni $I+1$, si bien que l' indice $I+1$ n'attend pas que l'affectation du $\{step\ I+1,1\}$ soit effectuée avant de faire celle du $\{step\ I+1,2\}$. Cet exemple illustre l'importance du choix du placement pour garantir une exécution performante.

Placement 4. Le diagramme suivant le résultat d'un placement des tableaux A et B sur deux indices différents. Dans ce cas les exécutions sont séparées: il n'y a pas de dépendance d'horloge. Les dépendances de flot sont confondues avec les dépendances séquentielles. Le coût en temps n'est pas plus élevé que pour l'exemple précédent. Le placement est plus économique que les précédents: les seules dépendances restantes sont des dépendances séquentielles.



Exemple 2.4.4 Dans cet exemple, nous comparons la traduction de code séquentiel en code $\mathcal{SCP}$ avec le résultat d'une parallélisation classique [21, 22]. Nous considérons plusieurs placements. Le but est de montrer qu'il est aisé de trouver un placement permettant

d'exprimer uniquement les dépendances de flot du programme.

Le programme séquentiel

for $I = 1, N$ do
  for $J = 1, 4$ do
  $A[J, I] = A[J-1, I] + A[1, I-1]$
  end
end

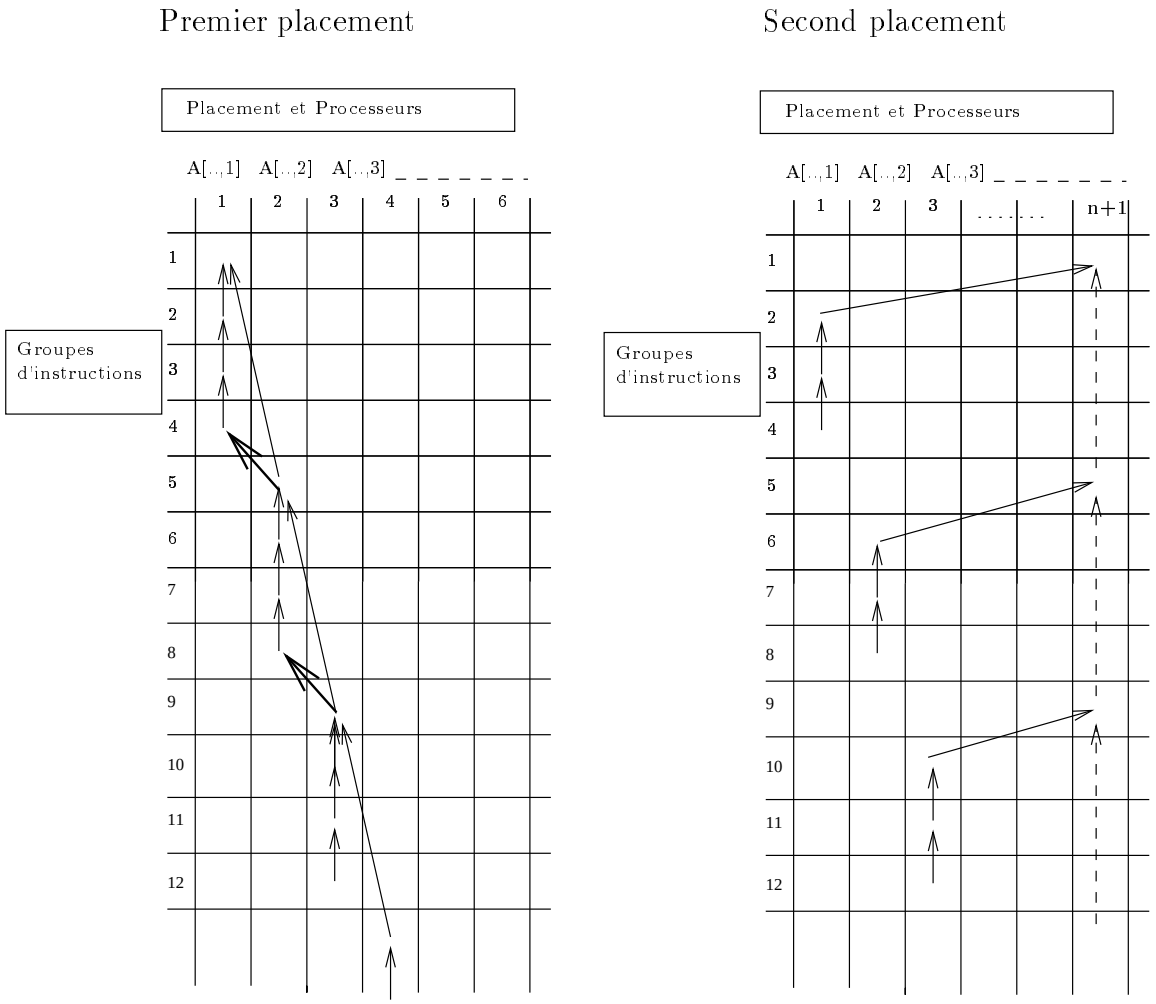
Le programme traduit en $\mathcal{SCP}$

forwhere $I = 1, N$ do
  forwhere $J = 1, 4$ do
  $step(A[J, I] = A[J-1, I] + A[1, I-1])$
  end
end

Placement 1. Choisissons le placement suivant: $Owner(A[j, i]) = i$.

Etudions le diagramme de dépendance. Les affectations de $A[1, 2]$ et $A[1, 1]$ sont respectivement exécutées par les indices 2 et 1. Or l'indice 1 doit aussi exécuter les affectations de $A[2, 1]$, $A[3, 1]$ et $A[4, 1]$. Celles-ci interviennent selon l'ordre séquentiel avant celle de $A[1, 2]$. Le mécanisme de synchronisation par horloge impose que l'indice 2 attende que l'indice 1 ait atteint le même point dans l'exécution des itérations, il impose donc une dépendance d'horloge entre l'affectation de $A[4, 1]$ et celle de $A[1, 2]$. Le même schéma est répété pour chaque itération. Le chemin de dépendances d'horloges et de dépendances séquentielles le plus long correspond à une exécution séquentielle. L'exécution n'est pas plus performante qu'en séquentiel. Considérant le critère de l'asynchronisme, le diagramme de dépendances montre que ce placement est mal choisi.

Placement 2. Le diagramme de dépendances nous permet de faire un choix de placement plus judicieux. Nous choisissons un indice pour les exécutions des affectations des variables $A[1, i]$ qui sont nécessairement en séquence. Puis nous plaçons les autres variables sur les mêmes indices que dans le placement précédent. Voici le diagramme de dépendances correspondant à ce placement:



Les affectations de $A[1,2]$ et $A[1,1]$ sont exécutées par l'indice $N+1$. L'indice i doit exécuter les affectations de $A[2,i]$, $A[3,i]$ et $A[4,i]$. Celles-ci interviennent selon l'ordre séquentiel avant celle de $A[1,i]$. Le mécanisme de synchronisation par horloge impose que l'indice i attende que l'indice $N+1$ ait atteint le même point dans l'exécution des itérations, il n'impose donc qu'une dépendance d'horloge entre l'affectation de $A[1,i]$ par $N+1$ et celle de $A[2,i]$. Cette dépendance d'horloge est exactement confondue avec la dépendance de flot. Les itérations $A[2,i]$, $A[3,i]$ et $A[4,i]$ sont poursuivies sur l'indice i sans synchronisation avec un autre indice. Le même schéma est répété pour chaque itération.

Le chemin de dépendances d'horloges et de dépendances séquentielles le plus long correspond à une exécution séquentielle des $A[1,i]$ sur $N+1$ plus l'exécution en séquence des $A[2,N]$, $A[3,N]$ et $A[4,N]$. Cette fois l'exécution est bien plus performante qu'en séquentiel ($N+3$ pas contre $4*N$ en séquentiel). Le coût de l'exécution peut être

obtenu en construisant le diagramme de coût correspondant à ce placement. Nous obtenons alors le diagramme suivant.

Coût du programme $\mathcal{SCP}$

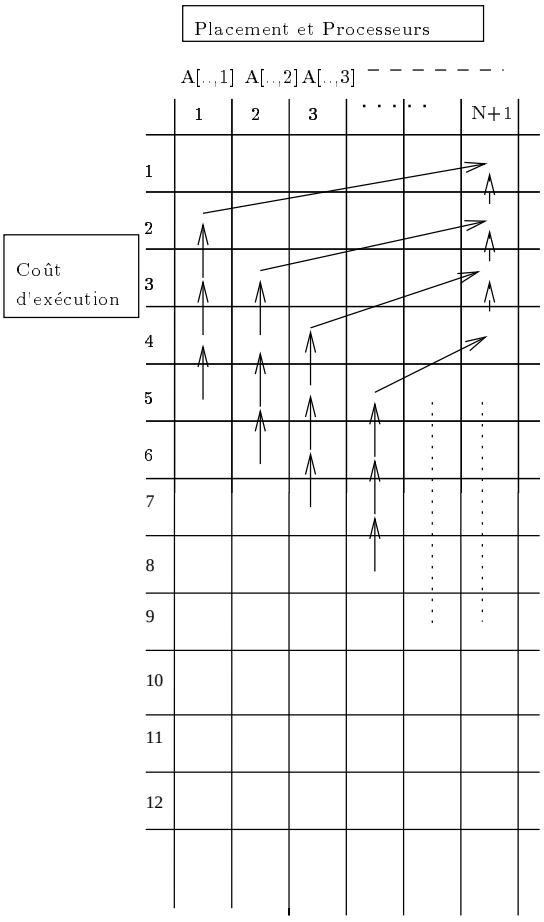
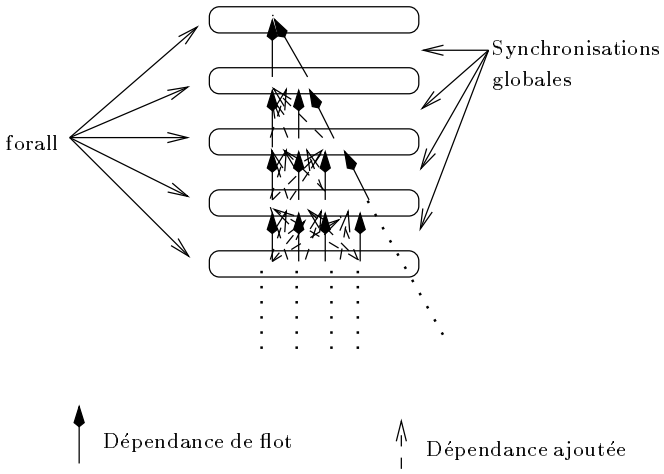


Diagramme de dépendance pour une parallélisation classique



Il nous reste à comparer ce code $\mathcal{SCP}$ placé avec l'exécution engendrée par une parallélisation classique. Une analyse de dépendance classique, conduisant à l'expression d'un code dataparallèle, donne le diagramme de dépendance de droite. Nous constatons que le programme $\mathcal{SCP}$ a le même diagramme coût que le code dataparallèle issu d'une parallélisation classique. Toutefois à l'exécution, le code $\mathcal{SCP}$ n'effectue que les synchronisations point à point imposées par le flot de données alors que le code dataparallèle doit encore être désynchronisé pour s'affranchir des synchronisations globales. Une traduction classique en code dataparallèle n'exprime pas autant d'indépendance que celle en $\mathcal{SCP}$.

Exemple 2.4.5

Programme séquentiel

for $I = 1, N-2$ do
  for $J = 1, N-2$ do
  $A[I,J] = A[I,J-1] * A[I+1,J] * A[I,J+1] * A[I-1,J]$
  end
end

Programme traduit en $\mathcal{SCP}$

forwhere $I = 1, N-2$ do
  forwhere $J = 1, N-2$ do
  $step(A[I,J] = A[I,J-1] * A[I+1,J] * A[I,J+1] * A[I-1,J])$
  end
end

Cet exemple est inspiré par l'algorithme de relaxation de Gauss-Seidel. Il consiste à parcourir un tableau bidimensionnel depuis la gauche vers la droite et de haut en bas tout en calculant le produit des valeurs des quatre voisins. L'analyse de cet exemple montre qu'il existe des dépendances de données et des anti-dépendances:

- Les itérations a, b et $a, b+1$ sont en antidépendance sur la variable lue au cours de a, b et initialisée avant le nid de boucle: $A[a, b+1]$

- Les itérations a, b et $a, b+1$ sont en dépendance sur la variable écrite au cours de a, b et lue au cours de $a, b+1$: $A[a, b]$

Les dépendances de flot imposent que l'itération $(a, b+1)$ soit exécutée après l'itération (a, b). A chaque antidépendance correspond une dépendance qui, si elle est respectée, force le respect de l'antidépendance. Alors le système d'empilement mémoire n'est pas nécessaire.

Exemple 2.4.6

Programme séquentiel

```
for I = 1, N − 2 do
  for J = 1, N − 2 do
    if A[I, J − 1] ≠ 0 then
      if A[I + 1, J] ≠ 0 then
        if A[I, J + 1] ≠ 0 then
          if A[I − 1, J] ≠ 0 then
            A[I, J] = A[I, J − 1] ∗ A[I + 1, J] ∗ A[I, J + 1] ∗ A[I − 1, J]
          end
        end
      end
    end
  end
end
```

Programme traduit en $\mathcal{SCP}$

```
forwhere I = 1, N − 2 do
  forwhere J = 1, N − 2 do
    where A[I, J − 1] ≠ 0 do
      where A[I + 1, J] ≠ 0 do
        where A[I, J + 1] ≠ 0 do
          where A[I − 1, J] ≠ 0 do
            step(A[I, J] = A[I, J − 1] ∗ A[I + 1, J] ∗ A[I, J + 1] ∗ A[I − 1, J])
          end
        end
      end
    end
  end
end
```

Cet exemple reprend l'algorithme précédent en l'optimisant. On évite certains accès distants inutiles en conditionnant le calcul par les valeurs des voisins qui doivent toutes être différentes de zéro. L'inconvénient principal réside dans l'évaluation des conditionnelles par tous les indices. Dans ce cas nous proposons une optimisation supplémentaire en ajoutant

une conditionnelle écartant les indices qui ne sont pas susceptibles d'effectuer l'affectation.

Programme séquentiel

```
for I = 1, N − 2 do
  for J = 1, N − 2 do
    if A[I, J − 1] ≠ 0 then
      if A[I + 1, J] ≠ 0 then
        if A[I, J + 1] ≠ 0 then
          if A[I − 1, J] ≠ 0 then
            A[I, J] = A[I, J − 1] ∗ A[I + 1, J] ∗ A[I, J + 1] ∗ A[I − 1, J]
          end
        end
      end
    end
  end
end
```

Programme traduit en $\mathcal{SCP}$

```
forwhere I = 1, N − 2 do
  forwhere J = 1, N − 2 do
    where This = Owner(A[I, J]) do
      where A[I, J − 1] ≠ 0 then
        where A[I + 1, J] ≠ 0 then
          where A[I, J + 1] ≠ 0 then
            where A[I − 1, J] ≠ 0 then
              step(A[I, J] = A[I, J − 1] ∗ A[I + 1, J] ∗ A[I, J + 1] ∗ A[I − 1, J])
            end
          end
        end
      end
      )
    end
  end
end
```

2.5 Conclusion

Dans ce chapitre nous avons présenté un cadre unifié pour la traduction de code séquentiel irrégulier en code parallèle. La syntaxe du langage $\mathcal{SCP}$ structure les programmes de manière à dégager un ordre logique sur les instructions. Il formalise la notion de précédence, permettant ainsi d'encapsuler les informations de dépendance ou d'indépendance au sein d'un programme. Cet ordre signifie intuitivement qu'il n'y a jamais de retour en arrière au cours de l'exécution, les boucles étant supposées déroulées. Cette caractéristique

différencie notre approche des modèles à passage de messages classiques du type MPI [61] où la sémantique des communications ne suit pas toujours la structure syntaxique des programmes.

La machine abstraite de $\mathcal{SCP}$ est basée sur l'attente de signaux diffusant des compteurs multi niveaux, les *horloges structurelles*. Elles permettent de maintenir la cohérence entre des indices exécutant des structures de contrôle dynamiques (where/elsewhere, loopwhere) en autorisant, pour un indice distant, le masquage temporaire de l'exécution d'une structure dynamique. Il est alors possible d'exprimer et de gérer des exécutions faiblement synchronisées, rendant le modèle de programmation et la machine abstraite adaptés aux schémas de synchronisation fins et dynamiques.

Nous avons présenté un schéma de traduction suffisamment général pour être appliqué à des programmes à schéma de dépendance complexe ou évalué dynamiquement. La traduction est fondée sur la distribution des données. Les communications sont des lectures en mémoire distante autorisant par là même la résolution dynamique des accès dans le cas de dépendances imprévisibles.

La sémantique des accès aux données dirigés par l'ordre logique permet de supprimer les synchronisations liées aux anti-dépendances tout en garantissant le déterminisme des exécutions. Le mécanisme d'assignation unique dynamique permet de gérer l'assignation unique de façon transparente, aussi bien dans le cadre régulier, que dans le cadre de structures de contrôle dynamiques et de dépendances imprévisibles. Nous avons montré que ce mécanisme simple ne permet pas toujours de dégager le parallélisme supplémentaire qu'apporte une assignation unique statique. Par contre il est possible d'exprimer le parallélisme supplémentaire qu'apporte l'assignation unique statique quand elle est praticable.

La qualité de notre traduction en terme de minimisation des synchronisations est dépendante du choix du placement. Comme le code généré est indépendant du placement, la traduction en $\mathcal{SCP}$ peut bénéficier en aval de toute analyse visant à choisir un "bon" placement. Nous avons proposé un modèle simple, fondé sur l'analyse des dépendances permettant d'évaluer et de réduire le coût des synchronisations. Une voie prometteuse serait de mettre à profit l'analyse de dépendance de flot, quand elle est possible, pour aboutir à des heuristiques de placement.

Il est certain que le schéma de traduction est loin d'être optimisé. Au lieu d'associer un groupe à chaque affectation, il peut être souhaitable de faire croître la taille des groupes d'instructions de façon à exprimer plus d'indépendance et de réduire le coût de gestion des horloges. De même, nous avons vu qu'il est souhaitable de faire ressortir les gardes au delà des structures de contrôle afin de limiter le nombre de gardes évaluées par chaque indice. L'analyse de dépendance à posteriori permet de disposer d'un cadre adapté pour exprimer finement l'indépendance au moyen de transformation du code parallèle. Cette approche peut se révéler particulièrement intéressante pour des programmes complexes mélangeant des sections de code aux degrés d'irrégularité divers.

Une approche plus générale de l'écriture d'un groupe d'instructions est possible. Si l'on peut connaître statiquement le code que chaque indice doit exécuter, alors il est possible d'écrire les groupes d'instructions sous la forme d'un code à parallélisme de contrôle. Nous évitons ainsi la résolution dynamique des gardes implicites de chaque affectation. Dans ce

cas, seul le squelette du programme reste commun à tous les indices.

Il serait intéressant d'enrichir le langage hôte $\mathcal{LH}$ avec des structures de contrôle afin de pouvoir factoriser les gardes implicites en les faisant sortir des structures de contrôle de $\mathcal{SCP}$. Le langage $\mathcal{LH}$ peut être aisément étendu puisque nous n'imposons que très peu de contraintes sur la sémantique de $\mathcal{LH}$. Les seules contraintes portent sur l'évaluation des expressions qui doivent respecter la sémantique des accès structurés par la syntaxe de $\mathcal{SCP}$.

En $\mathcal{SCP}$, l'exécution d'un groupe d'instructions peut être vue comme une opération atomique. Tous les groupes sont clos à l'entrée d'une structure de contrôle: toutes les valeurs calculées sont alors disponibles. Il serait possible de découpler les mises à dispositions des données et les structures de contrôle [33, 47]. Toutefois une telle approche compliquerait la lecture séquentielle et modulaire du code.

L'inconvénient majeur des approches classiques de la parallélisation de code séquentiel par distribution de données est le parcours obligatoire de toutes les itérations des boucles par tous les indices afin d'évaluer les gardes. Une traduction efficace doit éviter de faire parcourir à l'ensemble des indices la totalité des itérations d'un nid de boucles s'il est possible de prédire statiquement, en fonction d'un placement donné, les itérations pour lesquelles un indice n'a aucun calcul à exécuter [43]. Il est très facile d'exprimer en $\mathcal{SCP}$ des optimisations de ce type. L'horloge structurelle d'un indice peut être incrémentée de façon à simuler le parcours de ces itérations, sans que celles-ci n'aient été réellement exécutées. Nous décrivons plus en détail cette méthode dans le chapitre 5.

Nos exemples mettent en lumière les optimisations qui peuvent être réalisées en aval de la traduction. Elles s'expriment aisément dans le cadre du modèle $\mathcal{SCP}$.

Chapitre 3

Modèle d'exécution de $\mathcal{SCP}$: une sémantique formelle

Ce chapitre est de nature plus théorique. Notre but est de fonder les propriétés d'absence de blocage et de déterminisme des programmes $\mathcal{SCP}$. A cette fin, nous dotons le langage d'une sémantique opérationnelle à la Plotkin. Le chapitre est composé de trois parties.

Dans la première partie nous définissons la sémantique opérationnelle de $\mathcal{SCP}$. Elle formalise sur un premier niveau les règles d'exécution du langage hôte $\mathcal{LH}$. Nous décrivons ensuite les règles propres à $\mathcal{SCP}$ dans les deuxième et troisième niveaux de la sémantique.

Dans la deuxième partie nous prouvons l'absence de blocage (Théorème 3.2.1 page 67). La preuve est basée sur l'impossibilité de blocage d'un indice dont la position courante est minimale au sens de l'ordre logique. Cette preuve est simple; elle permet cependant d'introduire la notion de contexte et de transition contextuelle (Définition 3.2.2 page 62), des outils indispensables pour prouver la correction de la machine abstraite et de la fonction de traduction. Intuitivement, la notion de contexte permet de séparer l'ensemble des indices en deux sous ensembles, les indices inactifs et les indices actifs. Lors d'une transition contextuelle seul un indice actif peut effectuer un pas de calcul.

La troisième partie traite du déterminisme. Nous montrons que, bien qu'un programme $\mathcal{SCP}$ puisse donner lieu à plusieurs calculs différents, cette multiplicité est sans effet sur les résultats: toutes les exécutions possibles conduisent au même résultat (Théorème 5.2.2 page 123). La preuve suit un schéma classique basé sur la mise en évidence de la propriété diamant (Définition 3.3.1 page 67). Habituellement, la démonstration de cette propriété passe par la preuve d'absence de conflits (conditions de Berstein). L'originalité de notre démonstration repose sur la mise en évidence de conflits sans effet sur la propriété diamant. Un outil essentiel est la stabilité de la condition d'attente (Proposition 2.2.2 page 25)

3.1 Sémantique opérationnelle

La sémantique opérationnelle formalise le modèle d'exécution des programmes $\mathcal{SCP}$. Nous donnons une sémantique à un langage de coordination, tout en spécifiant les contraintes

sur l'exécution du code hôte $\mathcal{LH}$. C'est pourquoi nous décomposons la sémantique en trois niveaux. Le premier niveau définit la règle d'évaluation des expressions lors de l'exécution d'un programme en code hôte à l'intérieur d'un groupe d'instructions pour un indice donné. Le deuxième niveau définit les règles de transition entre les groupes d'instructions pour un indice donné. Le troisième niveau concerne tous les indices.

Chaque niveau définit de nouvelles transitions réutilisant les règles du niveau inférieur. Le fonctionnement de la machine abstraite est modélisé par les règles du troisième niveau, agissant sur des états globaux. Cette séparation permet de visualiser clairement la gestion des horloges qui est circonscrite au deuxième niveau.

Nous commençons par introduire quelques définitions.

- Nous rassemblons sous le vocable *variable scalaire* les variables scalaires simples ou les éléments de tableaux dont les indices sont connus. Par exemple les variables X, Y, $T[1]$ et $T[Exp]$ (où la valeur de Exp est connue) sont des variables scalaires. Un élément de tableau dont un indice contient une indirection non encore évaluée n'est pas assimilé à une variable scalaire. Il le sera lorsque tous ces indices serons évalués.

- La distribution des données sur les indices est exprimée par le biais de la fonction $Owner$. Elle est définie sur l'ensemble des variables. L'expression $Owner(V)$ désigne l'indice possédant la variable V.

  On note $Var(E)$ l'ensemble des variables qu'il est nécessaire d'évaluer pour calculer l'expression E.

- Nous notons $\tau|_u$ l'environnement multi niveau local à un indice u. Il associe à chaque variable $V \in Var$, telle que $Owner(V) = u$, une liste de couples $\tau|_u(V)$ constitués d'une valeur et d'une horloge structurelle. La fonction $\mathcal{T}(\tau, V)$ désigne l'ensemble des horloges structurelles qui sont associées à la variable V. Elles apparaissent au sein de la liste $\tau|_{Owner(V)}(V)$.

  L'environnement local $\tau|_u$ associe à chaque variable $V \in Ind$ une valeur $\tau|_u(V)$ correspondant à la valeur de la composante $V|_u$ de V.

- Nous notons $MemEnv$ l'ensemble des *environnements multi niveaux* dans lesquels sont sauvegardées les valeurs des différentes variables. Un *environnement multi niveaux* τ, est un vecteur d'*environnements multi niveaux locaux* . Par la suite, lorsqu'il n'y a pas d'ambiguïté, nous désignons un environnement multi niveaux par le terme environnement.

- Nous définissons à présent la fonction permettant d'extraire les valeurs des environnements multi niveaux $max(\tau, u, V, t)$:

  Si $V \in Var$: $max(\tau, u, V, t)$ retourne la valeur associée à V et à l'horloge $t'' = max\{t' \in \mathcal{T}(\tau, V) \ / \ t' \preceq t\}$ dans l'environnement local $\tau|_{Owner(V)}$.

  Si $V \in Ind$: La fonction $max(\tau, u, V, t)$ retourne la valeur associée à $V|_u$ dans l'environnement local $\tau|_u$: $\tau|_u(V)$.

Notons que l'existence et l'unicité de t'' sont liées à la croissance des horloges sur un indice. Au départ, toutes les variables sont initialisées à nil, et les horloges à (0,0): $max(\tau, u, V, (0,0)) = nil$.

La fonction $maxstrict(\tau, u, V, t)$ n'est utilisée que sur des variables scalaires. $maxstrict(\tau, u, V, t)$ retourne la valeur associée à V et à l'horloge $t'' = max\{t' \in \mathcal{T}(\tau, V) \ / \ t' \prec t\}$ dans l'environnement local $\tau|_{Owner(V)}$.

- Nous définissons la fonction $Top(\tau, V)$ prenant en entrée un environnement multi niveaux τ et une variable scalaire V et retournant l'horloge la plus récente associé à V dans τ: $Top(\tau, V) = max\{t' \in \mathcal{T}(\tau, V)\}$.

  Nous définissons la fonction $Change(l, Va)$ prenant en entrée une liste l de couples valeur-horloge et une valeur Va et retournant une liste l' égale à l sauf pour le couple dont l'horloge t'' est la plus récente au sein de la liste. Dans l', la valeur associée à t'' est Va.

- Un *environnement d'horloges* T, est un vecteur d'horloges possédant une composante par indice. Nous notons $T|_u$ l'horloge structurelle d'un indice u et H l'ensemble des environnement d'horloges.

- Nous notons respectivement $Proc$, Exp, $Bool$ et Val les ensembles de tous les indices, expressions, booléens et valeurs que prennent les variables ou les composantes d'un vecteur.

- Le langage hôte $\mathcal{LH}$ permet l'affectation d'un scalaire et celle d'un vecteur.

3.1.1 L'évaluation des expressions

Dans cette partie, nous définissons la sémantique de l'évaluation des expressions. Le langage autorise les expressions *non pures*, par opposition aux expressions *pures* au sens de HPF, c'est-à-dire sans effet de bord. Si une expression E n'est pas pure, son évaluation nécessite celles de variables distantes.

Nous définissons maintenant la fonction $Evalop$ qui permet d'évaluer les expressions non pures. L'idée intuitive est la suivante, $Evalop$ prend en entrée un environnement multi niveaux et retourne un couple de valeurs. Il est nécessaire de n'utiliser cette fonction que sur un environnement multi niveaux cohérent, c'est-à-dire possédant toutes les valeurs dont la fonction a besoin. Au cours de l'évaluation de $Evalop$, il faut donc vérifier que tous les indices distants qui possèdent des variables nécessaires à cette évaluation ne sont pas en retard par rapport à l'indice qui effectue l'évaluation. Cette condition est garantie par un test sur les horloges structurelles, évalué à $vrai$ lorsque toutes les horloges des indices qui possèdent des variables sur lesquelles $Evalop$ fait des accès ne sont plus inférieures à l'horloge de l'indice courant. La valeur de ce test est la première valeur du couple de sortie. La seconde valeur de sortie de $Evalop$ est le résultat de l'évaluation d'une expression.

Nous extrayons les résultats de $Evalop$ à l'aide de fonctions de projections $Proj_k$ telles que $Proj_k$ retourne la k^{ieme} composante d'un n-uplet.

Nous définissons formellement la fonction *Evalop* par induction sur la structure des expressions.

Définition 3.1.1 *La fonction Evalop est une fonction de* $(MemEnv \times Proc \times Exp \times H)$ *dans* $(Bool \times Val)$.

- *Si* V *est une variable scalaire* ($V \in Var$):

$$Evalop(\tau, u, V, T) = \ (\neg(T|_{Owner(V)} \ \prec \ T|_u), X),$$

$$avec\ X = \begin{cases} max(\tau, u, V, T|_u) \text{ si } Owner(V) = u, \\ maxstrict(\tau, u, V, T|_u) \text{ sinon.} \end{cases}$$

Remarque: *Si* $Owner(V) = u$, *alors la valeur du test est toujours vrai et la valeur retournée pour* V *peut avoir été affectée dans le même groupe que celui d'où l'on appelle la fonction Evalop et qui correspond à l'horloge* $T|_u$.

- *Si* V *est une variable vectorielle* ($V \in Ind$):

$$Evalop(\tau, u, V, T) = \ (vrai, X),$$

$$avec\ X = max(\tau, u, V, T|_u)$$

- *Si op est un opérateur binaire, et* E_1 *et* E_2 *deux expressions, alors:*

$$Evalop(\tau, u, E_1\ op\ E_2, T) = \\ (Proj_1(Evalop(\tau, u, E_1, T)) \ \wedge \ Proj_1(Evalop(\tau, u, E_2, T)), X),$$

$$avec\ X = \begin{cases} Proj_2(Evalop(\tau, u, E_1, T))\ op\ Proj_2(Evalop(\tau, u, E_2, T)) \\ \text{ si } Proj_2(Evalop(\tau, u, E_1, T)) \neq nil \text{ et } Proj_2(Evalop(\tau, u, E_2, T)) \neq nil, \\ nil\ sinon. \end{cases}$$

- *Si op est un opérateur unaire, et* E *une expression, alors:*

$$Evalop(\tau, u, op(E), T) = \\ (Proj_1(Evalop(\tau, u, E, T)), X),$$

$$avec\ X = \begin{cases} op(Proj_2(Evalop(\tau, u, E, T))) \text{ si } Proj_2(Evalop(\tau, u, E, T)) \neq nil, \\ nil \text{ sinon.} \end{cases}$$

- *Si* $Y[E]$ *n'est pas une variable scalaire (la valeur de l'expression* E *n'est pas encore connue) alors nous définissons récursivement l'évaluation de* $Y[E]$:

$$Evalop(\tau, u, Y[E], T) = \\ (Proj_1(Evalop(\tau, u, E, T)) \ \wedge \ Proj_1(Evalop(\tau, u, Y[Proj_2(Evalop(\tau, u, E, T))], T)), X),$$

$$avec\ X = \begin{cases} Proj_2(Evalop(\tau, u, Y[Proj_2(Evalop(\tau, u, E, T))], T)) \\ \text{ si } Proj_2(Evalop(\tau, u, E, T)) \neq nil, \\ nil \text{ sinon.} \end{cases}$$

Remarques.

- Dans le cas général, le résultat de l'évaluation d'une expression dépend de l'indice qui l'exécute.
- Si l'on évalue une expression réduite à une variable vectorielle, alors la projection $Proj_1$ sur le résultat de $Evalop$ donne toujours la valeur $vrai$ car la lecture de la composante du vecteur est locale.

3.1.2 Règles du premier niveau

Les règles du premier niveau définissent la sémantique du langage hôte. Il est parallèle, SPMD et implicitement gardé. Il est synchronisé par le biais de l'évaluation des expressions. Les transitions portent sur des états locaux. Un état local $\langle S, \tau, u, T\rangle$ est constitué d'un programme S, d'un environnement multi niveaux τ, d'un indice u et d'un environnement d'horloges.

Nous explicitons à présent les règles d'exécution des affectations, pour un indice donné. Nous distinguons trois types d'instructions au sein du code $\mathcal{LH}$.

Affectation d'un élément de tableau : $X[E_1] = E_2$**.** Avant d'exécuter l'affectation, on doit garantir que les valeurs attendues des variables lues à distance ont été affectées. Pour cela la règle d'affectation est conditionnée par la conjonction des tests sur les horloges des indices possédants les variables lues pour évaluer E_1 ($Proj_1(Evalop(\tau, u, E_1, T))$ et E_2 ($Proj_1(Evalop(\tau, u, E_2, T))$).

$$\frac{Proj_1(Evalop(\tau, u, E_1, T)) \ \wedge \ Proj_1(Evalop(\tau, u, E_2, T))}{\langle X[E_1] \ = \ E_2, \tau, u, T\rangle \mapsto \langle \bullet, \tau', u, T\rangle}$$

avec τ' est tel que pour toute variable Z:

$$\tau'|_{Owner(Z)}(Z) = \begin{cases} \tau|_{Owner(Z)}(Z)(Proj_2(Evalop(\tau, u, E_2, T)), T|_{Owner(Z)}) \\ \qquad \text{si} \begin{cases} Z = X[Proj_2(Evalop(\tau, u, E_1, T))] \text{ et} \\ u = Owner(Z) \end{cases}, \\ Change(\tau|_{Owner(Z)}(Z), Proj_2(Evalop(\tau, u, E_2, T))) \\ \qquad \text{si} \begin{cases} Z = X[Proj_2(Evalop(\tau, u, E_1, T))] \text{ et} \\ T|_u = Top(\tau, Z) \text{ et} \\ u = Owner(Z) \end{cases}, \\ \tau'|_{Owner(Z)}(Z) \text{ sinon.} \end{cases}$$

Nous explicitons la signification intuitive de cette règle. Les expressions E_1 et E_2 sont évaluées dans τ à la position courante dans l'exécution. Cette position est repérée par l'environnement d'horloges T. L'affectation est implicitement gardée. L'environnement τ n'est modifié que si l'indice u exécutant l'affectation est celui qui possède la variable $X[E_1]$. Si τ est modifié, ce n'est que sur sa composante $\tau|_u$ et pour la valeur de la variable $X[E_1]$

$(Z = X[E_1])$. Lorsque $X[E_1]$ à déjà été modifiée au cours de l'exécution du groupe d'instructions courant, il existe un couple valeur-horloge dans $\tau|_{Owner(Z)}(Z)$ dont l'horloge est égale à l'horloge $T|_u$ $(T|_u = Top(\tau, Z))$. Dans ce cas on n'ajoute pas un nouveau couple à la liste mais on écrase la valeur associée à $T|_u$ au sein de la liste. Ainsi on garantit l'unicité de $t'' = max\{t' \in \mathcal{T}(\tau, Z)\}$.

Affectation d'une variable simple: $X = E$, où $X \in Var$. La règle de cette affectation est la même que dans le cas précédent. Nous avons simplement supprimé les contraintes liées à l'évaluation des indices dans la partie gauche de l'affectation.

$$\frac{Proj_1(Evalop(\tau, u, E, T))}{\langle X = E, \tau, u, T\rangle \mapsto \langle \bullet, \tau', u, T\rangle}$$

avec τ' est tel que pour toute variable Z:

$$\tau'|_{Owner(Z)}(Z) = \begin{cases} \tau|_{Owner(Z)}(Z)(Proj_2(Evalop(\tau, u, E, T)), T|_{Owner(Z)}) \\ \qquad \text{si} \begin{cases} Z = X \text{ et} \\ u = Owner(Z) \end{cases}, \\ Change(\tau|_{Owner(Z)}(Z), Proj_2(Evalop(\tau, u, E, T))) \\ \qquad \text{si} \begin{cases} Z = X \text{ et} \\ T|_u = Top(\tau, Z) \text{ et} \\ u = Owner(Z) \end{cases}, \\ \tau'|_{Owner(Z)}(Z) \text{ sinon.} \end{cases}$$

La proposition suivante se déduit immédiatement des deux précédentes règles.

Proposition 3.1.1 *Au cours de l'exécution d'un programme, la liste de couples valeur/horloge qui est associée à une variable scalaire donnée dans l'environnement multi niveaux, contient des horloges toutes distinctes.*

Preuve

La preuve est immédiate si l'on considère que l'on n'ajoute jamais dans la liste un nouveau couple comportant une horloge qui existe déjà dans la liste.

■

Affectation d'un vecteur : $X = E$, où $X \in Ind$.

$$\frac{Proj_1(Evalop(\tau, u, E, T))}{\langle X = E, \tau, u, T\rangle \mapsto \langle \bullet, \tau', u, T\rangle}$$

avec τ' est tel que pour tout indice v et pour toute variable Z:

$$\tau'|_v(Z) = \begin{cases} \tau|_v(E) \text{ si } v = u \text{ et} Z = X, \\ \tau'|_v(Z) \text{ sinon.} \end{cases}$$

Nous explicitons la signification intuitive de cette règle. L'expression E est évaluée dans τ à la position courante dans l'exécution. Cette position est repérée par l'environnement d'horloges T. L'affectation n'est pas gardée. L'environnement τ est toujours modifié sur la composante $\tau|_u$ et pour la valeur de la variable X $(Z = X)$.

Remarques.

- Lorsqu'un indice ne possède pas le scalaire $X[E_1]$, il n'a rien d'autre à faire pour franchir l'affectation que de déterminer qu'il n'est pas possesseur de $X[E_1]$ et donc qu'il n'est pas concerné par l'affectation.
- Lorsque l'évaluation des expressions est locale, il n'est nullement besoin d'attendre d'autres indices. Dans ce cas les fonctions *Evalop* n'engendrent pas d'attente. La décision d'effectuer l'affectation reste locale elle aussi.
- Le mécanisme d'attente des indices distants et celui de l'évaluation des expressions suivent la même descente récursive basée sur la structure des expressions. Dans ces conditions, une implantation fine de cette sémantique pourrait coupler les deux démarches. Elle commencerait par l'évaluation de $X[E_1]$ en respectant les attentes induites puis déciderait s'il est nécessaire d'évaluer E_2 et donc d'attendre les indices nécessaires à son évaluation. Notons qu'une optimisation possible consisterait à recouvrir les temps d'attente sur l'évaluation de certaines variables de base par la poursuite de l'évaluation du reste de l'expression ou même de celle de l'autre expression.

3.1.3 Règles du deuxième niveau: Les transitions entre groupes

Le deuxième niveau de la sémantique opérationnelle permet de définir les transitions au niveau des groupes d'instructions et des structures de contrôles sur ces groupes. La flèche $\hookrightarrow$ symbolise une transition du deuxième niveau.

step() L'exécution d'un groupe d'instructions fait appel aux transitions du premier niveau. Les horloges structurelles sont mises à jour à la sortie du groupe d'instructions.

$$\frac{\langle S, \tau, u, T\rangle \mapsto \langle S', \tau', u, T\rangle}{\langle \mathsf{step}\ (S), \tau, u, T\rangle \hookrightarrow \langle \mathsf{step}\ (S'), \tau', u, T\rangle}$$

$$\langle \mathsf{step}\ (\bullet), \tau, u, T\rangle \hookrightarrow \langle \bullet, \tau, u, T'\rangle$$

avec pour tout j, $T'|_j = \begin{cases} t_u(x, y+1) \text{ si } T|_u = t_u(x,y) \text{ et } j = u, \\ T|_j \text{ sinon.} \end{cases}$

loopwhere B do S end. En vue d'optimiser la gestion de l'incrémentation des horloges structurelles au cours de l' exécution du loopwhere, celle-ci varie en fonction de l'itération effectuée. Seule la première itération empile un nouveau niveau d'horloge. Les itérations suivantes n'interviennent pas dans l'incrémentation des horloges. La taille de la pile des niveaux est d'autant réduite.

Nous introduisons une nouvelle instruction next_iteration qui ne fait pas partie du langage puisque le programmeur ne peut pas l'utiliser. Contrairement au loopwhere,

elle ne modifie pas les horloges structurelles. Notons que cette instruction n'apparaît pas dans la sémantique dénotationnelle (voir chapitre 4). Si B est évalué à vrai, alors S est exécuté une première fois puis l'instruction loopwhere est réécrite avec l'instruction next_iteration.

L'indice u qui exécute le loopwhere doit évaluer l'expression booléenne B. Cette expression n'est pas nécessairement pure. L'indice u doit donc attendre que les indices sur lesquels il doit faire un accès distant mettent à jour leur environnement. Il attend donc que les horloges structurelles des émetteurs ne soient plus inférieures à la sienne. Ces attentes sont réalisées lors de l'exécution de la fonction $Evalop(\tau, u, B, T)$. De la seconde valeur résultante de la fonction dépend l'exécution de la boucle. Un indice qui ne vérifie pas la condition B termine la boucle en une seule transition.

$$\frac{Proj_1(Evalop(\tau,u,B,T)) \ \wedge \ Proj_2(Evalop(\tau,u,B,T))}{\langle \textsf{next\_iteration} \ \ B \ \ \textsf{do} \ \ S \ \ \textsf{end}, \tau, u, T\rangle \hookrightarrow \langle S; \textsf{next\_iteration} \ \ B \ \ \textsf{do} \ \ S \ \ \textsf{end}, \tau, u, T\rangle}$$

$$\frac{Proj_1(Evalop(\tau,u,B,T)) \ \wedge \ \neg \ Proj_2(Evalop(\tau,u,B,T))}{\langle \textsf{next\_iteration} \ \ B \ \ \textsf{do} \ \ S \ \ \textsf{end}, \tau, u, T\rangle \hookrightarrow \langle \bullet, \tau, u, T\rangle}$$

avec pour tout j, $T'|_j = \begin{cases} t_u(x, y+1) \text{ si } T|_u = t_u(x,y)(a,b) \text{ et } j = u, \\ T|_j \text{ sinon.} \end{cases}$

$$\frac{Proj_1(Evalop(\tau,u,B,T)) \ \wedge \ Proj_2(Evalop(\tau,u,B,T))}{\langle \textsf{loopwhere} \ \ B \ \ \textsf{do} \ \ S \ \ \textsf{end}, \tau, u, T\rangle \hookrightarrow \langle S; \textsf{next\_iteration} \ \ B \ \ \textsf{do} \ \ S \ \ \textsf{end}, \tau, u, T'\rangle}$$

avec pour tout j, $T'|_j = \begin{cases} T|_u(1,0) \text{ si } j = u, \\ T|_j \text{ sinon} \end{cases}$

$$\frac{Proj_1(Evalop(\tau,u,B,T)) \ \wedge \ \neg \ Proj_2(Evalop(\tau,u,B,T))}{\langle \textsf{loopwhere} \ \ B \ \ \textsf{do} \ \ S \ \ \textsf{end}, \tau, u, T\rangle \hookrightarrow \langle \bullet, \tau, u, T'\rangle}$$

avec pour tout j, $T'|_j = \begin{cases} t_u(x, y+1) \text{ si } T|_u = t_u(x,y) \text{ et } j = u, \\ T|_j \text{ sinon.} \end{cases}$

where B do S elsewhere Q end. Cette structure de contrôle divise les indices actifs en deux groupes qui exécutent concurremment les branches S et Q. L'indice u qui exécute le where doit évaluer l'expression booléenne B. Cette expression n'est pas nécessairement pure. L'indice u doit donc attendre que les indices sur lesquels il doit faire un accès distant mettent à jour leur environnement. Il attend donc que les horloges structurelles des émetteurs ne soient plus inférieures à la sienne. Ces attentes sont réalisées lors de l'exécution de la fonction $Evalop(\tau, u, B, T)$. Suivant le résultat de la seconde valeur résultante de la fonction, l'indice u exécute l'une des deux branches de la conditionnelle.

Les horloges structurelles de deux indices exécutant chacun une branche différente du même where/elsewhere sont mises à jour de telle sorte qu'elles deviennent incomparables. La paire $(1,0)$ (resp. $(2,0)$) est concaténée à l'horloge structurelle de l'indice u s'il exécute la première branche (resp. la seconde branche). Rappelons que l'accès aux données est déduit de l'ordre structurel. Les écritures qui interviennent dans un groupe d'instructions ne peuvent donc être visibles pour les autres indices que pour des lectures qui appartiennent à un groupe postérieur toujours au sens de l'ordre structurel. Par conséquent, les deux branches d'un même where/elsewhere sont indépendantes.

Les horloges structurelles sont mises à jour à l'exécution de la marque de fin de bloc end qui correspond à la sortie de la conditionnelle.

$$\frac{Proj_1(Evalop(\tau,u,B,T)) \ \wedge \ Proj_2(Evalop(\tau,u,B,T))}{\langle \textsf{where} \ \ B \ \ \textsf{do} \ \ S \ \ \textsf{elsewhere} \ \ Q \ \ \textsf{end}, \tau, u, T\rangle \hookrightarrow \langle S; \ \textsf{end}, \tau, u, T'\rangle}$$

avec pour tout j, $T'|_j = \begin{cases} T|_u(1,0) \text{ si } j = u, \\ T|_j \text{ sinon.} \end{cases}$

$$\frac{Proj_1(Evalop(\tau,u,B,T)) \ \wedge \ \neg \ Proj_2(Evalop(\tau,u,B,T))}{\langle \textsf{where} \ \ B \ \ \textsf{do} \ \ S \ \ \textsf{elsewhere} \ \ Q \ \ \textsf{end}, \tau, u, T\rangle \hookrightarrow \langle Q; \ \textsf{end}, \tau, u, \rangle}$$

avec pour tout j, $T'|_j = \begin{cases} T|_u(2,0) \text{ si } j = u, \\ T|_j \text{ sinon.} \end{cases}$

Fin d'un bloc d'instructions : end. Cette instruction marque la fin d'une branche d'une structure de contrôle where/elsewhere. L'indice doit mettre à jour son horloge structurelle en dépilant un niveau d'horloge puis en incrémentant le niveau courant.

$$\langle \textsf{end}, \tau, u, T\rangle \hookrightarrow \langle \bullet, \tau, u, T'\rangle$$

avec pour tout j, $T'|_j = \begin{cases} t_u(x, y+1) \text{ si } T|_u = t_u(x,y)(a,b) \text{ et } j = u, \\ T|_j \text{ sinon.} \end{cases}$

3.1.4 Règle du troisième niveau: Les transitions entre états globaux

Le troisième niveau de la sémantique correspond aux transitions d'un état global dans un autre. Il permet de modéliser les calculs.

Un état global est un triplet. Sa première composante est la liste des programmes $P_1 \ : \ ... \ : \ P_n$ où P_u est le programme exécuté par l'indice u. Sa seconde composante est un environnement multi niveaux τ. Sa troisième composante est un environnement d'horloges T. Toute exécution d'un programme P se fait à partir d'un état initial $\langle P \ : \ ... \ : \ P, \ \tau, T\rangle$

où les couples constitués d'une valeur et d'une horloge structurelle, et stockés dans l'environnement τ, sont initialisés à $(nil, (0,0))$.

Trois règles composent ce niveau. La première permet de réécrire une instruction ou une structure de contrôle. La seconde et la troisième règle se rapportent à l'opérateur de séquence. La flèche $\longrightarrow$ symbolise une transition de ce niveau.

Bloc d'instructions : Cette règle permet de réécrire une instruction ou une structure de contrôle.

$$\frac{\langle P_u, \tau, u, T\rangle \hookrightarrow \langle P'_u, \tau', u, T'\rangle}{\langle P_1 \ : \ ... \ : \ P_u \ : \ ... \ : \ P_n, \tau, T\rangle \longrightarrow \langle P_1 \ : \ ... \ : \ P'_u \ : \ ... \ : \ P_n, \tau', T'\rangle}$$

Séquence $\mathcal{SCP}$: $S;T$. Ces règles permettent de réécrire une séquence entre groupes d'instructions. Elles sont placées à ce niveau de sémantique afin que l'exécution de chaque instruction soit visible au niveau le plus haut de la sémantique.

$$\frac{\langle P_u, \tau, u, T\rangle \hookrightarrow \langle P'_u, \tau', u, T'\rangle}{\langle P_1 \ : \ ... \ : \ P_u; S \ : \ ... \ : \ P_n, \tau, T\rangle \longrightarrow \langle P_1 \ : \ ... \ : \ P'_u; S \ : \ ... \ : \ P_n, \tau', T'\rangle}$$

$$\langle P_1 \ : \ ... \ : \ \bullet; S \ : \ ... \ : \ P_n, \tau, T\rangle \longrightarrow \langle P_1 \ : \ ... \ : \ S \ : \ ... \ : \ P_n, \tau, T\rangle$$

Séquence $\mathcal{LH}$: $step(S;T)$. Ces règles permettent de réécrire une séquence entre instructions $\mathcal{LH}$. Elles sont placées à ce niveau de sémantique afin que l'exécution de chaque instruction soit visible au niveau le plus haut de la sémantique.

$$\frac{\langle P_u, \tau, u, T\rangle \hookrightarrow \langle P'_u, \tau', u, T'\rangle}{\langle P_1 \ : \ ... \ : \ \mathsf{step}\ (P_u; S) \ : \ ... \ : \ P_n, \tau, T\rangle \longrightarrow \langle P_1 \ : \ ... \ : \ \mathsf{step}\ (P'_u; S) \ : \ ... \ : \ P_n, \tau', T'\rangle}$$

$$\langle P_1 \ : \ ... \ : \ \mathsf{step}\ (\bullet; S) \ : \ ... \ : \ P_n, \tau, T\rangle \longrightarrow \langle P_1 \ : \ ... \ : \ \mathsf{step}\ (S) \ : \ ... \ : \ P_n, \tau, T\rangle$$

3.2 Absence de blocage

La formalisation de l'exécution des programmes $\mathcal{SCP}$ par le biais d'une sémantique opérationnelle permet de prouver l'absence de blocage.

Le théorème fondamental sur l'absence de blocage peut être trouvé page 67. L'idée intuitive est qu'un indice ne peut être bloqué que par un autre indice en retard par rapport à lui.

Nous commençons par introduire quelques définitions.

Définition 3.2.1 (Contexte) *Un contexte est un sous ensemble de l'ensemble des indices.*

Nous définissons à partir de la relation $\longrightarrow$ une relation relative à l'exécution d'instructions par les indices d'un contexte c uniquement. Nous l'appelons *relation de transition contextuelle* et on la note $\overset{c}{\longrightarrow}$.

Définition 3.2.2 (Transition contextuelle) *Nous appelons transition contextuelle $\overset{c}{\longrightarrow}$ toute transition $\longrightarrow$ d'un indice appartenant au contexte c.*

Exemple 3.2.1 Une seule transition contextuelle $\xrightarrow{\{2\}}$ sur le contexte contenant uniquement l'indice 2 est réalisable à partir d'un état $\langle \mathsf{step()} : \mathsf{step()} : ..., \tau, T\rangle$:

$$\langle \mathsf{step()} : \mathsf{step()} : ..., \tau, T\rangle \longrightarrow \langle \mathsf{step()} : \bullet : ..., \tau, T\rangle$$

Par contre, aucune transition $\xrightarrow{\{2\}}$ n'est possible sur l'état $\langle \mathsf{step()} : \bullet : ..., \tau, T\rangle$.

Remarquons que la transition $\longrightarrow$ correspond à la transition contextuelle $\xrightarrow{true}$ où *true* désigne le contexte contenant tous les indices. On note $\xrightarrow{c^*}$ la fermeture réflexive transitive de la relation $\xrightarrow{c}$ et $\xrightarrow{*}$ celle de $\longrightarrow$.

Pour des programmes $S_1 : ... : S_n$, un vecteur d'horloges T et un contexte c, on introduit une notion de triplet *non bloquant*. Intuitivement, un triplet $(S_1 : ... : S_n, T, c)$ est non bloquant si, lors de l'exécution de leurs programmes, les indices du contexte c ne peuvent pas être bloqués par les indices hors contexte.

Définition 3.2.3 (Triplet non bloquant) *Un triplet $(S_1 : ... : S_n, T, c)$ est non bloquant si pour tout état initial $\langle S_1 : ... : S_n, \tau, T\rangle$ et toute transition $\xrightarrow{c^*}$ ne portant que sur les indices du contexte c telle que*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle,$$

on a pour tout indice $i \in c$ et $j \notin c$, $\neg(T'|_j \prec T'|_i)$.

Exemple 3.2.2 Le triplet $(\mathsf{step()}{:}\mathsf{step(B{=}X)}, T, \{2\})$ est non bloquant si $T|_1 = (0,2)$ et $T|_2 = (0,0)$ car $T|_1 \succ T|_2$ et l'exécution du $\mathsf{step(B{=}X)}$ conduit aux horloges $T'|_1 \succ T'|_2$ ($T'|_1 = (0,2)$ et $T'|_2 = (0,1)$). Par contre, avec la distribution $Owner(X) = 1$ et $Owner(B) = 2$, le triplet $(\mathsf{step()}{:}\mathsf{step(B{=}X)}, T, \{2\})$ n'est pas *non bloquant* si $T|_1 = (0,0)$ et $T|_2 = (0,1)$ car $T|_1 \prec T|_2$. Les seules transitions autorisées par $\xrightarrow{\{2\}^*}$ portent sur l'exécution de $\mathsf{step(B{=}X)}$. Or les valeurs des horloges structurelles empêchent l'évaluation de $Evalop(X)$ sur 2 si $Owner(X) = 1$. On se trouve donc en situation de blocage.

Remarques.

- Tout triplet $(S_1 : ... : S_n, T, true)$ où *true* désigne le contexte formé de tous les indices est non bloquant.
- Si $(S_1 : ... : S_n, T, c)$ est non bloquant et si

  $$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle,$$

  alors le triplet $(S'_1 : ... : S'_n, T', c)$ est non bloquant.

Lors de l'exécution d'un programme $\mathcal{SCP}$, la sémantique opérationnelle conduit à des états intermédiaires où le programme à exécuter ne correspond pas à un programme syntaxiquement correct. Par la suite on réservera le terme «programme $\mathcal{SCP}$» aux programmes syntaxiquement corrects. Par exemple, après le calcul d'une itération du programme $\mathcal{SCP}$

```
loopwhere true do
   step (P)
end
```

la sémantique opérationnelle conduit au programme (syntaxiquement incorrect) suivant :

```
next_iteration true do
   step (P)
end
```

Parmi l'ensemble des états possibles, seul un sous-ensemble correspond aux états qui peuvent apparaître au cours de l'exécution d'un programme par les indices d'un contexte donné. On appelle ces états les *états accessibles.*

Définition 3.2.4 (Etat accessible) *Un état $\langle S_1 : ... : S_n, \tau, T\rangle$ est accessible pour un contexte c, s'il existe un triplet $(S'_1 : ... : S'_n, T', c)$ non bloquant, où tous les indices du contexte c ont le même programme $\mathcal{SCP}$ et la même horloge structurelle tel que :*

$$\langle S'_1 : ... : S'_n, \tau', T'\rangle \xrightarrow{c^*} \langle S_1 : ... : S_n, \tau, T\rangle$$

Exemple 3.2.3 L'état $\langle$end : end$, \tau, T\rangle$ où $T|_1 = T|_2 = (0,1)$ n'est pas accessible pour le contexte *true* formé des deux indices 1 et 2. On ne peut pas construire un programme $\mathcal{SCP}$ syntaxiquement correct qui conduise à end avec l'horloge $(0,1)$. Par contre si $T|_1 = T|_2 = (0,0)(1,0)$, alors l'état devient accessible. L'état $\langle S : S, \tau, T'\rangle$ où $S =$where *true* do step() end et $T'|_1 = T'|_2 = (0,0)$ conduit à $\langle$end : end$, \tau, T\rangle$ par une transition $\xrightarrow{*}$.

Lemme 3.2.1 *Tout triplet $(S_1 : ... : S_n, T, c)$ correspondant à un état accessible $\langle S_1 : ... : S_n, \tau, T\rangle$ pour un contexte c est non bloquant.*

Preuve

La preuve découle des définitions 3.2.3 et 3.2.4 de triplet non bloquant et d'état accessible.

■

On présente deux lemmes sur l'évolution de la forme des horloges structurelles au cours du calcul d'un programme $\mathcal{SCP}$. Ils sont très proches, mais on a choisi de les distinguer pour plus de clarté. Le premier concerne les horloges au cours d'un calcul, alors que le second traite les cas où le programme est terminé. Les deux lemmes sont basés sur le nombre de structures non imbriquées dans des structures de contrôle.

Lemme 3.2.2 *Soit un programme $\mathcal{SCP}$ S, un état $\langle S_1 : ... : S_n, \tau, T\rangle$ et un indice u tel que $S_u = S$ et $T|_u = t(a,b)$ (t peut être vide). S'il existe une transition $\stackrel{*}{\longrightarrow}$ telle que :*

$$\langle S_1 : ... : S_n, \tau, T\rangle \stackrel{*}{\longrightarrow} \langle S'_1 : ... : S'_n, \tau', T'\rangle,$$

alors $T'|_u = t(a, b+d)q$ où d est le nombre de structures non imbriquées exécutées par u.

Preuve

Le programme S est réduit à une structure. *Telle qu'a été définie la sémantique opérationnelle de $\mathcal{SCP}$ (voir section 3.1), si un indice u exécutant une structure (*step(), where/elsewhere, loopwhere*) a une horloge structurelle $t(a,b)$, il termine l'exécution avec une horloge $t(a,b+1)$:*

- *Le cas est trivial si l'instruction est un* step().
- *Chaque fois que l'on rentre dans une structure de contrôle, on empile une nouvelle paire en sommet d'horloge. Cette paire est ensuite dépilée en sortie de structure de contrôle, et le compteur terminal de l'horloge est incrémenté de 1.*

Cas général. *Supposons maintenant que la transition $\stackrel{*}{\longrightarrow}$ ait permis à l'indice u d'exécuter d structures de contrôle non imbriquées. En généralisant le cas précédent par une récurrence simple, on montre que si u a initialement une horloge $t(a,b)$, alors il a une horloge $T(a,b+d)$ après l'exécution de la dernière structure.*

Reste le cas où, a partir de la structure d après laquelle u a une horloge $T(a,b+d)$, u peut avoir fait d'autres transitions élémentaires $\longrightarrow$. Détaillons les différents cas en fonction des transitions possibles:

- *Les transitions ne portent pas sur des structures de contrôles ou des franchissement de groupes d'instructions. Donc on a $T'|_u = t(a, b+d)$.*
- *Les transitions ont conduit u dans une structure de contrôle non imbriquée, mais ne lui ont pas permis d'en ressortir : $T'|_u = t(a, b+d)q$.*
- *Aucune structure n'a pu être exécutée (complètement) sinon cette instruction est comptabilisée dans d.*

■

Lemme 3.2.3 *Soit un programme $\mathcal{SCP}$ S ayant d structures non imbriquées et un indice u avec l'horloge $t(a,b)$ en début d'exécution. Si u termine S, alors il termine avec l'horloge $t(a, b+d)$.*

Preuve

La preuve est directe à partir de celle du lemme précédent. ■

Nous avons aussi besoin du lemme suivant prouvant qu'à partir d'un état accessible, si un indice du contexte a terminé son programme, alors il a une horloge structurelle plus grande que les indices du contexte qui n'ont pas fini.

Lemme 3.2.4 *Soit un état accessible $\langle S_1 : ... : S_n, \tau, T\rangle$ pour un contexte c. Pour tous les indices $u \in c$ et $v \in c$, si $S_u = \bullet$ et $S_v \neq \bullet$ alors $T|_v \preceq T|_u$.*

Preuve

L'état $\langle S_1 : ... : S_n, \tau, T\rangle$ est accessible, donc il existe une transition $\xrightarrow{c^*}$ et un état $\langle S'_1 : ... : S'_n, \tau', T'\rangle$ où tous les indices u du contexte c ont la même horloge $T|_u = t(a,b)$ et le même programme $\mathcal{SCP}$ S ($S'_u = S$) tel que :

$$\langle S'_1 : ... : S'_n, \tau', T'\rangle \xrightarrow{c^*} \langle S_1 : ... : S_n, \tau, T\rangle$$

D'après le lemme 3.2.3, l'indice u du contexte c ayant terminé le programme S, on a $T|_u = t(a, b+n)$ où n est le nombre de structures non imbriquées de S. Si l'indice v n'a pas terminé S, il ne peut pas avoir exécuté plus de structures non imbriquées que u, donc $T|_v = t(a, b+n')q$ où $n' \leq n$ d'après le lemme 3.2.2. On en conclut que $T|_v \leq T|_u$. ∎

La proposition suivante assure l'absence de blocage pour les transitions se rapportant aux indices d'un contexte c si ces indices ne peuvent être bloqués par ceux hors contexte.

Proposition 3.2.1 (Absence de blocage) *Soit un état accessible $\langle S_1 : ... : S_n, \tau, T\rangle$ pour un contexte c. S'il existe un indice $i \in c$ tel que $S_i \neq \bullet$, alors il existe une transition $\xrightarrow{c}$ possible à partir de $\langle S_1 : ... : S_n, \tau, T\rangle$.*

Preuve

Supposons qu'aucune transition sur un indice du contexte c ne soit possible à partir de l'état accessible $\langle S_1 : ... : S_n, \tau, T\rangle$, et qu'au moins un programme S_u où $u \in c$ ne soit pas terminé. Considérons l'ensemble $m \subseteq c$ des indices ayant une horloge minimale : si $i \in m$ alors il n'existe pas d'indice $j \in c$ tel que $T|_j \prec T|_i$. Cet ensemble est non vide car le contexte est formé d'un nombre fini d'indices.

- *Si pour tout $i \in m$, $S_i = \bullet$, alors par hypothèse il existe un indice $u \notin c \backslash m$ tel que $S_u \neq \bullet$. Donc il existe un indice $i \in m$ vérifiant $T|_u \succ T|_i$. C'est absurde d'après le lemme 3.2.4.*
- *S'il existe un indice $i \in m$ qui n'a pas terminé sont programme, l'indice i est bloqué par une condition $Proj_1(Evalop(\tau, u, E, T))$ dans une instruction du langage hôte $\mathcal{LH}$ puisque ce sont les seules transitions bloquantes de la sémantique opérationnelle de $\mathcal{SCP}$. L'indice attendu ne peut être un indice inactif car le triplet $(S_1 : ... : S_n, \tau, T, c)$ est non bloquant (Lemme 3.2.1). Donc l'indice attendu est actif. C'est absurde, l'horloge de i étant minimale, la condition d'attente est nécessairement vérifiée.*

■

Le théorème suivant est un cas particulier de la proposition 3.2.1. Il prouve l'absence de blocage pour toute exécution d'un programme $\mathcal{SCP}$.

Théorème 3.2.1 *Soit un programme $\mathcal{SCP}$ S et un vecteur d'horloges structurelles T toutes initialisées à la même valeur. Soit la transition $\langle S : ... : S, \tau, T\rangle \xrightarrow{*} \langle S'_1 : ... : S'_n, \tau', T'\rangle$. S'il existe un indice v tel que $S'|_v \neq \bullet$, alors il existe aussi une transition $\longrightarrow$ possible à partir de l'état $\langle S'_1 : ... : S'_n, \tau', T'\rangle$.*

Preuve

La preuve est immédiate à partir de la proposition 3.2.1, en considérant le contexte true contenant tous les indices. ■

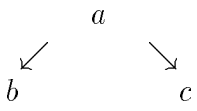
3.3 Déterminisme

L'exécution d'un programme $\mathcal{SCP}$ peut donner lieu à plusieurs calculs différents. Nous allons montrer que la multiplicité des calculs est sans effet sur les résultats, c.-à-d. que les programmes $\mathcal{SCP}$ sont déterministes (Théorème 3.3.2 page 71): pour un programme donné, soit tous les calculs sont infinis, soit ils terminent tous, rendant le même résultat final. La démonstration de cette propriété repose principalement sur la preuve que les transitions contextuelles $\xrightarrow{c}$ vérifient la propriété diamant (ou propriété du losange) (Théorème 3.3.1).

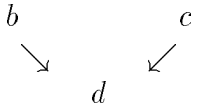
3.3.1 Propriétés sur les systèmes de réductions abstraits

La preuve du déterminisme est obtenue à partir de la propriété diamant [1].

Définition 3.3.1 (Propriété diamant) *Soit une relation binaire non vide $\rightarrow$ et $\xrightarrow{*}$ sa fermeture réflexive transitive. La relation $\rightarrow$ vérifie la propriété diamant si pour tout a, b, et c $(b \neq c)$ où*

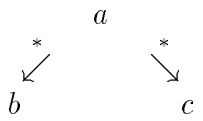


alors il existe d tel que

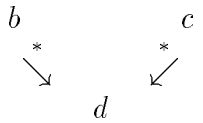


Le lemme suivant est dû à Newman cf. [1].

Lemme 3.3.1 (Confluence) *Si la relation $\rightarrow$ vérifie la propriété diamant alors elle est confluente, autrement dit pour tout a, b, et c tels que*



alors il existe d tel que



Pour prouver le déterminisme des programmes $\mathcal{SCP}$, il suffit alors de vérifier que la relation $\longrightarrow$ de la sémantique opérationnelle a la propriété diamant (cf. [1]). Cette vérification consiste à passer en revue l'ensemble des conflits possibles entre deux transitions. Nous commençons par les transitions contextuelles.

Théorème 3.3.1 (Propriété diamant) *La relation $\stackrel{c}{\longrightarrow}$ vérifie la propriété diamant.*

Preuve

Soit un état initial $\langle S_1 : ... : S_n, \tau, T\rangle$. La paire d'états obtenus par deux transitions $\stackrel{c}{\longrightarrow}$ à partir d'un tel état est appelée paire critique. *Les seules paires critiques possibles sont obtenues par des transitions concurrentes relatives à deux indices i et j distincts.*

Les conflits possibles entre deux transitions concurrentes sont du type écriture/écriture ou lecture/écriture. Lorsque les deux transitions concurrentes lisent ou écrivent sur des éléments distincts, nous aboutissons à des états équivalents pour ce qui concerne les éléments lus ou écrits. C'est toujours le cas pour les programmes. Reste le cas de lectures ou écritures sur les horloges et les variables. Distinguons les cas:

Les conflits écriture/écriture

Horloges. *Une horloge n'est écrite que par l'indice qui la possède. Il n'y a pas de conflit écriture/écriture possible pour les horloges.*

Variables. *Une variable n'est écrite que par l'indice qui la possède et dans son environnement local. Il n'y a pas de conflit écriture/écriture possible pour les variables.*

Les conflits lecture/écriture *il n'existe pas de transition modifiant à la fois une horloge et l'environnement local. Par contre il en existe qui lisent horloges distantes et variables distantes. Nous allons les opposer à des transitions qui mo-*

difient l'environnement local ou bien l'horloge de l'indice sur lequel est effectué la transition.

1. **La transition sur j modifie l'horloge de j.** *Le conflit survient si l'autre transition lit des horloges distantes. C'est le cas si l'indice i réalise une transition nécessitant l'évaluation d'une expression E.*

   Supposons que l'évaluation de l'expression E ne nécessite pas d'évaluation de variables appartenant à l'indice j: il n'existe pas de variable $V \in Var(E)$ tel que $Owner(V) = j$. Dans ce cas la fonction $Evalop(\tau, u, E, T)$ évaluée sur i n'engendre aucune lecture sur l'horloge de j. Il n'y a pas de conflit lecture/écriture possible.

   Supposons que l'évaluation de l'expression E nécessite l'évaluation de variables appartenant à l'indice j: il existe une variable $V \in Var(E)$ telle que $Owner(V) = j$. Dans ce cas la fonction $Evalop(\tau, i, E, T)$ évaluée sur i engendre des lectures sur l'horloge de j.

   La première valeur de sortie de $Evalop$ est évaluée via la fonction $Proj_1(Evalop(\tau, i, E, T))$. Cette valeur dépend des horloges distantes de l'environnement d'horloges T. La fonction $Proj_1(Evalop(\tau, i, E, T))$ évaluée sur i engendre des lectures sur l'horloge de j en testant la condition $\neg(T|_j \prec T|_i)$. Par hypothèse, nous savons qu'à partir de l'état initial, indice i réalise la transition nécessitant l'évaluation de E. Cela signifie que la fonction $Proj_1(Evalop(\tau, i, E, T))$ est évaluable à $vrai$ à partir de l'état initial. Par construction de $Evalop$ nous avons pour tout $V \in Var(E)$, $\neg(T|_{Owner(V)} \prec T|_i)$. Donc à l'état initial nous avons la condition d'attente vérifiée $\neg(T|_j \prec T|_i)$. La proposition 2.2.2 (page 25) de stabilité assure que la condition d'attente est encore vérifiée après mise à jour de $T|_j$. Donc le résultat du test après lecture de l'horloge est le même, que la transition nécessitant l'évaluation de E soit exécutée avant, ou après la transition mettant à jour l'horloge. La seconde valeur de sortie de $Evalop$ est évaluée via la fonction $Proj_2(Evalop(\tau, i, E, T))$. Le résultat de cette fonction ne dépend pas de la valeur des horloges distantes si la condition $Proj_1(Evalop(\tau, i, E, T))$ est vérifiée. Or les règles de la sémantique imposent que le résultat de cette fonction ne soit utilisé que si $Proj_1(Evalop(\tau, i, E, T))$ est évaluée à $vrai$. Donc comme le résultat de $Proj_1(Evalop(\tau, i, E, T))$ ne dépend pas de l'ordre d'exécution des transitions, le résultat de $Proj_2(Evalop(\tau, i, E, T))$ n'en dépend pas plus.

2. **La transition sur j modifie l'environnement local de j.** *Le conflit survient si l'autre transition lit des variables distantes. C'est le cas si l'indice i réalise une transition nécessitant l'évaluation d'une expression E.*

Supposons que l'évaluation de l'expression E ne nécessite pas l'évaluation de la variable V appartenant à l'indice j: $V \notin Var(E)$. Dans ce cas la fonction $Evalop(\tau, i, E, T)$ évaluée sur i n'engendre aucune lecture sur l'environnement local de j. Il n'y a pas de conflit lecture/écriture possible.

Supposons que l'évaluation de l'expression E nécessite l'évaluation de V appartenant à l'indice j. D'une part cela peut modifier le résultat de l'évaluation de E. D'autre part, si l'évaluation de l'expression E passe par l'évaluation d'une indirection fonction de V, l'ensemble des indices dont les horloges sont lues par i dépendent de V. Les deux composantes du résultat de $Evalop(\tau, i, E, T)$ peuvent être affectées. Il nous faut à nouveau distinguer deux cas:

1. *Supposons que les indices i et j réalisent leurs transitions au sein d'un même groupe d'instructions possédant l'horloge t. Dans ce cas j met à jour son environnement local en ajoutant aux valeurs associées à V le couple formé de la nouvelle valeur affectée à V et de l'horloge courante t. L'indice i réalise une transition nécessitant l'évaluation de V. La valeur de cette variable est celle retournée par la fonction $maxstrict(\tau, i, V, T|_u)$ puisque $Owner(V) \neq i$. Il s'agit de la valeur associée à l'horloge t', telle que t' est le maximum parmi l'ensemble des horloges associées à V dans $\tau|_{Owner(V)}$ et qui sont strictement inférieures à t. Il ne peut donc s'agir de la valeur écrite par j dans le même groupe d'instructions.*
2. *Supposons que les indices i et j réalisent leurs transitions au sein de deux groupes d'instructions distincts possédant les horloges t_i et t_j.*

   Si t_i et t_j sont incomparables, ou si $t_i \prec t_j$ *, alors suivant un argument semblable à celui développé précédemment, i lit une valeur associée à une horloge antérieure à t_j alors que j écrit une valeur associée à l'horloge t_j.*

   Si $t_j \prec t_i$ *, alors i peut lire une valeur associée à l'horloge t_j tandis que j écrit une valeur associée à la même horloge. Comme deux transitions forment une paire critique, la fonction $Evalop(\tau, i, V, T)$ avec $T|_i = t_i$ et $T|_j = t_j$ est évaluée à vrai. Nous en déduisons $\neg(T|_{Owner(V)} \prec T|_i)$, donc nous avons $\neg(T|_j \prec T|_i)$, ce qui contredit les hypothèses.*

La valeur de V est lue par i dans un couple valeur/horloge de $\tau|_j$ différent de celui où j écrit.

■

Déterminisme

Nous déduisons de la propriété diamant le déterminisme des exécutions restreintes à un contexte donné.

Proposition 3.3.1 (Déterminisme) *Les exécutions restreintes à un contexte c donné sont déterministes: pour un état initial donné, ou bien tous les calculs sont infinis, ou bien ils terminent tous en donnant le même résultat final.*

Preuve

Le résultat découle des lemme 3.3.1 et théorème 3.3.1. ■

En choisissant $c = true$, il découle du résultat précédent que, pour un état initial donné, toutes les exécutions d'un programme $\mathcal{SCP}$ conduisent au même état final.

Théorème 3.3.2 (Déterminisme) *$\mathcal{SCP}$ est déterministe: pour tout programme $\mathcal{SCP}$ et pour un état initial donné, ou bien tous les calculs sont infinis, ou bien ils terminent tous en donnant le même résultat final.*

Preuve

La preuve est directe par la proposition 3.3.1 en considérant le contexte true.

■

3.4 Conclusion

Dans ce chapitre nous avons formalisé la machine abstraite de $\mathcal{SCP}$, par le biais d'une sémantique opérationnelle à la Plotkin. Ce cadre formel permet de prouver les propriétés fondamentales des exécutions: absence de blocage et déterminisme. Ces propriétés intrinsèquement garanties en $\mathcal{SCP}$, facilitent la maîtrise des programmes et leur validation formelle en s'affranchissant de la multiplicité des exécutions possibles.

La preuve de l'absence de blocage des exécutions de programmes $\mathcal{SCP}$ est basée sur l'existence d'un indice dont la position courante est minimale au sens de l'ordre logique. Cette preuve est simple et générique dans le sens où elle est applicable pour tout langage utilisant un mécanisme de synchronisation basé sur les horloges structurelles. Elle a été développée pour le langage $\mathcal{SCL} - \mathcal{C}han$ [48].

La preuve de déterminisme repose principalement sur la propriété diamant, elle-même reposant sur l'examen des conflits entre les transitions. L'originalité de la preuve réside dans la mise en évidence de conflits sur les horloges résolus grâce à la stabilité de la condition d'attente. Cette preuve est sensible aux hypothèses faites sur les lectures en mémoire distantes. Elle suppose un examen très fin de l'absence de conflits lors de ces accès.

La sémantique opérationnelle du langage $\mathcal{SCP}$ est présentée sur trois niveaux. Cela permet de distinguer la sémantique opérationnelle propre au langage $\mathcal{LH}$. Il est alors très simple d'enrichir $\mathcal{LH}$ en ne modifiant que le niveau de sémantique qui lui est propre.

Chapitre 4

Sémantique dénotationnelle

Ce chapitre est de nature théorique. Notre but est de fonder le modèle de programmation de $\mathcal{SCP}$ par rapport à sa machine abstraite. A cette fin, nous dotons ce langage d'une sémantique dénotationnelle structurée par la syntaxe. Le chapitre est composé de trois parties.

Dans la première partie nous décrivons la sémantique dénotationnelle de $\mathcal{SCP}$. Elle formalise le modèle de programmation. Elle comporte deux niveaux. Le premier niveau correspond à la sémantique dénotationnelle du langage hôte de $\mathcal{SCP}$. Le second niveau correspond à la sémantique de $\mathcal{SCP}$.

La seconde partie constitue la preuve de l'équivalence entre la sémantique dénotationnelle et la sémantique opérationnelle fondant le modèle d'exécution et de programmation. Le théorème central est trouvé page 80 (théorème 4.2.1). La preuve repose sur la considération de calculs moins asynchrones que les autres correspondant intuitivement à une exécution pas à pas de la sémantique dénotationnelle. Nous introduisons successivement les notions de correspondance des environnements (4.2.1 page 78), de cohérence des environnements (4.2.2 page 79) et d'état synchrone (4.2.3 page 79). Le théorème d'équivalence est prouvé en deux étapes, les propositions (4.2.1 page 80) et (4.2.2 page 98) en constituent respectivement les conditions nécessaire et suffisante.

La troisième partie traite de la correction de la fonction de traduction de code séquentiel en code parallèle. Le théorème central est trouvé page 104 (Théorème 4.3.1).

4.1 Sémantique dénotationnelle

Le but est de fonder le modèle de programmation par l'introduction d'une sémantique dénotationnelle. Elle donne pour chaque programme la fonction calculée.

Commençons par introduire quelques notations.

- Un environnement, est une fonction ψ renvoyant pour toute variable X la valeur contenue dans la mémoire. On note $MemEnv$ l'ensemble des *environnements* dans lesquels sont sauvegardées les valeurs des différentes variables.

- On note Ctx l'ensemble des *contextes*. Un contexte c est un vecteur de booléens. Il représente l'activité des indices. Un indice u est actif si $c|_u = true$ (noté aussi $u \in c$).

4.1.1 Sémantique du langage hôte

Pour chaque instruction, le résultat calculé dépend de l'indice ayant exécuté cette instruction. Nous introduisons la fonction $Eval(\psi_1, \psi_2, u, V)$ qui permet d'évaluer une variable dans le code hôte. Cette fonction retourne la valeur de V en fonction des deux environnements ψ_1, ψ_2 et de l'indice u. L'idée intuitive est la suivante, l'environnement ψ_1 représente la mémoire des indices distants alors que ψ_2 représente la mémoire locale. La valeur d'une variable scalaire ($V \in Var$) dépend de l'indice u qui l'évalue.

- Si $Owner(V) = u$, V est évaluée dans ψ_2.
- Si $Owner(V) \neq u$, V est évaluée dans ψ_1.

Si la variable représente un vecteur ($V \in Ind$), alors elle est toujours évaluée dans ψ_2.

On note $[\![S]\!]_{\mathcal{LH}}$ la sémantique d'un programme S du langage hôte $\mathcal{LH}$. A partir d'un élément du domaine $(MemEnv \times MemEnv \times Ctx)$ elle renvoie un élément de $(MemEnv \times MemEnv)$. Du point de vue du programme global, il suffit de préciser qu'elle ne modifie pas le contexte et que toute variable distante est lue dans un environnement qui n'est jamais modifié. Les valeurs affectées le sont dans un second environnement.

Définition 4.1.1 *La fonction Eval est une fonction de* $(MemEnv \times MemEnv \times Proc \times Var \cup Ind)$ *dans* Val. *Nous avons:*

$$Eval(\psi_1, \psi_2, u, V) = \begin{cases} \psi_2(V) & si\ u = Owner(V)\ et\ V \in Var), \\ \psi_2(V|u) & si\ V \in Ind, \\ \psi_1(V) & sinon \end{cases}$$

Il est aisé d'étendre la fonction $Eval$ des variables aux expressions. Soit $Var(E)$ l'ensemble des variables qu'il est nécessaire d'évaluer pour calculer l'expression E. Pour évaluer $Eval(\psi_1, \psi_2, u, E)$ chaque variable V de $Var(E)$ est évaluée par le biais de $Eval(\psi_1, \psi_2, u, V)$. Ensuite l'ensemble des valeurs calculées est utilisé pour évaluer E.

Nous donnons la sémantique de $\mathcal{LH}$ pour les affectations et leur séquence:

- Affectation d'un élément de tableau:

$$[\![X[E_1] = E_2]\!]_{\mathcal{LH}}(\psi_1, \psi_2, c) = (\psi_1, \psi_2'),$$

avec ψ_2' définit de la façon suivante. On pose pour tout $Z \in Var \cup Ind$ et indice u:

$$Eval(\psi_1, \psi_2', u, Z) = Eval(\psi_1, \psi_2, u, E_2) \text{ si } \begin{cases} u \in c \text{ et} \\ Z = X[Eval(\psi_1, \psi_2, u, E_1)] \text{ et} \\ Owner(Z) = u \end{cases},$$

sinon

$$Eval(\psi_1, \psi_2', u, Z) = \ Eval(\psi_1, \psi_2, u, Z).$$

- Affectation d'une variable: $X = E$ avec $X \in Var$. Cette règle se déduit aisément de la règle précédente.

$$[\![X = E]\!]_{\mathcal{LH}}(\psi_1, \psi_2, c) = (\psi_1, \psi_2'),$$

avec ψ_2' définit de la façon suivante. On pose pour tout $Z \in Var \cup Ind$ et indice u:

$$Eval(\psi_1, \psi_2', u, Z) = Eval(\psi_1, \psi_2, u, E) \text{ si } \begin{cases} u \in c \text{ et} \\ Z = X \text{ et} \\ Owner(Z) = u \end{cases},$$

sinon

$Eval(\psi_1, \psi_2', u, Z) = \ Eval(\psi_1, \psi_2, u, Z).$

- Affectation d'un vecteur:$I = E$ avec $I \in Ind$. Toutes les composantes appartenant au indices du contexte sont modifiées.

$$[\![I = E]\!]_{\mathcal{LH}}(\psi_1, \psi_2, c) = (\psi_1, \psi_2'),$$

avec ψ_2' définit de la façon suivante. On pose pour tout $Z \in Var \cup Ind$ et indice u:

$$Eval(\psi_1, \psi_2', u, Z) = Eval(\psi_1, \psi_2, u, E) \text{ si } \begin{cases} u \in c \text{ et} \\ Z = I \end{cases},$$

sinon

$Eval(\psi_1, \psi_2', u, Z) = \ Eval(\psi_1, \psi_2, u, Z).$

- Séquence d'instructions $\mathcal{LH}$: $S; T$

$$[\![S; T]\!]_{\mathcal{LH}}(\psi_1, \psi_2, c) = (\psi_1, \psi_2''),$$

avec ψ_2'' tel que:

$$[\![S]\!]_{\mathcal{LH}}(\psi_1, \psi_2, c) = (\psi_1, \psi_2'),$$

et

$$[\![T]\!]_{\mathcal{LH}}(\psi_1, \psi_2', c) = (\psi_1, \psi_2'')$$

4.1.2 Sémantique de $\mathcal{SCP}$

Pour symboliser les programmes qui ne terminent pas, nous introduisons une valeur spéciale $\perp$ modélisant l'état indéfini. On note $[\![S]\!]$ la sémantique d'un programme S du langage $\mathcal{SCP}$. La sémantique dénotationnelle associe à tout programme S la fonction qu'il calcule, notée $[\![S]\!]$. La fonction $[\![S]\!]$ est définie par induction sur la structure des groupes d'instructions du programme. A partir d'un élément du domaine $(MemEnv \times Ctx) \cup \{\perp\}$

elle renvoie un élément de $MemEnv \cup \{\bot\}$. L'état $\bot$ est stable: pour tout programme S : $[\![S]\!](\bot) = \bot$.

Groupe d'instructions : step() Cette primitive correspond à l'exécution d'un groupe d'instructions.

$$[\![\mathsf{step}\ (S)]\!](\psi, c) = \psi'$$

avec $[\![S]\!]_{\mathcal{LH}}(\psi, \psi, c) = (\psi, \psi')$

Séquence : $S;T$. Tous les indices actifs exécutent le programme S. Ensuite, ils exécutent T.

$$[\![S;T]\!](\psi, c) = [\![T]\!]([\![S]\!](\psi, c), c)$$

Boucle : loopwhere B do S end. Tant qu'il existe des indices actifs pour lesquels l'expression booléenne B s'évalue à vrai, ces indices exécutent S. La boucle termine lorsque tous les indices actifs au début de l'exécution évaluent B à faux. Nous étendons la fonction $Eval(\psi_1, \psi_2, u, B)$ de manière à ce que $Eval(\psi_1, \psi_2, B)$ renvoie un vecteur booléen à une composante par indice: $Eval(\psi_1, \psi_2, B)|_u = Eval(\psi_1, \psi_2, u, B)$. On utilise de manière classique une sémantique dénotationnelle du plus petit point fixe [57]. Lorsque la boucle termine, le calcul de la sémantique dénotationnelle est basé sur le plus petit point fixe, noté $fix(F)$, de la fonctionnelle F définie par:

$$\begin{aligned} F = \lambda f\ \lambda(\psi, c)\ if\ (\exists u\ tel\ que\ (c \wedge Eval(\psi, \psi, u, B))_u = true) \\ then f([\![S]\!](\psi, c \wedge Eval(\psi, \psi, B)), c \wedge Eval(\psi, \psi, B)) \\ else \psi \end{aligned}$$

$$[\![\mathsf{loopwhere}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end}]\!](\psi, c) = fix(F)(\psi, c)$$

Si la boucle ne termine pas, la sémantique renvoie l'état indéfini $\bot$:

$$[\![\mathsf{loopwhere}\ \ B\ \ \mathsf{do}\ \ S\ \ \mathsf{end}]\!](\psi, c) = \bot$$

Conditionnelle : where B do S elsewhere Q end. La conditionnelle conduit à une exécution indépendante des deux branches. Les instructions exécutées dans une branche modifiant des variables accédées dans l'autre branche sont masquées jusqu'à la fin de l'exécution des deux branches. Les données modifiées par de telles instructions sont enregistrées dans les environnements. En sortie de conditionnelle, le nouvel environnement est recomposé.

Pour simplifier les expressions, on utilise le vecteur booléen $b_1 = c \wedge Eval(\psi, \psi, B)$ (resp. $b_2 = c \wedge \neg Eval(\psi, \psi, B)$) représentant le contexte associé à la première branche (resp. la seconde branche) de la conditionnelle. On applique $[\![]\!]$ à la première branche avec le contexte b_1 :

$$[\![S]\!](\psi, b_1) = \psi'_1$$

Afin d'exprimer l'indépendance entre les exécutions des branches, on reprend l'environnement initial ψ pour le calcul de la seconde branche.

$$[\![Q]\!](\psi, b_2) = \psi'_2$$

Voici la sémantique du where/elsewhere:

$$[\![\textsf{where}\ \ B\ \ \textsf{do}\ \ S\ \ \textsf{elsewhere}\ \ Q\ \ \textsf{end}]\!](\psi, c) = \psi'$$

L'environnement final ψ' est obtenu à partir des environnements ψ'_1 et ψ'_2. Pour toute variable ou composante de vecteur X on a:

$$\psi'(X) = \begin{cases} \psi'_1(X) & \text{si} \begin{cases} (X \in Var\ et\ b_1|_{Owner(X)} = true)\ ou \\ (X = I|_u\ (tel\ que\ I \in Ind)\ et\ b_1|_u = true) \end{cases} \\ \psi'_2(X) & \text{sinon} \end{cases},$$

La sémantique du where se déduit aisément de celle du where/elsewhere:

$$[\![\textsf{where}\ \ B\ \ \textsf{do}\ \ S\ \ \textsf{end}]\!](\psi, c) = [\![\textsf{where}\ \ B\ \ \textsf{do}\ \ S\ \ \textsf{elsewhere step ()}\ \ \textsf{end}]\!](\psi, c) = \psi'$$

avec:

$$[\![S]\!](\psi, b_1) = \psi'_1$$

L'environnement final ψ' est construit à partir de l'environnement ψ'_1. Pour toute variable ou composante de vecteur X on a:

$$\psi'(X) = \begin{cases} \psi'_1(X) & \text{si} \begin{cases} (X \in Var\ et\ b_1|_{Owner(X)} = true)\ ou \\ (X = I|_u\ (tel\ que\ I \in Ind)\ et\ b_1|_u = true) \end{cases} \\ \psi(X) & \text{sinon} \end{cases},$$

4.2 Equivalence fondamentale

Le but de cette section est de fonder la correction du modèle d'exécution par rapport au modèle de programmation. On montre que le résultat donné par la sémantique dénotationnelle pour un programme S est identique au résultat donné par les calculs de la sémantique opérationnelle : ou bien on obtient les mêmes environnements privés ou bien la sémantique dénotationnelle renvoie l'état indéfini $\perp$ et tous les calculs sont infinis.

Le théorème central (théorème 4.2.1 page 80) est prouvé en distinguant la condition nécessaire (proposition 4.2.1 page 80) et la condition suffisante (proposition 4.2.2 page 98).

Chaque preuve est faite par induction sur la structure du programme. L'outil essentiel est la notion *d'état synchrone* (définition 4.2.3 page 79). Intuitivement, les états synchrones correspondent aux étapes d'un calcul tel qu'il est décrit par la sémantique dénotationnelle. La preuve est faite par induction sur la structure du programme en prouvant à chaque étape la stabilité de la notion d'état synchrone.

La *cohérence des environnements* multi niveaux assure qu'à chaque étape de l'induction, toutes les données sont estampillées par des horloges strictement plus petites que l'horloge courante de l'émetteur et pas plus grandes que l'horloge courante des éventuels récepteurs (définition 4.2.2 page 79). Cette propriété est essentielle pour prouver la stabilité de la correspondance des environnements au cours des différentes étapes de l'induction.

La preuve est structurée en plusieurs parties. On introduit dans la première partie une notion d'équivalence entre les environnements multi niveaux de la sémantique opérationnelle et les environnements de la sémantique dénotationnelle (définition 4.2.1). La seconde partie traite de la cohérence des environnements (définition 4.2.2). La troisième présente la définition d'état synchrone (définition 4.2.3). Le théorème d'équivalence est énoncé dans la quatrième partie (théorème 4.2.1). La condition nécessaire est prouvée dans la cinquième (proposition 4.2.1) et la condition suffisante dans la sixième (proposition 4.2.2).

4.2.1 Correspondance des environnements

Pour montrer la correction du modèle de programmation, il est nécessaire de pouvoir s'assurer qu'une lecture dans un environnement renvoie la même valeur dans la sémantique opérationnelle et la sémantique dénotationnelle. Or les environnements de communication de la sémantique opérationnelle et de la sémantique dénotationnelle ont des structures distinctes. On formalise, par une fonction $Equiv$, une correspondance entre les deux types d'environnement.

L'idée intuitive de cette correspondance est la suivante: à partir de son horloge, chaque indice du contexte évalue une variable distante dans l'environnement multi niveaux à la même valeur que celle stockée dans l'environnement ψ_1 représentant la mémoire des indices distants. Symétriquement, à partir de son horloge, chaque indice du contexte évalue une variable locale dans l'environnement multi niveaux à la même valeur que celle stockée dans l'environnement ψ_2 représentant la mémoire locale de l'indice.

La fonction $Equiv$ est définie à partir des deux fonctions $Evalop$ (3.1.1 page 55) et $Eval$ (4.1.1 page 74).

Définition 4.2.1 (Equivalence des environnements) *Soit un environnement multi niveaux τ, des environnements ψ_1 et ψ_2, un environnement d'horloges structurelles T et un contexte c. Pour toute variable $V \in Var \cup Ind$, pour tout indice u du contexte c, on dit que τ est équivalent au couple (ψ_1, ψ_2) si $Proj_2(Evalop(\tau, u, V, T)) = Eval(\psi_1, \psi_2, u, V)$.*

Nous le notons: $Equiv(\tau, \psi_1, \psi_2, T, c)$

4.2.2 Cohérence des environnements

La cohérence de l'environnement multi niveaux assure que toutes les données sont estampillées par des horloges strictement plus petites que l'horloge de l'émetteur et pas plus grandes que celle du récepteur.

Définition 4.2.2 *Un environnement τ est* cohérent *vis-à-vis d'un environnement d'horloges T si pour tout $X \in Var$, pour tout $i \in Proc$ tel que $i \neq Owner(X)$ et pour toute paire (Val, t) de la liste $\tau|_{Owner(X)}(X)$, on a $t \preceq T|_{Owner(X)}$ et $\neg(T|_i \prec t)$.*

Le lemme simple suivant exprime la stabilité de la cohérence par croissance des horloges.

Lemme 4.2.1 *Soit un environnement τ cohérent vis-à-vis de T. Si un environnement d'horloges T' vérifie pour tout indice u $T'|_u \succeq T|_u$ alors τ est cohérent vis-à-vis de T'.*

Preuve

La preuve est décomposée en deux phases correspondant aux deux propriétés des états cohérents.

1. *Pour toute paire (Val, t) de la liste $\tau|_{Owner(X)}(X)$, on a: $t \preceq T|_{Owner(X)}$. Or par hypothèse on a $T'|_{Owner(X)} \succeq T|_{Owner(X)}$ donc $t \preceq T'|_{Owner(X)}$*
2. *On prouve la seconde propriété par l'absurde. Supposons que $T'|_{Owner(X)} \preceq t$. Or par hypothèse on a $T'|_{Owner(X)} \succeq T|_{Owner(X)}$ donc $T|_{Owner(X)} \preceq t$, ce qui contredit les hypothèses.*

■

4.2.3 Etat synchrone

On définit un état particulier, appelé *état synchrone*, qui correspond intuitivement à une exécution pas à pas de la sémantique dénotationnelle. La notion d'état synchrone regroupe un ensemble de propriétés sur les éléments de $(S, S_1 : .... : S_n, T, c, \tau, \psi_1, \psi_2)$.

Définition 4.2.3 (Etat synchrone) *Un état $(S, S_1 : .... : S_n, T, c, \tau, \psi_1, \psi_2)$ est synchrone s'il vérifie les conditions suivantes :*

- *pour tout $i \in c$, $S_i = S$,*
- *les indices du contexte ont la même horloge : il existe une horloge t telle que pour tout indice $i \in c$, $T|_i = t$,*
- *le triplet $(S_1 : ... : S_n, T, c)$ est non bloquant,*
- *l'environnement multi niveaux τ est cohérent vis-à-vis de T,*
- *les environnements τ et (ψ_1, ψ_2) sont équivalents: $Equiv(\tau, \psi_1, \psi_2, T, True)$.*

4.2.4 Théorème d'équivalence

On est maintenant à même d'énoncer le théorème d'équivalence.

Théorème 4.2.1 (Equivalence sémantique) *Soit S un programme $\mathcal{SCP}$, T une horloge dont toutes les composantes sont égales, $True$ un contexte où tous les indices sont actifs, τ un environnement multi niveaux, et ψ un environnement tels que: $Evalop(\tau, u, V, T) = nil$, $Eval(\psi, \psi, u, V) = nil$. Soit un environnement multi niveaux τ', et un environnement ψ' tels que: $Equiv(\tau', \psi', \psi', T', True)$. Alors:*

$$[\![S]\!](\psi, True) = \psi' \neq \bot$$

si et seulement si

$$\langle S : ... : S, \tau, T\rangle \xrightarrow{*} \langle \bullet : ... : \bullet, \tau', T'\rangle.$$

On fait une preuve par induction sur la structure du programme. Pour plus de clarté on traite la condition nécessaire et la condition suffisante séparément. Après avoir noté que l'état initial $(S, S : .... : S, T, True, \tau, \psi, \psi)$ est synchrone, on applique la proposition 4.2.1 page 80 pour prouver la condition nécessaire et la proposition 4.2.2 page 98 pour prouver la condition suffisante.

4.2.5 Théorème d'équivalence : condition nécessaire

Par souci de clarté, on introduit une *propriété d'induction*. A partir d'un état synchrone, la propriété d'induction associe au résultat de la sémantique dénotationnelle, un calcul de la sémantique opérationnelle produisant un nouvel état synchrone.

Définition 4.2.4 (Propriété d'induction) *Un programme $\mathcal{SCP}$ S vérifie la propriété d'induction s'il vérifie la propriété suivante :*
pour tout état synchrone $(S, S_1 : .... : S_n, T, c, \tau, \psi, \psi)$ si

$$[\![S]\!](\psi, c) = \psi' \neq \bot,$$

alors il existe $\xrightarrow{c^*}$ tel que

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle$$

où $(\bullet, S'_1 : .... : S'_n, T', c, \tau', \psi', \psi')$ est un état synchrone.

Proposition 4.2.1 (Condition nécessaire) *Tout programme $\mathcal{SCP}$ S vérifie la propriété d'induction.*

Preuve

La preuve se fait par induction sur la structure du programme. Les différentes étapes de l'induction sont obtenues à partir des lemmes 4.2.2 à 4.2.20. ■

Un premier groupe de lemmes traite des exécutions au sein d'un groupe d'instructions (lemmes 4.2.2 à 4.2.9). Les lemmes suivants traitent des exécutions entre groupes d'instructions.

Remarques.

- Tout état $\langle S'_1 : ... : S'_n, \tau', T' \rangle$ issu de l'état $\langle S_1 : ... : S_n, \tau, T \rangle$ par une transition $\xrightarrow{c^*}$ correspond à un triplet $(S'_1 : ... : S'_n, T', c)$ non bloquant d'après le lemme 3.2.1 page 64. Dans les preuves qui suivent, on omettra de justifier à nouveau le non blocage pour ce type de situation, afin d'alléger les démonstrations.
- La plupart des preuves des lemmes techniques nécessaires à la démonstration de la condition nécessaire sont reportées en annexe (section 4.5 page 110). Elles ne présentent pas de difficulté particulière.

Exécution au sein d'un groupe d'instructions

On se préoccupe maintenant de l'étape d'induction relative aux groupes d'instructions. Les environnements en entrée de la sémantique dénotationnelle du langage hôte ne sont pas nécessairement égaux comme dans le cas de $\mathcal{SCP}$. On définit une propriété d'induction propre aux exécutions dans le langage hôte.

Définition 4.2.5 (Propriété d'induction de $\mathcal{LH}$) *Un programme $\mathcal{LH}$ S vérifie la propriété d'induction de $\mathcal{LH}$ s'il vérifie la propriété suivante :*
pour tout état synchrone $(S, S_1 : .... : S_n, T, c, \tau, \psi_1, \psi_2)$ si

$$[\![S]\!]_{\mathcal{LH}}(\psi_1, \psi_2, c) = (\psi'_1, \psi'_2),$$

alors il existe $\xrightarrow{c^*}$ tel que

$$\langle S_1 : ... : S_n, \tau, T \rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T' \rangle$$

où $(\bullet, S'_1 : .... : S'_n, T', c, \tau', \psi'_1, \psi'_2)$ est un état synchrone.

Le lemme intermédiaire suivant donne le résultat de l'évaluation d'un environnement multi niveaux par la fonction *Evalop* après l'exécution d'une affectation de tableau. Il fait le lien entre la définition de la fonction Evalop et la construction des environnements multi niveaux.

Lemme 4.2.2 *Soit τ cohérent, et T tel que pour tout indice i appartenant au contexte c, il existe une horloge t telle que $T|_i = t$. Soit S_i tel que pour tout i appartenant au contexte c, $S_i = X[E_1] = E_2$. Considérons la transition contextuelle $\xrightarrow{c^*}$ telle que:*

$$\langle S_1 : ... : S_n, \tau, T \rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T \rangle$$

Posons $w = Owner(X[Proj_2(Evalop(\tau, v, E_1, T))])$. Alors, τ' est tel que pour toute variable Z, tout indice v, tout environnement d'horloges T' on a:

$$Proj_2(Evalop(\tau', v, Z, T')) =$$
$$\begin{cases} Proj_2(Evalop(\tau, w, E_2, T)) \text{ si } \begin{cases} w \in c \text{ et } ((v = w \text{ et } T'|_v = T|_w) \text{ ou } T'|_v \succ T|_w) \text{ et} \\ Z = X[Proj_2(Evalop(\tau, v, E_1, T))]) \end{cases} \\ Proj_2(Evalop(\tau, v, Z, T')) \text{ sinon} \end{cases},$$

Preuve

Voir section 4.5.1 page 110. ■

Lemme 4.2.3 *Le programme $\mathcal{LH}$ réduit à une affectation $X[E_1] = E_2$ vérifie la propriété d'induction de $\mathcal{LH}$.*

Preuve

Tous les indices actifs i vérifient la condition d'attente $Proj_1(Evalop(\tau, i, E_1, T)) \wedge Proj_1(Evalop(\tau, i, E_2, T))$. Les indices j auquels i se réfère peuvent être actifs ou non. Si j n'est pas actif alors, par hypothèse de non blocage, on a $\neg(T|_j \prec T|_i)$. Si j est actif, alors il a la même horloge que i. Chaque indice du contexte peut donc exécuter l'instruction $S: X[E_1] = E_2$ par une transition $\xrightarrow{c^*}$:

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T\rangle$$

Considérons maintenant le résultat de la sémantique dénotationnelle.

$$[\![S]\!]_{\mathcal{LH}}(\psi_1, \psi_2, c) = (\psi_1, \psi'_2)$$

avec pour toute variable Z et pour tout indice v:

$$Eval(\psi_1, \psi'_2, v, Z) = \begin{cases} Eval(\psi_1, \psi_2, v, E) & si \begin{cases} v \in c \text{ et} \\ Z = X \text{ et} \\ Owner(Z) = v \end{cases} \\ Eval(\psi_1, \psi_2, v, Z) & sinon \end{cases}, \tag{4.1}$$

Montrons que l'état résultant est synchrone. Pour le prouver, nous montrons que l'environnement résultant de la sémantique opérationnelle est cohérent et qu'il correspond aux environnements de la sémantique dénotationnelle.

1. *Montrons que l'environnement τ' est cohérent. Par hypothèse, l'environnement τ est cohérent vis-à-vis de l'environnement d'horloges T. Rappelons la sémantique de l'affectation d'un élément de tableau $X[E_1] = E_2$:*

$$\frac{Proj_1(Evalop(\tau, u, E_1, T)) \wedge Proj_1(Evalop(\tau, u, E_2, T))}{\langle X[E_1] = E_2, \tau, u, T\rangle \mapsto \langle \bullet, \tau', u, T\rangle}$$

avec τ' est tel que pour toute variable Z:

$$\tau'|_{Owner(Z)}(Z) = \begin{cases} \tau|_{Owner(Z)}(Z)(Proj_2(Evalop(\tau, u, E_2, T)), T|_{Owner(Z)}) \\ \qquad si \begin{cases} Z = X[Proj_2(Evalop(\tau, u, E_1, T))] \text{ et} \\ u = Owner(Z) \end{cases} \\ Change(\tau|_{Owner(Z)}(Z), Proj_2(Evalop(\tau, u, E_2, T))) \\ \qquad si \begin{cases} Z = X[Proj_2(Evalop(\tau, u, E_1, T))] \text{ et} \\ T|_u = Top(\tau, Z) \text{ et} \\ u = Owner(Z) \end{cases} \\ \tau'|_{Owner(Z)}(Z) \text{ sinon.} \end{cases},$$

Seules les listes $\tau|_i(Z)$ *avec* $Z = X[Proj_2(Evalop(\tau, i, E_1, T))]$ *et* $i = Owner(Z)$, *sont modifiées. Puisque l'environnement d'horloges n'est pas modifié, les autres restent cohérentes. Les listes* $\tau|_i(Z)$ *avec* $Z = X[Proj_2(Evalop(\tau, i, E_1, T))]$ *et* $i = Owner(Z)$, *sont modifiées par la transition* $\xrightarrow{c^*}$ *par l'ajout on la modification d'une paire* $(X, T|_i)$. *Par hypothèse pour tout* $j \in Proc$ *tel que* $j \neq Owner(Z)$ *et pour toute paire* (Val, t) *de* $\tau|_{Owner(Z)}(Z)$, *on a* $t \preceq T|_{Owner(Z)}$ *et* $\neg(T|_j \prec t)$. *Il reste à verifier que* $T|_i \preceq T|_{Owner(Z)}$ *(immédiat) et* $\neg(T|_j \prec |T_i)$.

- *Si* $j \in c$, *on a:* $T|_j = T|_i$, *donc:* $\neg(T|_j \prec T|_i)$
- *Si* $j \notin c$, *on ne peut avoir* $T|_j \prec T|_i$ *puisque le triplet* $(S'_1 : ... : S'_n, T, c)$ *est non bloquant.*

2. *Montrons que:* $Equiv(\tau', \psi_1, \psi'_2, T, True)$. *Posons* $w = Owner(X[Proj_2(Evalop(\tau, v, E_1, T))])$. *D'après le lemme 4.2.2, l'environnement* τ' *est tel que pour toute variable* Z, *tout indice* v, *tout environnement d'horloges* T' *on a:*

$$Proj_2(Evalop(\tau', v, Z, T')) = \begin{cases} Proj_2(Evalop(\tau, w, E_2, T)) \\ \quad si \begin{cases} w \in c \text{ et } ((v = w \text{ et } T'|_v = T|_w) \text{ ou } T'|_v \succ T|_w) \text{ et} \\ Z = X[Proj_2(Evalop(\tau, v, E_1, T))]) \end{cases} \\ Proj_2(Evalop(\tau, v, Z, T')) \text{ sinon} \end{cases}, \tag{4.2}$$

Soit une variable Z *et un indice* v. *Par hypothèse on a* $Equiv(\tau, \psi_1, \psi_2, T, True)$. *Donc:*

$$X[Eval(\psi_1, \psi_2, v, E_1)] = X[Proj_2(Evalop(\tau, v, E_1, T))], \tag{4.3}$$

et

$$Owner(X[Eval(\psi_1, \psi_2, v, E_1)]) = Owner(X[Proj_2(Evalop(\tau, v, E_1, T))]). \tag{4.4}$$

Distinguons le cas où v *est dans le contexte* c *et celui où il n'en fait pas partie:*

- *D'après (4.1) si* $v \notin c$, *on a:* $Eval(\psi_1, \psi'_2, v, Z) = Eval(\psi_1, \psi_2, v, Z)$. *Par hypothèse* $Equiv(\tau, \psi_1, \psi_2, T, True)$, *donc* $Eval(\psi_1, \psi_2, v, Z) = Proj_2(Evalop(\tau, v, Z, T))$. *D'après (4.2) on a* $Proj_2(Evalop(\tau', v, Z, T)) = Proj_2(Evalop(\tau, v, Z, T))$. *On en déduit* $Proj_2(Evalop(\tau', v, Z, T)) = Eval(\psi_1, \psi'_2, v, Z)$.
- *Si* $v \in c$, *alors nous distinguons le cas où* $Z = X[Eval(\psi_1, \psi_2, v, E_1)]$ *de celui où cette égalité n'est pas vérifiée.*
  - ▷ *Si* $Z \neq X[Eval(\psi_1, \psi_2, v, E_1)]$ *on a d'après (4.1)* $Eval(\psi_1, \psi'_2, v, Z) = Eval(\psi_1, \psi_2, v, Z)$. *Par hypothèse* $Equiv(\tau, \psi_1, \psi_2, T, c)$, *donc on a* $X[Eval(\psi_1, \psi_2, v, E_1)] = X[Proj_2(Evalop(\tau, v, E_1, T))]$. *D'après (4.3) on* a

$Z \neq X[Proj_2(Evalop(\tau, v, E_1, T))]$, *donc d'après (4.2), pour toute horloge T', on a* $Proj_2(Evalop(\tau', v, Z, T')) = Proj_2(Evalop(\tau, v, Z, T'))$. *En particulier on a l'égalité suivante:* $Proj_2(Evalop(\tau', v, Z, T)) = Proj_2(Evalop(\tau, v, Z, T))$. *De* $Equiv(\tau, \psi_1, \psi_2, T, c)$, *on déduit* $Proj_2(Evalop(\tau', v, Z, T)) = Eval(\psi_1, \psi_2', v, Z)$

- ▷ *Si* $Z = X[Eval(\psi_1, \psi_2, v, E_1)]$, *alors nous distinguons le cas où* $v = Owner(X[Eval(\psi_1, \psi_2, v, E_1)])$ *de celui où cette égalité n'est pas vérifiée.*
  - ∗ *Si* $v \neq Owner(X[Eval(\psi_1, \psi_2, v, E_1)])$, *on a d'après (4.1)* $Eval(\psi_1, \psi_2', v, Z) = Eval(\psi_1, \psi_2, v, E_2)$. *D'après (4.3) et (4.4) on a* $Z = X[Proj_2(Evalop(\tau, v, E_1, T))]$ *et* $v \neq Owner(X[Proj_2(Evalop(\tau, v, E_1, T))])$. *Donc pour T on a d'après (4.2)* $Proj_2(Evalop(\tau', v, Z, T)) = Proj_2(Evalop(\tau, v, Z, T))$. *De* $Equiv(\tau, \psi_1, \psi_2, T, c)$, *on déduit* $Proj_2(Evalop(\tau', v, Z, T)) = Eval(\psi_1, \psi_2', v, Z)$.
  - ∗ *Si* $v = Owner(X[Eval(\psi_1, \psi_2, v, E_1)])$ *on a d'après (4.1)* $Eval(\psi_1, \psi_2', v, Z) = Eval(\psi_1, \psi_2, v, E_2)$. *D'après (4.3) et (4.4) on a* $Z = X[Proj_2(Evalop(\tau, v, E_1, T))]$ *et* $v = Owner(X[Proj_2(Evalop(\tau, v, E_1, T))])$. *Donc pour T on a d'après (4.2)* $Proj_2(Evalop(\tau', v, Z, T)) = Proj_2(Evalop(\tau, v, E_2, T))$. *De* $Equiv(\tau, \psi_1, \psi_2, T, c)$, *on déduit* $Proj_2(Evalop(\tau, v, E_2, T)) = Eval(\psi_1, \psi_2, v, E_2)$ *et donc* $Proj_2(Evalop(\tau', v, Z, T)) = Eval(\psi_1, \psi_2', v, Z)$.

■

Le lemme intermédiaire suivant donne le résultat de l'évaluation d'un environnement multi niveaux par la fonction *Evalop* après l'exécution d'une affectation de scalaire. Il fait le lien entre la définition de la fonction Evalop et la construction des environnements multi niveaux.

Lemme 4.2.4 *Soit* τ *cohérent, et T tel que pour tout indice i appartenant au contexte c, il existe une horloge t telle que* $T|_i = t$. *Soit* S_i *tel que pour tout i appartenant au contexte c,* $S_i = X = E$ *où* $X \in Var$. *Considérons la transition contextuelle* $\xrightarrow{c^*}$ *telle que:*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S_1' : ... : S_n', \tau', T\rangle$$

Posons $w = Owner(X[Proj_2(Evalop(\tau, v, E, T))])$. *Alors,* τ' *est tel que pour toute variable Z, tout indice v, tout environnement d'horloges T' on a:*

$$Proj_2(Evalop(\tau', v, Z, T')) = \begin{cases} Proj_2(Evalop(\tau, w, E, T)) \text{ si } \begin{cases} w \in c \text{ et } ((v = w \text{ et } T'|_v = T|_w) \text{ ou } T'|_v \succ T|_w) \text{ et} \\ Z = X) \end{cases} \\ Proj_2(Evalop(\tau, v, Z, T')) \text{ sinon} \end{cases},$$

Preuve

La preuve est similaire à celle du lemme 4.2.2 page 81. ■

Lemme 4.2.5 *Le programme $\mathcal{LH}$ réduit à une affectation de scalaire $X = E$ vérifie la propriété d'induction de $\mathcal{LH}$.*

Preuve

On ne détaille pas cette démonstration qui est très similaire à celle donnée pour le lemme 4.2.3 page 82. Il suffit d'appliquer le lemme 4.2.4 et de remplacer $X[Proj_2(Evalop(\tau, v, E_1, T))]$ par X. L'étape de vérification de correspondance entre $X[Proj_2(Evalop(\tau, v, E_1, T))]$ et $X[Eval(\psi_1, \psi_2, v, E_1))]$ devient inutile.

■

Le lemme intermédiaire suivant donne le résultat de l'évaluation d'un environnement multi niveaux par la fonction *Evalop* après l'exécution d'une affectation de vecteur. Il fait le lien entre la définition de la fonction Evalop et la construction des environnements multi niveaux.

Lemme 4.2.6 *Soit S_i tel que pour tout i appartenant au contexte c, $S_i = S_i : I = E$ où $I \in Ind$. Considérons la transition contextuelle $\xrightarrow{c^*}$ telle que:*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T\rangle$$

Alors, τ' est tel que pour toute variable Z, tout indice v, tout environnement d'horloges T' on a:

$$Proj_2(Evalop(\tau', v, Z, T')) =$$
$$\begin{cases} Proj_2(Evalop(\tau, v, E, T)) \text{ si } v \in c \text{ et } Z = I, \\ Proj_2(Evalop(\tau, v, Z, T')) \text{ sinon} \end{cases}$$

Preuve

Voir section 4.5.1 page 111. ■

Lemme 4.2.7 *Le programme $\mathcal{LH}$ réduit à une affectation de vecteur $I = E$ vérifie la propriété d'induction de $\mathcal{LH}$.*

Preuve

Cette preuve est similaire à celle du lemme 4.2.3.

■

Lemme 4.2.8 *La séquence de deux programmes $\mathcal{LH}$ qui vérifient la propriété d'induction de $\mathcal{LH}$, vérifie la propriété d'induction de $\mathcal{LH}$.*

Preuve

Soit S et P deux programmes $\mathcal{LH}$ qui vérifiant la propriété d'induction de $\mathcal{LH}$. On a par hypothèse d'induction:

Pour tout état synchrone $(S, S_1 : .... : S_n, T, c, \tau_S, \psi_{S1}, \psi_{S2})$, si

$$[\![S]\!]_{\mathcal{LH}}(\psi_{S1}, \psi_{S2}, c) = (\psi'_{S1}, \psi'_{S2}) \neq \perp,$$

alors il existe $\xrightarrow{c^*}$ tel que

$$\langle S_1 : ... : S_n, \tau_S, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau'_S, T\rangle,$$

où $(\bullet, S'_1 : .... : S'_n, T, c, \tau'_S, \psi'_{S1}, \psi'_{S2})$ est un état synchrone.

Soit l'état synchrone: $(S; P, S_1; P_1 : .... : S_n; P_n, T, c, \tau_S, \psi_{S1}, \psi_{S2})$.

Si $[\![S; P]\!]_{\mathcal{LH}}(\psi_{S1}, \psi_{S2}, c) = (\psi''_1, \psi''_2) \neq \perp$, alors il existe ψ'_1 et ψ'_2 tels que:

$$[\![S]\!]_{\mathcal{LH}}(\psi_{S1}, \psi_{S2}, c) = (\psi'_1, \psi'_2) \neq \perp, \tag{4.5}$$

et ψ''_1 et ψ''_2 tels que:

$$[\![P]\!]_{\mathcal{LH}}(\psi'_1, \psi'_2, c) = (\psi''_1, \psi''_2) \neq \perp. \tag{4.6}$$

En utilisant l'hypothèse d'induction on voit qu'il existe une transition:

$$\langle S_1; P_1 : .... : S_n; P_n, \tau_S, T\rangle \xrightarrow{c^*} \langle P_1 : ... : P_n, \tau'_S, T\rangle$$

où $(P, P_1 : .... : P_n, T, c, \tau'_S, \psi'_1, \psi'_2)$ est synchrone. De même, il existe une transition:

$$\langle P_1 : ... : P_n, \tau'_S, T\rangle \xrightarrow{c^*} \langle P'_1 : ... : P'_n, \tau''_S, T\rangle$$

où $(\bullet, P'_1 : .... : P'_n, T, c, \tau''_S, \psi''_1, \psi''_2)$ est synchrone.

■

Lemme 4.2.9 *Tout programme $\mathcal{LH}$ vérifie la propriété d'induction de $\mathcal{LH}$.*

Preuve

La preuve découle des lemmes 4.2.3, 4.2.5, 4.2.7 et 4.2.8. ■

Exécution entre groupes d'instructions

Cette section permet de passer de la preuve concernant $\mathcal{LH}$ à la preuve concernant $\mathcal{SCP}$. On prouve la propriété d'induction pour le franchissement d'un groupe entier. Cette preuve nécessite un lemme intermédiaire prouvant la cohérence de la fonction Evalop lors de l'incrément des horloges.

Lemme 4.2.10 *Soit un environnement τ cohérent vis-à-vis de T, un indice j et une horloge t_j tels que $T|_j \prec t_j$. Soit T' tel que $T'|_j = t_j$ et $\forall i \neq j,\ T'|_i = T|_i$. Alors*

$$\forall u \in Proc, \forall X \in Var \cup Ind,\ Proj_2(Evalop(\tau, X, u, T)) = Proj_2(Evalop(\tau, X, u, T'))$$

Preuve

Voir section 4.5.1 page 113. ■

Lemme 4.2.11 *Si le programme P vérifie la propriété d'induction de $\mathcal{LH}$, alors le programme* step *(P) vérifie la propriété d'induction.*

Preuve

Chaque indice du contexte peut exécuter l'instruction $S =$ step *(P) par une transition contextuelle $\xrightarrow{c^*}$:*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T\rangle$$

Considérons maintenant le résultat de la sémantique dénotationnelle. Notons ψ' l'environnement résultant de la sémantique dénotationnelle.

$$[\![\text{step } (S)]\!](\psi, c) = \psi'$$

avec $[\![S]\!]_{\mathcal{LH}}(\psi, \psi, c) = (\psi, \psi')$. Montrons que l'état résultant est synchrone. Pour le prouver, nous montrons que l'environnement résultant de la sémantique opérationnelle est cohérent et qu'il correspond aux environnements de la sémantique dénotationnelle.

- *Montrons d'abord que l'environnement τ' est cohérent. Par hypothèse d'induction, l'environnement τ est cohérent vis-à-vis de l'environnement d'horloges T. Dans la sémantique du franchissement du* step *(voir 3.1.3 page 59), seules les horloges des indices $i \in c$ sont modifiées. Par hypothèse τ' est cohérent vis-à-vis de T et pour tout indice $i \in c$, $T'|_i \succeq T|_i$ et pour tout indice $i \notin c$, $T'|_i = T|_i$. Alors τ' est cohérent vis-à-vis de T' (lemme 4.2.1).*
- *Il reste à vérifier que: $Equiv(\tau', \psi', \psi', T', True)$. Par hypothèse d'induction, l'environnement τ' est cohérent vis-à-vis de T. De plus, pour tout indice $i \in c$, on a $T'|_i \succeq T|_i$ et pour tout indice $i \notin c$, on a $T'|_i = T|_i$. Alors d'après le lemme 4.2.10, on a:*

$\forall u \in Proc, \forall X \in Var \cup Ind,\ Proj_2(Evalop(\tau', X, u, T)) = Proj_2(Evalop(\tau', X, u, T'))$

Par hypothèse, on a $Equiv(\tau', \psi, \psi', T, True)$, *donc:*

$\forall u \in c, X \in Var \cup Ind, ona : Proj_2(Evalop(\tau', u, X, T)) = Eval(\psi, \psi', u, X),$

et donc:

$\forall u \in c, X \in Var \cup Ind, ona : Proj_2(Evalop(\tau', X, u, T')) = Eval(\psi_1, \psi_2', u, X)$

■

Séquence

Le lemme suivant concerne l'instruction de séquence.

Lemme 4.2.12 *Si les programmes $\mathcal{SCP}$ P_1 et P_2 vérifient la propriété d'induction, alors $S = P_1; P_2$ vérifie la propriété d'induction.*

Preuve

Supposons que l'on ait:

$$[\![P_1; P_2]\!](\psi, c) = \psi' \neq \perp$$

Le calcul peut se décomposer en deux phases:

$$[\![P_1]\!](\psi, c) = \psi'' \text{ et } [\![P_2]\!](\psi'', c) = \psi'$$

On distingue deux transitions contextuelles issues de $\xrightarrow{c^*}$ *en les notant* $\xrightarrow{c(1)^*}$ *et* $\xrightarrow{c(2)^*}$. *L'état* $(P_1; P_2, S_1 : .... : S_n, T, c, \tau, \psi, \psi)$ *est synchrone. Par hypothèse d'induction, il existe donc une transition* $\xrightarrow{c(1)^*}$ *telle que:*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c(1)^*} \langle S_1'' : ... : S_n'', \tau'', T''\rangle$$

où $(\bullet; P_2, S_1'' : .... : S_n'', T'', c, \tau'', \psi'', \psi'')$ *est un état synchrone. La transition* $\xrightarrow{c(2)^*}$ *permet aux indices actifs d'exécuter la séquence:*

$$\langle S_1'' : ... : S_n'', \tau'', T''\rangle \xrightarrow{c(2)^*} \langle S_1''' : ... : S_n''', \tau'', T''\rangle$$

L'état résultant $(P_2, S_1''' : .... : S_n''', T'', c, \tau'', \psi'', \psi'')$ *est évidemment synchrone.*

On applique une nouvelle fois l'hypothèse d'induction pour l'exécution de P_2. *Il existe une transition* $\xrightarrow{c(3)^*}$ *telle que:*

$$\langle S_1''' : ... : S_n''', \tau'', T''\rangle \xrightarrow{c(3)^*} \langle S_1' : ... : S_n', \tau', T'\rangle$$

où $(\bullet, S_1' : .... : S_n', T', c, \tau', \psi', \psi')$ *est un état synchrone.* ■

Instruction where/elsewhere

On se préoccupe maintenant de l'étape d'induction traitant de la conditionnelle. Cette étape nécessite un lemme intermédiaire prouvant qu'un triplet reste non bloquant (voir définition 3.2.3 page 63) si l'on considère un contexte réduit où les indices exclus du contexte initial sont dans une branche concurrente d'un where/elsewhere.

Lemme 4.2.13 *Soit une triplet* $(S_1 : ... : S_n, T, c)$ *non bloquant et un contexte réduit* $c' \subseteq c$. *Soit un programme* $\mathcal{SCP}$ S *tel que pour tout* $i \in c'$, $S_i = S$. *Si pour tout* $i \in c \backslash c'$ *et* $j \in c'$ *tels que* $T|_i = w(a,b)$ *et* $T|_j = w(e,f)q$ *(où* q *peut être vide) on a* $a \neq e$. *Alors le triplet* $(S_1 : ... : S_n, T, c')$ *est non bloquant.*

Preuve

Voir section 4.5.2 page 114. ■

Lemme 4.2.14 *Soit un environnement* τ *cohérent vis-à-vis d'un environnement d'horloges* T. *Si on a* $\forall\, i \in c, T'|_i \succeq T|_i$ *et* $Equiv(\tau, \psi, \psi, T, c)$ *alors on a* $Equiv(\tau, \psi, \psi, T', c)$.

Preuve

Voir section 4.5.2 page 114. ■

Le lemme suivant démontre la propriété d'induction pour l'instruction where/elsewhere.

Lemme 4.2.15 *Si les programmes* P_1 *et* P_2 *vérifient la propriété d'induction alors* $S =$where B do P_1 elsewhere P_2 end *vérifie la propriété d'induction.*

Preuve

Supposons que l'on a :

$$[\![\text{where } B \text{ do } P_1 \text{ elsewhere } P_2 \text{ end}]\!](\psi, c) = \psi'$$

On note $b_1 = c \wedge Eval(\psi, \psi, B)$ *(resp.* $b_2 = c \wedge \neg Eval(\psi, \psi, B)$*) représentant le contexte associé à la première branche (resp. la seconde branche) de la conditionnelle.*

On applique $[\![]\!]$ *à la première branche avec le contexte* b_1 *:*

$$[\![P_1]\!](\psi, b_1) = \psi'_1$$

Pour la seconde branche on reprend l'environnement initial ψ *pour masquer aux indices de* b_2 *les copies qui ont pu avoir lieu pendant le calcul de* $[\![P_1]\!]$.

$$[\![P_2]\!](\psi, b_2) = \psi'_2$$

L'environnement final ψ' est obtenu à partir des environnements ψ'_1 et ψ'_2 :

$$\textit{pour tout } X,\ \psi'(X) = \begin{cases} \psi'_1(X) & \textit{si } \begin{cases} (X \in Var \textit{ et } Owner(X) \in b_1) \textit{ ou} \\ (X = I|_u \textit{ (tel que } I \in Ind) \textit{ et } u \in b1) \end{cases} \\ \psi'_2(X) & \textit{sinon} \end{cases},$$

On construit $\xrightarrow{c^*}$ à partir de quatre transitions $\xrightarrow{c(0)^*}$, $\xrightarrow{b_1{}^*}$, $\xrightarrow{b_2{}^*}$ et $\xrightarrow{c(3)^*}$.

Transition $\xrightarrow{c(0)^*}$. *A partir de l'état initial, chaque indice actif peut faire une première transition pour l'exécution du* where/elsewhere *:*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c(0)^*} \langle S_1^0 : ... : S_n^0, \tau, T^0\rangle$$

où

- $S_i^0 = P_1$; end *et* $T^0|_i = t(1,0)$ *si* $i \in b_1$ *(t est l'horloge initiale des indices du contexte c),*
- $S_i^0 = P_2$; end *et* $T^0|_i = t(2,0)$ *si* $i \in b_2$.

Montrons que l'état $(P_1, Q_1^0 : ... : Q_n^0, T^0, b_1, \tau, \psi, \psi)$ où $Q_i^0 = P_1$ si $i \in b_1$ est synchrone.

- *Le triplet $(S_1 : ... : S_n, T, c)$ est non bloquant, donc $(S_1^0 : ... : S_n^0, T^0, c)$ aussi. On en déduit que $(Q_1^0 : ... : Q_n^0, T^0, c)$ est non bloquant. Pour les indices $i \in b_1$ et $j \in b_2$, $T^0|_i$ est incomparable à $T^0|_j$ et pour $j \notin c$ on a $T^0|_j = T|_j$, donc d'après le lemme 4.2.13, $(Q_1^0 : ... : Q_n^0, T^0, b_1)$ est non bloquant.*
- *D'après le lemme 4.2.1, τ est cohérent vis-à-vis de T^0.*
- *De $Equiv(\tau, \psi, \psi, T^0, True)$ et du lemme 4.2.14, on déduit: $Equiv(\tau, \psi, \psi, T^0_{b_1}, True)$.*

Transition $\xrightarrow{b_1{}^*}$. *On applique l'hypothèse d'induction à l'état :*

$$(P_1, Q_1^0 : ... : Q_n^0, T^0, b_1, \tau, \psi, \psi).$$

Il existe $\xrightarrow{b_1{}^*}$ tel que :

$$\langle Q_1^0 : ... : Q_n^0, \tau, T^0\rangle \xrightarrow{b_1{}^*} \langle Q_1^1 : ... : Q_n^1, \tau'_1, T^1\rangle$$

où $(\bullet, Q_1^1 : ... : Q_n^1, T^1, b_1, \tau'_1, \psi'_1, \psi'_1)$ est un état synchrone et où tous les indices du contexte b_1 ont terminé leur programme P_1 avec la même horloge $t(1, y_1)$. On dénote par $T^1_{b_1}$ (resp. $T^1_{b_2}$) l'horloge T^1_i telle que $i \in b_1$ (resp. l'horloge T^1_j telle que $j \in b_2$).

On montre maintenant que $(P_2, Q_1^1 : ... : Q_n^1, T^1, b_2, \tau'_1, \psi'_1, \psi'_1)$ ou $Q_i^1 = P_2$ si $i \in b_2$ est synchrone. Trois points essentiels sont à vérifier, la preuve du non blocage, la cohérence des environnements et la propriété $Equiv(\tau'_1, \psi'_1, \psi'_1, T^1_{b_1}, c)$.

- *Le triplet $(S_1 : ... : S_n, T, c)$ est non bloquant, donc $(S_1^0 : ... : S_n^0, T^0, c)$ aussi. On en déduit que $(Q_1^0 : ... : Q_n^0, T^0, c)$ est non bloquant. Pour les indices*

$i \in b_1$ et $j \in b_2$, $T^0|_i$ est incomparable à $T^0|_j$ et pour $j \notin c$ on a $T^0|_j = T|_j$, donc d'après le lemme 4.2.13, $(Q_1^0 : ... : Q_n^0, T^0, b_1)$ est non bloquant.

- *Le triplet $(S_1^0 : ... : S_n^0, T^0, c)$ est non bloquant, donc $(Q_1^1 : ... : Q_n^1, T^0, b_2)$ aussi (lemme 4.2.13). Pour les indices $i \in b_1$ et $j \in b_2$, $T^1|_i$ est incomparable à $T^1|_j$ et pour $j \notin c$ on a $T^1|_j = T^0|_j$, donc $(Q_1^1 : ... : Q_n^1, T^1, b_2)$ est non bloquant.*
- *L'environnement τ_1' est cohérent vis-à-vis de T^1 car $(\bullet; \mathsf{end}, S_1^1 : ... : S_n^1, T^1, b_1, \tau_1', \psi_1', \psi_1')$ est synchrone.*
- *De $Equiv(\tau, \psi, \psi, T^0, True)$ et du lemme 4.2.14, on déduit: $Equiv(\tau, \psi, \psi, T^0_{b_1}, True)$.*
  Par hypothèse on a: $Equiv(\tau_1', \psi_1', \psi_1', T^1_{b_1}, True)$. Lors de l'exécution de P_1, $i \in b_1$ ajoute seulement des copies estampillées par des horloges du type $t(1, x)q$ dans sa mémoire publique (lemme 3.2.2). Donc, ces horloges étant incomparables avec $T^1_{b_2} = T^0_{b_2}$, on a $Equiv(\tau_1', \psi_1', \psi_1', T^1_{b_2}, True)$.

Transition $\xrightarrow{b_2^*}$. *On applique l'hypothèse d'induction à l'état*

$$(P_2, Q_1^1 : ... : Q_n^1, T^1, b_2, \tau_1', \psi_1', \psi_1').$$

Il existe $\xrightarrow{b_2^*}$ tel que:

$$\langle Q_1^1 : ... : Q_n^1, \tau_1', T^1 \rangle \xrightarrow{b_2^*} \langle Q_1' : ... : Q_n', \tau', T^2 \rangle$$

où $(\bullet, Q_1' : ... : Q_n', T^2, b_2, \tau', \psi', \psi')$ est un état synchrone et où tous les indices du contexte b_2 ont terminé leur programme P_2 avec la même horloge $t(2, y_2)$. En appliquant les transitions $\xrightarrow{b_1^*}$ et $\xrightarrow{b_2^*}$ à l'état $\langle S_1^0 : ... : S_n^0, \tau, T^0 \rangle$ on déduit:

$$\langle S_1^0 : ... : S_n^0, \tau, T^0 \rangle \xrightarrow{b_1^*} \xrightarrow{b_2^*} \langle S_1^2 : ... : S_n^2, \tau', T^2 \rangle$$

Transition $\xrightarrow{c(3)^*}$. *On associe une dernière transition pour l'exécution du $\bullet; \mathsf{end}$ par les indices du contexte c:*

$$\langle S_1^2 : ... : S_n^2, \tau', T^2 \rangle \xrightarrow{c(3)^*} \langle S_1' : ... : S_n', \tau', T' \rangle$$

avec pour les indices $i \in c$, $S_i' = \bullet$ et $T'|_i = k(a, b+1)$ si $T = k(a, b)$.

Montrons que l'état résultant est synchrone.

- *Le triplet $(S_1' : ... : S_n', T', c)$ est non bloquant car l'état $\langle S_1' : ... : S_n', \tau', T' \rangle$ est issu de l'état $\langle S_1 : ... : S_n, \tau, T \rangle$ où le triplet $(S_1 : ... : S_n, T, c)$ est non bloquant (lemme 3.2.1).*
- *τ' est cohérent vis-à-vis de T^2. Tout indice i vérifie $T'|_i \succeq T^2|_i$, donc d'après le lemme 4.2.1, τ' est cohérent vis-à-vis de T'.*

- *Il nous reste à montrer que* $Equiv(\tau', \psi', \psi', T'_c, True)$. *Détaillons les différents cas possibles en fonction des programmes exécutés par l'indice* i*:*
  - ▷ *Si* $i \in b_1$, *on a:* $Equiv(\tau'_1, \psi'_1, \psi'_1, T^1_{b_1}, True)$ *et* $T^2|_i = T^1|_i$.
    Donc: $Equiv(\tau'_1, \psi'_1, \psi'_1, T^2_{b_1}, b_1)$. *L'environnement* τ'_1 *est cohérent vis-à-vis de* T^1, *de plus on a* $T^2|_i \prec T'|_i$ *donc par le lemme 4.2.14 on a* $Equiv(\tau'_1, \psi'_1, \psi'_1, T', b_1)$.
    Comme $\tau'|i = \tau'_1|i$, *on a* $Equiv(\tau', \psi'_1, \psi'_1, T', b_1)$.
  - ▷ *Si* $i \in b_2$, *on a* $Equiv(\tau', \psi'_2, \psi'_2, T^2_{b_2}, b_2)$. *L'environnement* τ' *est cohérent vis-à-vis de* T^2, *de plus on a* $T^2|_i \prec T'|_i$ *donc par le lemme 4.2.14 on a* $Equiv(\tau', \psi'_2, \psi'_2, T', b_2)$.
  - ▷ *Si* $i \in \bar{c}$, *on a* $\tau'|i = \tau|i$, *donc* $Equiv(\tau', \psi, \psi, T, \bar{c})$. *L'environnement* τ *est cohérent vis-à-vis de* T, *de plus on a* $T|_i \prec T'|i = k(a, b+1)$ *donc par le lemme 4.2.14 on a* $Equiv(\tau', \psi, \psi, T', \bar{c})$

  Les environnements τ'_1 *et* τ' *sont cohérents vis-à-vis de* T'. *De plus on a:*

$$Equiv(\tau'_1, \psi'_1, \psi'_1, T', b_1)$$

$$Equiv(\tau', \psi'_2, \psi'_2, T', b_2)$$

$$Equiv(\tau', \psi, \psi, T', \bar{c})$$

  Nous en déduisons $Equiv(\tau', \psi', \psi', T', True)$.

■

Instruction loopwhere

Les lemmes qui suivent se réfèrent à l'étape d'induction du loopwhere. On traite cette étape en deux lemmes pour tenir compte de la particularité du loopwhere qui est réécrit par la sémantique opérationnelle en next_iteration après la première itération. L'instruction next_iteration évite l'ajout d'une paire aux horloges au début de chaque nouvelle itération de boucle. Cette instruction étant spécifique à la sémantique opérationnelle, on est amené à prouver une variante de la propriété d'induction (définition 4.2.4 page 80). Les horloges des indices du contexte ont au moins deux paires et un label terminal 1. Cette paire terminale est celle ajoutée en début de loopwhere, avant d'exécuter next_iteration. Les deux paires sont nécessaires pour ne pas aboutir à une horloge sans paire en sortie de boucle. Nous aboutissons à la définition *d'état synchrone étendu :*

Définition 4.2.6 (Etat synchrone étendu) *La liste* $(S, S_1 : .... : S_n, T, c, \tau, \psi)$ *est un état synchrone étendue si elle vérifie les conditions suivantes :*

- $(S, S_1 : .... : S_n, T, c, \tau, \psi, \psi)$ *est un état synchrone.*
- *Les horloges des indices du contexte* c *sont du type* $t(a, b)(1, d)$.

Avant de présenter le lemme correspondant à l'étape d'induction de next_iteration, on introduit quatre lemmes intermédiaires. Les trois premiers se rapportent à la stabilité du non blocage sur un contexte réduit, le dernier concerne la fonction *Equiv* après évaluation de la condition d'une instruction next_iteration. Par souci de clarté, les preuves de ces lemmes sont reportées section 4.5.3 page 115.

Comme pour la conditionnelle, on a besoin d'un lemme sur la stabilité du non blocage sur des contextes réduits. On se place dans le cas où les indices exclus du contexte initial ont une horloge plus grande que celle des indices du nouveau contexte.

Lemme 4.2.16 *Soit un triplet $(S_1 : ... : S_n, T, c)$ non bloquant et un contexte réduit $c' \subseteq c$. Soit un programme $\mathcal{SCP}$ S tel que pour tout $i \in c'$, $S_i = S$. Si pour tout $j \in c \backslash c'$ et $i \in c'$ on a $T|_j = w(a,b)$ et $T|_i = w(a,f)q$ où q est non vide et $b > f$, alors le triplet $(S_1 : ... : S_n, T, c')$ est non bloquant.*

Preuve

Voir section 4.5.3 page 115. ■

Le lemme précédent n'est cependant pas suffisant. Il ne peut pas s'appliquer à un programme de la forme next_iteration B do P end qui n'est pas un programme $\mathcal{SCP}$. On présente donc un second lemme de stabilité du non blocage. Pour le démontrer, il est nécessaire d'introduire un lemme sur la stabilité de la forme des horloges.

Lemme 4.2.17 *Soit un programme $\mathcal{SCP}$ P, un état $\langle S_1 : ... : S_n, \tau, T \rangle$ et un indice i tel que $S_i =$ next_iteration B do P end et $T|_i = t(a,b)(e,f)$ (t peut être vide). S'il existe une transition $\stackrel{*}{\longrightarrow}$ telle que :*

$$\langle S_1 : ... : S_n, \tau, T \rangle \stackrel{*}{\longrightarrow} \langle S'_1 : ... : S'_n, \tau', T' \rangle,$$

alors on a :

- $T'|_i = t(a,b)(e,f')q'$ *où* $f' \geq f$ *si* $S'_i \neq \bullet$,
- $T'|_i = t(a,b+1)$ *sinon.*

Preuve

Voir section 4.5.3 page 115. ■

Lemme 4.2.18 *Soit une triplet $(S_1 : ... : S_n, T, c)$ non bloquant et un contexte réduit $c' \subseteq c$. Soit un programme $S =$ next_iteration B do P end où P est un programme $\mathcal{SCP}$ et où pour tout $i \in c'$, $S_i = S$. Si pour tout $j \in c \backslash c'$ et $i \in c'$ on a $T|_j = w(a,b)$ et $T|_i = w(a,f)q$ où q est non vide et $b > f$, alors le triplet $(S_1 : ... : S_n, T, c')$ est non bloquant.*

Preuve

Voir section 4.5.3 page 116. ■

Le lemme suivant correspond à l'étape d'induction pour l'instruction next_iteration. L'instruction next_iteration n'existant pas pour la sémantique dénotationnelle, cette dernière utilise un loopwhere.

Lemme 4.2.19 *Soit* $S =$ next_iteration B do P end *où le programme* $\mathcal{SCP}$ P *vérifie la propriété d'induction. Pour tout état synchrone étendu* $(S, S_1 : .... : S_n, T, c, \tau, \psi)$, *si*

$$[\![\text{loopwhere } B \text{ do } P \text{ end}]\!](\psi, c) = \psi' \neq \bot$$

alors il existe $\xrightarrow{c^*}$ *tel que*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle$$

où $(\bullet, S'_1 : .... : S'_n, T', c, \tau', \psi', \psi')$ *est un état synchrone.*

Preuve

De façon classique [57], on définit une suite de fonctions f_k *de* $MemEnv \times Ctx$ *vers* $MemEnv$ *convergeant vers le plus petit point fixe* $fix(F)$ *de la fonction* F *lorsque la boucle termine (voir section 4.1) :*

$$\begin{array}{rrcl} & f_{-1}(\psi, c) & = & \bot \\ et & f_k(\psi, c) & = & F(f_{k-1})(\psi, c) \\ & & = & if\ (\exists i \in c \wedge Eval(\psi, \psi, i, B)) \\ & & & then\ f_{k-1}([\![P]\!](\psi, c \wedge Eval(\psi, \psi, B)), c \wedge Eval(\psi, \psi, B)) \\ & & & else\ \psi \end{array} \quad (4.7)$$

On suppose que $[\![\text{loopwhere } B \text{ do } P \text{ end}]\!](\psi, c) \neq \bot$. *Il existe donc un entier* k *tel que :*

$$[\![\text{loopwhere } B \text{ do } P \text{ end}]\!](\psi, c) = f_k(\psi, c) = \psi'$$

On fait alors une preuve par récurrence. On montre que pour tout entier $k \geq 0$ *si* $f_k(\psi, c) = \psi'$ *alors il existe* $\xrightarrow{c^*}$ *tel que*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle$$

où $(\bullet, S'_1 : .... : S'_n, T', c, \tau', \psi', \psi')$ *est un état synchrone.*

- $k = 0$. *Il n'existe pas d'indice* i *tel que* $i \in c \wedge Eval(\psi, \psi, B)$. *Le calcul de la sémantique opérationnelle ne modifie pas les environnements :*

$$\begin{array}{rcl} [\![\text{loopwhere } B \text{ do } P \text{ end}]\!](\psi, c) & = & f_0(\psi, c) \\ & = & \psi \end{array}$$

On construit simplement $\xrightarrow{c^*}$ *en exécutant la boucle pour chaque indice du contexte* c. *Puisque aucun d'entre eux ne vérifie la condition* B, *une seule transition par indice est nécessaire pour terminer la boucle et aucun environnement n'est modifié :*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau, T'\rangle$$

$$\textit{pour tout } i,\ T'|_i = \begin{cases} t(x,y+1) \textit{ si } T|_i = t(x,y)(a,b) \textit{ et } i \in c, \\ T|_i \textit{ sinon} \end{cases}$$

- ▷ *D'après le lemme 4.2.1 page 79, τ est cohérent vis-à-vis de T'.*
- ▷ *D'après le lemme 4.2.14, on a $Equiv(\tau, \psi, \psi, T', True)$.*

- **$k > 0$.** *On suppose la propriété vraie pour $k-1 \geq 0$. Montrons qu'elle est vérifiée pour k. On note $b_1 = c \wedge Eval(\psi, \psi, B)$ et $b_2 = c \wedge \neg Eval(\psi, \psi, B)$. La sémantique dénotationnelle nous donne :*

$$\begin{array}{rcl} [\![\textsf{loopwhere } B \textsf{ do } P \textsf{ end}]\!](\psi, c) & = & f_k(\psi, c) \\ & = & f_{k-1}([\![P]\!](\psi, b_1), c \wedge Eval(\psi, \psi, B)) \\ & = & f_{k-1}(\psi'', b_1) \\ & = & (\psi') \end{array}$$

Remarquons que les résultats de $[\![P]\!](\psi, b_1)$ et de $f_{k-1}(\psi'', b_1) = [\![\textsf{loopwhere } B \textsf{ do } P \textsf{ end}]\!](\psi'', b_1)$ sont évidemment différents de $\perp$ sinon l'hypothèse de terminaison sur S en k itérations n'est plus vérifiée.

On construit $\xrightarrow{c^*}$ à partir de trois transitions $\xrightarrow{c(1)^*}$, $\xrightarrow{c(2)^*}$ et $\xrightarrow{c(3)^*}$.

Transition $\xrightarrow{c(1)^*}$. *La première transition $\xrightarrow{c(1)^*}$ associe une transition pour chaque indice actif à partir de l'instruction* next_iteration *:*

$$\langle S_1 : ... : S_n, \tau, T \rangle \xrightarrow{c(1)^*} \langle S_1^1 : ... : S_n^1, \tau, T^1 \rangle$$

$$\textit{avec pour tout } i,\ T^1|_i = \begin{cases} t(x,y+1) \textit{ si } T|_i = t(x,y)(a,b) \textit{ et } i \in b_2, \\ T|_i \textit{ sinon} \end{cases}$$

$$\textit{et pour tout } i,\ S_i^1 = \begin{cases} P; \textsf{next\_iteration } B \textsf{ do } P \textsf{ end} & \textit{si } i \in b_1, \\ \bullet & \textit{si } i \in b_2, \\ S_i & \textit{sinon} \end{cases}$$

Montrons que l'état $(P, Q_1^1 : ... : Q_n^1, T^1, b_1, \tau, \psi, \psi)$ où $Q_i = P$ si $i \in b_1$, est synchrone.

- ▷ *Le triplet $(S_1^1 : ... : S_n^1, T^1, c)$ est non bloquant, donc $(Q_1^1 : ... : Q_n^1, T^1, c)$ aussi. Les indices de b_2 ont une horloge $t(x, y+1)$. Les indices de b_1 ont une horloge $t(x,y)(a,b)$ plus petite. Donc, d'après le lemme 4.2.16, le triplet $(Q_1^1 : ... : Q_n^1, T^1, b_1)$ est non bloquant.*
- ▷ *D'après le lemme 4.2.1 page 79, τ est cohérent vis-à-vis de T^1.*
- ▷ *Le lemme 4.2.14 assure que $Equiv(\tau, \psi, \psi, T', True)$.*

Transition $\xrightarrow{c(2)^*}$. *On applique la propriété d'induction à partir de l'état $(P, Q_1^1 : ... : Q_n^1, T^1, b_1, \tau, \psi, \psi)$. Il existe donc une transition $\xrightarrow{{b_1}^*}$ telle que :*

$$\langle Q_1^1 : ... : Q_n^1, \tau, T^1 \rangle \xrightarrow{{b_1}^*} \langle Q_1^2 : ... : Q_n^2, \tau'', T^2 \rangle$$

où $(\bullet, Q_1^2 : ... : Q_n^2, T^2, b_1, \tau'', \psi'', \psi'')$ *est synchrone.*

On construit pour l'état $\langle S_1^1 : ... : S_n^1, \tau, T^1\rangle$ *la transition* $\xrightarrow{c(2)^*}$ *à partir de la transition* $\xrightarrow{b_1{}^*}$ *suivie d'une transition permettant à chaque indice de* b_1 *d'exécuter la séquence suivant le programme* P *:*

$$\langle S_1^1 : ... : S_n^1, \tau, T^1\rangle \xrightarrow{c(2)^*} \langle S_1^2 : ... : S_n^2, \tau'', T^2\rangle$$

$$\text{où pour tout } i,\ S_i^2 \begin{cases} \mathsf{next\_iteration}\ B\ \mathsf{do}\ P\ \mathsf{end} & \text{si } i \in b_1, \\ \bullet & \text{si } i \in b_2, \\ S_i & \text{sinon} \end{cases}$$

Les indices de b_1 *ont maintenant à exécuter l'instruction* next_iteration B do P end*. Pour pouvoir appliquer l'hypothèse de récurrence il faut vérifier que*

$$(\mathsf{next\_iteration}\ \ B\ \ \mathsf{do}\ \ P\ \ \mathsf{end}, S_1^2 : ... : S_n^2, T^2, b_1, \tau'', \psi'')$$

est un état synchrone étendu.

- ▷ *Le triplet* $(S_1^2 : ... : S_n^2, T^2, c)$ *est non bloquant, donc* $(S_1^2 : ... : S_n^2, T^2, b_1)$ *aussi d'après le lemme 4.2.18.*
- ▷ *D'après le lemme 3.2.3 page 65, si les indices* $i \in b_1$ *ont une horloge initiale* $T|_i = t(x,y)(1,b)$*, alors* $T^2|_i = t(x,y)(1,h)$*. Donc les horloges des indices du contexte ont au moins deux paires.*

Transition $\xrightarrow{c(3)^*}$**.** *On est donc en mesure d'appliquer la propriété d'induction de* next_iteration*. Il existe une transition* $\xrightarrow{c(3)^*}$ *telle que :*

$$\langle S_1^2 : ... : S_n^2, \tau'', T^2\rangle \xrightarrow{c(3)^*} \langle S_1' : ... : S_n', \tau', T'\rangle$$

$$\text{où pour tout } i,\ T'|_i = \begin{cases} t(x,y+1) \text{ si } T|_i = t(x,y)(a,b) \text{ et } i \in b_1, \\ T|_i \text{ sinon} \end{cases}$$

$$\text{et pour tout } i,\ S_i' = \begin{cases} \bullet & \text{si } i \in c, \\ S_i & \text{sinon} \end{cases}$$

L'état $(\bullet, S_1' : ... : S_n', T', b_1, \tau', \psi', \psi')$ *est synchrone. La seule difficulté pour montrer que l'état* $(\bullet, S_1' : ... : S_n', T', c, \tau', \psi', \psi')$ *est synchrone réside dans la vérification du non blocage. Il n'existe pas d'indice tel que* $i \in c$ *et* $S_i' \neq \bullet$ *donc d'après la proposition 3.2.1, le triplet* $(S_1' : ... : S_n', T', c)$ *est non bloquant.*

■

Le lemme suivant achève la démonstration de la proposition 4.2.1 page 80. Il démontre la propriété d'induction pour l'instruction loopwhere.

Lemme 4.2.20 *Si le programme $\mathcal{SCP}$ P vérifie la propriété d'induction, alors* $S =$loopwhere B do P end *vérifie la propriété d'induction.*

Preuve

La preuve est similaire à celle du lemme précédent. On se permettra donc d'être très concis dans l'argumentation. Le calcul de la sémantique dénotationnelle est identique. Au niveau opérationnel, au lieu de débuter avec une instruction next_iteration *on travaille avec l'instruction* loopwhere. *Il n'est pas nécessaire de faire une récurrence sur le nombre d'itérations exécutées, on étudie seulement deux cas :*

1. *Si $k = 0$ alors la boucle termine directement en donnant un état synchrone.*
2. *Si $k > 0$ on déroule la première itération de la boucle qui se réécrit avec la structure* next_iteration *:*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c(1)^*} \langle S_1^1 : ... : S_n^1, \tau, T^1\rangle$$

$$\text{avec pour tout } i,\ T^1|_i = \begin{cases} t(a,b+1) \text{ si } T|_i = t(a,b) \text{ et } i \in b_2, \\ t(a,b)(1,0) \text{ si } T|_i = t(a,b) \text{ et } i \in b_1, \\ T|_i \text{ sinon} \end{cases}$$

$$\text{et pour tout } i,\ S_i^1 = \begin{cases} P; \text{next\_iteration } B \text{ do } P \text{ end} & \text{si } i \in b_1, \\ \bullet & \text{si } i \in b_2, \\ S_i & \text{sinon} \end{cases}$$

Montrons que l'état $(P, Q_1^1 : ... : Q_n^1, T^1, b_1, \tau, \psi, \psi)$ où $Q_i^0 = P$ si $i \in b_1$ est synchrone :

- *Le lemme 4.2.14 assure que $Equiv(\tau, \psi, \psi, T^1, True)$.*
- *Le triplet $(S_1 : ... : S_n, T, c)$ est non bloquant, donc $(S_1^1 : ... : S_n^1, T^1, c)$ aussi. On déduit que $(Q_1^1 : ... : Q_n^1, T^1, c)$ est non bloquant. En appliquant le lemme 4.2.16, on montre que $(Q_1^1 : ... : Q_n^1, T^1, b_1)$ est non bloquant.*

Par hypothèse il existe une transition qui, suite à l'exécution de P par les indices de b_1, donne un état synchrone $(\bullet, Q_1^2 : ... : Q_n^2, T^2, b_1, \tau'', \psi'', \psi'')$.

On applique à l'état $\langle S_1^1 : ... : S_n^1, \tau, T^1\rangle$ cette transition et une transition pour l'exécution de la séquence suivant P. Montrons que l'état résultant

$$(\text{next\_iteration } B \text{ do } P \text{ end}, S_1^2 : ... : S_n^2, T^2, b_1, \tau'', \psi'')$$

est un état synchrone étendu :

- *Le triplet $(S_1^2 : ... : S_n^2, T^2, c)$ est non bloquant, donc le lemme 4.2.18 assure que le triplet $(S_1^2 : ... : S_n^2, T^2, b_1)$ est non bloquant.*

- *Avant l'exécution du programme P, les horloges des indices de b_1 étaient au moins de taille deux. Donc, par stabilité de la forme initiale des horloges 3.2.3 page 65, ce point est toujours valide avant l'exécution de* next_iteration.

On peut donc appliquer le lemme 4.2.19 qui nous conduit au résultat attendu.

■

4.2.6 Théorème d'équivalence : condition suffisante

La preuve par induction de la condition suffisante du théorème 4.2.1 est pour une large part similaire à la preuve de la condition nécessaire. Pour chaque étape de l'induction, on suppose que l'on a une transition $\xrightarrow{c^*}$ terminant le programme à partir d'un état synchrone. Il faut alors montrer que la sémantique dénotationnelle permet de produire un nouvel état synchrone. Lorsqu'on aborde les structures de contrôles, il est nécessaire de montrer qu'il existe des transitions intermédiaires permettant d'appliquer les hypothèses d'induction. On a recours à la proposition 3.2.1 page 66 d'absence de blocage et à la proposition 3.3.1 page 70 de déterminisme pour prouver que ces transitions existent. La dernière proposition est aussi pour montrer que la suite des transitions intermédiaires conduisent au même état que la transition $\xrightarrow{c^*}$.

Par souci de clarté, on introduit une *seconde propriété d'induction*. A partir d'un état synchrone, la seconde propriété d'induction associe au calcul de la sémantique opérationnelle, un résultat dénotationnel produisant un nouvel état synchrone.

Définition 4.2.7 (Seconde propriété d'induction) *Un programme $\mathcal{SCP}$ S vérifie la seconde propriété d'induction s'il vérifie la propriété suivante :*
pour tout état synchrone $(S, S_1 : .... : S_n, T, c, \tau, \psi, \psi)$, s'il existe $\xrightarrow{c^*}$ tel que

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle$$

alors

$$[\![S]\!](\psi, \psi, c) = (\psi', \psi') \neq \bot,$$

où $(\bullet, S'_1 : .... : S'_n, T', c, \tau', \psi', \psi')$ est un état synchrone

Proposition 4.2.2 (Condition suffisante) *Tout programme $\mathcal{SCP}$ S vérifie la seconde propriété d'induction.*

Preuve

La preuve se fait par induction sur la structure du programme. Les étapes de l'induction sont obtenues à partir des lemmes qui suivent. ■

On définit une propriété d'induction propre aux exécutions dans le langage hôte.

Définition 4.2.8 (Seconde propriété d'induction de $\mathcal{LH}$) *Un programme $\mathcal{LH}$ S vérifie la propriété d'induction de $\mathcal{LH}$ s'il vérifie la propriété suivante :*
pour tout état synchrone $(S, S_1 : .... : S_n, T, c, \tau, \psi_1, \psi_2)$, s'il existe $\xrightarrow{c^*}$ tel que

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle$$

alors

$$[\![S]\!]_{\mathcal{LH}}(\psi_1, \psi_2, c) = (\psi'_1, \psi'_2)$$

où $(\bullet, S'_1 : .... : S'_n, T', c, \tau', \psi'_1, \psi'_2)$ est un état synchrone.

Les cinq premiers lemmes traitent des programmes $\mathcal{LH}$.

Lemme 4.2.21 *le programme $X[E_1] = E_2$ vérifie la seconde propriété d'induction de $\mathcal{LH}$.*

Preuve
La preuve est identique à celle donnée pour le lemme 4.2.3 page 82. ■

Lemme 4.2.22 *le programme $X := E$ vérifie la seconde propriété d'induction de $\mathcal{LH}$.*

Preuve
La preuve est identique à celle donnée pour le lemme 4.2.4 page 84. ■

Lemme 4.2.23 *le programme $I := E$ vérifie la seconde propriété d'induction de $\mathcal{LH}$.*

Preuve
La preuve est identique à celle donnée pour le lemme 4.2.6 page 85. ■

Lemme 4.2.24 *La composition de deux programmes $\mathcal{LH}$ qui vérifient la seconde propriété d'induction de $\mathcal{LH}$, vérifient la seconde propriété d'induction de $\mathcal{LH}$.*

Preuve
La preuve est identique à celle donnée pour le lemme 4.2.8 page 85. ■

Lemme 4.2.25 *Tout programme $\mathcal{LH}$ P vérifie la seconde propriété d'inductionde $\mathcal{LH}$.*

Preuve
Un programme $\mathcal{LH}$ est composé d'affectations en séquence. Ce lemme est déduit des lemmes 4.2.21, 4.2.22, 4.2.23 et 4.2.24. ■

Exécution entre groupes d'instructions

Cette section permet de passer de la preuve concernant $\mathcal{LH}$ à la preuve concernant $\mathcal{SCP}$. On prouve la seconde propriété d'induction pour le franchissement d'un groupe entier.

Lemme 4.2.26 *Si le programme $\mathcal{LH}$ P vérifie la seconde propriété d'induction de $\mathcal{LH}$, alors le programme* step (P) *vérifie la seconde propriété d'induction.*

Preuve

La preuve est identique à celle donnée pour le lemme 4.2.11 page 87. ■

Le lemme suivant concerne l'instruction de séquence.

Lemme 4.2.27 *Si les programmes $\mathcal{SCP}$ P_1 et P_2 vérifient la seconde propriété d'induction alors $S = P_1; P_2$ vérifie la seconde propriété d'induction .*

Preuve

Supposons qu'il existe $\xrightarrow{c^*}$ tel que

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle$$

On montre que ce calcul peut se décomposer en deux étapes. Soit $\xrightarrow{c(1)^*}$ une transition telle que

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c(1)^*} \langle S''_1 : ... : S''_n, \tau'', T''\rangle$$

où $S''_i = \bullet; P_2$ si $i \in c$.

La transition $\xrightarrow{c(1)^*}$ existe car l'exécution du programme P_1 par les indices du contexte termine :

- *Les indices du contexte c ne peuvent être bloqués (proposition 3.2.1 page 66).*
- *Si l'exécution de P_1 par les indices du contexte c conduit à des calculs infinis, alors tous les calculs à partir de $\langle S_1 : ... : S_n, \tau, T\rangle$ sont infinis (proposition 3.3.1 page 70), ce qui contredit les hypothèses.*

On applique la seconde propriété d'induction sur le programme P_1. On obtient un état synchrone $(\bullet; P_2, S''_1 : .... : S''_n, T'', c, \tau'', \psi'', \psi'')$ et

$$[\![P_1]\!](\psi, c) = \psi''$$

La seconde transition $\xrightarrow{c(2)^*}$ termine l'exécution de S pour les indices du contexte :

$$\langle S''_1 : ... : S''_n, \tau'', T''\rangle \xrightarrow{c(2)^*} \langle S'_1 : ... : S'_n, \tau''', T'''\rangle$$

où $S'_i = \bullet$ *si* $i \in c$. *On justifie l'existence de* $\xrightarrow{c(2)^*}$ *en utilisant le même type d'arguments que pour* $\xrightarrow{c(1)^*}$. *On applique la seconde propriété d'induction sur* P_2. *On obtient un état synchrone* $(\bullet, S'_1 : .... : S'_n, T''', c, \tau''', \psi''', \psi''')$ *et*

$$[\![P_2]\!](\psi''', c) = \psi'''$$

La proposition 3.3.1 page 70 apporte le dernier argument pour conclure. Il assure l'unicité de l'état terminal pour les transitions $\xrightarrow{c^*}$ *et* $\xrightarrow{c(1)^*}\xrightarrow{c(2)^*}$: $\psi''' = \psi'$, $\tau''' = \tau'$ *et* $T''' = T'$.

■

On se préoccupe maintenant de la conditionnelle.

Lemme 4.2.28 *Si les programmes SCP* P_1 *et* P_2 *vérifient la seconde propriété d'induction, alors* $S =$where B do P_1 elsewhere P_2 end *vérifie la seconde propriété d'induction.*

Preuve

On ne détaille pas cette démonstration. Elle est très similaire à celle du lemme 4.2.15 page 89. On suppose que l'on a une transition $\xrightarrow{c^*}$ *conduisant à un état terminal. On montre que l'on peut décomposer cette transition en quatre transitions* $\xrightarrow{c(0)^*}$, $\xrightarrow{b_1{}^*}$, $\xrightarrow{b_2{}^*}$ *et* $\xrightarrow{c(3)^*}$. *On justifie l'existence de ces transitions comme dans la preuve précédente, à partir des propositions de déterminisme et d'absence de blocage (propositions 3.3.1 et 3.2.1).*

■

La boucle est traitée en deux étapes comme pour la preuve de la propriété d'induction. On définit une variante de la seconde propriété d'induction pour l'instruction next_iteration.

Lemme 4.2.29 *Soit un programme* $S =$next_iteration B do P end *où le programme SCP* P *vérifie la seconde propriété d'induction. Pour tout état synchrone étendu* $(S, S_1 : .... : S_n, T, c, \tau, \psi, \psi)$, *s'il existe* $\xrightarrow{c^*}$ *tel que*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T'\rangle,$$

alors

$$[\![\text{loopwhere } B \text{ do } P \text{ end}]\!](\psi, c) = \psi' \neq \bot$$

où $(\bullet, S'_1 : .... : S'_n, T', c, \tau', \psi', \psi')$ *est un état synchrone.*

Preuve

On ne détaille pas non plus cette démonstration. Elle est très similaire à celle du lemme 4.2.19 page 94. La démonstration est basée sur une récurrence sur le nombre k *d'itérations exécutées par le calcul de la sémantique opérationnelle. Dans le cas où* $k > 0$, *on décompose la transition initiale* $\xrightarrow{c^*}$ *en trois transitions* $\xrightarrow{c(1)^*}$,$\xrightarrow{c(2)^*}$ *et* $\xrightarrow{c(3)^*}$. *L'existence de ces transitions repose sur les propositions de déterminisme et d'absence de blocage (propositions 3.3.1 et 3.2.1).* ■

Le lemme suivant achève la démonstration de la proposition 4.2.2 page 98. Il prouve la seconde propriété d'induction pour l'instruction loopwhere.

Lemme 4.2.30 *Si le programme $\mathcal{SCP}$ P vérifie la seconde propriété d'induction, alors* $S =$loopwhere B do P end *vérifie la seconde propriété d'induction.*

Preuve

On ne détaille pas la démonstration qui s'appuie sur la preuve du lemme 4.2.19 page 94. La décomposition de la transition initiale est justifiée par les propositions 3.3.1 et 3.2.1. Dans le cas où au moins un indice exécute une itération, on fait appel au lemme 4.2.29 pour traiter l'instruction next_iteration *après avoir exécuté la première itération.* ■

4.3 Correction de la traduction

Avant de prouver la correction de la traduction, nous formalisons le modèle de programmation séquentiel par l'introduction d'une sémantique dénotationnelle pour un langage séquentiel simple $\mathcal{S}$.

4.3.1 Sémantique dénotationnelle pour $\mathcal{S}$

La sémantique dénotationnelle associe à tout programme S la fonction qu'il calcule, notée $[\![S]\!]_{\mathcal{S}}$. La fonction $[\![S]\!]_{\mathcal{S}}$ est définie par induction sur la structure des groupes d'instructions du programme. A partir d'un élément du domaine $(MemEnv_{\mathcal{S}}) \cup \{\perp\}$ elle renvoie un élément de $MemEnv_{\mathcal{S}} \cup \{\perp\}$. L'élément $\perp$ modélise l'état indéfini. L'état $\perp$ est stable : pour tout programme S : $[\![S]\!]_{\mathcal{S}}(\perp) = \perp$. L'ensemble $MemEnv_{\mathcal{S}}$ représente l'ensemble des *environnements simples* dans lesquels sont sauvegardées les valeurs des différentes variables. Un environnement simple φ est une fonction renvoyant pour toute variable X la valeur contenue dans la mémoire. Dans le modèle séquentiel on n'utilise que des variables scalaires (ensemble Var). Ce sont les seules variables sauvegardées dans un environnement simple. Notons que dans les environnements de la sémantique dénotationnelle de $\mathcal{SCP}$, nous sauvegardons à la fois des scalaires et des vecteurs.

Nous donnons la sémantique de $\mathcal{S}$ pour les affectations, la séquence, la conditionnelle et la boucle:

- Affectation d'un élément de tableau.

$$[\![X[E_1] = E_2]\!]_{\mathcal{S}}(\varphi) = \varphi',$$

avec pour tout $Z \in Var$:

$$\varphi'(Z) = \begin{cases} \varphi(E_2) & \text{si } Z = X[\varphi(E_1)], \\ \varphi(Z) & \text{sinon} \end{cases}$$

- Affectation d'une variable simple:$X = E$ avec $X \in Var$.

$$[\![X = E]\!]_{\mathcal{S}}(\varphi) = \varphi',$$

avec pour tout Z: $\varphi'(Z) = \begin{cases} \varphi(E) & \text{si } Z = X, \\ \varphi(Z) & \text{sinon} \end{cases}$

- Séquence d'instructions $\mathcal{S}$: $S;T$

$$[\![S;T]\!]_{\mathcal{S}}(\varphi) = [\![T]\!]_{\mathcal{S}}([\![S]\!]_{\mathcal{S}}(\varphi)),$$

- Conditionnelle : if Exp do S end.

  La conditionnelle conduit à une exécution de S conditionnée par la valeur de l'expression booléenne Exp.

$$[\![\text{if } Exp \text{ do } S \text{ end}]\!]_{\mathcal{S}}(\varphi) = \varphi'$$

avec: $\varphi' = \begin{cases} [\![S]\!]_{\mathcal{S}}(\varphi) & \text{si } \varphi(Exp) = True, \\ \varphi & \text{sinon} \end{cases}$

- Boucle : while Exp do S end.

  La fonction associée à la boucle est récursive et termine lorsque Exp est évaluée à faux. On utilise de manière classique une sémantique dénotationnelle du plus petit point fixe [57]. Lorsque la boucle termine, le calcul de la sémantique dénotationnelle est basé sur le plus petit point fixe de la fonction F notée $fix(F)$:

$$\begin{array}{c} F = \lambda f \ \lambda\varphi \ if \ \varphi(Exp) \\ then \quad f([\![S]\!]_{\mathcal{S}}(\varphi)) \\ else \quad \varphi \end{array}$$

$$[\![\text{while } Exp \text{ do } S \text{ end}]\!]_{\mathcal{S}}(\varphi) = fix(F)(\varphi)$$

  Si la boucle ne termine pas, la sémantique renvoie l'état indéfini $\perp$:

$$[\![\text{while } Exp \text{ do } S \text{ end}]\!]_{\mathcal{S}}(\varphi) = \perp$$

4.3.2 Preuve formelle de correction de la traduction

Dans cette section nous prouvons la correction de la traduction d'un programme séquentiel écrit en $\mathcal{S}$, en un programme parallèle écrit en $\mathcal{SCP}$. Nous prouvons le théorème central (Théorème 4.3.1) en vérifiant l'équivalence des sémantiques lorsque le programme séquentiel termine.

Théorème d'équivalence de la traduction

Théorème 4.3.1 *Soit un programme séquentiel, S, alors l'exécution du programme $\mathcal{SCP}$, $trans(S)$, donne le même résultat final.*

Preuve

La preuve est déduite de la proposition 4.3.1 page 104 qui prouve l'équivalence des résultats des sémantiques dénotationnelles de $\mathcal{S}$ et $\mathcal{SCP}$.

■

Nous supposons que les environnements simples et les environnements de la sémantique dénotationnelle de $\mathcal{SCP}$ utilisent un même ensemble $Vari$ de noms de variables. Dans ce cas il est possible de comparer les valeurs de ces variables. Dans le cas où un même nom de variable représente un vecteur pour $\mathcal{SCP}$ et un scalaire pour $\mathcal{S}$, l'intuition est la suivante: on considérera un vecteur comme équivalent à un scalaire si toutes les composantes du vecteurs sont égales à la valeur du scalaire.

Définition 4.3.1 *La variable X à la même valeur dans un environnement simple φ et un environnement ψ si: $\forall u \in Proc : \varphi(X) = Eval(\psi, \psi, u, X)$. Dans ce cas on notera $\varphi(X) = \psi(X)$.*

On étend naturellement cette définition aux expressions. Une expression a la même valeur dans φ et ψ si pour tout indice u, on a $\varphi(Exp) = Eval(\psi, \psi, u, Exp)$. Dans ce cas on notera $\varphi(Exp) = \psi(Exp)$.

Définition 4.3.2 *Un environnement étendu aux vecteurs ψ est équivalent à un environnement simple φ si chaque variable de $Vari$ a la même valeur dans les deux environnements:*
$\forall u \in Proc, X \in Vari : \varphi(X) = Eval(\psi, \psi, u, X)$
Nous le notons: $Eq(\psi, \varphi)$

La proposition suivante énonce l'équivalence des résultats des sémantiques dénotationnelles de $\mathcal{S}$ et $\mathcal{SCP}$ lorsqu'on part d'environnements équivalents.

Proposition 4.3.1 *Pour tout programme séquentiel S, si les environnements φ et ψ sont équivalents alors:*

$$Eq([\![S]\!]_{\mathcal{S}}(\varphi), [\![Trans(S)]\!](\psi, True))$$

Preuve

Nous construisons la preuve de ce théorème par induction sur la structure des programmes. A chaque étape de l'induction nous comparons les résultats des sémantiques dénotationnelles. Cette preuve est facilitée par la structure de la fonction de traduction de $\mathcal{S}$ vers $\mathcal{SCP}$, qui elle-même est construite par induction sur la structure des programmes. Il suffit donc de vérifier l'équivalence des environnements retournés par les sémantiques pour chaque étape de la preuve (lemmes 4.3.1, 4.3.2, 4.3.4, 4.3.5).

■

4.3.3 La traduction d'une affectation

Lemme 4.3.1 *Pour tout programme séquentiel $S = X[Exp_1] = Exp_2$, si les environnements φ et ψ sont équivalents, alors:*

$$Eq([\![S]\!]_{\mathcal{S}}(\varphi), [\![Trans(S)]\!](\psi, True))$$

Preuve

La sémantique dénotationnelle de $\mathcal{S}$ donne:

$$[\![X[E_1] = E_2]\!]_{\mathcal{S}}(\varphi) = \varphi',$$

avec φ' tel que pour tout $Z \in Var$:

$$\varphi'(Z) = \begin{cases} \varphi(E_2) & si\ Z = X[\varphi(E_1)], \\ \varphi(Z) & sinon \end{cases}$$

La fonction de traduction donne $Trans(S) = step(X[Exp_1] = Exp_2)$. La sémantique dénotationnelle de $\mathcal{SCP}$ donne: $[\![\mathsf{step}\ (X[E_1] = E_2)]\!](\psi, True) = \psi''$ (voir section 4.1.1 page 74). Par hypothèse on a: $Eq(\psi, \varphi)$. Nous en déduisons que pour tout indice u on a: $Eval(\psi, \psi, u, E_1) = \varphi(E_1)$. Nous en déduisons pour tout u que $X[Eval(\psi, \psi, u, E_1)]$ et $X[\varphi(E_1)]$ représentent la même variable scalaire.

Soit un indice u et une variable Z:

1. *Si $Z = X[Eval(\psi, \psi, u, E_1)]$, alors: $Eval(\psi'', \psi'', u, Z) = Eval(\psi, \psi, u, E_2)$ et donc: $Eval(\psi'', \psi'', u, Z) = \varphi(E_2)$ et donc: $Eval(\psi'', \psi'', u, Z) = \varphi'(Z)$*
2. *Si $Z \neq X[Eval(\psi, \psi, u, E_1)]$, alors: $Eval(\psi'', \psi'', u, Z) = Eval(\psi, \psi, u, Z)$ et donc: $Eval(\psi'', \psi'', u, Z) = \varphi'(Z)$*

■

Lemme 4.3.2 *Pour tout programme séquentiel $S = X = Exp$, si les environnements φ et ψ sont équivalents, alors:*

$$Eq([\![S]\!]_{\mathcal{S}}(\varphi), [\![Trans(S)]\!](\psi, True))$$

Preuve

La preuve est une version simplifiée de celle du lemme 4.3.1. ■

4.3.4 La traduction d'une conditionnelle

Lemme 4.3.3 *Pour tout programme séquentiel* S = if Exp then S end, *si les environnements* φ *et* ψ *sont équivalents, alors:*

$$Eq([\![S']\!]_{\mathcal{S}}(\varphi), [\![Trans(S')]\!](\psi, True))$$

Preuve

Par hypothèse les environnements φ *et* ψ *sont équivalents, alors :* $\varphi(Exp) = \psi(Exp)$. *On en déduit que pour tout indice* u*:*

$$Eval(\psi, \psi, u, Exp)) = \varphi(Exp) \tag{4.8}$$

La sémantique du where *(voir 4.1.2 page 77) en partant d'un contexte* $True$ *est la suivante:*

$$[\![\text{where } B \text{ do } S \text{ end}]\!](\psi, True) = [\![\text{where } B \text{ do } S \text{ elsewhere step () end}]\!](\psi, True) = \psi'$$

avec:

$$[\![S]\!](\psi, Eval(\psi, \psi, Exp)) = \psi'_1$$

L'environnement final ψ' *est construit à partir de l'environnement* ψ'_1 *:*

$$\text{pour tout } X,\ \psi'(X) = \begin{cases} \psi'_1(X) & si \begin{cases} (X \in Var \text{ et } Owner(X) \in \psi(Exp)) \text{ ou} \\ (X = I|_u \text{ (tel que } I \in Ind) \text{ et } u \in \psi(Exp)) \end{cases} \\ \psi(X) & sinon \end{cases},$$

Utilisons (4.8), on en déduit: $[\![Trans(S')]\!](\psi, True) = [\![Trans(S)]\!](\psi, Eval(\psi, \psi, Exp))$

La sémantique de $\mathcal{S}$ *donne:* $[\![\text{if } Exp \text{ do } S \text{ end}]\!]_{\mathcal{S}}(\varphi) = \varphi'$

$$\text{avec: } \varphi' = \begin{cases} [\![S]\!]_{\mathcal{S}}(\varphi) & si\ \varphi(Exp) = True, \\ \varphi & sinon \end{cases}$$

Il faut alors distinguer le cas où la condition est vérifiée du cas où elle ne l'est pas:

- *Si* $\varphi(Exp)$, *alors par hypothèse on a* $\psi(Exp)$ *et donc* $[\![S']\!]_{\mathcal{S}}(\varphi) = [\![S]\!]_{\mathcal{S}}(\varphi)$. *Par hypothèse d'induction on a* $[\![S]\!]_{\mathcal{S}}(\varphi) = [\![Trans(S)]\!](\psi, True)$. *Or par hypothèse d'induction on a* $[\![Trans(S)]\!](\psi, True) = [\![Trans(S')]\!](\psi, True)$, *donc* $[\![S']\!]_{\mathcal{S}}(\varphi) = [\![Trans(S')]\!](\psi, True)$.
- *Si* $\neg\varphi(Exp)$, *alors par hypothèse on a:* $\neg\psi(Exp)$ *donc par les mêmes arguments que précédemment:*

$$[\![S']\!]_{\mathcal{S}}(\varphi) = \varphi = [\![S]\!](\psi, False) = [\![Trans(S')]\!](\psi, True)$$

■

4.3.5 La traduction d'une boucle while

Lemme 4.3.4 *Pour tout programme séquentiel* S = while Exp do S end, *si les environnements* φ *et* ψ *sont équivalents, on a:*

$$Eq([\![S']\!]_{\mathcal{S}}(\varphi), [\![Trans(S')]\!](\psi, True))$$

Preuve

Par hypothèse les environnements φ *et* ψ *sont équivalents, donc* $\varphi(Exp) = \psi(Exp)$, *on en déduit que pour tout indice* u

$$Eval(\psi, \psi, u, Exp)) = \varphi(Exp) \tag{4.9}$$

La sémantique dénotationnelle du while *est donnée en 4.3.1 (page 103).*

On suppose que $[\![\text{while } Exp \text{ do } S \text{ end}]\!]_{\mathcal{S}}(\varphi) \neq \perp$. *Il existe donc un entier* k *tel que*

$$[\![\text{while } Exp \text{ do } S \text{ end}]\!]_{\mathcal{S}}(\varphi) = f_k(\varphi) = \varphi'$$

La sémantique dénotationnelle du loopwhere *est donnée en 4.1.2 (page 76).*

On fait alors une preuve par récurrence. On montre que pour tout entier $k \geq 0$, *si* $f_k(\varphi) = \varphi'$, *alors* $g_k(\psi, True) = \psi'$ *et* $Eq(\varphi', \psi')$.

- $k = 0$. *Dans ce cas* $\neg\varphi(Exp)$ *et* $f_k(\varphi) = \varphi' = \varphi$. *Par hypothèse on a* $Eq(\psi, \varphi)$, *donc* $\neg\psi(Exp)$ *et* $g_k(\psi, True) = \psi' = \psi$. *On en déduit que les environnements* φ' *et* ψ' *sont équivalents.*
- $k > 0$. *On suppose la propriété vraie pour* $k - 1$. *Montrons qu'elle est vérifiée pour* k.

  La sémantique dénotationnelle de $\mathcal{S}$ *nous donne*

$$\begin{aligned} [\![\text{while } Exp \text{ do } S \text{ end}]\!]_{\mathcal{S}}(\varphi) &= f_k(\varphi) \\ &= f_{k-1}([\![S]\!]_{\mathcal{S}}(\varphi)) \\ &= f_{k-1}(\varphi'') \\ &= (\varphi') \end{aligned} \tag{4.10}$$

  La sémantique dénotationnelle de $\mathcal{SCP}$ *nous donne*

$$[\![\text{loopwhere } B \text{ do } S \text{ end}]\!](\psi, True) = g_k(\psi, True)$$

  De (4.10) on déduit $\varphi(Exp) = True$. *Par hypothèse d'induction on a* $Eq(\varphi, \psi)$, *donc* $\psi(Exp) = True$. *On en tire*

$$[\![\text{loopwhere } B \text{ do } S \text{ end}]\!](\psi, True) = g_{k-1}([\![S]\!](\psi, True), True)$$

De l'hypothèse d'induction on déduit

$$[\![\textsf{loopwhere}\ \ B\ \ \textsf{do}\ \ S\ \ \textsf{end}]\!](\psi, True) = g_{k-1}(\psi'', True),$$

avec φ'' et ψ'' équivalents.
De l'hypothèse d'induction sur les boucles on déduit

$$[\![\textsf{loopwhere}\ \ B\ \ \textsf{do}\ \ S\ \ \textsf{end}]\!](\psi, True) = \psi',$$

avec φ' et ψ' équivalents.

■

Remarque. Nous ne détaillons pas la preuve de la traduction d'une boucle for par un forwhere. Il suffit de traduire la boucle for en boucle while, puis de traduire cette boucle en loopwhere. Il reste alors à traduire la boucle loopwhere en forwhere.

4.4 Conclusion

Dans ce chapitre, nous avons introduit une formalisation du modèle de programmation de $\mathcal{SCP}$, par le biais d'une sémantique dénotationnelle structurée par la syntaxe.

La preuve d'équivalence entre la sémantique dénotationnelle et la sémantique opérationnelle valide le modèle de programmation de $\mathcal{SCP}$ par rapport à sa machine abstraite. Elle repose sur l'évolution de calculs moins asynchrones que les autres correspondant intuitivement à une "exécution" pas à pas de la sémantique dénotationnelle. La principale difficulté consiste à identifier les propriétés stables à chaque étape de ces exécutions. Cette preuve est transposable à d'autres langages dont les communications sont dirigées par la syntaxe, si l'on identifie les propriétés stables qui leurs sont propres. Dans le cas de $\mathcal{SCP}$, la preuve est facilitée par la structure en étages de la sémantique opérationnelle qui permet de distinguer clairement les transitions modifiant l'environnement de celles modifiant les horloges.

Cette preuve réutilise le schéma de la preuve d'équivalence des sémantiques pour le langage $\mathcal{SCL} - \mathcal{C}han$ [48]. Les deux langages utilisent le même mécanisme de synchronisation par le biais d'horloges structurelles. Les preuves spécifiques aux horloges sont semblables. Cela concerne la preuves de non blocage, et certains lemmes techniques. Les deux langages utilisent les mêmes structures de contrôles. Les lemmes concernant les structures de contrôles sont donc voisins. Ils diffèrent toutefois notablement pour ce qui concerne la gestion de la mémoire et des communications. $\mathcal{SCP}$ autorise les accès distants et les communications implicites alors que les communications passent par des envois de messages explicites à travers des canaux en $\mathcal{SCL} - \mathcal{C}han$. De plus, en $\mathcal{SCL} - \mathcal{C}han$, une donnée lue dans un canal est consommée. Cette différence entre les deux modèles introduit de notables différences entre les deux preuves en ce qui concerne le déterminisme et les lemmes de l'équivalence liées au where/elsewhere. De plus, elle influe sur le choix des propriétés

stables à vérifier à chaque étape. La différence essentielle entre les deux preuves concerne l'ajout d'un langage hôte qui impose la définition d'une condition de stabilité propre à ce langage. Il reste que la proximité des deux démonstrations illustre la généralité de leur structure commune. L'hypothèse commune pour les deux langages concerne les accès distants dirigés par la syntaxe. Une étape supplémentaire serait de dégager une méthodologie générale de preuves pour ce type de langage.

Nous donnons les propriétés minimales sur la sémantique dénotationnelle du langage hôte $\mathcal{LH}$, afin que la sémantique de $\mathcal{SCP}$ soit valide. Nous pouvons largement réutiliser le même schéma de preuve avec un langage hôte étendu. Par exemple l'adjonction de structures de contrôle dynamiques dont les expressions sont évaluées comme celles des affectations ne modifie pas fondamentalement la preuve. Les propriétés du langage hôte $\mathcal{LH}$ sont très peu contraignantes. Tout langage hôte "raisonnable" peut les vérifier.

La validation du langage $\mathcal{SCP}$ permet de se restreindre au formalisme de la sémantique dénotationnelle pour aborder la preuve de la correction de traduction de code séquentiel en code parallèle. Puisque la traduction reproduit la structure du code séquentiel, nous obtenons un cadre formel qui autorise des preuves par induction, réduisant ainsi la complexité de la validation de la traduction.

Devant la complexité théorique des preuves il pourrait être intéressant d'en automatiser certaines étapes grace à un "theorem prover".

4.5 Annexe : preuves des lemmes techniques

On présente les preuves des lemmes techniques de la section 4.2. Les énoncés des différents lemmes sont repris en conservant leur numérotation initiale.

4.5.1 Lemmes relatifs à l'étape d'induction sur les affectations de $\mathcal{LH}$

Lemme 4.2.2 page 81

Lemme 4.2.2 *Soit τ cohérent, et T tel que pour tous indice i appartenant au contexte c, il existe une horloge t telle que $T|_i = t$. Soit S_i tel que pour tout i appartenant au contexte c, $S_i = X[E_1] = E_2$. Considérons la transition contextuelle $\xrightarrow{c^*}$ telle que:*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T\rangle$$

Posons $w = Owner(X[Proj_2(Evalop(\tau, v, E_1, T))])$. Alors, τ' est tel que pour toute variable Z, tout indice v, tout environnement d'horloges T' on a:

$$Proj_2(Evalop(\tau', v, Z, T')) =$$
$$\begin{cases} Proj_2(Evalop(\tau, w, E_2, T)) \ si \begin{cases} w \in c \ et \ ((v = w \ et \ T'|_v = T|_w) \ ou \ T'|_v \succ T|_w) \ et \\ Z = X[Proj_2(Evalop(\tau, v, E_1, T))]) \end{cases} \\ Proj_2(Evalop(\tau, v, Z, T')) \ sinon \end{cases},$$

Preuve

Rappelons la sémantique de l'affectation d'un élément de tableau $X[E_1] = E_2$:

$$\frac{Proj_1(Evalop(\tau, u, E_1, T)) \ \wedge \ Proj_1(Evalop(\tau, u, E_2, T))}{\langle X[E_1] = E_2, \tau, u, T\rangle \mapsto \langle \bullet, \tau', u, T\rangle}$$

avec τ' est tel que pour toute variable Z:

$$\tau'|_{Owner(Z)}(Z) = \begin{cases} \tau|_{Owner(Z)}(Z)(Proj_2(Evalop(\tau, u, E_2, T)), T|_{Owner(Z)}) \\ \qquad si \begin{cases} Z = X[Proj_2(Evalop(\tau, u, E_1, T))] \ et \\ u = Owner(Z) \end{cases} \\ Change(\tau|_{Owner(Z)}(Z), Proj_2(Evalop(\tau, u, E_2, T))) \\ \qquad si \begin{cases} Z = X[Proj_2(Evalop(\tau, u, E_1, T))] \ et \\ T|_u = Top(\tau, Z) \ et \\ u = Owner(Z) \end{cases} \\ \tau'|_{Owner(Z)}(Z) \ sinon. \end{cases},$$

Pour tout indice v distinguons les cas selon la valeur de Z:

- *si $Z \neq X[Proj_2(Evalop(\tau, v, E_1, T))]$ alors les transitions sur l'indice u ne changent pas l'environnement $\tau|_{Owner(Z)}(Z)$, donc on a $\tau'|_{Owner(Z)}(Z) = \tau|_{Owner(Z)}(Z)$. On en déduit par définition de Evalop:*

$$Proj_2(Evalop(\tau', v, Z, T')) = Proj_2(Evalop(\tau, v, Z, T')).$$

- *si* $Z = X[Proj_2(Evalop(\tau, v, E_1, T))]$ *alors suivant l'indice* v,
  - ▷ *si* v *tel que* $Owner(Z) \notin c$, *alors* $\forall u \in c, u \neq Owner(Z)$ *et donc les transitions sur l'indice* u *ne changent pas l'environnement* τ, *et donc* $\tau'|_{Owner(Z)}(Z) = \tau|_{Owner(Z)}(Z)$. *Par définition de Evalop, on en déduit* $Proj_2(Evalop(\tau', v, Z, T')) = Proj_2(Evalop(\tau, v, Z, T'))$.
  - ▷ *si* v *tel que* $Owner(Z) \in c$, *alors* $\exists u \in c, u = Owner(Z)$ *et donc la transition sur l'indice* u *change l'environnement* $\tau|_u = \tau|_{Owner(Z)}$ *pour* Z. *On a*

    $$\tau'|_{Owner(Z)}(Z) = \tau|_{Owner(Z)}(Z)(Proj_2(Evalop(\tau, u, E_2, T)), T|_{Owner(Z)}),$$

    ou bien

    $$\tau'|_{Owner(Z)}(Z) = Change(\tau|_{Owner(Z)}(Z), Proj_2(Evalop(\tau, u, E_2, T))).$$

    Suivant la valeur de $T'|_v$ *nous distinguons trois cas.*

    1. *Si* $T'|_v = T|_{Owner(Z)}$ *alors comme par hypothèse,* τ *est cohérent, on a* $max\{t' \in \mathcal{T}(\tau', Z) \ / \ t' \preceq T'|_v\} = T|_{Owner(Z)}$. *Dans ce cas d'après la définition de Evalop:*

       $$Proj_2(Evalop(\tau', v, Z, T')) = Proj_2(Evalop(\tau, u, E_2, T))$$

    2. *Si* $T'|_v \succ T|_{Owner(Z)}$ *alors comme par hypothèse,* τ *est cohérent, on a* $max\{t' \in \mathcal{T}(\tau', V) \ / \ t' \prec T|_u\} = T|_{Owner(Z)})$. *Dans ce cas d'après la définition de Evalop:*

       $$Proj_2(Evalop(\tau', v, Z, T')) = Proj_2(Evalop(\tau, u, E_2, T))$$

    3. *Si* $\neg(T'|_v \succeq T|_{Owner(Z)})$ *alors comme par hypothèse,* τ *est cohérent, on a* $max\{t' \in \mathcal{T}(\tau', V) \ / \ t' \preceq T|_u\} = max\{t' \in \mathcal{T}(\tau, V) \ / \ t' \preceq T|_u\}$. *Dans ce cas d'après la définition de Evalop:*

       $$Proj_2(Evalop(\tau', v, Z, T')) = Proj_2(Evalop(\tau, v, Z, T'))$$

■

Lemme 4.2.6 page 85

Lemme 4.2.6 *Soit* S_i *tel que pour tout* i *appartenant au contexte* c, $S_i = S_i : I = E$ *où* $I \in Ind$. *Considérons la transition contextuelle* $\xrightarrow{c^*}$ *telle que:*

$$\langle S_1 : ... : S_n, \tau, T\rangle \xrightarrow{c^*} \langle S'_1 : ... : S'_n, \tau', T\rangle$$

Alors, τ' *est tel que pour toute variable* Z, *tout indice* v, *tout environnement d'horloges* T' *on a:*

$$Proj_2(Evalop(\tau', v, Z, T')) =$$
$$\begin{cases} Proj_2(Evalop(\tau, v, E, T)) \ si\ v \in c\ et\ Z = I, \\ Proj_2(Evalop(\tau, v, Z, T')) \ sinon \end{cases}$$

Preuve

Rappelons la sémantique de l'affectation d'un vecteur : $I = E$ où $I \in Ind$.

$$\frac{Proj_1(Evalop(\tau, u, E, T))}{\langle I = E, \tau, u, T\rangle \mapsto \langle \bullet, \tau', u, T\rangle}$$

avec τ' tel que tout indice v, variable Z, :

$$\tau'|_v(Z) =$$
$$\begin{cases} \tau|_v(E) \ si \begin{cases} v = u\ et \\ Z = I) \end{cases} \\ \tau'|_v(Z) \ sinon \end{cases},$$

Pour tout indice v distinguons les cas selon la variable Z:

- *si $Z \neq I$ alors les transitions sur l'indice u ne changent pas l'environnement $\tau|_v(Z)$, on a $\tau'|_v(Z) = \tau|_v(Z)$. On en déduit par définition de Evalop*

$$Proj_2(Evalop(\tau', v, Z, T')) = Proj_2(Evalop(\tau, v, Z, T')).$$

- *si $Z = I$ alors distinguons les cas selon le contexte.*
  - ▷ *Si $v \notin c$, les transitions sur l'indice u ne changent pas l'environnement τ, on a $\tau'|_v(Z) = \tau|_v(Z)$. On en déduit par définition de Evalop*

$$Proj_2(Evalop(\tau', v, Z, T')) = Proj_2(Evalop(\tau, v, Z, T')).$$

  - ▷ *Si $v \in c$, alors la transition sur l'indice v change l'environnement $\tau|_v$ pour Z. On a*

$$\tau'|_v(Z) = Proj_2(Evalop(\tau, u, E, T)).$$

■

Lemme 4.2.10 page 87

Lemme 4.2.10 *Soit un environnement τ cohérent vis-à-vis de T, un indice j et une horloge t_j tels que $T|_j \prec t_j$. Soit T' tel que $T'|_j = t_j$ et $\forall i \neq j,\ T'|_i = T|_i$. Alors on a*

$$\forall u \in Proc, \forall X \in Var \cup Ind,\ Proj_2(Evalop(\tau, X, u, T)) = Proj_2(Evalop(\tau, X, u, T'))$$

Preuve

Distinguons les cas où $X \in Var$ et $X \in Ind$

- $X \in Var$.
  D'après la définition de la fonction Evalop:

$$Proj_2(Evalop(\tau, u, X, T)) = \begin{cases} max(\tau, u, V, T|_u) \text{ si } Owner(X) = u, \\ maxstrict(\tau, u, V, T|_u) \text{ sinon.} \end{cases}$$

  Lorsque $u \neq j$, la fonction renvoie clairement le même résultat quelque soit $T'|_j$. La seule difficulté de la preuve correspond au cas : $j = u$. Nous avons alors:

$$Proj_2(Evalop(\tau, j, X, T)) = \begin{cases} max(\tau, j, X, t_j) \text{ si } Owner(X) = j, \\ maxstrict(\tau, j, X, t_j) \text{ sinon.} \end{cases}$$

  Par hypothèse, l'environnement τ est cohérent *vis-à-vis de T. Donc pour tout $X \in Var$, $i \in Proc$ tel que $i \neq Owner(X)$ et pour toute paire associée à X $(Val, t) \in \tau$, on a $t \preceq T|_{Owner(X)}$ et $\neg(T|_i \prec t)$.*
  donc il n'existe pas de t tel que: pour toute paire associée à X $(Val, t) \in \tau$ et $\forall i \in Proc$ tel que $i \neq Owner(X)$ on a: $T|_i \prec t$.
  Donc si $j \neq Owner(X)$:

$$max\{t' \in \mathcal{T}(\tau, X) \ / \ t' \preceq t_j\} = max\{t' \in \mathcal{T}(\tau, X) \ / \ t' \preceq T|_j\}$$
$$max\{t' \in \mathcal{T}(\tau, X) \ / \ t' \prec t_j\} = max\{t' \in \mathcal{T}(\tau, X) \ / \ t' \prec T|_j\}$$

  si $j = Owner(X)$ alors pour tout t tel que: pour toute paire associée à X $(Val, t) \in \tau$ on a: $t \preceq T|_j$
  Donc:

$$max\{t' \in \mathcal{T}(\tau, X) \ / \ t' \preceq t_j\} = max\{t' \in \mathcal{T}(\tau, X) \ / \ t' \preceq T|_j\}$$
$$max\{t' \in \mathcal{T}(\tau, X) \ / \ t' \prec t_j\} = max\{t' \in \mathcal{T}(\tau, X) \ / \ t' \prec T|_j\}$$

- $X \in Ind$. *D'autre part, Si X est une variable vectorielle ($X \in Ind$):*
  La fonction Evalop est définie comme telle:

$$Evalop(\tau, u, V, T) = \ (vrai, X),$$

avec $X = max(\tau, u, V, T|_u)$

La fonction $max()$ renvoi une seule valeur pour τ, u, V quelque soit l'horloge T_u.

■

4.5.2 Lemmes relatifs à l'étape de la conditionnelle

Lemme 4.2.13 page 89

Lemme 4.2.13 *Soit un triplet $(S_1 : ... : S_n, T, c)$ non bloquant et un contexte réduit $c' \subseteq c$. Soit un programme $\mathcal{SCP}$, S, tel que pour tout $i \in c'$, $S_i = S$. Si pour tout $j \in c \backslash c'$ et $i \in c'$ on a $T|_j = w(a, b)$ et $T|_i = w(e, f)q$ (où q peut être vide) et $a \neq e$, alors le triplet $(S_1 : ... : S_n, T, c')$ est non bloquant.*

Preuve

Montrons que les indices de $c \backslash c'$ ne bloquent pas ceux du contexte c'. Soit un indice $j \in c \backslash c'$ et un indice $i \in c'$ tels que $T|_j = w(a, b)$, $T|_i = w(e, f)q$ et $a \neq e$. Les horloges $T|_j$ et $T|_i$ sont donc incomparables. Considérons une transition $\xrightarrow{c'^*}$ appliquée aux indices de c' telle que :

$$\langle S_1 : ... : S_n, \tau, T \rangle \xrightarrow{c'^*} \langle S'_1 : ... : S'_n, \tau', T' \rangle.$$

D'après le lemme 3.2.2 page 64, l'horloge de i est du type $T'|_i = w(e, f')q'$, donc $T'|_j$ et $T'|_i$ sont incomparables ($T'|_j = T|_j$ car $j \notin c'$). Finalement $\neg(T'|_j \prec T'|_i)$, ainsi le triplet $(S_1 : ... : S_n, T, c')$ est non bloquant. ■

Lemme 4.2.14 page 89

Lemme 4.2.14 *Soit un environnement τ cohérent vis-à-vis d'un environnement d'horloges structurelles T. Si on a $\forall\, i \in c, T'|_i \succeq T|_i$ et $Equiv(\tau, \psi, \psi, T, c)$ alors on a: $Equiv(\tau, \psi, \psi, T', c)$.*

Preuve

D'après le lemme 4.2.1 on déduit que τ est cohérent vis-à-vis de T'.

Donc d'après le lemme 4.2.10:

$$\forall V \in Var \cup Ind, i \in c, ona : Proj_2(Evalop(\tau, i, V, T)) = \ Proj_2(Evalop(\tau, i, V, T'))$$

De $Equiv(\tau, \psi, \psi, T, c)$ on déduit:

$$\forall V \in Var \cup Ind, i \in c, ona : Proj_2(Evalop(\tau, i, V, T)) = Eval(\psi, \psi, i, V).$$

Donc:

$$\forall V \in Var \cup Ind, i \in c, ona : Proj_2(Evalop(\tau, i, V, T')) = Eval(\psi, \psi, i, V).$$

Et donc: $Equiv(\tau, \psi, \psi, T', c)$.

■

4.5.3 Lemmes relatifs à l'étape d'induction de la boucle

Lemme 4.2.16 page 93

Lemme 4.2.16 *Soit une triplet $(S_1 : ... : S_n, T, c)$ non bloquant et un contexte réduit $c' \subseteq c$. Soit un programme $\mathcal{SCP}$ S tel que pour tout $i \in c'$, $S_i = S$. Si pour tout $j \in c \backslash c'$ et $i \in c'$ on a $T|_j = w(a, b)$ et $T|_i = w(a, f)q$ où q est non vide et $b > f$, alors le triplet $(S_1 : ... : S_n, T, c')$ est non bloquant.*

Preuve

Montrons que les indices de $c \backslash c'$ ne bloquent pas ceux du contexte c'. Soit un indice $j \in c \backslash c'$ et un indice $i \in c'$ tels que $T|_j = w(a, b)$, $T|_i = w(a, f)q$ où q est non vide et $b > f$. Considérons une transition $\xrightarrow{c'*}$ appliquée aux indices de c' telle que :

$$\langle S_1 : ... : S_n, \tau, T \rangle \xrightarrow{c'*} \langle S'_1 : ... : S'_n, \tau', T' \rangle$$

Par hypothèse $b > f$, donc l'horloge de j est supérieure à celle de i : $T|_j \succ T|_i$. D'après le lemme 3.2.2 page 64, l'horloge de i est du type $T'|_i = w(a, f)q'$ par conséquent $T'|_j \succ T'|_i$. Le triplet $(S_1 : ... : S_n, T, c)$ est donc non bloquant. ■

Lemme 4.2.17 page 93

Lemme 4.2.17 *Soit un programme $\mathcal{SCP}$ P, un état $\langle S_1 : ... : S_n, \tau, T \rangle$ et un indice i tel que $S_i =$*next_iteration *B* do *P* end *et $T|_i = t(a, b)(e, f)$ (t peut être vide). S'il existe une transition $\xrightarrow{*}$ telle que :*

$$\langle S_1 : ... : S_n, \tau, T \rangle \xrightarrow{*} \langle S'_1 : ... : S'_n, \tau', T' \rangle,$$

alors on a :

- $T'|_i = t(a, b)(e, f')q'$ *où* $f' \geq f$ *si* $S'_i \neq \bullet$,
- $T'|_i = t(a, b + 1)$ *sinon.*

Preuve

- **Si** $S'_i \neq \bullet$, *l'évaluation du test B à vrai ne modifie pas l'horloge de i. De plus, la forme des horloges reste inchangée par l'exécution de P (lemme 3.2.2 page 64). On a donc le résultat attendu.*
- **Si** $S'_i = \bullet$, *avant d'évaluer à faux la condition B l'indice i a une horloge de la forme $T''|_i = t(a,b)(e,f'')$ car l'exécution de P est terminée (lemme 3.2.3 page 65). Donc en sortie de* next_iteration, *i possède l'horloge $T''|_i = t(a,b)$.*

■

Lemme 4.2.18 page 93

Lemme 4.2.18 *Soit une triplet $(S_1 : ... : S_n, T, c)$ non bloquant et un contexte réduit $c' \subseteq c$. Soit un programme $S =$*next_iteration *B* do *P* end *où P est un programme $\mathcal{SCP}$ et où pour tout $i \in c'$, $S_i = S$. Si pour tout $j \in c \backslash c'$ et $i \in c'$ on a $T|_j = w(a,b)$ et $T|_i = w(a,f)q$ où q est non vide et $b > f$, alors le triplet $(S_1 : ... : S_n, T, c')$ est non bloquant.*

Preuve

On ne détaille pas la preuve, très similaire à celle du lemme 4.2.16. On reprend le même raisonnement en utilisant le lemme 4.2.17. ■

Chapitre 5

Modèle à passage de message

Le début de ce chapitre jusqu'à la section 5.2 incluse, a fait l'objet d'une publication [49] et d'un rapport de recherche [48]. Les sections suivantes développent les travaux présentés dans les articles [51, 50]. Ces publications sont cosignées avec Xavier Rebeuf, Bruno Raffin et Bernard Virot.

Dans ce chapitre, nous nous intéressons au problème de l'expansion de la mémoire dans le cadre de la traduction de code séquentiel en code parallèle fondée sur le modèle de synchronisation introduit dans $\mathcal{SCP}$.

Dans le première partie du chapitre, nous introduisons le langage $\mathcal{SCL}-\mathcal{C}han$ (*Structural Clock Language Channel*). Il permet de remplacer l'empilement systématique des valeurs des variables par un empilement ad hoc. Tout en conservant le principe des communications dirigées par la structure du programme, on remplace les lectures en mémoire distantes par un mécanisme de communication explicite par passage de message. L'empilement des valeurs est alors conditionné par les envois explicites de données, ouvrant ainsi la possibilité de n'empiler en mémoire que les données utiles pour le calcul. Cet avantage est toutefois conditionné par la nécessité pour chaque indice, de connaître les indices distants susceptibles d'utiliser ses données locales.

$\mathcal{SCL}-\mathcal{C}han$ reprend le mécanisme de synchronisation de $\mathcal{SCP}$. Comme en $\mathcal{SCP}$, la précédence entre les instructions à l'exécution respecte un ordre logique fondé sur leurs positions dans la syntaxe du programme. La sémantique des communications respecte encore la structure syntaxique du programme.

A la différence du modèle $\mathcal{SCP}$, les programmes $\mathcal{SCL}-\mathcal{C}han$ ne sont pas structurés en groupes d'instructions. C'est l'absence de communication entre des instructions exécutées par des indices différents qui exprime l'indépendance entre ces instructions.

La machine abstraite de $\mathcal{SCL}-\mathcal{C}han$ reprend le même mécanisme de masquage temporaire de l'exécution d'une structure pour un indice distant, basé sur les horloges structurelles. Il est possible d'abstraire le mécanisme de synchronisation fondé sur l'ordre structurel, grâce à une vision synchrone de l'exécution. La transmission de signaux portant les horloges structurelles autorise l'implantation d'un mécanisme d'attente entre les in-

dices, exploitant la précédence entre les instructions: le récepteur attend que l'émetteur ait exécuté toutes les instructions d'envoi *précédant* logiquement l'instruction de réception. L'attente n'est donc pas conditionnée par la réception d'une donnée comme dans le modèle standard de MPI [61]. On évite ainsi les blocages causés par l'absence d'envoi correspondant à une réception. Les envois asynchrones accumulent les données en transit dans des canaux gérés en mode LIFO. Il permet d'absorber automatiquement les anciennes valeurs envoyées, pour ne recevoir que la plus récente au sens de l'ordre logique.

La deuxième partie du chapitre énonce les résultats fondamentaux concernant le langage $\mathcal{SCL}-Chan$: déterminisme, absence de blocage et équivalence des modèles d'exécution et de programmation.

La troisième partie concerne la traduction de code séquentiel en code $\mathcal{SCL}-Chan$. Comme avec le modèle $\mathcal{SCP}$, elle est fondée sur la distribution des données. Elle suit le principe des écritures locales. Des gardes sont explicitement insérées pour gérer dynamiquement les communications. L'exécution des affectations est gardée. Le code final est totalement indépendant du choix de la distribution des données. Grâce au schéma de communication dirigé par la syntaxe, la traduction de code séquentiel en code $\mathcal{SCL}-Chan$ est modulaire.

Nous montrons qu'il est possible de tirer parti d'une analyse floue des dépendances permettant de connaître statiquement un sur ensemble des variables affectées et un sur ensemble des variables lues. Nous montrons qu'il est possible de tirer parti de la possibilité de non correspondance entre les émissions et les réceptions en épargnant le parcours de structures de contrôle "inutiles" aux indices qui n'ont aucune affectation à y réaliser.

Dans la dernière partie, nous discutons l'intérêt comparé des approches $\mathcal{SCP}$ et $\mathcal{SCL}-Chan$ pour l'optimisation de la traduction automatique de code séquentiel en code parallèle. Le modèle $\mathcal{SCL}-Chan$ paraît bien convenir aux programmes permettant l'optimisation des gardes d'émissions, grâce à une analyse statique des dépendances. Au contraire, le mode d'accès du langage $\mathcal{SCP}$ le rend bien adapté lorsque cette optimisation échoue, au prix toutefois d'une augmentation du prix de la gestion mémoire.

5.1 Présentation du langage $\mathcal{SCL}-Chan$

Dans cette section nous présentons le langage $\mathcal{SCL}-Chan$. La machine abstraite est semblable à celle de $\mathcal{SCP}$. Les définitions de Var, Ind, et $This$ sont les mêmes. L'allocation de la mémoire est uniforme. Chaque variable est systématiquement allouée dans chaque mémoire locale. A la différence de $\mathcal{SCP}$, les expressions légales sont les expressions *pures* au sens de HPF c'est à dire sans effet de bord.

L'échange des données passe par des communications explicites de type *send/receive*. Une donnée stockée en mémoire privée de u devient accessible pour un indice distant v si elle a été explicitement *envoyée* à v. Un indice v peut stocker une telle valeur que si il l'a explicitement *reçue*. Les données en transit (celle envoyées mais non encore reçues) sont stockées dans des canaux (éventuellement infinis). Chaque canal est une liste LIFO vis-à-vis de *l'ordre structurel*. Il existe exactement un canal par variable et par couple

d'indices. Nous dénotons par (u, v, X), le canal associé à l'émetteur u, le récepteur v et la variable X. Au début de l'exécution tous les canaux sont vides. Les communications sont asynchrones: seules les réceptions sont bloquantes. Le langage $\mathcal{SCL}-Chan$ suit le modèle SPMD. Toutes les instructions exécutent le même programme. Un indice exécutant une instance d'instruction est dit actif pour cette instance.

Contrairement à $\mathcal{SCP}$, un programme $\mathcal{SCL}-Chan$ n'est pas structuré en groupes d'instructions. En $\mathcal{SCL}-Chan$, l'envoi de message est la structure qui correspond en $\mathcal{SCP}$ à la mise à jour des valeurs en fin de groupe d'instructions. La sémantique de la réception de message est fonction de la position des envois. Les instructions *significatives* pour coder la position des indices vis-à-vis de ces envois sont les instructions d'envoi elles-mêmes et les structures de contrôle susceptibles de les englober. Tout comme le modèle $\mathcal{SCP}$, le codage de la position des indices est réalisé par les horloges structurelles. La méthode de construction est tout à fait similaire à celle donnée au chapitre 2.2.1. La notion d'ordre structurel est la même que celle décrite en 2.2.1 page 22.

5.1.1 Les instructions du langage

Nous définissons les instructions du langage:

Pas d'action: skip. Cette instruction ne fait rien et termine immédiatement.

Affectation: $X[E_1] := E_2$. Un indice actif u met à jour la valeur de $X[E_1]$ dans sa mémoire locale avec la valeur de E_2. Les expressions E_1 et E_2 doivent être *pures*, i.e. évaluables dans la mémoire locale de l'indice u.

Séquence: $S;T$. Quand un indice actif finit l'exécution de S, il commence l'exécution de T. Cette instruction n'implique pas de synchronisation implicite.

Conditionnelle: where B do S elsewhere T end. Si un indice actif u évalue la condition pure B à $vrai$, alors il exécute le bloc d'instructions S, sinon il exécute T.

Remarquons qu'il est possible de définir comme en $\mathcal{SCP}$ une conditionnelle à une seule branche where B do S end de la façon suivante: where B do S elsewhere skip end.

Boucle: loopwhere B do S end. L'itération est exprimée par un dépliage classique. Un indice actif exécute répétitivement S tant qu'il évalue l'expression pure $B|_u$ à $vrai$.

Remarquons qu'il est possible de définir comme en $\mathcal{SCP}$ une structure de contrôle forwhere (voir 2.4.2 page 39).

Envoi: send X to A. L'expression A doit être pure et X est une variable. Cette instruction n'est pas bloquante. Un indice actif u ajoute la valeur de X au canal (u, v, X). La valeur de l'horloge courante de u est associée dans le canal à la valeur envoyée.

Réception: receive X from A into Y. Cette instruction est la seule instruction bloquante. L'expression A doit être pure et X, Y sont des variables. La sémantique de cette

instruction est proche de celle des lectures à distance dans le modèle $\mathcal{SCP}$. Comme en $\mathcal{SCP}$, elle dépend de l'ordre structurel. Son exécution, comme en $\mathcal{SCP}$, se déroule en deux phases. Un indice u doit d'abord vérifier une *condition d'attente* avant de pouvoir sélectionner une donnée dans le canal de réception.

- **Condition d'attente.** L'indice récepteur u attend que l'indice émetteur $v = A$ ait exécuté toutes les instructions send *précédant* le receive au sens de l'ordre logique (définition 2.2.1).
- **Sélection de la donnée reçue.** Après que la condition d'attente soit satisfaite, l'indice u examine le canal (v, u, X). Les données du canal sont ordonnées, via les horloges structurelles associées à chaque donnée, par ordre strictement croissant par rapport à l'ordre structurel. En effet, l'indice v, seul indice pouvant ajouter des données dans le canal, exécute toujours les instructions dans un ordre croissant. A partir de l'horloge du récepteur u on divise la liste des données du canal en trois sous listes BE, NC et AF. La liste BE correspond aux données envoyées par v par des instructions send *précédant* la position actuelle de u. La liste NC correspond aux données envoyées par des instructions send *incomparables* à la position de u, et AF aux données envoyées par des instructions *suivant* la position de u (Fig. 5.1).

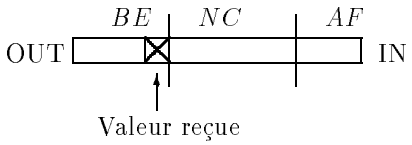


FIG. 5.1 - *Répartition des données dans un canal*

Nous distinguons les deux cas possibles:

- ▷ Si l'ensemble BE est vide alors l'indice u reçoit la valeur spéciale `nil` dans Y.
- ▷ Dans le cas contraire, l'indice u reçoit la valeur la plus *récente* de BE et l'affecte à Y. Toutes les données appartenant à BE sont alors supprimées.

La condition d'attente garantit que l'ensemble BE ne peut plus être modifié par l'indice v. Cela signifie que toutes les données envoyées par v et qui doivent être lues à la position de u dans le programme ont été reçues. Après que la condition soit vérifiée, les valeurs envoyées v sont stockées dans les ensembles NC ou AF.

Remarques.

- La réception de `nil` lors de l'exécution d'un receive signifie qu'aucun envoi ne lui correspond.

- Les canaux sont gérés comme des listes LIFO vis-à-vis de l'ordre structurel dans le sens où la valeur reçue est la dernière envoyée par un send *précédant* la position courante du récepteur.
- Le where/elsewhere de $\mathcal{SCL}-Chan$ remplit le même rôle que celui de $\mathcal{SCP}$. Les deux branches sont incomparables dans l'ordre structurel. Cela interdit toute communication entre les branches d'un même where/elsewhere. Le where/elsewhere exprime du parallélisme disjoint au sens d'Apt et Olderog [1]. Les librairies classiques à passage de messages n'intègrent pas de structure pour l'expression de parallélisme disjoint.
- Comme en $\mathcal{SCP}$, la réception est unilatérale. Elle peut être effectuée sans qu'une émission lui corresponde (absence d'interblocage). De même plusieurs émissions peuvent être exécutées sans qu'il existe des réceptions leur correspondant. Il n'y a aucune correspondance obligatoire entre envois et réceptions contrairement aux librairies classiques de passage de message telles que MPI [61]. Cette fonctionnalité facilite l'expression de schémas de dépendances irréguliers (voir les exemples de la section 5.3.2).
- Contrairement à $\mathcal{SCP}$, $\mathcal{SCL}-Chan$ consomme les données accédées à distance. Une valeur envoyée est soit reçue une seule fois, soit détruite sans avoir été reçue. Ce choix va dans le sens d'une gestion plus économique de l'espace mémoire que celle de $\mathcal{SCP}$. D'autre part, l'idée intuitive qui justifie ce choix est qu'une valeur supprimée est une valeur périmée. Si une réception a lieu sur une variable dont la valeur à été deux fois envoyée alors la valeur reçue est la dernière. La première valeur est caduque, elle est donc supprimée. Par contre si deux réceptions sur une variable ont lieu, alors qu'il n'y a pas d'envoi entre les deux, la seconde valeur reçue est `nil`. Un mécanisme de sauvegarde local des valeurs déjà reçues permet dans ce cas de réutiliser la valeur transmise lors de la première réception. A condition de gérer un tel mécanisme, nous ne perdons pas la possibilité de faire plusieurs accès distants sur une même valeur comme il est possible de le faire en $\mathcal{SCP}$. Cette remarque est fondamentale pour la traduction de programme séquentiel en programme $\mathcal{SCL}-Chan$.

5.1.2 Construction des horloges

Le modèle d'exécution asynchrone de $\mathcal{SCL}-Chan$ est basé sur l'utilisation des horloges. Leur construction est semblable à celle de $\mathcal{SCP}$. Le langage $\mathcal{SCL}-Chan$ abandonne la structure en groupes d'instructions de $\mathcal{SCL}$. Les incréments d'horloges ne comptent plus des groupes d'instructions, mais des envois de messages. Le send est la structure $\mathcal{SCL}-Chan$ qui correspond à la mise à jour des valeurs distantes. Les instructions *significatives* pour coder la position des indices vis-à-vis de ces envois sont les instructions d'envoi elles-mêmes (send) et les structures de contrôle susceptibles de les englober(where/elsewhere, where, loopwhere et forwhere). A partir de ces instructions, la construction des horloges structurelles suit la méthode énoncée en (2.2.1) page 21.

send. Après l'exécution d'un send, le compteur terminal de l'horloge courante est incrémenté.

where/elsewhere. En début de conditionnelle on ajoute la paire $(1,0)$ à l'horloge courante si on vérifie la condition, sinon on ajoute $(2,0)$. En sortie de conditionnelle, on dépile un niveau d'horloge et on incrémente le nouveau compteur terminal de 1.

loopwhere. Un indice qui débute l'exécution d'un loopwhere ajoute une paire $(1,0)$ en sommet d'horloge s'il vérifie la condition. Il n'empile plus de niveau au début des itérations suivantes. Dès qu'un indice sort de la boucle, il dépile le niveau empilé en entrée et incrémente de 1 son compteur terminal. Un indice qui n'exécute aucune itération incrémente directement de 1 son compteur terminal.

5.1.3 Exemple

Reprenons l'exemple de pipeline entre deux indices, codé en $\mathcal{SCP}$ (voir 2.2.3, page 26). Il donne un exemple de la manière dont les horloges sont gérées en $\mathcal{SCL} - \mathcal{C}han$. A la droite de chaque instruction nous donnons l'horloge de l'indice quand il a fini de l'exécuter. La valeur i correspond au numéro de l'itération dans la boucle. En utilisant la structure where, nous répartissons une tâche spécifique sur chaque indice. Le premier indice calcule S depuis ses données locales puis envoie sa valeur vers l'indice 2. L'indice 2 la reçoit et l'additionne à $A[i]$. Chaque tâche est répétée 100 fois.

Comme dans l'exécution du programme $\mathcal{SCP}$, l'indice 1 ne génère aucune attente. Pour chaque itération i il empile dans le canal $(1,2,S)$ le résultat des affectations de S. Quand la donnée S à l'itération i est envoyée dans le canal, l'horloge $(0,1)(1,2i)(1,0)$ lui est associée. Ensuite, l'indice 1 incrémente son horloge à $(0,1)(1,2i)(1,1)$. Lorsque l'indice 2 exécute l'instruction receive de l'itération i, il possède l'horloge $(0,1)(1,2i+1)(1,0)$. Pour commencer, il attend que l'index 1 ait une horloge qui ne soit plus strictement inférieure à la sienne, i.e. que 1 ait commencé l'exécution du second where de l'itération i. Alors, il affecte la dernière valeur stockée dans la canal $(1,2,S)$ au regard de sa propre horloge et de l'ordre structurel. Cette valeur correspond à celle estampillée par l'horloge $(0,1)(1,2i)(1,0)$. Notons que contrairement à la gestion de la mémoire en $\mathcal{SCL}$, il n'existe pas de donnée plus ancienne dans le canal puisque chaque valeur envoyée est reçue et consommée

$Owner(S) = 1$	
$Owner(Z) = 1$	
$Owner(A) = 2$	
$Owner(R) = 2$	
forwhere $i := 0$ to $i < 100$ do	$(0,0)(1,2i)$
where $This = 1$ do	$(0,1)(1,2i)(1,0)$
$S := i * Z[i];$	$(0,1)(1,2i)(1,0)$
send S to $This + 1;$	$(0,1)(1,2i)(1,1)$
end;	$(0,1)(1,2i+1)$
where $This = 2$ do	$(0,1)(1,2i+1)(1,0)$
receive S from $This - 1$ into $R;$	$(0,1)(1,2i+1)(1,0)$
$A[i] := A[i] + R;$	$(0,1)(1,2i+1)(1,0)$
end;	$(0,1)(1,2i+2)$
end;	$(0,2)$

5.2 Propriétés fondamentales

Dans cette section, nous énonçons sous la forme de théorèmes les propriétés fondamentales de $\mathcal{SCL} - Chan$. Les preuves peuvent être trouvées dans le rapport de recherche [48] et dans [55].

5.2.1 Absence de blocage

Le théorème suivant exprime l'absence de blocage des calculs des programmes $\mathcal{SCL} - Chan$.

Théorème 5.2.1 *Soit un programme $\mathcal{SCL} - Chan$ S et un vecteur d'horloges structurelles T toutes initialisées à la même valeur. Au cours de tout calcul, s'il existe un état pour lequel un indice v n'a pas terminé son calcul, alors il existe aussi une transition à partir de cet état.*

5.2.2 Déterminisme

Le théorème suivant exprime le déterminisme de tous les calculs des programmes $\mathcal{SCL} - Chan$. Il repose sur la notion d'équivalence entre des calculs. Deux calculs sont équivalents si, partant de mémoires privées identiques, soit ils terminent en donnant le même résultat final, soit les deux ne terminent pas.

Théorème 5.2.2 (Déterminisme) *$\mathcal{SCL} - Chan$ est déterministe: pour tout programme $\mathcal{SCL} - Chan$ et pour un état initial donné, tous les calculs sont équivalents.*

5.2.3 Calculs synchrones

Le modèle d'exécution de $\mathcal{SCL} - \mathcal{C}han$ permet de définir un modèle de programmation synchrone, c'est-à-dire qu'une lecture séquentielle et dirigée par la syntaxe du programme est toujours possible. L'idée centrale est qu'une exécution synchrone est une exécution asynchrone particulière.

Le déterminisme des exécutions asynchrones garantit que toutes les calculs sont équivalents du point de vue du résultat final. Par conséquent le programmeur peut créer ses programmes dans le cadre synchrone alors que le modèle d'exécution est faiblement synchrone.

Dans le modèle synchrone, chaque instruction est exécutée de façon synchrone par tous les indices actifs. Comme tous les indices ont la même position la vision des communications par canaux est simplifiée. On considère que chaque canal ne contient qu'une seule valeur. L'envoi d'une valeur dans un canal écrase la valeur contenue dans ce canal. La réception d'une valeur dans un canal récupère la valeur contenue dans le canal puis la remplace par la valeur nil. La seule difficulté pour faire correspondre le modèle de programmation synchrone avec le modèle d'exécution asynchrone consiste à conserver l'indépendance entre les deux branches d'un where/elsewhere. Pour cela, dans le modèle de programmation synchrone, on séquentialise l'exécution des deux branches. On masque les communications entre des branches différentes par l'utilisation de canaux temporaires. A la fin de l'exécution du where/elsewhere, les canaux sont recomposés à partir des valeurs des canaux avant l'exécution du where/elsewhere et de celles des canaux temporaires.

5.2.4 Equivalence fondamentale

Le théorème d'équivalence exprime que l'exécution synchrone d'un programme $\mathcal{SCL} - \mathcal{C}han$ conduit au même résultat que toutes les exécutions asynchrones possibles.

Théorème 5.2.3 (Equivalence sémantique) *Soit une exécution synchrone E_s partant d'un programme S, tous les indices étant actifs et tous les canaux étant initialisés avec* nil. *Soit une exécution asynchrone E_a telle que tous les indices partent du même programme S, de la même horloge structurelle, avec des canaux $LIFO$ vides et avec la même mémoire privée que dans le cas synchrone. Alors, l'exécution synchrone E_s et l'exécution asynchrone E_a sont équivalentes.*

La preuve est basée sur la définition de sémantiques formelles. On définit une sémantique opérationnelle formalisant le modèle d'exécution de $\mathcal{SCL} - \mathcal{C}han$. Une sémantique dénotationnelle formalise le modèle de programmation synchrone. Une preuve par induction sur la structure du programme permet de faire correspondre les résultats des deux sémantiques.

5.3 La fonction de traduction en $\mathcal{SCL}-Chan$

Une solution au problème du surcoût induit par la gestion de la mémoire en $\mathcal{SCP}$ consiste à remplacer les accès en mémoire distante par un mécanisme de passage de messages. Au lieu de stocker l'ensemble de sa mémoire en prévision d'éventuels accès, un indice se contente d'envoyer les valeurs nécessaires aux indices distants. Dans ce cas le stock inutile de mémoire est considérablement réduit. Par contre cela suppose une information supplémentaire à la disposition du processus de traduction. Il faut connaître toutes les communications éventuelles. C'est l'approche de la traduction avec $\mathcal{SCL}-Chan$.

Comme en $\mathcal{SCP}$, la distribution des données sur les indices est exprimée par le biais de la fonction $Owner$. L'expression $Owner(V)$ note l'indice possédant la variable V. Les composantes de V sur les autres indices contiennent des copies de la composante $V_{Owner(V)}$ qui ne sont pas nécessairement à jour. La définition de $Owner$ s'étend naturellement aux ensembles de variables. Ainsi, si Q est un ensemble de variables, $Owner(Q)$ note l'ensemble des indices possédant au moins une variable de Q.

Soit un programme S. Nous notons $Concern(S)$ un surensemble de l'ensemble des variables qui apparaissent à gauche des affectations dans S. L'ensemble $Concern(S)$ contient les variables de S *susceptibles* d'être affectées, nous les désignons comme les indices *concernés* par S. Si Exp est une expression apparaissant dans un programme, nous notons $Ref(Exp)$ un surensemble de l'ensemble des variables qui doivent être référencées par un indice pour évaluer Exp. Les variables appartenant à $Ref(Exp)$ sont celles *susceptibles* d'être référencées par l'indice évaluant Exp. La définition de Ref s'étend naturellement des expressions aux programmes: $Ref(S)$ note un surensemble de l'ensemble des variables qui sont accédées en lecture dans le programme S.

Exemple 5.3.1

```
for i = 1 to N do
  A[i+1] = A[i-1];
  if A[i+1] ≠ 0 do
    A[i] = A[i-1] * A[i+1];
  end
end
```

$= S$

Considérons l'exemple 5.3.1. A est un tableau mono-dimensionnel. Le programme itère N fois le code S. Pour chaque itération, nous pouvons choisir: $Concern(S) = \{A[i], A[i+1]\}$ et $Ref(S) = \{A[i-1], A[i+1]\}$.

Par souci de concision, nous introduisons de nouvelles instructions SPMD. Nous définissons l'instruction **msend** Q **to** T où Q et T sont des ensembles de variables. Un indice exécutant une instruction **msend**, envoie toutes les valeurs des variables lui appartenant et appartenant à Q vers tous les indices possédant au moins une variable appartenant à T. Nous définissons de même l'instruction **mreceive** Q où Q est un ensemble de variables. Un indice exécutant cette instruction reçoit la valeur de chaque variable $V \in Q$ depuis l'indice $Owner(V)$ et la stocke dans sa propre composante de V. Si une valeur `nil` est reçue, alors la composante locale de V n'est pas mise à jour. Ce choix correspond à l'idée intuitive

suivante: si un indice u reçoit la valeur `nil`, cela signifie que la valeur de V n'a pas été mise à jour par l'indice $Owner(V)$ depuis la dernière communication. Dans ce cas la valeur locale courante de V correspond à la valeur distante.

msend Q to T end	mreceive Q

foreach $V \in Q$ do
  foreach $W \in T$ do
  where $Owner(V) = This$
  $and\ Owner(W) \neq This$ do
  send V to $Owner(W)$;
  end;
  end;
end;
	foreach $V \in Q$ do
	  where $Owner(V) \neq This$ do
	  receive V from $Owner(V)$ into Tmp;
	  where $Tmp \neq$ `nil` do
	  $V := Tmp$;
	  end;
	  end;
	end;

Passons à présent à la distribution de programmes séquentiels. Elle reprend le même schéma que la traduction en $\mathcal{SCP}$. Par contre en $\mathcal{SCL} - \mathcal{C}han$ les communications et les gardes sont explicites. Il revient à la traduction de les gérer. Pour chaque affectation les gardes sont ajoutées. Pour chaque affectation ou structure de contrôle les communications nécessaires sont ajoutées.

5.3.1 La traduction d'une affectation

Nous explicitons la traduction d'une affectation dans un élément de tableau $X[Exp_1] = Exp_2$. Le cas d'une variable simple peut en être aisément déduit. En accord avec la règle des écritures locales, l'affectation est réalisée par l'indice $u = Owner(X[Exp_1])$. La gestion des communications et gardes est décomposée en trois étapes (voir plus bas) correspondant à trois phases de communications.

Etape 1 La première étape de la traduction consiste à envoyer les valeurs nécessaires à l'évaluation des expressions Exp_1 et Exp_2.

Chaque indice possédant une variable de $Ref(Exp_2)$ envoie sa valeur à tous les indices susceptibles d'affecter la valeur de Exp_2. Le surensemble des variables lues est donné par l'ensemble $Ref(Exp_2)$. Le surensemble des indices susceptibles d'affecter la valeur de Exp_2 est donné par l'ensemble des indices possédant des variables apparaissant dans $Concern(X[Exp_1] = Exp_2)$.

Chaque indice possédant une variable de $Ref(Exp_1)$ envoie sa valeur à l'indice désigné responsable de l'évaluation de Exp_1. Nous définissons une fonction $Resp$ de l'ensemble des variables vers les indices qui permet de choisir cet indice. Pour un ensemble Q de variables , $Resp(Q)$ choisit une variable $V \in Q$ et retourne $v = Owner(V)$. Pour un ensemble de variables Q fixé, $Resp(Q)$ retourne toujours le même indice. Donc chaque indice possédant une variable éventuellement accédée en lecture $V \in Ref(Exp_1)$ l'envoie vers $v = Resp(Concern(X[Exp_1] = Exp_2))$.

Etape 2 La deuxième étape concerne l'évaluation par l'indice v de l'indice de la variable modifiée. Une garde assure que seul l'indice v exécute cette étape. Il reçoit les valeurs envoyées à la première étape et évalue localement l'expression Exp_1. Il devient alors possible d'évaluer l'indice responsable de l'affectation de la partie gauche de l'affectation: $u = Owner(X[Exp_1])$. Enfin, la valeur de l'expression Exp_1 est envoyée vers l'indice u.

Etape 3 C'est au cours de la troisième étape que l'affectation proprement dite est réalisée. Une garde assure que seuls les indices susceptibles de posséder la variable affectée exécutent cette étape. Cette sélection est assurée par la condition: $This \in Owner(Concern(X[Exp_1] = Exp_2))$.

Tous les indices exécutant cette étape tentent une réception sur la valeur de l'expression Exp_1 envoyée à l'étape 2. Un seul de ces indices reçoit cette valeur. Il s'agit de l'indice qui est responsable de l'affectation. Les autres reçoivent la valeur nil.

L'indice responsable de l'affectation reçoit les valeurs nécessaires pour évaluer Exp_2 qui ont été préalablement envoyées à l'étape 1. Il est alors à même d'effectuer l'affectation.

$Trans($ $X[Exp_1] = Exp_2$ $)$	*Etape 1*	msend $Ref(Exp_2)$ to $Concern(X[Exp_1] = Exp_2)$; msend $Ref(Exp_1)$ to $Resp(Concern(X[Exp_1] = Exp_2))$;
	Etape 2	where $This = Resp(Concern(X[Exp_1] = Exp_2))$ do mreceive $Ref(Exp_1)$; $Tmp = Exp_1$; send Tmp to $Owner(X[Tmp])$; end;
	Etape 3	where $This \in Owner(Concern(X[Exp_1] = Exp_2))$ do receive Tmp from $Resp(Concern(X[Exp_1] = Exp_2))$ *into* Tmp; where $Tmp \neq$ nil do mreceive $Ref(Exp_2)$; $X[Tmp] = Exp_2$; end; end

Traduction d'une affectation

Remarque. Si Exp_1 est une fonction des compteurs de boucles, il est possible de déterminer localement quel élément d'un tableau doit être affecté. Alors, $Owner(Exp_1)$ est aussi une fonction des compteurs de boucles et il est judicieux de choisir $Resp$ tel que $Resp(Concern(X[Exp_1] := Exp_2)) = Owner(X[Exp_1])$. La troisième étape peut alors être modifiée de façon à ce que les indices ne reçoivent plus Tmp mais évaluent localement Exp_1 afin de déterminer si ils doivent effectuer l'affectation.

5.3.2 La traduction d'une conditionnelle

La traduction de if Exp do S end suit un schéma similaire à celui de l'affectation. La gestion des communications et gardes est décomposée suivant trois étapes similaires.

Etape 1 La première étape de la traduction consiste à envoyer les valeurs nécessaires à l'évaluation des expressions référencées en lecture dans les expressions appartenant à S et dans Exp.

Chaque indice possédant une variable de $Ref(S)$ envoie sa valeur à tous les indices possédant une variable éventuellement affectée au cours de l'exécution de S. Le surensemble des variables lues est donné par l'ensemble $Ref(S)$. Le surensemble des indices possédant une variable éventuellement affectée au cours de l'exécution de S est donné par l'ensemble $Concern(S)$.

Chaque indice possédant une variable de $Ref(Exp)$ envoie sa valeur à l'indice désigné responsable de l'évaluation de Exp. Cet indice est désigné via la fonction $Resp$. Donc chaque indice possédant une variable éventuellement accédée en lecture $V \in Ref(Exp)$ l'envoie vers $v = Resp(Concern(S))$.

Etape 2 La deuxième étape concerne l'évaluation par l'indice v de la condition Exp. Une garde assure que seul l'indice v exécute cette étape. Il reçoit les valeurs envoyées à la première étape et évalue localement l'expression Exp. Le résultat de cette évaluation est alors envoyé à tous les indices possédant une variable éventuellement affectée au cours de l'exécution de S.

Etape 3 C'est au cours de la troisième étape que le résultat de l'évaluation de Exp est utilisé pour localement décider l'exécution du code conditionné. Une garde assure que seuls les indices susceptibles de posséder une variable affectée au cours de l'exécution de S exécutent cette étape. Cette sélection est assurée par la condition: $This \in Owner(Concern(S))$.

Tous les indices exécutant cette étape réceptionnent la valeur de l'expression Exp envoyée à l'étape 2. Si la valeur autorise l'exécution du code conditionné tous les indices l'exécutent, sinon aucun ne le fait. La suite de la traduction est enchaînée récursivement.

$Trans($ if Exp do S end $)$	$Etape\ 1$	msend $Ref(S)$ to $Concern(S)$; msend $Ref(Exp)$ to $Resp(Concern(S))$;
	$Etape\ 2$	where $This = Resp(Concern(S))$ do mreceive $Ref(Exp)$; $Tmp = Exp$; msend $\{Tmp\}$ to $Concern(S)$; end;
	$Etape\ 3$	where $This \in Owner(Concern(S))$ do receive Tmp from $Resp(Concern(S))$ into Tmp; where Tmp do; $Trans(S)$; end; end
Traduction d'une conditionnelle		

Remarque. Si un seul indice doit exécuter S, la seconde étape devient inutile. L'évaluation de Exp peut alors être effectuée au cours de la troisième étape. En outre, les instructions msend de la première étape peuvent être fusionnées en un unique msend.

5.3.3 La traduction d'une boucle

La traduction de while Exp do S end, où S représente un code séquentiel, suit un schéma similaire à celui de la conditionnelle. La gestion des communications et gardes est décomposée suivant six étapes.

Etape 1 La première étape de la traduction consiste à envoyer les valeurs nécessaires à l'évaluation des expressions référencées en lecture dans les expressions appartenant à S et dans Exp.

Chaque indice possédant une variable éventuellement accédée en lecture au cours de l'exécution de S envoie sa valeur à tous les indices possédant une variable éventuellement affectée au cours de l'exécution de S. Le surensemble des variables lues est donné par l'ensemble $Ref(S)$. Le surensemble des indices possédant une variable éventuellement affectée au cours de l'exécution de S est donné par l'ensemble $Concern(S)$.

Chaque indice possédant une variable éventuellement accédée en lecture au cours de l'évaluation de la condition de terminaison Exp envoie sa valeur à l'indice désigné responsable de l'évaluation de Exp. Cet indice est désigné via la fonction $Resp$. Donc chaque indice possédant une variable éventuellement accédée en lecture $V \in Ref(Exp)$ l'envoie vers $v = Resp(Concern(S))$.

Etape 2 La deuxième étape concerne l'évaluation par l'indice v de la condition Exp. Une garde assure que seul l'indice v exécute cette étape. Il reçoit les valeurs envoyées à la

première étape et évalue localement l'expression Exp. Le résultat de cette évaluation est alors envoyé à tous les indices possédant une variable éventuellement affectée au cours de l'exécution de S.

Etape 3 C'est au cours de la troisième étape que le résultat de l'évaluation de Exp est utilisé pour localement décider si l'exécution du code à insérer le sera au moins une fois. Une garde assure que seuls les indices susceptibles de posséder une variable affectée au cours de l'exécution de S exécutent cette étape. Cette sélection est assurée par la condition: $This \in Owner(Concern(S))$.

Tous les indices exécutant cette étape réceptionnent la valeur de l'expression Exp envoyée à l'étape 2. Si la valeur autorise le calcul de la première itération tous les indices l'exécutent, sinon aucun ne le fait. La suite de la traduction est enchaînée récursivement.

Etape 4 Cette étape interne aux itérations est similaire à la première étape. Elle en reprend totalement la forme mais seulement une partie de la fin. Les indices qui ont exécutés une itération S et qui font partie de ceux qui possèdent des variables éventuellement référencées dans S ou dans Ref communiquent à nouveau leurs valeurs.

Etape 5 Cette étape interne aux itérations est similaire à la deuxième étape. Comme celle-ci elle concerne l'évaluation de la condition Exp. Cette évaluation tient compte des nouvelles valeurs calculées au cours de l'itération précédente.

Etape 6 Tous les indices exécutant cette étape réceptionnent la valeur de l'expression Exp envoyée à l'étape 5. Si la valeur autorise le calcul de l'itération suivante, tous les indices l'exécutent, sinon aucun ne le fait.

$Trans($ while Exp do S end)	*Etape 1* msend $Ref(S)$ to $Concern(S)$; msend $Ref(Exp)$ to $Resp(Concern(S))$; *Etape 2* where $This = Resp(Concern(S))$ do mreceive $Ref(Exp)$; $Tmp = Exp$; msend $\{Tmp\}$ to $Concern(S)$; end; *Etape 3* where $This \in Owner(Concern(S))$ do receive Tmp from $Resp(Concern(S))$ *into* Tmp; loopwhere tmp do $Trans(S)$; *Etape 4* msend $Ref(S)$ to $Concern(S)$; msend $Ref(Exp)$ to $Resp(Concern(S))$; *Etape 5* where $This = Resp(Concern(S))$ do mreceive $Ref(Exp)$; $Tmp = Exp$; msend $\{Tmp\}$ to $Concern(S)$; end; *Etape 6* receive Tmp from $Resp(Concern(S))$ *into* Tmp; end; end
Traduction d'une boucle while	

Remarque. Les indices non concernés par les itérations de S n'exécutent pas l'étape 3. Ils ne remettent pas à jour leur mémoire ils n'ont donc pas besoin de diffuser à chaque itération une information inchangée.

La traduction de for $i = B_{inf} to B_{sup}$ do S end, où S représente un code séquentiel, suit le même principe. Elle passe par l'utilisation de la boucle forwhere. La traduction introduit une variable vectorielle $I \in Ind$ variant entre B_{inf} et B_{sup}. Puisque le nombre d'itérations est connu avant la première itération, la traduction évalue les bornes une seule fois au début de la boucle.

$Trans($ for $i = B_{inf} to B_{sup}$ do S end)	*Etape 1* msend $Ref(S)$ to $Concern(S)$; msend $Ref(B_{inf}) \cup Ref(B_{sup})$ to $Resp(Concern(S))$; *Etape 2* where $This = Resp(Concern(S))$ do mreceive $Ref(B_{inf}) \cup Ref(B_{sup})$; $Tmp1 = B_{inf}$; $Tmp2 = B_{sup}$; msend $\{Tmp1, Tmp2\}$ to $Concern(S)$; end; *Etape 3* where $This \in Owner(Concern(S))$ do receive $\{Tmp1\}$ from $Resp(Concern(S))$ *into* $Tmp1$; receive $\{Tmp2\}$ from $Resp(Concern(S))$ *into* $Tmp2$; forwhere $i = Tmp1$ *to* $Tmp2$ do $Trans(S)$; end; end
Traduction d'une boucle for	

Remarques.

- Notons que les envois de variables sont effectués a priori. Les réceptions correspondantes ne sont pas nécessairement exécutées. Par conséquent, comme l'instruction receive est la seule instruction bloquante, ce schéma de traduction économise à l'exécution des synchronisations inutiles. Ne plus rendre systématique la correspondance entre émission et réception constitue l'avantage déterminant de notre schéma de communication et de synchronisation. Le modèle d'exécution de $\mathcal{SCL} - \mathcal{C}han$ garantit que les messages inutiles n'interfèrent pas avec les communications utiles.

- Au contraire des approches classiques de la distribution de code dirigée par les données [3], un indice qui n'a pas à modifier des variables dans un code conditionné par une expression booléenne, ne reçoit pas les valeurs nécessaires à l'évaluation de la condition.

- Notre schéma de traduction tient compte de l'information fournie par la connaissance statique des ensembles $Concern$. Pour chacune de nos traductions, un indice ne faisant pas partie de $Concern$ n'a pas besoin d'exécuter les étapes deux et trois.

- La correction de fonction de traduction peut être prouvée formellement. La preuve repose sur les sémantiques dénotationnelles du langage séquentiel et de $\mathcal{SCL} - \mathcal{C}han$. Nous ne donnons ici que la justification intuitive.

  La correction est prouvée par induction. Comme $\mathcal{SCL} - \mathcal{C}han$ est déterministe, il suffit de considérer les calculs synchrones du programme traduit. Chaque étape de l'induction repose sur l'observation suivante: comme pour tout programme S, l'ensemble $Ref(S)$ (resp. $Concern(S)$) contient toutes les variables accédées en lecture dans S (resp. toutes les variables affectées dans S), toutes les valeurs attendues sont envoyées et reçues.

Exemple 5.3.2 Afin d'illustrer notre schéma de traduction, nous utilisons l'exemple page 125 dont nous avons explicité les ensembles $Ref(S)$ et $Concern(S)$. Le programme ci-dessous correspond à sa traduction en $\mathcal{SCL}-Chan$.

```
for i := 1 to 100 do
  msend {A[i]} to {A[i+1]};
  msend {} to {A[i+1]};
  where This = Owner(A[i+1]) do
    mreceive {};
    Tmp := i+1;
    send (Owner(A[i+1]), Tmp);
  end;
  where This = Owner(A[i+1]) do
    receive (Owner(A[i+1]), Tmp, Tmp);
    where Tmp ≠nil do
      mreceive {A[i]};
      A[Tmp] := A[i];
    end;
  end;
  msend {A[i+1], A[i-1]} to {A[i]};
  msend {A[i+1]} to {A[i]};
  where This = Owner(A[i]) do                    1
    mreceive {A[i+1]};                           2
    Tmp := (A[i+1] ≠ 0);
    msend {Tmp} to {A[i]};
  end;
  where This ∈ {A[i]} do
    receive (Owner(A[i]), Tmp, Tmp);
    where Tmp do
      msend {A[i-1], A[i+1]} to {A[i]};
      msend {} to {A[i]};
      where This = Owner(A[i]) do
        mreceive {};
        Tmp := i;
        send (Owner(A[i]), Tmp);
      end;
      where This = Owner(A[i]) do
        receive (Owner(A[i]), Tmp, Tmp);
        where Tmp ≠nil do
          mreceive {A[i-1], A[i+1]};
          A[Tmp] := A[i-1] * A[i+1];
        end;
      end;
    end;
```

Programme traduit

L'indice $Owner(A[i])$ peut exécuter l'instruction mreceive $\{A[i+1]\}$ (marquée **2**) si et seulement si l'indice $Owner(A[i+1])$ a évalué la conditionnelle (marquée **1**). Cela garantit que l'indice $Owner(A[i])$ reçoit la valeur de $A[i+1]$ mise à jour.

Les instructions soulignées sont inutiles: par exemple les instructions de communications depuis un indice vers lui-même. Elles peuvent être aisément supprimées via une analyse statique à la compilation.

Exemple 5.3.3 Reprenons l'exemple 2.4.4 page 44. Nous lui appliquons la fonction de traduction en $\mathcal{SCL} - \mathcal{C}han$.

Le programme séquentiel

```
for I = 1, N do
  for J = 1, 4 do
    A[J, I] = A[J − 1, I] + A[1, I − 1]
  end
end
```

Le programme traduit en $\mathcal{SCL} - \mathcal{C}han$

```
forwhere I = 1, N do
  forwhere J = 1, 4 do
    msend (A[J − 1, I], A[1, I − 1]) to (A[J, I]);
    where This = Owner(A[J, I]) do
      mreceive (A[J − 1, I], A[1, I − 1]);
      A[J, I] = A[J − 1, I] + A[1, I − 1];
    end
  end
end
```

Notons que les bornes de la boucle for sont connues statiquement, nous avons supprimées les communications liées à leur évaluation. De même, si nous notons S l'affectation $A[J, I] = A[J - 1, I] + A[1, I - 1]$. Comme nous avons $Concern(S) = \{A[J, I]\}$, nous choisissons $Owner(A[J, I])$ pour $Resp(Concern(S))$. Par conséquent, la seconde étape dans la transformation de l'affectation devient inutile. (cf. Remarque 5.3.1 page 127).

Explicitons à présent les instructions msend et mreceive:

```
forwhere I = 1, N do
  forwhere J = 1, 4 do
    where This = Owner(A[J − 1, I]) do
      send A[J − 1, I] to Owner(A[J, I]);
    end
    where This = Owner(A[1, I − 1]) do
      send A[1, I − 1] to Owner(A[J, I]);
    end
    where This = Owner(A[J, I]) do
      receive A[J − 1, I] from Owner(A[J − 1, I]) into A[J − 1, I];
      receive A[1, I − 1] from Owner(A[1, I − 1]) into A[1, I − 1];
      A[J, I] = A[J − 1, I] + A[1, I − 1];
    end
  end
end
```

Considérons les placements que nous avons étudiés pour la traduction de cet exemple en $\mathcal{SCP}$. Le premier placement ($Owner(A[j, i]) = i$) peut paraître séduisant puisque la

communication send $A[J-1,I]$ to $Owner(A[J,I])$; devient alors locale. On obtient le code suivant:

```
forwhere I = 1, N do
  forwhere J = 1, 4 do
    where This = Owner(A[1, I − 1]) do
      send A[1, I − 1] to Owner(A[J, I]);
    end
    where This = Owner(A[J, I]) do
      receive A[1, I − 1] from Owner(A[1, I − 1]) into A[1, I − 1];
      A[J, I] = A[J − 1, I] + A[1, I − 1];
    end
  end
end
```

On supprime une partie des communications, mais comme dans le cas du programme $\mathcal{SCP}$, le mécanisme de synchronisation par horloge impose que l'indice 2 attende que l'indice 1 ait atteint le même point dans l'exécution des itérations, il impose une dépendance entre l'affectation de $A[4,1]$ et celle de $A[1,2]$. Le code est séquentialisé par le mécanisme de synchronisation par horloges structurelles.

Par contre si l'on reprend le second placement décrit page 45, le code obtenu conserve ses communications mais l'exécution est parallèle. Un fois le choix du placement effectué, on peut alors optimiser le code $\mathcal{SCL}-Chan$ en tenant compte de celui-ci.

Cet exemple montre que l'analyse, basée sur les diagrammes de dépendances, qui à été mise en place dans le cadre de $\mathcal{SCP}$ peut être utile pour faire un choix de distribution de données dans le cadre de $\mathcal{SCL}-Chan$.

5.4 Exemple : Gauss-Seidel modifié

Dans cette section nous présentons un algorithme inspiré de la relaxation de Gauss-Seidel. Il présente exactement le même schéma de dépendances. L'algorithme comporte un parcours de chaque élément d'un tableau à deux dimensions depuis la droite vers la gauche et de haut en bas tout en calculant le produit des valeurs des quatre voisins. Pour mettre en évidence l'intérêt des communications unilatérales nous ne modifions la valeur d'un élément que si le résultat est non nul. Cela permet de ne pas propager les valeurs nulles du tableau.

La parallélisation de cet algorithme est un problème intéressant puisqu'il comporte de nombreuses dépendances et anti-dépendances. Nous appliquons la fonction de traduction au programme séquentiel donné page 137. Toutes les références sont directes, elles ne dépendent que des compteurs de boucles. Par conséquent une simple analyse statique permet de calculer Ref (respectivement $Concern$) qui correspondent exactement aux ensembles de variables référencées (respectivement assignées) au cours de l'exécution. Nous notons S l'affectation $A[I,J] := A[I,J-1] * A[I+1,J] * A[I,J+1] * A[I-1,J]$. Comme nous

avons $Concern(S) = \{A[I,J]\}$, nous choisissons $Owner(A[I,J])$ pour $Resp(Concern(S))$. Par conséquent, la seconde étape dans la transformation de l'affectation devient inutile. (cf. Remarque 5.3.1 page 127). Pour la même raison, la seconde étape de la traduction de la conditionnelle devient inutile (cf. Remarque 5.3.2 page 129).

Remarques

- Notons que les tests imbriqués $This = Owner(A[I,J])$ sont redondants.
- La première instruction msend communique toutes les variables nécessaires à l'indice $Owner(A[I,J])$. Comme ces variables ne sont pas modifiées au cours de l'itération courante de la boucle, les autres instructions d'envois (soulignées dans le programme) deviennent inutiles. De plus, comme de part notre schéma de traduction tous les canaux sont vides, les instructions soulignées mreceive ne font rien sinon terminer. L'indice $Owner(A[I,J])$ utilise sa propre composante des variables appartenant aux autres indices. Par conséquent il est possible de supprimer ces instructions.
- Dans l'algorithme séquentiel, les conditionnelles imbriquées pourraient être réduites à une seule par conjonction des tests. Comme dans ce cas toutes les communications doivent être faites avant dévaluer le test, cette transformation interdit l'économie des communications inutiles.

Le programme séquentiel

```
real:A[0:N,0:N]
do I=1,N-1
   do J=1,N-1
     if A[I,J-1]≠ 0 then
       if A[I+1,J]≠ 0 then
         if A[I,J+1]≠ 0 then
           if A[I-1,J] ≠ 0 then
             A[I,J]=A[I,J-1]*A[I+1,J]*A[I,J+1]*A[I-1,J]
           endif
         endif
       endif
     endif
   enddo
enddo
```

La traduction

```
forwhere I := 1 to N − 1 do
  forwhere J := 1 to N − 1 do
    msend(A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]) to Owner(A[I, J]);
    where This = Owner(A[I, J]) do
      mreceive(A[I, J − 1]);
      where A[I, J − 1] ≠ 0 do
        msend(A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]) to Owner(A[I, J]);
        where This = Owner(A[I, J]) do
          mreceive(A[I + 1, J]);
          where A[I + 1, J] ≠ 0 do
            msend(A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]) to Owner(A[I, J]);
            where This = Owner(A[I, J]) do
              mreceive((A[I, J + 1]);
              where A[I, J + 1] ≠ 0 do
                msend(A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]) to Owner(A[I, J]);
                where This = Owner(A[I, J]) do
                  mreceive((A[I − 1, J]);
                  where A[I − 1, J] ≠ 0 do
                    msend(A[I, J − 1], A[I + 1, J], A[I, J + 1], A[I − 1, J]) to Owner(A[I, J]);
                    where This = Owner(A[I, J]) do
                      mreceive(A[I, J − 1], A[I + 1, J], (A[I, J + 1], (A[I − 1, J]);
                      A[I, J] := A[I, J − 1] ∗ A[I + 1, J] ∗ A[I, J + 1] ∗ A[I − 1, J]
                    end
                  end
                end
              end
            end
          end
        end
      end
    end
  end
end
```

5.5 Exemple: factorisation de Cholesky

Dans cette section, nous distribuons l'algorithme classique de la factorisation de Cholesky. Contrairement aux exemples précédents, le programme n'est pas réduit à un nid de boucle parfait. Dans le but d'exploiter le creux, nous introduisons une conditionnelle: la valeur $A[j,k]$ est lue si $A[i,k]$ n'est pas nulle (1) (voir le programme ci-dessous). Comme la valeur $A[i,k]$ n'est pas modifiée dans la boucle la plus interne, le test peut être réalisé à l'extérieur de cette boucle.

Programme séquentiel

```
REAL,SPARSE:: A[N,N]
do i = 1,N
   do k = 1,i-1
     if A[i,k] ≠ 0 then                (1)
       do j = i,N
         A[j,i]=A[j,i]-A[j,k]*A[i,k]      }=S
       enddo
     endif
   enddo
   A[i,i]=sqrt(A[i,i])
   do d = i+1,N
     A[d,i]=A[d,i]/A[i,i]
   enddo
enddo
```

Programme traduit

```
forwhere I := 1 to N + 1 do
   forwhere K := 1 to I − 1 do
     msend (A[I,I],..,A[N+1,I]) to (A[I,I],..,A[N+1,I]);
     msend (A[I,K],..,A[N+1,K]) to (A[I,I],..,A[N+1,I]);
     msend (A[I,K]) to (A[I,I]);
     where This = Owner(A[I,I]) do
       mreceive (A[I,K]);
       Tmp := (A[I,K] ≠ 0);
       msend Tmp to (A[I,I],..,A[N+1,I]);
     end;
     where This ∈ {Owner(A[I,I]),..,Owner(A[N+1,I])} do
       receive Tmp from Owner(A[I,I]);
       where Tmp do
         forwhere J := I to N do
           msend (A[J,I],A[I,K],A[J,K]) to A[J,I];
           where This = Owner(A[J,I]) do
             mreceive (A[J,I],A[J,K],A[I,K]);
             A[J,I] := A[J,I] − A[J,K] * A[I,K]
           end
         end
       end
     end;
     msend (A[I,I]) to Owner(A[I,I]);
     where This = Owner(A[I,I] do
       mreceive (A[I,I]);
       A[I,I] := √A[I,I];
     end;
     forwhere D := I + 1 to N do
       msend (A[D,I],A[I,I]) to Owner(A[D,I]);
       where This = Owner(A[D,I]) do
         mreceive (A[D,I],A[I,I]);
         A[D,I] := A[D,I]/A[I,I]
       end
     end
end
```

Comme dans l'exemple précédent, toutes les variables sont des tableaux dont les indices sont des compteurs de boucle. Nous pouvons alors choisir les sur ensembles Ref et $Concern$ qui correspondent exactement aux variables lues et affectées durant l'exécution (par exemple $Ref(A(I,J) := A(I,J) - A(J,K) * A(I,K)) = \{A(I,J), A(J,K), A(I,K)\}$). Nous avons choisit $Resp(S) = Owner(A[I,I])$.

Remarques.

- Des communications et des calculs sont évités si la valeur communiquée $A[i,k]$ est nulle. Cette possibilité est directement liée à la sémantique des communications de $\mathcal{SCL} - \mathcal{C}han$. Comme les instructions send peuvent être effectuées sans instructions receive correspondantes, il est possible de gérer des conditionnelles avec des réceptions imbriquées dedans.

- Comme dans l'exemple précédent, des optimisations syntaxiques évidentes permettent de supprimer du programme les instructions soulignées.

5.6 Optimisations

Dans cette section nous présentons des optimisations sur les programmes traduits. Nous les introduisons sur le programme inspiré de la relaxation de Gauss-Seidel. Nous montrons d'abord qu'il est possible de réduire le coût en facilitant le recouvrement entre calcul et communication. Ensuite nous prenons en compte le placement pour effectuer des optimisations supplémentaires. Finalement cette section termine par la présentation d'optimisations liées à la gestion des horloges structurelles.

5.6.1 Recouvrement calcul/communication

Il est possible de tirer partie d'une analyse de dépendance très simple pour modifier la place de certaines instructions send pour favoriser le recouvrement de communications par le calcul. Chaque élément de la matrice est modifié une seule fois. Par conséquent chaque donnée peut être envoyée dès qu'elle a été calculée. Pour une itération donnée (I, J), les instructions d'envoi $A[I, J-1]$ et $A[I-1, J]$ sont replacées par deux instructions send respectivement à la fin des itérations $(I, J-1)$ et $(I-1, J)$. Pour les même raisons, comme toutes les modifications des données $A[I, J+1]$ et $A[I+1, J]$ interviennent après qu'elles soient envoyées, les instructions d'envoi sont remplacées par les instructions send au début du programme (page 141).

5.6.2 Optimisations liées au placement

Afin d'implémenter l'algorithme distribué, nous devons résoudre le problème du choix du placement des données, i.e. nous devons distribuer les données sur l'ensemble des processeurs disponibles. Le placement intervient via la fonction *Owner* qui peut être choisie après la génération du code.

Nous notons Pe_Nb le nombre de processeurs disponibles. Nous choisissons un placement consistant en une distribution de la matrice A par blocs de colonnes selon le vecteur de Pe_Nb indices numérotés de 0 à $Pe\_Nb - 1$. Par souci de clarté, nous supposons que N est un multiple de Pe_Nb et nous notons L le nombre $L = N/Pe\_Nb$. E Chaque indice possède $N \times L$ composants. Ce placement correspond à une fonction *Owner* très simple: $Owner(A[I, J]) = (J \; div \; L)$.

Une fois la fonction de placement choisie, il est possible d'ajouter des optimisations additionnelles en tenant compte de la sémantique de la fonction *Owner*.

- **Optimisations sur les communications.** Comme tous les éléments d'une colonne sont possédés par le même indice, toutes les communications à travers cette colonne deviennent des accès locaux. Par conséquent elles peuvent être supprimées (exemple: l'instruction msend $(A[I, J])$ to $Owner(A[I+1, J])$).

- **Optimisations sur le calcul.** Comme la fonction de traduction distribue chaque itération de chaque boucle sur tous les indices, un indice u peut exécuter des itéra-

tions inutiles. Toutefois il est souvent possible de détecter à la compilation de telles itérations par le biais d'une analyse statique.

Nous considérons qu'une itération est *inutile* pour un indice si cet indice n'a rien d'autre à faire dedans sinon exécuter les tests et gérer la progression des horloges. Nous introduisons une nouvelle structure de boucle appelée *jump*. Cette boucle conserve la sémantique de la boucle *for* mais chaque indice u n'effectue que les itérations qui peuvent être *utiles*. Un indice omettant une itération risquerait de ne pas mettre son horloge à jour. Par conséquent, dans le but de conserver la cohérence des horloges, un tel indice doit mettre à jour son horloge locale en tenant compte du nombre d'itérations non exécutées n. Cette opération sur les horloges est réalisée dans le code par la fonction scl_c_inc (n). Comme nous distribuons la matrice par blocs de L colonnes, un indice u doit communiquer avec l'indice $u-1$ (respectivement $u+1$) pour calculer les éléments de la première (respectivement dernière) colonne de son bloc. Pour les colonnes intermédiaires, toutes les données nécessaires sont locales. Par conséquent, le placement induit la distinction de trois codes différents: première iteration, dernière iteration et celles intermédiaires. Afin d'implémenter la boucle *jump* dans le cas d'une distribution par blocs, nous introduisons une macro appelée jump_block. Notons que la définition de *jump* n'est pas liée à cette distribution. Par exemple, il est possible de définir l'équivalent de —jump_block—, pour les distributions cycliques.

```
#define jump_block(jump_1,jump_2,first,last,clock_increment,
{first_iter},{intermediate_iter},{last_iter}){

int j;
scl_c_inc((jump_1)*clock_increment);
j=first;
{first_iter}
for(++j;j<last;j++){
   {intermediate_iter}}
j=last;
{last_iter}
Scl_c_inc((jump_2)*clock_increment);}
```

Où les paramètres ont le rôle suivant:

- `first` et `last` représentent respectivement le nombre de la première et de la dernière itération à exécuter.
- `jump_1` et `jump_2` dénotent respectivement le nombre d'iterations inutiles à éviter avant et après les itérations utiles qui sont de `first` à `last`.
- Le nombre d'incréments d'horloges à l'intérieur d'une itération du for initial correspond à `clock_increment` (il ne tient pas compte du scl_c_inc (N) à l'intérieur du *jump*). L'intuition est la suivante: un indice n'exécutant pas une itération incrémente son horloge de `clock_increment`.

- {first_iter}, {intermediate_iter} et {last_iter} notent les codes correspondant respectivement à la première, aux intermédiaires et à la dernière itération. Notons que les horloges sont incrémentées de la même valeur dans tous ces codes et que cette valeur peut toujours être évaluée à la compilation.

5.6.3 Optimisations liées à la communication des horloges structurelles

Comme les horloges sont incrémentées et diffusées à tous les indices après chaque instruction significative send, loopwhere, where, nous introduisons des communications inutiles. Nous avons développé plusieurs optimisations pour réduire ce surcoût. Comme les horloges sont utilisées uniquement par les indices qui reçoivent, il est suffisant d'envoyer une horloge uniquement une seule fois, entre un send et le receive correspondant, après la dernière mise à jour d'horloge.

Le programme optimisé final est donné ci-dessous. Il est traduit dans une librairie C implémentant le langage $\mathcal{SCL}-\mathcal{C}han$. Nous donnons les principales fonctions de cette librairie en décrivant l'action réalisée par un indice u qui les exécute.

Scl_send (v, X) . Correspond à l'instruction send X to v.

Scl_receive (v, X, Y) . Correspond à l'instruction receive X from v into Y.

Scl_c_send (v) . L'indice u envoi son horloge courante à l'indice v.

Scl_c_send_all . L'indice u envoi son horloge courante à tous les autres indices.

Scl_c_inc . L'indice u incrémente son horloge de 1.

Notons que la gestion des horloges par le programmeur peut conduire à des blocages et des exécutions indéterministes. Cette approche est réservée à la phase d'optimisation.

```
for(i=1;i<N-1;i++){
  if( 0<This && This<Pe_nb){
    Scl_send(This-1,a[i][This*L]);
  }
  Scl_c_inc(1);

  Scl_jump_block(This==0 ? 0 : This*L-1,
  This==Pe_nb-1 ? 0 : (Pe_nb-This-1)*L-1,
  This==0 ? 1 : 0,This==Pe_nb-1 ? L-2 : L-1,1,
    { /*code for the first iteration*/
      scl_c_send(this-1);
      if(a[i-1][j]!=0 && a[i+1][j] !=0
         && a[i][j+1]!=0 )
        {
          Scl_receive(This-1,a[i][j-1],tmp1);
          if(tmp1!=0){
            a[i][j]=a[i-1][j]*a[i+1][j]*
              tmp1*a[i][j+1];
          }
    Scl_c_inc(1);
    },
    { /*code for the intermediate iterations*/
      if(a[i-1][j]!=0 && a[i+1][j] !=0
         && a[i][j-1]!=0  && a[i][j+1]!=0 ){
        a[i][j]=a[i-1][j]*a[i+1][j]
          *a[i][j-1]*a[i][j+1];
      }
      Scl_c_inc(1);
    },
    { /*code for the last iteration*/
      if((a[i-1][j]!=0 && a[i+1][j]!=0)
         && a[i][j-1]!=0)
        {
        Scl_receive(This+1, a[i][j+1],tmp2);
        if(tmp2!=0){
          a[i][j]=a[i-1][j]*a[i+1][j]*a[i][j-1]*tmp2;
        }
        Scl_send(This+1,a[i][j]);
        Scl_c_inc(1);
    }
   )
}
```

5.7 Comparaison entre $\mathcal{SCL}-\mathcal{C}han$ et $\mathcal{SCP}$

Les langages $\mathcal{SCP}$ et $\mathcal{SCL}-\mathcal{C}han$ sont basés sur un modèle de synchronisation commun fondé sur l'ordre logique et les horloges structurelles. Dans les deux langages, la mémoire est gérée sur le mode LIFO: la valeur lue est la plus récente (selon la précédence) mise à disposition par un indice distant. Il est alors possible de faire une traduction où les structures des boucles et conditionnelles correspondent exactement à celles de leurs sources séquentielles, rendant par là même la parallélisation modulaire.

Néanmoins, $\mathcal{SCP}$ et $\mathcal{SCL}-\mathcal{C}han$ ne sont pas destinés à distribuer les mêmes codes séquentiels. En $\mathcal{SCP}$, les communications sont implicites, unilatérales et initialisées par l'indice qui doit faire une lecture à distance. On construit ainsi un mécanisme dynamique d'évaluation des expressions. Son implantation pourrait optimiser les communications en fonction de paramètres seulement connus à l'exécution (suppressions de communications inutiles, regroupement de communications, ...). $\mathcal{SCP}$ introduit un schéma de traduction suffisamment général pour être appliqué à des programmes à schéma de dépendance complexe ou évalué dynamiquement.

Au contraire, en $\mathcal{SCL}-\mathcal{C}han$, les communications sont bilatérales et explicites. Il est alors indispensable de procéder à une analyse statique minimum afin de définir des sur ensembles des indices participant éventuellement à l'évaluation d'une expression. Des sur ensembles des communications nécessaires à la traduction peuvent alors être générés statiquement dans le code.

L'utilisation de gardes implicites en $\mathcal{SCP}$ ne permet pas d'exprimer dans le code les optimisations qu'il est possible de déterminer statiquement sur ces gardes (factorisation de gardes). En $\mathcal{SCL}-\mathcal{C}han$ par contre les gardes sont explicitement dans le code. Ce type d'optimisation est possible.

Le langage $\mathcal{SCL}-\mathcal{C}han$ permet d'exprimer statiquement plus d'optimisations dans le cadre de programmes où une analyse, même partielle, est praticable.

L'intérêt du modèle $\mathcal{SCP}$ est plus global. Il permet d'effectuer un étude simple du coût des synchronisations. Cette étude est ensuite utilisable dans le cadre des programmes $\mathcal{SCL}-\mathcal{C}han$ afin de faire un choix de distribution des données.

5.8 Conclusion

Nous avons montré que le mécanisme de communication et de synchronisation par le biais des horloges structurelles n'est ni cantonné à un langage ni au mode de communication par accès en mémoire distante. Nous avons présenté un langage à passage de messages explicites basé sur cette approche. Dans le cadre de la traduction de code séquentiel en code parallèle fondée sur le modèle de synchronisation structuré par la syntaxe, $\mathcal{SCL}-\mathcal{C}han$ résout le problème de l'expansion de la mémoire en permettant de n'empiler que les données utiles pour le calcul. Cette optimisation de la gestion mémoire passe par l'optimisation des gardes d'émissions qui ne peut être réalisée que lorsqu'il est possible de faire une analyse statique des données qui sont accédées par les indices distants. Néanmoins l'analyse requise

n'a nullement besoin d'être exacte. Nous tirons parti de la possibilité de non correspondance entre les émissions et les réceptions pour montrer qu'il suffit de mettre en évidence un sur ensemble de ces données. Evidemment plus l'analyse est précise, meilleure est l'économie mémoire réalisée.

Les approches classiques de la communication par passage de messages utilisent des canaux FIFO. Bruno Raffin a montré que, dans l'approche $\mathcal{SCL} - \mathcal{C}han$, les deux gestions des canaux sont possibles et qu'elles ont la même expressivité [55]. Avec le mode FIFO, tous les envois doivent être absorbés par des réceptions correspondantes. Au contraire, avec le mode LIFO, il peut y avoir plus d'envois que de réceptions, les envois excédentaires n'étant naturellement pas pris en compte par les réceptions. La gestion LIFO est plus naturelle pour la traduction de code séquentiel par distribution de données. La valeur d'une variable distante correspond à la valeur envoyée la plus récente (selon l'ordre logique) parmi celles qui précèdent la réception. Cela correspond à l'idée intuitive d'écrasement de l'ancienne valeur d'une variable par la nouvelle au cours d'une exécution.

Au contraire des approches classiques de la parallélisation par distribution des données [14, 2, 26, 43], nous tirons parti de la possibilité de non correspondance entre les émissions et les réceptions en épargnant le parcours de structures de contrôle "inutiles" aux indices qui n'ont aucune affectation à y réaliser. Toutefois, cette optimisation n'est effective que s'il est possible de restreindre les envois de messages par une analyse, éventuellement floue, des dépendances de flots.

L'indépendance entre des instructions exécutées par des processeurs différents conduit à des optimisations supplémentaires qui s'expriment simplement en «remontant» les envois et en «descendant» les réceptions au sein du code. Ces optimisations permettent d'augmenter le recouvrement entre calculs et communications.

L'inconvénient majeur de la parallélisation de code séquentiel par distribution de données est le parcours obligatoire de toutes les itérations des boucles par tous les indices afin d'évaluer les gardes. Nous avons montré qu'il est très facile d' éviter de faire parcourir à l'ensemble des indices la totalité des itérations s'il est possible de prédire statiquement, en fonction d'un placement donné, les itérations pour lesquelles un indice n'a aucun calcul à exécuter. On utilise alors les horloges structurelles de façon à simuler le parcours de ces itérations, sans que celles-ci n'aient été réellement exécutées. Ainsi l'ordre logique initial entre les itérations est respecté alors que la structure de la boucle est modifiée.

Chapitre 6

Implantation et tests sur Cray T3D

Dans ce chapitre nous montrons l'intérêt de l'approche sur des exemples classiques tirés de l'algèbre linéaire: la relaxation de Gauss-Siedel et la factorisation de Cholesky. Ces exemples ont été extensivement étudiés dans le cadre des approches classiques de la parallélisation et de la compilation de programmes dataparallèles. La relaxation de Gauss-Siedel est citée comme exemple par Boulet et Brandes [8] pour évaluer les stratégies de parallélisation automatique pour HPF. Dans sa version creuse, la factorisation de Cholesky comporte des imbrications de boucles DOACROSS donc les dépendances sont déterminées par la structure du creux [36]. L'expression de tels schémas de synchronisation est difficile dans le cadre dataparallèle. Nous avons testé ces exemples sur un Cray T3D. CRAFT fournit une librairie d'extensions dataparallèles pour Fortran 77 sur le Cray T3D [17]. Pour les comparaisons, nous avons utilisé la parallélisation générée par le compilateur PAF (*Paralléliseur Automatique pour Fortran*) développé à l'Université de Versailles [21, 22]. Il permet de traduire des programmes Fortran en MP Fortran [44]. Les programmes MP Fortran ne peuvent être directement compilés pour le Cray T3D. Nous avons traduit les programmes MP Fortran en programmes CRAFT. Il est alors possible de comparer les performances de notre approche avec celles des approches classiques.

Dans une première partie nous présentons l'implantation de $\mathcal{SCL}-\mathcal{C}han$ sur le Cray T3D. Nous avons utilisé la librairie décrite dans la section 5.6.3 page 141 implantée par Bruno Raffin [55]. La librairie $\mathcal{SCL}-\mathcal{C}han$ constitue un outil simple, efficace et compact (moins de 1000 lignes), pour tester l'approche de la synchronisation et de la communication utilisant les horloges structurelles. Elle permet d'écrire des programmes C [39] augmentés des fonctions de gestion des horloges structurelles.

Dans une seconde partie nous présentons les résultats de l'exécution du programme $\mathcal{SCL}-\mathcal{C}han$ de relaxation de Gauss-Siedel.

La troisième partie donne les résultats de l'exécution du programme $\mathcal{SCL}-\mathcal{C}han$ de factorisation de Cholesky. La parallélisation d'une version non creuse de l'algorithme permet la comparaison avec l'approche mise en oeuvre dans le compilateur PAF [42].

Pour chacun des exemples nous évaluons l'efficacité du code en fonction du nombre de processeurs et du degré de creux des données.

6.1 Implantation sur Cray T3D

Nous avons testé nos programmes sur le Cray T3D de l'IDRIS («Institut du Développement et des Ressources en Informatique Scientifique», Orsay) qui dispose de 128 processeurs parmi lesquels 64 étaient à notre disposition pour les tests. Dans cette section nous décrivons le Cray T3D ainsi que l'implantation de notre librairie.

6.1.1 Le T3D

Le Cray T3D est une machine à mémoire répartie équipée de processeurs Alpha cadencés à 150 Mhz. L'adressage de la mémoire est global. Il offre des moyens performants pour la gestion des communications unilatérales.

L'essentiel des fonctionnalités de bas niveau du Cray peuvent être accédées par des appels à la librairie SHMEM (SHared MEMory library). Elle exploite au mieux l'adressage global et spécificités matérielles du T3D dans les échanges de messages. En particulier, les communications sont de simples transferts de mémoire asynchrones entre les processeurs. Ils offrent de bien meilleurs performances que les communications basées sur des librairies de passage de message telle que MPI [13] qui est, du reste, implantée sur SHMEM. Par exemple l'instruction d'écriture distante `shmem_put` a un temps de latence proche de $1\mu s$ et une bande passante de 120 MB/s.

Il est possible de programmer le T3D dans le style dataparallèle en utilisant la librairie CRAFT [17]. Cette librairie est appelée depuis les programme Fortran 77. Elle intègre des directives de distribution des données et du travail, une interprétation dataparallèle des tableaux Fortran 90 et des facilités concernant les entrées-sorties dans un environnement multi processeurs. La librairie CRAFT est optimisée pour le Cray T3D: elle est construite au dessus de SHMEM. Le portage de code MP Fortran vers CRAFT est, la plupart du temps, immédiat.

6.1.2 Implantation de la librairie $\mathcal{SCL} - \mathcal{C}han$

L'implantation de la librairie $\mathcal{SCL} - \mathcal{C}han$ est principalement basée sur l'instruction de communication `shmem_put` de la librairie SHMEM. Elle autorise l'écriture directe dans la mémoire d'un processeur distant sans que ce dernier n'en soit informé. Lors de la transmission des horloges, chaque processeur stocke une copie des horloges des processeurs distants dans un emplacement mémoire réservé. Tout nouvel envoi écrase l'ancienne horloge sans que le processeur distant en soit averti. Une horloge est toujours écrasée par une autre plus grande. La vérification de la condition d'attente par un indice donné n'a besoin que de l'horloge la plus grande qu'il connaisse. Notons que ce protocole implique qu'une horloge envoyée après une autre doit arriver après cette dernière. L'ordre des messages d'envoi d'horloges et de données doit être conservé à la réception. Cette propriété est garantie par les protocole de routages du Cray T3D. Le protocole de communication garantit l'absence de perte de donnée ainsi que l'absence de blocage bien que la taille des canaux soit finie si celle-ci est strictement supérieure à deux [55].

6.2 Relaxation de Gauss-Seidel

Nous présentons à présent les expérimentations avec la version parallèle de l'algorithme de relaxation de Gauss-Seidel optimisé (voir section 5.6.3 page 141).

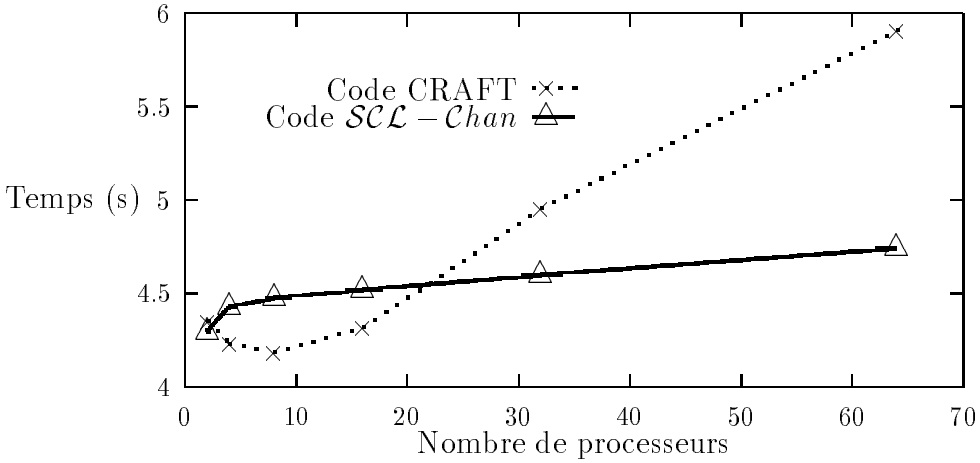


FIG. 6.1 - *Gauss-Seidel sans valeur nulles.*

6.2.1 Tests sans gestion du creux

Pour mettre en évidence les variations du surcoût engendrées par les communications, nous présentons dans un premier temps les résultats sans données creuses mais en faisant varier le nombre d'indices. (Fig. 6.1). Chaque indice possède le même nombre de coefficients (1000×2048 coefficients): lorsque l'on augmente le nombre de processeurs, la taille de la matrice augmente. Pour un faible nombre de processeurs, notre programme parallèle atteint des performances similaires à celles du programme CRAFT généré à partir de PAF. Toutefois ce dernier semble moins extensible: le temps de calcul augmente fortement avec le nombre de processeurs.

6.2.2 Tests avec gestion du creux

Nous avons testé notre programme sur 32 processeurs avec différents taux de valeurs nulles et avec toujours le même nombre d'éléments par processeur (Fig. 6.2). L'amélioration est déjà significative avec 20% de creux. Elle croît quasi linéairement avec le creux. De coûteuses phases d'attente sont évitées puisque nous n'exécutons pas les instructions receive inutiles.

Le paralléliseur PAF ne gère pas les structures de contrôle dynamiques. La version de Gauss-Seidel parallélisée par PAF est celle sans conditionnelle. Elle ne tient pas compte du degré de creux. Le programme CRAFT généré à partir de PAF donne un temps d'exécution constant.

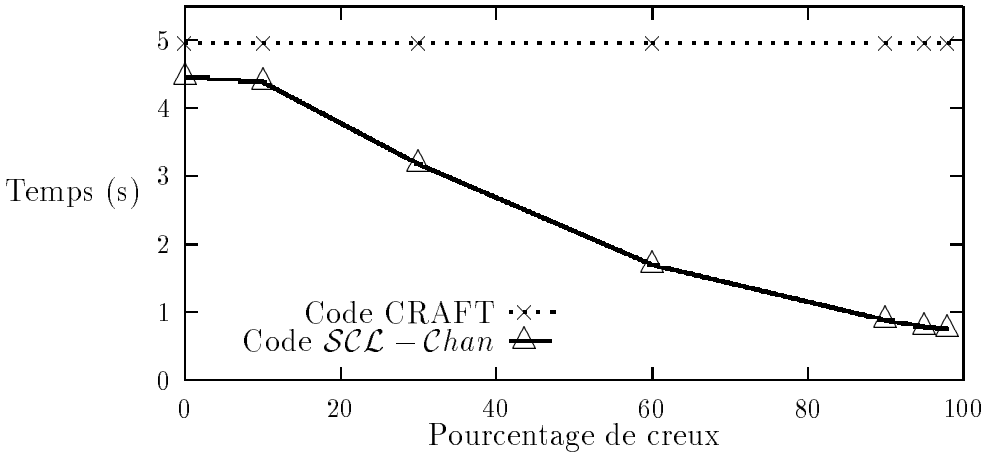


FIG. 6.2 - *Gauss-Seidel avec valeurs nulles.*

6.3 Factorisation de Cholesky

Dans cette section nous considérons le programme $\mathcal{SCL}-Chan$, issu de la parallélisation de la factorisation de Cholesky, présenté en 5.5 page 137. Nous avons réalisé nos tests avec une matrice de 3200×3200 composants. Dans cet exemple, lorsque l'on augmente le nombre de processeurs, la taille de la matrice ne change pas.

6.3.1 Distribution des données et optimisations

Le paralléliseur PAF proposant une distribution cyclique par lignes, nous avons choisi pour la comparaison une distribution identique. Chaque indice possède un tableau de $L \times N$ coefficients avec $N = 3200$ et $L = N/Pe\_Nb$. Cette distribution correspond à une fonction $Owner$ très simple: $Owner(A[I,J]) = (I \ mod \ L)$. La figure 6.3 donne la factorisation de Cholesky dans sa version optimisée.

Dans l'algorithme de Gauss-Siedel, nous avons renommé les éléments de la matrice de façon à utiliser des indices de matrice locaux. Les bornes des boucles ont été modifiées en conséquence. Dans la factorisation de Cholesky, nous allons illustrer l'autre choix: conserver dans le code parallèle les indices originaux de la matrice. Le passage aux indices locaux est automatisé par le biais des fonctions $Transi(x) = x \ div \ Pe\_nb$ et $Transj(x) = x$. La traduction est simplifiée puisque les boucles n'ont pas besoin d'être modifiées pour tenir compte d'indices locaux.

Au cours de la factorisation de Cholesky, les calculs ont lieu sur la partie triangulaire basse de la matrice et les dépendances de données partent de la première colonne vers la dernière.

La première optimisation concerne les accès locaux. Elle est basée sur l'observation suivante. Les calculs nécessaires à la modification d'un élément de la colonne I font intervenir tous les éléments à gauche sur la même ligne. Ces accès n'engendrent pas de communications.

La seconde optimisation concerne les accès distants. Pour chaque élément de la colonne I on a besoin des éléments de la première ligne de la partie triangulaire basse de la matrice: les pivots. On doit communiquer la ligne des pivots à tous les indices possédant des éléments de la colonne I qui doivent être modifiés. En conséquence les communications intervenant lors de la première phase de chaque itération générale sont celles concernant la ligne des pivots. Nous avons étendu notre librairie afin de pouvoir effectuer des communications de tableaux. La ligne des pivots est envoyée en une seule communication. De plus nous envoyons les pivots une seule fois pour chaque indices.

En utilisant la structure de contrôle jump (cf. 5.6.2), nous optimisons le parcours des itérations. Nous adaptons le jump à une distribution cyclique: jump_cyclic. Pour chaque indice, seules sont effectuées les itérations *utiles* des boucles for utilisant les compteurs j et d. Notons que dans notre algorithme les codes internes au jump ne comportent pas d'incréments d'horloges. La gestion des horloges au sein du jump est ici inutile.

Nous introduisons ici le code du jump_cyclic. Ses paramètres sont les mêmes que ceux du jump_block.

```
#define jump_cyclic(jump_1,jump_2,first,last,clock_increment,
{first_iter},{intermediate_iter},{last_iter}){

int j;
j=This+ (first/Pe_Nb + ((( first mod Pe_Nb )>(This))?1:0))*(Pe_Nb);
scl_c_inc(j*clock_increment);
j=first;
{first_iter}
for(j=j;j<last+1-Pe_Nb;j=j+Pe_Nb){
   {intermediate_iter}
   Scl_c_inc((nb_proc - 1)*clock_increment);}
}
{last_iter}
Scl_c_inc((nb_proc - 1)*clock_increment);
Scl_c_inc(jump_2*clock_increment);}
```

Dans l'algorithme $\mathcal{SCL} - Chan$ initial (cf. 5.5), le pivot n'est communiqué que s'il n'est pas nul. Le programme effectue plus de réceptions que d'envois: la réception d'une valeur *nil* est interprétée comme la réception d'un pivot nul. Dans le programme optimisé, nous avons factorisé les envois de pivots en une seule communication. Un envoi n'est effectué que si l'ensemble des éléments à envoyer sont nuls.

```
for(i=0;i<N;i++){
not_sparse_line=0;
  if(i!=0){
    if(This == OWNER(i,i) ){
      not_sparse_line=0;
      for(k=0;k<i;k++){
        if(a[TRANSI(i)][TRANSJ(k)]!=0.0){
          not_sparse_line=1;
          tmp2[k]=a[TRANSI(i)][TRANSJ(k)];
        }else{
         tmp2[k]=0.0;
        }
      }
    }

    for(j=i+1;j<(((i+Pe_nb)<N)?i+Pe_nb:N);j++){
      if(This == OWNER(i,i)){
        if(not_sparse_line){
          Scl_send( OWNER(j,i),tmp2);
        }
        Scl_c_inc(1);
        Scl_c_send(OWNER(j,i));
      }else{
        Scl_c_inc(1);
      }

      if( This == OWNER(j,i)){
        not_sparse_line=Scl_receive(OWNER(i,i),tmp2,tmp2);
      }
    }
  }else{
    Scl_c_inc((((i+Pe_nb)<N)?Pe_nb-1:N-i-1));
  }
  if (not_sparse_line){
    Scl_jump_cyclic(i-1,0,i,N-1,0, {
      for(k=0;k<i;k++){
        a[TRANSI(j)][TRANSJ(i)]=a[TRANSI(j)][TRANSJ(i)]
        -(a[TRANSI(j)][TRANSJ(k)]*tmp2[TRANSJ(k)]);
      },{
        for(k=0;k<i;k++){
          a[TRANSI(j)][TRANSJ(i)]=a[TRANSI(j)][TRANSJ(i)]
          -(a[TRANSI(j)][TRANSJ(k)]*tmp2[TRANSJ(k)]);
        },{
        for(k=0;k<i;k++){
          a[TRANSI(j)][TRANSJ(i)]=a[TRANSI(j)][TRANSJ(i)]
          -(a[TRANSI(j)][TRANSJ(k)]*tmp2[TRANSJ(k)]);
        })
  }
  if( This == OWNER(i,i)){
    a[TRANSI(i)][TRANSJ(i)]=sqrt(a[TRANSI(i)][TRANSJ(i)]);
  }
  if( This == OWNER(i,i)){
    tmp=a[TRANSI(i)][TRANSJ(i)];
  }
  for(j=i+1;j<(((i+Pe_nb)<N)?i+Pe_nb:N);j++){
    if( This == OWNER(i,i)){
      Scl_send(OWNER(j,i),tmp);
      Scl_c_inc(1);
      Scl_c_send(OWNER(j,i));
    }else{
      Scl_c_inc(1);
    }
    if( This == OWNER(j,i)){
      Scl_receive(OWNER(i,i),tmp,tmp);
    }
  }
  Scl_jump_cyclic(i,0,i+1,N-1,0, {
    a[TRANSI(j)][TRANSJ(i)]=a[TRANSI(j)][TRANSJ(i)]
                            /tmp;
    },{
    a[TRANSI(j)][TRANSJ(i)]=a[TRANSI(j)][TRANSJ(i)]
                            /tmp;
    },{
    a[TRANSI(j)][TRANSJ(i)]=a[TRANSI(j)][TRANSJ(i)]
                            /tmp;
    })
  }
```

FIG. 6.3 - *Algorithme $\mathcal{SCL}$ – Chan de Cholesky optimisé*

6.3.2 Tests avec et sans gestion du creux

Nous présentons dans la figure 6.4 les résultats des expérimentations en faisant varier le nombre de processeurs. Quel que soit le nombre de processeurs, le programme $\mathcal{SCL}-Chan$ a de meilleures performances que son équivalent CRAFT généré à partir de PAF (fig. 6.4).

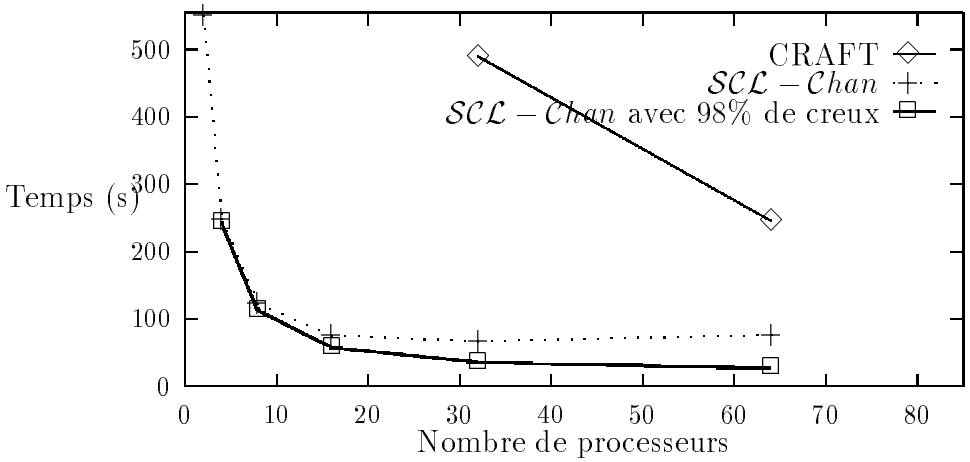


FIG. 6.4 - *Cholesky avec une matrice sans valeurs nulles et avec 98% de valeurs nulles.*

6.3.3 Evolution en fonction du creux

La figure 6.5 donne le résultat des tests avec différents taux de valeurs nulles et sur 32 processeurs. L'amélioration n'est significative qu'à partir d'un taux important de creux (60%). Ce résultat est dû au regroupement des envois de pivot au sein d'une même communication. Il faut que l'ensemble des valeurs d'une ligne soient nulles pour qu'il y ait économie d'envoi. Le programme CRAFT donne un temps constant de l'ordre de 500 secondes: nous ne l'avons pas fait figurer sur le graphique.

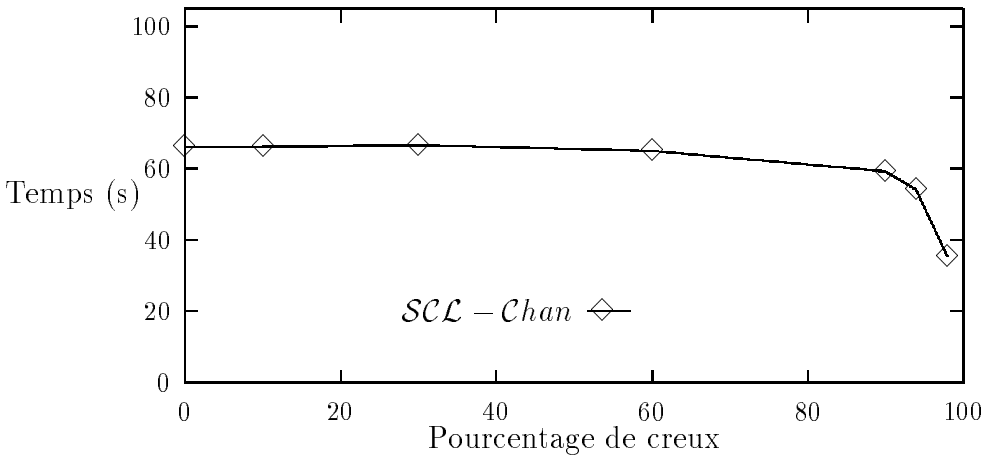


FIG. 6.5 - *Exploitation du creux dans l'algorithme $\mathcal{SCL}-Chan$ de Cholesky.*

6.4 Conclusion

Notre but principal était de vérifier la faisabilité de notre approche. Nous fournissons une réponse partielle. L'approche est pertinente sur la machine T3D. Nous constatons une amélioration très significative des performances entre les programmes $\mathcal{SCL}-\mathcal{C}han$ comparés aux programmes parallélisés avec PAF et traduits en CRAFT. Les tests prouvent l'intérêt de notre approche de la parallélisation. Il reste à poursuivre les expérimentations pour démontrer sa faisabilité sur d'autres architectures.

Le compilateur PAF ne gère pas les structures de contrôle dynamiques. Collard [16] a proposé une extension des concepts de la parallélisation basée sur un ordonnancement des instances d'instructions vers une classe de programmes à contrôle dynamique [16]. Il serait intéressant de tester cette approche afin de comparer ses résultats avec les nôtres.

Le compilateur CRAFT nous était imposé par la machine que nous avons choisie. Il serait prudent de coder les programmes générés par PAF dans d'autres langages dataparallèles et pour d'autres compilateurs.

Chapitre 7

Conclusions et perspectives

Nous avons présenté un cadre unifié pour la traduction de code séquentiel irrégulier en code parallèle. Notre approche consiste essentiellement à structurer les programmes de manière à dégager un ordre logique sur les instructions. L'ordre logique formalise la notion de précédence, permettant ainsi d'encapsuler les informations de dépendance ou d'indépendance au sein d'un programme. Les machines abstraites de $\mathcal{SCP}$ et $\mathcal{SCL}-Chan$ sont basées sur l'attente de signaux diffusant des compteurs multi niveaux, les *horloges structurelles*. Elles permettent de maintenir la cohérence entre des indices exécutant des structures de contrôle dynamiques en autorisant le masquage temporaire, pour un indice distant, de l'exécution d'une structure dynamique. Il est alors possible d'exprimer et de gérer des exécutions faiblement synchronisées, rendant, par là même, le modèle de programmation et la machine abstraite adaptés à l'expression de schémas de synchronisation complexes. La sémantique des accès distants respecte la structure syntaxique du programme.

Nous proposons une approche générale de la traduction de code séquentiel en code parallèle par distribution des données. Elle est fondée sur la sémantique des accès distants dirigés par l'ordre logique. Les structures des boucles et conditionnelles correspondent exactement à celles de leurs sources séquentielles, rendant par là même la parallélisation modulaire. Contrairement aux approches classiques où le code est réorganisé afin d'en extraire le parallélisme et de l'exprimer selon le modèle dataparallèle, il n'y a pas de réorganisation du code. Chaque indice respecte l'ordre séquentiel d'exécution des instructions. Le schéma de traduction est suffisamment général pour être appliqué à des programmes à schémas de dépendances complexes ou évalués dynamiquement. La traduction est fondée sur la distribution des données. La mémoire est gérée sur le mode LIFO: la valeur lue est la plus récente (selon la précédence) mise à disposition par un indice distant. La sémantique des accès aux données permet de supprimer les synchronisations liées aux anti-dépendances. La gestion des communications est assimilable à un mécanisme d'assignation unique dynamique. Il permet de gérer l'assignation unique de façon transparente, aussi bien dans le cadre régulier, que dans le cadre de structures de contrôle dynamiques et de dépendances imprévisibles.

L'inconvénient majeur des approches classiques de la parallélisation de code séquentiel par distribution de données est le parcours obligatoire de toutes les itérations des boucles

par tous les indices, afin d'évaluer les gardes. Notre approche permet d'éviter les parcours inutiles lorsqu'il est possible de prédire statiquement, en fonction d'un placement donné, les itérations pour lesquelles un indice n'a aucun calcul à exécuter. On utilise alors les horloges structurelles de façon à simuler le parcours de ces itérations, sans que celles-ci n'aient été réellement exécutées. Ainsi, l'ordre logique initial entre les itérations est respecté alors que la structure de la boucle est modifiée.

La traduction est fondée sur la distribution des données. En $\mathcal{SCP}$ l'utilisation de gardes implicites et d'une mémoire partagée réduit la parallélisation à un simple problème de distribution des données et d'optimisations. Le modèle introduit avec $\mathcal{SCP}$ permet de séparer l'analyse de dépendance de la génération de code. Nous avons introduit avec $\mathcal{SCP}$ un modèle simple d'évaluation du coût d'une distribution en terme de synchronisation. Manipuler, maîtriser, et tenir compte du coût des communications est une tâche ardue. L'utilisation d'une mémoire partagée permet de s'abstraire de ce problème. Une telle approche a pour inconvénient majeur de ne pas tenir compte du temps réel d'accès à la mémoire. Par contre le modèle de coût reste facile à manipuler. A l'instar du modèle PRAM, notre modèle de coût peut servir d'étalon pour évaluer l'efficacité d'une implantation d'un algorithme parallèle. L'inefficacité étant représentée par le ratio du travail (produit du temps par le nombre de processeurs) de l'implantation par le travail sur le modèle PRAM [59].

L'utilisation du langage $\mathcal{SCL}-\mathcal{C}han$ permet d'apporter une réponse au problème de l'expansion de la mémoire causé par le mécanisme d'empilement des valeurs. $\mathcal{SCL}-\mathcal{C}han$ permet de remplacer l'empilement systématique des valeurs des variables par un empilement ad hoc. Tout en conservant le principe des communications dirigées par la structure du programme, on remplace les lectures en mémoire distantes par un mécanisme de communication bilatérale et explicite par passage de message. L'empilement des valeurs est alors conditionné par les envois explicites de données, ouvrant ainsi la possibilité de n'empiler en mémoire que les données utiles pour le calcul. Cet avantage est toutefois conditionné par la nécessité, pour chaque indice, de connaître les indices distants susceptibles d'utiliser ses données locales. Il est alors indispensable de pratiquer une analyse statique, même partielle, afin de définir des sur ensembles des indices recevant ou émettant des valeurs lors de l'évaluation d'une expression. L'attente n'est pas conditionnée par la réception d'une donnée comme dans le modèle standard de MPI [61]. On évite ainsi les blocages causés par l'absence d'envoi correspondant à une réception. Des sur ensembles des communications nécessaires à la traduction peuvent alors être générés statiquement dans le code.

Lors de la traduction de code séquentiel en code $\mathcal{SCL}-\mathcal{C}han$, des gardes sont explicitement insérées pour la gestion dynamique des communications et pour faire respecter le principe des écritures locales. Si une analyse statique permet de mettre à jour des optimisations sur les gardes, celles-ci peuvent être exprimées dans le code $\mathcal{SCL}-\mathcal{C}han$. En $\mathcal{SCP}$, les gardes sont implicites. Les optimisations sur les gardes ne peuvent être exprimées dans le code. Ainsi, le modèle $\mathcal{SCL}-\mathcal{C}han$ paraît mieux convenir aux programmes permettant l'optimisation des gardes, grâce à une analyse statique des dépendances.

Nous avons montré qu'il est possible de tirer parti d'une analyse, même floue, des dépendances permettant de connaître statiquement un sur ensemble des variables affectées et un sur ensemble des variables lues.

Finalement $\mathcal{SCL}-\mathcal{C}han$ est adapté à la parallélisation d'algorithmes dont l'irrégularité n'interdit pas totalement une analyse statique des dépendances. L'intérêt du modèle $\mathcal{SCP}$ est plus global. Le schéma de traduction en code $\mathcal{SCP}$ est suffisamment général pour être appliqué à des programmes très irréguliers. De plus, il permet d'effectuer une étude simple du coût des synchronisations. Cette étude est ensuite utilisable dans le cadre des programmes $\mathcal{SCL}-\mathcal{C}han$ afin de faire un choix de distribution des données, le critère étant ici la réduction des attentes.

Nous avons montré qu'il est possible d'associer à notre modèle un cadre théorique simple fondé sur des sémantiques opérationnelle et dénotationnelle dirigées par la syntaxe. Ces dernières permettent de faire des preuves par induction sur la structure du programme. La preuve de correction du schéma de traduction est conduite en deux temps. On prouve l'équivalence de la sémantique dénotationnelle avec la sémantique opérationnelle, puis on prouve l'équivalence sémantique du programme séquentiel et du programme parallèle par le biais des sémantiques dénotationnelles. Comparé aux preuves d'équivalence utilisant directement la sémantique opérationnelle [2], nos preuves paraissent plus simples et elles offrent une meilleure indépendance par rapport aux schémas de compilation.

Nous avons conduit sur le Cray T3D des expérimentations à partir de programmes $\mathcal{SCL}-\mathcal{C}han$, issus de notre traduction, puis optimisés en fonction de la distribution de données. Nous comparons ces exemples avec les parallélisations classiques réalisées par le paralléliseur PAF développé à l'Université de Versailles. Les résultats observés sont en faveur des programmes $\mathcal{SCL}-\mathcal{C}han$ pour les données pleines comme pour les données creuses. Nous observons toutefois une très nette amélioration des performances pour ce qui concerne les tests sur des données creuses. Ces résultats sont une première indication en faveur de l'utilisation d'un modèle de communications structuré par la syntaxe et implanté par les horloges structurelles, pour effectuer des parallélisations raisonnablement performantes dirigées par les données. Elles montrent que notre traduction est capable d'exploiter le contrôle irrégulier et qu'il est possible d'en tirer parti pour améliorer les performances des algorithmes lorsque les données sont creuses. Les expériences demanderaient à être complétées en appliquant notre parallélisation sur d'autres algorithmes et en étendant nos tests à d'autres machines.

Perspectives

La méthode de traduction présentée s'étendrait aisément au cadre de l'exécution spéculative de structures de contrôle dynamiques. Une exécution spéculative consiste à ne pas prendre en compte les dépendances de données lors de l'évaluation d'une expression dont dépend le contrôle. Dans ce cas on exécute les opérations contrôlées jusqu'à ce qu'il soit possible de connaître la valeur réelle de l'expression. On doit alors rétablir la cohérence du mécanisme de synchronisation selon que les exécutions s'avèrent prédites correctement ou non. Notre modèle constitue une base solide à la fois pour valider et pour implanter ce mécanisme: en maintenant la cohérence entre des indices exécutant des structures de contrôle dynamiques, les horloges permettent de synchroniser les indices, que les exécutions soient spéculatives ou non. La structure multi niveaux des horloges structurelles autorise

le masquage d'exécutions spéculatives qui se révèlent incorrectes. On doit alors rétablir la cohérence des données en validant les valeurs prédites correctement et en éliminant celles prédites incorrectement. Il est alors possible de s'appuyer sur l'estampillage des données par les horloges structurelles pour rétablir la cohérence des données.

Avec $\mathcal{SCP}$, nous avons fourni une sémantique opérationnelle pour les accès en mémoire distante. En réutilisant un mécanisme similaire, il est possible de réaliser une mémoire virtuellement partagée sur les machines à mémoire distribuée.

Les expérimentations conduites sur le Cray T3D abondent dans le sens de l'efficacité de la méthode de traduction des programmes séquentiels. L'étude expérimentale doit être poursuivie sur d'autres architectures parallèles afin de confirmer cette première série de tests. Il serait alors possible de déterminer précisément les besoins matériels liés au modèle. Par ailleurs, afin de poursuivre les tests avec des codes totalement irréguliers il serait intéressant de disposer d'un run-time pour $\mathcal{SCP}$.

Malgré l'introduction de structures de contrôle dynamiques dans les exemples testés, ceux-ci présentaient des schémas de dépendances en grande partie prévisibles. Il serait intéressant d'effectuer des tests sur des applications au schéma de dépendance totalement imprévisible. Dans ce cas il faudrait implanter le langage $\mathcal{SCP}$. La seule difficulté concernant la faisabilité de l'implantation pourrait provenir de l'espace mémoire nécessaire au mécanisme d'assignation unique dynamique. Il faudrait alors contrôler l'expansion mémoire par blocage temporaire des indices, associé à une récupération mémoire exploitant l'information globale donnée par les horloges. Un tel mécanisme à été implanté par Bruno Raffin [55] pour le langage $\mathcal{SCL}-Chan$. Il ne semble pas qu'il y ait de difficulté majeure à l'étendre au cadre de $\mathcal{SCP}$.

Tel que nous avons présenté $\mathcal{SCL}-Chan$ et $\mathcal{SCP}$, seul le premier permet d'effectuer explicitement des factorisations sur les gardes liées au principe des écriture locales. Il est cependant possible d'effectuer ce type d'optimisation dans $\mathcal{SCP}$ en factorisant les gardes implicites au sein d'un même groupe d'instructions. Afin d'être à même d'exprimer les mêmes optimisations que dans $\mathcal{SCL}-Chan$, il serait intéressant d'enrichir le langage hôte $\mathcal{LH}$ par des structures de contrôle afin de pouvoir faire sortir les gardes implicites des structures de contrôle. Le langage $\mathcal{LH}$ peut être aisément étendu puisque nous n'imposons que très peu de contraintes sur la sémantique de $\mathcal{LH}$. Les seules contraintes portent sur l'évaluation des expressions qui doivent respecter la sémantique des accès structurés par la syntaxe de $\mathcal{SCP}$.

Le modèle présenté constitue une base solide pour l'extension de la fonction de traduction aux programmes séquentiels comportant des procédures et fonctions. La modularité de la fonction de traduction facilite cette tâche: puisque les communications sont dirigées par la syntaxe, les communications au sein d'une procédure portent sur des valeurs évaluées en amont. Une première approche de l'introduction des procédures dans un modèle de communication dirigé par la syntaxe est présentée dans un article cosigné avec Bruno Raffin, Xavier Rebeuf et Bernard Virot [47].

Nous avons vu que la qualité des traductions que nous proposons, en terme de minimisation des synchronisations, est dépendante du choix de la distribution des données. Une voie prometteuse serait de développer l'analyse des dépendances, quand elle est possible,

pour aboutir à des heuristiques de placement. Des recherches en cours visent à établir un modèle de performance pour $\mathcal{SCP}$ ou $\mathcal{SCL} - \mathcal{C}han$. Celui-ci sera essentiel pour mettre en oeuvre ce type d'heuristique.

Bibliographie

[1] BK.R. Apt and E-R. Olderog. *Verification of Sequential and concurrent Programs*. Springer-Verlag, 1991.

[2] C. Bareau. *Distribution automatique de programmes séquentiels: étude structurelle et expérimentale*. PhD thesis, Université de Rennes 1, 1995.

[3] Cyrille Bareau, Benoit Caillaud, Claude Jard, and Rene Thoraval. Correctness of Automated Distribution of Sequential Programs. In *PARLE'93*, number 694 in LNCS. Springer Verlag, June 1993.

[4] Cyrille Bareau and Claude Jard. Dynamic Analysis of Parallelized Sequential Programs using Instrumented Semantics. In *European School of Computer Science: Parallel Programming Environnements For High Performance Computing*, Alpe d'Huez, April 1-5 1996.

[5] L. Bougé. The data-parallel programming model: a semantics perspective. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 4–26. Springer, 1996.

[6] L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard, and B. Virot. Formal validation of data-parallel programs: introducing the assertional approach. In G-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 197–219. Springer, 1996.

[7] L. Bougé, D. Cachera, Y. Le Guyadec, G. Utard, and B. Virot. Formal validation of data-parallel programs: a two-component assertional proof system for a simple language. *Theoretical Computer Science B*, 189(1-2):71–107, 1997.

[8] P. Boulet and T. Brandes. Evaluation of Automatic Parallelization Strategies for HPF Compilers. Technical Report 95-44, LIP, November 1995.

[9] T. Brandes. *ADAPTOR Programmer's Guide Version 4.0*. GMD, March 1996.

[10] Brezany and A. Choudhary. Techniques and Optimizations for Developing Irregular Out-of-Core Applications on Distributed-Memory Systems. Technical Report TR 96-4, Institute for Software Technology and Parallel Systems, University of Vienna, December 1996.

[11] P. Brezany, O. Chéron, K. Sanjari, and E. van Konijnenburg. Processing Irregular Codes Containing Arrays with Multi-Dimensional Distributions by the PREPARE HPF Compiler. In *HPCN Europe'95*, Milan, Italy, 2-5 May 1995.

[12] D. Callahan and K. Kennedy. Compiling programs for distributed-memory multiprocessors. *Journal of Supercomputing*, 2:151–169, 1988.

[13] K. Cameron, L. J. Clarke, and A. G. Smith. CRI/EPCC MPI for CRAY T3D. In *First European Cray T3D Workshop*. EPFL, September 1995.

[14] O. Chéron. *Pandore II: un compilateur dirigé par la distribution des données*. PhD thesis, Université de Rennes 1, 1993.

[15] F. Coelho, C. Germain, and J.-L. Pazat. State of the Art in Compiling HPF. In G.-R. Perrin and A. Darte, editors, *The Data Parallel Programming Model*, volume 1132 of *LNCS*. Springer, 1996.

[16] J.-F. Collard. *Parallélisation Automatique des Programmes à Contrôle Dynamique*. PhD thesis, École Normale Supérieure de Lyon, 1995.

[17] D. Henty. Craft Performance Optimisation. Technical Report EPCC-TR95-04, Edinburgh Parallel Computing Centre, September 1995.

[18] R. Das, R. von Hanxleden, K. Kennedy, C. Koelbel, and J. Saltz. Compiler Analysis for Irregular Problems in Fortran D. In *Proceedings of the Fifth Workshop on Languages and Compilers for Parallel Computing*, pages 97–111, New Haven, Connecticut, (revised January 1993).

[19] Guy Edjlali, Gagan Agrawal, Alan Sussman, and Joel Saltz. Data Parallel Programming in an Adaptive Environment. Technical Report TR-3350, University of Maryland Institute for Advanced Computer Studies, September 1994. (Also cross-referenced as UMIACS-TR-94-109).

[20] P. Feautrier. Array expansion. In *Int. Conf. on Supercomputing*, pages 429–441, St Malo, 1988. ACM.

[21] P. Feautrier. Compiling for massively parallel architectures: a perspective. *Microprocessing and Microprogramming*, 41, 1995.

[22] P. Feautrier. Automatic Parallelization in the Polytope Model. *LNCS*, 1132, 1996.

[23] C.J. Fidge. Time-stamps in message passing systems that preserve the partial ordering. In *11th Australian Computer Science Conf.*, pages 55–66, 1988.

[24] G. Fox, S. Hiranandani, Ken Kennedy, C. Koelbel, U. Kremer, C. Tseng, and M. Wu. Fortran D Language Specification. Technical Report CRPC-TR90079, Department of Computer Science, Rice University, April 1991.

[25] N. Francez. *Fairness*. Springer-Verlag, 1986.

[26] M. Le Fur. *Compilation de boucles dirigée par la distribution des données*. PhD thesis, Université de Rennes 1, 1995.

[27] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[28] V. Getov, T. Brandes, B. Chapman, A. Dunlop, T. Hey, and D. Pritchard. Comparison of HPF-like Systems. Technical Report D4.3.a, University of Southampton, GMD, University of Vienna., November 1993.

[29] M. W. Goudreau, J. M. D. Hill, K. Lang, B. McColl, S. B. Rao, D. C. Stafanescu, T. Suel, and T. Tsantilas. A Proposal for the BSP Worldwide Standard Library. `http://www.bps-worldwide.org/`, July 1996.

[30] G.Utard and G.Hains. Deadlock-free absorption of barrier synchronisations. *Information Processing Letters*, 56:221–227, 1995.

[31] Y. Le Guyadec and B. Virot. Sequential-like proofs of data-parallel programs. *Parallel Processing Letters*, 6(3):415–426, 1996.

[32] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Loosely Synchronized Execution Model for a Simple Data-Parallel Language. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *EuroPar'96 Parallel Processing*, number 1123 in LNCS. Springer-Verlag, 1996.

[33] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. Horloges structurelles pour la désynchronisation de programmes data-parallèles. In *Actes de RenPar'8*, pages 77–80. Université de Bordeaux, France, may 1996.

[34] Yann Le Guyadec, Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. Structural Clocks for a Loosely Synchronized Data-Parallel Language. In *MPCS'96*. EUROMICRO and Istituto di Ricerca su Sistemi Informatici Parallele, May 1996. Final proceedings published by IEEE Computer Society Press, to appear.

[35] High Performance Fortran Forum. *High Performance Fortran Language Specification*, may 1993.

[36] High Performance Fortran Forum. *HPF-2 Scope of Activities andd Motivating Applications*, november 1994.

[37] R-Y. Hwang. Synchronization Migration for Performance Enhancement in a DOACROSS Loop. In *IEE Proceedings-E Computers and Digital Techniques*, pages 107–116, March 1995.

[38] V.P. Krothapalli J.Thulasiraman and M. Giesbrecht. Run-time parallelization of irregular DOACROSS loops. Technical Report No. 95/03, Dept. of Computer Science, University of Manitoba., June 1995.

[39] B. W. Kerninghan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, 1988.

[40] C.H. Koelbel, D.B. Loveman, R.S. Schreiber, G.L. Steele Jr., and M.E. Zosel. *The High Performance Fortran Handbook*. The MIT Press, 1994.

[41] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–564, 1978.

[42] V. Lefebvre. Gestion optimisée des données dans les programmes parallélisés. *Technique et Science Informatique*, 16:1111–1139, Nov 1997.

[43] Y. Mahéo. *Environnement pour la compilation dirigée par les données : supports d'exécution et expérimentations*. PhD thesis, Université de Rennes 1, 1995.

[44] MasPar Computer Corporation, Sunnyvale, CA. *MasPar Fortran Reference Manual*, software version 1.1 edition, August 1991.

[45] F. Mattern. Virtual time and global states of distributed systems. In M. Cosnard and P. Quinton, editors, *Parallel and Distributed Algorithms*, pages 215–226. North Holland, 1989.

[46] W F McColl. General purpose parallel computing. In A M Gibbons and P Spirakis, editors, *Lectures on Parallel Computation. Proc. 1991 ALCOM Spring School on Parallel Computation*, pages 337–391. Cambridge University Press, 93.

[47] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A General but Simple Technique to Handle Asynchronous Data-Parallel Control Structures. In IEEE, editor, *Fifth Euromicro Workshop on Parallel and Distributed Processing - PDP'97*, pages 189–196, London, United Kingdom, January 1997. IEEE Computer Society Press.

[48] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Unifying Approach of Data and Task Parallelism Based on Structural Clock Communication Paradigm (Extended Version). Technical Report RR97-07, LIFO, Orléans, France, May 1997. `http://www.univ-orleans.fr/SCIENCES/LIFO/Members/virot/`.

[49] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. $\mathcal{SCL}-\mathcal{C}han$: An Asynchronous Data-Parallel Language for Irregular Algorithms. In IEEE, editor, *Second International Workshop on High-Level Parallel Programming Models and Supportive Environments - HIPS'97 (in conjonction with 11th International Parallel Processing Symposium - IPPS'97)*, Geneva, Switzerland, April 1997.

[50] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Simple Synchronization and Communication Multi-threaded Library for Automatic Distribution of Irregular Sequential Code. In *Third International Conference on Massively Parallel Computing Systems - MPCS'98*, Colorado Springs, Colorado U.S.A., April 1998.

[51] Emmanuel Melin, Bruno Raffin, Xavier Rebeuf, and Bernard Virot. A Structured Synchronization and Communication Model Fitting Irregular Data Accesses. *Journal of Parallel and Distributed Computing*, 50(1/2):1–27, April/May 1998.

[52] Bongki Moon, Mustafa Uysal, and Joel Saltz. Index Translation Schemes for Adaptive Computations on Distributed Memory Multicomputers. Technical Report CS-TR-3428, University of Maryland at College Park Department of Computer Science, February 1995.

[53] Shubhendu S. Mukherjee, Shamik D. Sharma, Mark D. Hill, James R. Larus, Anne Rogers, and Joel Saltz. Efficient Support for Irregular Applications on Distributed–Memory Machines. Technical Report TR-488-95, Princeton University Computer Science, April 1995.

[54] S. Prakash and R. Bagrodia. An Adaptative Synchronization Method for Unpredictable Communication Patterns in Data-Parallel Programs. In IEEE, editor, *Proceedings of the 9th International Paralle Processing Symposium - IPPS'95*, March 1995.

[55] B. Raffin. *Un modèle structuré de communication et de synchronisaion pour le parallélisme de tâche.* PhD thesis, Université d'Orléans, 1997.

[56] Joel Saltz, Kathleen Crowley, Ravi Mirchandaney, and Harry Berryman. Run-time scheduling and execution of loops on message passing machines. *Journal of Parallel ans Distributed Computing*, 8:303–312, 1990.

[57] David A. Schmidt. *DENOTATIONAL SEMANTICS: A Methodology for Language Development.* Wm. C. Brown, 1986.

[58] Shamik D. Sharma, Ravi Ponnusamy, Bongki Moon, Yuan-Shin Hwang, Raja Das, and Joel Saltz. Run-time and Compile-time Support for Adaptive Irregular Problems. Technical Report CS-TR-3266, University of Maryland at College Park Department of Computer Science, March 1996.

[59] D. Skillicorn. *Foundations of Parallel Programming.* Cambridge International Series on Parallel Computation. Cambridge University Press, 1994.

[60] D. B. Skillicorn. miniBSP: A BSP Language and Transformation System. `http://www.qucis.queensu.ca/home/skill`, 1996.

[61] M. Snir, S. W. Otto, S. Huss-Lederman, D. W. Walker, and J. Dongarra. *MPI, The Complete Reference.* Scientific and Engineering Computation. The MIT Press, 1996.

[62] A. Stewart and M. Clint. Synchronising Asynchronous Communications. In *Europar'97 Parallel Processing*, volume 1300 of *LNCS*, Passau, Germany, August 1997. Springer-Verlag.

[63] M. Ujaldón, E. L. Zapata, S. D. Sharma, and J. Saltz. Parallelization Techniques for Sparce Matrix Applications. *Journal of Parallel and Distributed Computing*, 38:256–266, 1996.

[64] Leslie G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

[65] A. Veen and M. de Lange. Overview of the PREPARE Project. In *4th Int. Workshop on Compilers for Parallel Computers*, pages 345–350, Delft, The Netherlands, December 1993.

[66] V.Lefebvre and P. Feautrier. Storage Management in Parallel Programs. In IEEE, editor, *Fifth Euromicro Workshop on Parallel and Distributed Processing - PDP'97*, London, United Kingdom, January 1997. IEEE Computer Society Press.

[67] R. von Hanxleden. Handling Irregular Problems with Fortran D-A Preliminary Report. Technical Report CRPC-TR93339-S, Center for Research on Parallel Computation, Rice University, 1993.

[68] M. Wolfe. *High Performance Compilers for Parallel Computing.* Addison–Wesley Publishing Company, 1996.

[69] C. Xu and V. Chaudhary. Time-Stamping Algorithms for Parallelization of Loops at Run-Time. In IEEE, editor, *11th International Parallel Processing Symposium - IPPS'97*, pages 443–450, Geneva, Switzerland, April 1997.

Résumé

Cette thèse est consacrée à l'étude d'un modèle cible pour la traduction de code séquentiel en code parallèle.

Les approches classiques de la parallélisation automatique sont basées sur un ordonnancement des instances d'instructions et la génération d'un code proche du modèle dataparallèle, conduisant à l'exécution de «fronts» d'opérations indépendantes. Elles impliquent l'analyse exacte des domaines d'itérations des nids de boucles, des ensembles d'opérations, et des dépendances de données entre les opérations. Si cette méthode fournit une réponse optimale pour une large classe de programmes à schémas de dépendances prévisibles, l'expérience a montré qu'elle est souvent inefficace dans le cas des calculs sur objets irréguliers (par exemple sur les matrices creuses).

Le principe de la résolution dynamique des accès offre une réponse simple pour la traduction de code présentant des accès irréguliers aux données. Toutefois, lorsque les communications sont des accès en mémoire distante, il est indispensable d'introduire des synchronisations pour garantir que les données ne sont pas écrasées avant d'être lues. Si, au contraire, les communications passent par des envois de messages, alors chaque processeur doit parcourir l'ensemble des structures de contrôle, afin de faire correspondre exactement les envois et les réceptions.

Nous proposons une méthode de traduction reprenant le principe de la résolution dynamique des accès. Notre approche consiste à structurer les programmes de telle sorte que les informations de dépendance et d'indépendance entre les instructions soient représentées par la syntaxe. Les accès en mémoire distante respectent la structure syntaxique du programme. Les synchronisations sont basées sur l'attente de signaux diffusant des compteurs multi niveaux appelés les *horloges structurelles*. Elles permettent de maintenir la cohérence entre des processeurs exécutant des structures de contrôle dynamiques, tout en autorisant le masquage temporaire de l'exécution de ces structures. Les lectures en mémoire distante dirigées par la syntaxe permettent d'éviter les synchronisations globales lors de la traduction de codes séquentiels. Lorsqu'une analyse de dépendance, même partielle, est praticable on peut remplacer les accès en mémoire distante par des envois de messages. Le modèle de communication dirigé par la syntaxe permet alors d'éviter le parcours de toutes les structures de contrôle par tous les indices.

Nous montrons qu'il est possible de fonder l'approche par la construction d'un cadre formel constitué d'un langage cible associé à des sémantiques opérationnelle et dénotationnelle. Nous prouvons l'équivalence des deux sémantiques, validant ainsi le modèle de programmation et sa machine abstraite.

Nous terminons par des exemples de programmes parallélisés selon notre méthode et nous comparons leurs performances avec celles de programmes parallélisés suivant des approches plus classiques.

Mots clés

Distribution automatique de programmes séquentiels ; Horloges structurelles ; Modèle de synchronisation et de communication ; Parallélisation automatique ; Algorithmes irréguliers ; Expression de dépendances imprévisibles ou irrégulières ; Implantations sur Cray T3D.