Certification de programmes BSML avec juxtaposition parallèle

Dimitri Louis-Régis

Stage de Master Recherche 2

sous la direction de
Frédéric Loulergue
Laboratoire d'Algorithmique, Complexité et Logique
61, avenue du général de Gaulle
94010 Créteil cedex – France
loulergue@univ-paris12.fr

Avril-Septembre 2005

Remerciements

Je remercie toute l'équipe du projet Propac pour leur aide à la réussite de se stage.

Tout particulièrement, je remercie mon encadrant Le Professeur Frédéric Loulergue qui a su me guider tout au long de ce stage.

Frédéric Gava dont les travaux en programmation fonctionnelle m'ont inspirés.

Et également Abdetouab Belbekkouche et Raddia Benheddi pour leur présence et leur soutient en temps que collègue tout au long de ce stage ainsi que pour les longues discutions qui ont beaucoup contribuées à la compréhension de notions pas toujours évidentes.

Table des matières

Chapitre 1

Introduction

Certains problèmes comme la simulation de phénomènes physiques ou chimiques ou la gestion de bases de données de grande taille nécessitent des performances que seules les machines massivement parallèles peuvent offrir. En plus de ces domaines d'application, on utilise les machines parallèles en recherche opérationnelle, en extraction de connaissances et en réalité virtuelle. Les machines parallèles servent ainsi à un sous-ensemble important de toutes les applications de l'informatique. Leurs programmations demeurent néanmoins plus difficile que celles des machines séquentielles. La conception de langages adaptés est un sujet de recherche actif.

Dans le spectre des solutions possibles pour programmer les systèmes parallèles, le parallélime de données est une voie intermédiaire. C'est un paradigme de programmation parallèle dans lequel un programme décrit une séquence d'actions sur des tableaux à acces parallèles. Le modèle Bulk Synchronous Parallel ou BSP vise à maximiser le parallélisme de données. Un programme BSP est écrit en fonction du nombre de processeurs de l'architecture sur laquelle il s'exécute. Le modèle d'exécution BSP sépare synchronisation et communication et oblige les deux à être des opérations collectives. Il propose un modèle de coût fiable et simple permettant de prévoir les performances de façon réaliste et portable.

Une extension d'Objective Caml basée sur ce modèle est actuellement en cours de dévellopement. Cette bibliothèque est la Bulk Synchronous parallel ML library ou BSMLlib. Ce projet poursuit deux objectifs principaux : parvenir à des langages universels et dans lesquels le programmeur peut se faire une idée du coût à partir du code source. Cette dernière exigence nécessite que soient explicites dans les programmes les lieux du reseau statique de processeurs de la machine.

C'est dans ce cadre que des opérations supplémentaires sont ajoutées à BSMLlib. La juxtaposition parallèle est l'une de ces opérations. Elle est issue de la composition parallèle et permet d'écrire des algorithmes parallèles diviser-pour-reigner en divisant la machine parallèle en sous machines parallèles indépendantes.

Le projet "Programmation parallèle certifié" ou Propac, développé au Laboratoire d'Algorithmique, Complexité et Logique (LACL) de l'université de Paris 12, s'interresse à la certification de programmes parallèles écrits en BSML à l'aide de l'assistant de preuve Coq.

Or la juxtaposition parallèle brise la sémantique purement fonctionnelle de BSML. Pour faire des preuves de programmes BSML avec juxtaposition parallèle en Coq il faut donc passer par une phase de transformation du programme contenant la juxtaposition parallèle en un programme sémantiquement équivalent du point de vue fonctionnel (mais avec des performances diffèrentes) qui ne contient pas de juxtaposition parallèle.

Mon stage se place dans le cadre du projet Propac de l'ACI Jeunes Chercheuse et Chercheurs. L'équipe de ce projet est constituée de Frédéric Loulergue et de Frédéric Gava en temps que membres permanents ainsi que des jeunes chercheurs stagiaires : Benheddi, Belbekkouche et moi-même. Ce stage s'effectue au LACL de l'université de Paris 12 situé à Créteil dans le Val-de-Marne pour une durée de 5 mois. Il se compose de deux phases. La première consiste à complèter la bibliothèque BSMLlib en y ajoutant l'opération de juxatposition. Il nous faut également concevoir la sémantique bas niveau associée. Ensuite, la deuxième phase consiste à concevoir la phase de transformation de programme contenant la juxtaposition afin de prouver la correction d'un ensemble de programme BSML avec juxtaposition parallèle.

Chapitre 2

Présentation de BSML

Notre présentation commencera par aborder des rappels sur le modèle BSP puis nous poursuivrons sur les primitives du langages Bulk Synchronous Parallel ML ou BSML.

2.1 Le modèle BSP

Le modèle *Bulk Synchronous Parallelism* ou BSP est introduit en 1990 par Valiant. Ce modèle de programmation parallèle offre un haut niveau d'abstraction tout en permettant des performances prévisibles et portables sur une large gamme de machines parallèles. Il décrit à la fois une architecture parallèle, un modèle d'exécution ainsi qu'un modèle de coût.

L'archithecture d'un ordinateur BSP est composé d'un ensemble homogène de paires processeur-mémoire, d'un réseau de communication permettant l'échange de messages inter-processeur et d'une unité de synchronisationn globale qui exécute des demandes collectives de barrières de synchronisation. Les performances d'un tel ordinateur sont caractérisées par le nombre de paires de processeur-mémoire p, le temps L nécessaire à la réalisation d'une barrière de synchronisation ainsi que le temps g nécessaire à une 1-relation (phase de communication dans laquelle chaque processeur envoie ou reçoit au plus un mot d'où 1-relation) ; le réseau peut également réaliser un échange, appelé h-relation (au plus h mots émis ou reçus par processeur) en temps $h \times g$.

Le modèle d'exécution d'un programme BSP le décrit comme une séquence de *super-étapes*. Chaque super-étape est divisée en trois phases successives et logiquement disjointes.

La phase de calculs séquentiels Chaque processeur utilise les données qu'il détient localement pour faire des calculs de façon séquentielle et pour demander des transferts depuis ou vers d'autres processeurs,

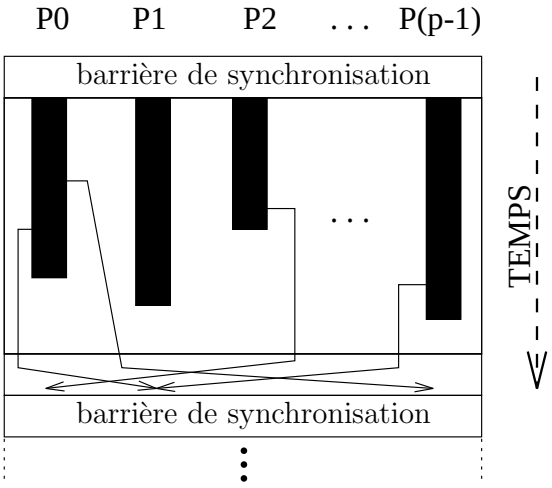


FIG. 2.1 – Une super-étape BSP

La phase de communication le réseau réalise les échanges de données demandés à la phase précédente,

La phase de synchronisation une barrière de synchronisation globale termine la super-étape. A l'issu de cette barrière de synchronisation globale, les données échangées sont effectivement disponibles. Une nouvelle super-tape peut alors commencer.

Le modèle de coût prévoit un temps d'exécution pour une super-étape qui dépend de la durée maximale de chacune de ses composantes : durée des calculs locaux, durée des communications, durée de la synchronisation globale.

$$\text{Time}(s) = \max_{i:processeur} w_i^{(s)} + \max_{i:processeur} h_i^{(s)} \times g + L$$

où $w_i^{(s)}$ est le temps de calcul local du processeur i durant la super-étape s,
où $h_i^{(s)}$ est le maximum entre le mots transmis $h_{i+}^{(s)}$ et les mots reçus $h_{i-}^{(s)}$ par le processeur i durant la super-étape s.

Le temps d'exécution $\sum_s \text{Time}(s)$ d'un progrmme BSP est donné par la formule suivante :

$$W + H \times g + S \times L$$

où S est le nombre de super-étapes du programme
où W = $\sum_s \max_i w_i^{(s)}$
où H = $\sum_s \max_i h_i^{(s)}$
L'optimisation de tel programme passe par la minimisation des trois paramètres qui sont le nombre de super-étapes, le déséquilibre des communications $h^{(s)}$ et le déséquilibre des calculs locaux $w^{(s)}$.

2.2 Bulk Synchronous Parallel ML

Le langage BSML est un langage data-parallèle fonctionnel pour la programmation d'algorithmes BSP. Il est basé sur une extension confluente du λ-calcul. Grâce à la structure BSP sousjacente de ses programmes, il est associé au modèle de coût BSP. On peut prouver le bon fonctionnement des programmes BSML avec l'assistant de peuve Coq. Le langage BSML ne possède pas d'implémentation en temps que langage à part entière mais sous forme d'une bibliothèque BSMLlib pour Objective Caml. La version 0.3 de cette bibliothèque est actuellement en cours de dévellopement. Elle proposera une programmation modulaire des différentes primitives présentées en figure 2.2 et intègrera entre autres les primitives liées à la composition parallèle (voir figure 2.3).

bsp_p: unit → int
bsp_g: unit → float
bsp_l: unit → float
mkpar: (int → α) → α **par**
apply: (α → β) **par** → α **par** → β **par**
put: (int → α option) **par** → (int → α option) **par**
at: α **par** → int → α

FIG. 2.2 – Primitives

Les programmes BSML s'écrivent comme les programmes Caml usuels.
Ce sont des programmes séquentiels qui utilisent des structures de données parallèle pour effectuer les calculs parallèles ainsi que les communications.

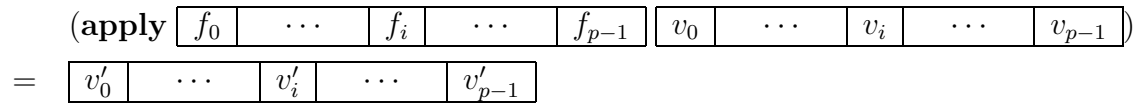
Les fonctions **bsp_p**(), **bsp_g**() et **bsp_l**() donnent respectivement acces aux paramètres BSP : p (nombre de processeurs constant durant l'exécution tant que la juxtaposition parallèle n'est pas utilisée, voir chapitre suivant), g et L.

On utilise une structure de donnée parallèle, le vecteur parallèle, pour effectuer les calculs parallèles. Le vecteur parallèle est de type α **par** : on a p valeurs (une valeur par processeur) de type α . L'imbrication des types par est interdit grâce à un système de typage introduit par F. Gava.

C'est la primitive **mkpar** qui crée les vecteurs parallèles. Pour chaque processeur de la machine parallèle, **mkpar** f applique la fonction f placée en paramètre à la valeur i du pid de ce processeur. Le processeur i contient donc (f i) évaluée en v_i.

v_0	$\cdots$	v_i	$\cdots$	v_{p-1}

La primitive **apply** permet d'appliquer un vecteur parallèle de fonctions à un vecteur parallèle d'arguments.Pour chaque processeur de la machine parallèle **apply** f v applique la i^{me} composante de la fonction f placée en paramètre à la i^{me} composante du vecteur argument v placé en paramètre. Le processeur i contient donc la valeur $(f_i\ v_i)$ évaluée en v'_i.



Ces deux primitives permettent le calcul des phases asynchrones (la première phase d'une super-étape).

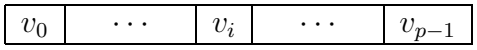
Les deux primitives restantes **put** et **at** permettent de programmer la phase synchrone elles font intervenir les communications.

put utilise le type α option défini par : **type** α option = None | Some **of** α

L'argument de cette primitive est un vecteur parallèle de fonctions qui décrivent les messages *à envoyer*. La fonction f_i au processeur i indique pour chaque processeur de destination j, soit la donnée à envoyer $(f_i\ j) = \text{Some}\, v$, soit la constante None signalant qu'aucune donnée ne sera envoyée de i vers j.

Le résultat est encore un vecteur parallèle de fonctions, qui décrivent les messages *reçus*. Ainsi la fonction g_j au processeur j appliquée à i donnera $\text{Some}\, v$ si le processeur i a envoyé la valeur v au processeur j, et donnera None si i n'a rien envoyé à j.

Enfin la primitive **at** permet de projeter la valeur d'un vecteur parallèle à un processeur donné :



Elle est généralement employée avec en combinaison avec la primitive de conditionnelle globale du BSλ-calcul ou avec un **match with**. Le système de typage interdit l'évaluation d'une expression contenant cette primitive dans le contexte d'une primitive **mkpar**.

2.3 La juxtaposition parallèle

L'écriture d'algorithme de type diviser-pour-reigner nécessite l'ajoute d'opérateurs de composition parallèle à BSMLlib. La juxtaposition est une opération de composition parallèle qui permet d'évaluer deux programmes sur la même machine parallèle. Elle sépare la machine parallèle en deux sous machines et évalue chacun des programmes sur une sous machine. Afin de conserver le modèle BSP d'exécution, les primitives de communication à savoir get et at effectuent des barières de synchronisation globale sur la totalité du reseau.

2.3.1 Présentation informelle

L'évaluation du terme (**juxta** $m\ E_1\ E_2$) se déroule comme suit :

– la première sous machine est constituée des proceseurs numérotés de 0 à m-1 soit un total de m processeurs. La valeur du paramètre p du modèle BSP est désormais m sur cette sous machine. L'évaluation du terme E_1 s'effectura sur cette sous machine.
– la seconde sous machine est constituée des proceseurs numérotés de m à p-1 soit un total de $p-m$ processeurs. La valeur du paramètre p du modèle BSP est désormais $p-m$ sur cette sous machine. Tous les processeurs i de cette sous machine sont dès lors renommés en $i-m$, ce qui transforme le processeur m en processeur 0 et le processeur (p-1) en (p-1)-m. L'évaluation du terme E_2 s'effectura sur cette sous machine.

Une différence entre le nombre de super-étapes s'effectuant sur chacune des sous machines peut générer des problèmes lors de la phase de synchronisation globale. La synchronisation globale nécessite que tous les processeurs du reseau demande une synchronisation. Si sur un des sous réseaux il n'y a plus de phases de communication à effectuer celui-ci ne fera plus de demande de synchronisation, ce qui génère un blocage pour les autres sous réseaux qui sont en attente d'une synchronisation. La primitive sync permet de contourner ce problème. Elle appelle en boucle des barrières de synchronisation jusqu'à ce que l'un des appels concerne tout le réseau. L'appel en boucle des barrières de synchronisations a pour effet de simuler des calculs sur les sous réseaux ayant fini leur calcul. Ainsi il y a à nouveau synchronisation globale du réseau. La primitive **sync** peut se réduire dès que tous les appels de synchronisation sont effectués par elle, ce qui signifie qu'il ne reste plus aucun tranfert à effectuer et que les calculs de la primitive **juxta** sont terminés.

Le résultat de l'évaluation d'une juxtaposition parallèle est un vecteur :

$$(\mathbf{juxta}\, m\ < v_0, \cdots, v_{m-1} > < v'_0, \cdots, v'_{p-m-1} >) =< v_0, \cdots, v_{m-1}, v'_0, \cdots, v'_{p-m-1} >$$

Pour éviter que les deux derniers arguments de la fonction **juxta** et l'argument de la fonction **sync** ne soient évalués il faut qu'ils soient des fonctions :

juxta: int $\rightarrow$ (unit $\rightarrow$ α **par**) $\rightarrow$ (unit $\rightarrow$ α **par**) $\rightarrow$ α **par**
sync: (unit $\rightarrow$ α **par**) $\rightarrow$ α **par**

FIG. 2.3 – Primitives de la juxtaposition

2.3.2 Example

L'exemple suivant (Figure 2.4) est une version diviser-pour-règner du calcul des préfixes en parallèle, appelé **scan**. Sa sémantique est définie par :

$$\text{scan}\ \oplus\ \langle\ v_0\ , \ldots,\ v_{p-1}\ \rangle\ =\ \langle\ v_0\ , \ldots,\ v_0 \oplus v_1 \oplus \ldots \oplus v_{p-1}\ \rangle$$

```
let rec scan op vec =
  if bsp_p()=1 then
    vec
  else
    let mid = bsp_p()/2 in
    let vec' = sync(fun() →  juxta mid (fun () → scan op vec)
                                       (fun () → scan op vec) ) in
    let msg vec = apply (mkpar(fun i v → if i=mid−1
    then fun dst →  if dst>=mid then Some v else None
    else fun dst →  None)) vec
    and parop = parfun2(fun x y → match x with None → y
                                             |Some v → op v y) in
    parop (apply(put(msg vec'))(mkpar(fun i → mid−1))) vec'
```

FIG. 2.4 – Calcul des préfixes avec la juxtaposition parallèle

où $\oplus$ est une opération binaire associative.

Le réseau est divisé en deux parties et la fonction **scan** est appliquée récursivement sur ces deux parties. La valeur au dernier processeur de la première partie est diffusée à tous les processeurs de la seconde partie. Puis cette valeur et la valeur locale calculée par l'appel récursif sont combinées avec l'opération **op** sur chaque processeur de la seconde partie.

Chapitre 3

Implémentation et test

La version 0.3 de la bibliothèque BSMLlib est actuellement en cours de développement. Elle proposera une version modulaire, au sens des modules d'Objective Caml, ce qui aura pour effet d'harmoniser les implantations et de faciliter la maintenance ainsi que les évolutions futurs de la bibliothèque. L'implantation de la juxtaposition parallèle dans la version 0.3 passe par une identification claire et précise des modifications à apporter dans chacuns des modules. Mon travail a donc consisté à identifier ces modifications en implantant la juxtaposition parallèle sur la dernière version stable (terminée) de BSMLlib, à savoir la version 0.25.

3.1 Présentation de la version 0.25

Contrairement à la vue du programmeur, l'implantation parallèle de BSML offre une vue SPMD des programmes. Un programme écrit en BSML commence par une fonction d'initialisation. Cette fonction met à jours les paramètres BSP et lance l'exécution d'une copie du programme sur chacun des processeurs de la machine parallèle via des primitives MPI.

Les primitives de base du langage sont codées en Caml dans le fichier *bsmllib.ml*. Certaines informations systèmes ainsi que les procédures MPI ne sont pas directement accessibles en Caml. Aussi le fichier *bsmlimpl.c* contient tous les codes implémentés en C. L'interface entre le code C et le code Caml est effectuée par le fichier *mpi.ml*.

Intégrer la juxtaposition nécessite de modifier chacun de ces fichiers ainsi que leurs interfaces associées.

1. Transformer les paramètres systèmes (pid, nombre de processeurs) en variable. En effet, tant que la juxtaposition ne rentre pas en jeu, ils restent constants. Mais la juxtaposition séparant le réseau en deux, le nombre de processeurs de chaque sous machine est donc réduit et les processeurs de celles-ci sont renumérotés. Il nous faut donc modifier le code afin que ces paramètres puissent varier suivant le sous réseau dans lequel ils se trouvent.

2. Gérer les communications de manière adéquate.
3. Gérer la primitive **sync**.

3.2 Transformation des paramètres systèmes

Il y a deux paramètres systèmes à modifier : **pid** et **bsp_p**.

Le pid est une valeur propre à chaque processeur. Cette valeur correspond à la numérotation du proceseur dans le reseau auquel il appartient. On distinguera deux pid.

- Le pid absolu qui correspond à la numérotation du processeur sur l'ensemble de la machine parallèle. Cette valeur demeure constante au cours de l'exécution du programme.
- Le pid relatif qui correspond à la numérotation du processeur sur le sous réseau auquel il appartient. Cette valeur varie suivant la composition du sous réseau.

La juxtaposition divise la machine parallèle en deux sous réseaux et les processeurs de chacun de ses sous réseaux sont renumérotés. C'est donc le pid relatif qui est modifié. Le renommage des numéros de processeurs dans la sous machine s'effectue par rapport au pid absolu du premier processeur (**first**) de la sous machine. Le pid relatif s'obtient donc par une soustraction de la valeur du **first**, de la valeur du pid absolue.

Lors de l'initialisation du programme **first** est initialisée à 0 ce qui nous donne un pid relatif identique au pid absolu tant qu'il n'y a pas de juxtaposition. A l'évaluation d'un **juxta**, la valeur **first** est mise à jours sur chacun des processeurs en fonction du sous réseau auquel il appartient.

Le second paramètre, **bsp_p**(paramètre p du modèle BSP) est une valeur propre à chaque sous-machine. Cette valeur correspond au nombre de processeurs présents dans le reseau auquel il appartient. La juxtaposition divise la machine parallèle en deux sous réseaux. La taille du réseau auquel appartient le processeur est donc modifiée.

Au début de tout programme la variable **bsp_p** est initialisée à la taille de la machine parallèle. Á l'évaluation d'un **juxta** la valeur du **bsp_p** est mise à jours sur chacun des processeurs en fonction de la sous machine à laquelle il appartient.

La mise à jour des valeurs est faite à l'aide de piles. A chaque appel de la fonction **juxta**, la nouvelle valeur est calculée puis empilée sur la pile correspondante. Une fois l'évaluation de la fonction **juxta** terminée, on dépile ; ainsi l'ancienne valeur est restaurée. L'accès à la variable, se fait par lecture de la tête de pile.

3.3 Gestion des communications

Le maintient du modèle BSP, impose que les communications s'effectuent au niveau global (c'est-à-dire sur l'ensemble de la machine parallèle). Les primitives qui génèrent des communications, à savoir **put** et **at**, reçoivent des paramètres qui tiennent compte uniquement du sous réseau. Il y a une différence de format entre les paramètres du niveau global et ceux du sous réseau. Cette différence tient à la taille des vecteurs. En effet les vecteurs issus du réseau global contiennent plus d'éléments. Il nous faut donc traiter les données venant du sous réseau pour les rendre cohérentes au niveau global afin que l'émision des données soit possible. Ensuite nous devrons traiter les données reçues du niveau global afin qu'elles soient exploitable par le sous réseau.

Les primitives **put** et **at** recoivent toutes les deux un vecteur en argument. Ce vecteur est aux dimentions du sous réseau à savoir un élément par processeur présent dans le sous réseau réseau. Le traitement des données issues du sous réseau vers le niveau global consite à augmenter le vecteur par des éléments correspondant à chacun des processeurs extérieurs au sous reseau. Ces éléments seront affectés de la valeur None. Ainsi ils ne génèreront aucune transmition de messages.

Soit une machine parallèle à 4 processeurs est séparée en deux sous réseaux
après un **juxta** 2 e_1 e_2

0	1	2	3

La primitive put évaluée dans le sous réseau 1
(processeurs 2 et 3 dans la numérotation d'origine) :
(put | $f\ 0$ | $f\ 1$ |)

L'argument | $f\ 0$ | $f\ 1$ | passe au niveau global et devient :

None	*None*	$f\ 0$	$f\ 1$

FIG. 3.1 – Transformation des vecteurs pour le niveau global

Le traitement des données issues du niveau global vers le sous réseau consite à reccupérer uniquement les éléments correspondant à des processeurs du sous réseau. La valeur **first** nous indique le décalage à effectuer afin de reccupérer la bonne valeur.

3.4 Gestion du sync

La primitive **sync** est chargée d'effectuer des synchronisations en boucles sur les processeurs ayant fini leurs évaluations afin de permettre aux autres processeurs de réaliser leurs communications. Dans la version 0.25 il n'y a pas de barrière de communication explicite. En effet les communications sont implémentées à l'aide des primitive MPI blocantes. Les primitives blocantes génèrent une barrière de synchronisation au début et à la fin de chaque communication. Les échanges de données sont implémentés comme suit :

- Dans un premier temps les processeurs s'échangent la taille des données qu'ils auront à transmettre via une diffusion. C'est la primitive *bsmlimpl_alltoall_int* qui effectue cette opération. Elle utilise un *MPI_alltoall* en C.
- Ensuite les données sont échangées entre les processeurs via une second diffusion. C'est la primitive *bsmlimpl_alltoall* qui effectue cette opération. Elle utilise un *MPI_Alltoallv* en C. Cette fonction permet un échange optimisé des données puisque leurs tailles sont connues d'avance grâce à la diffusion précédente.

Notre primitive **sync** doit donc réaliser deux diffusions afin de permettre au autres processeurs d'effectuer leurs communications. La première diffusion (celle chargée de l'échange de la taille des données) transmettra la valeur -1. La seconde diffusion ne transmettra aucune valeur puisqu'elle a indiqué lors de la diffusion précédent n'avoir aucune valeur à transmettre.

La primitive **sync** génère en boucle des barrières de synchronisation. La condition de sorite de la primitive **sync** est gérée lors de la première diffusion. La fonction *bsmlimpl_alltoall_int* qui effectue le broadcast renvoie 0 si tous les messages reçus ont la valeur -1. En effet la taille des données à transmettre n'est jamais négative par définition. La valeur -1 est forcément issue d'une primitive **sync**.

Remarques La première diffusion transmet les tailles des données. La réception d'une taille négative peut entrainer des erreurs. La valeur -1 est donc remplacée par la valeur 0 à la réception du message.

La seconde diffusion échange les données. La réception d'une donnée vide peut entrainer un décalage dans le vecteur résultat. La donnée vide est donc remplacée par la donnée None à la réception du message.

3.5 Test

La version 0.25 de BSMLlib offre deux modes de compilation. Un mode parallèle via MPI et un mode séquentiel. Le mode séquentiel permet une simulation du programme. Il est utilisé lors des évaluations dans l'interprèteur en ligne hérité d'Ob-

jective Caml. Le simulateur sèquentiel qui accepte les primitives de la juxtaposition parallèle fut implanté par F. Loulergue en 2002. Il a été intégré à la nouvelle version de BSML issue de se stage.

Cette implantation parallèle de la juxtapositon pour BSML a été testée sur une machine séquentielle à l'aide de processus virtuelles comme le permet MPI. Des tests sur les machines parallèles du LACL et du LIFO reste à venir.

Chapitre 4

Sémantique distribuée de BSML avec juxtaposition

L'évaluation distibuée permet de donner toutes les étapes du calcul sur chaque processeur et de produire une estimation du temps d'exécution du programme. Elle se place au plus près de l'implantation machine et présente le programme comme une composition parallèle de programmes s'exécutant sur chacun des processeurs de la machine parallèle. Cette vue de type SPMD de l'évaluation de termes comporte pour chaque processeur :

- un terme 'e' ,
- une variable $\mathcal{G}id$ qui contient la valeur du pid du processeur dans le réseau global (la totalité de la machine parallèle),
- une variable $\mathcal{L}id$ qui contient la valeur du pid du proceseur dans le sous réseau courant (sous réseau sollicité),
- une pile $\mathcal{E}_{first}$ dont la tête contient toujours la valeur $\mathcal{G}id$ du premier processeur du sous réseau courant (sous réseau sollicité),
- une pile $\mathcal{E}_{bsp\_p}$ dont la tête contient toujours le nombre de processeur du sous réseau courant (sous réseau sollicité).

La définition de la variable $\mathcal{L}id$ n'est pas indispensable car elle peut être facilement obtenue à l'aide des autres informations contenues dans le processeur donné. Néanmoins, nous l'incluons afin de rendre notre sémantique plus lisible.

4.1 Syntaxe

La syntaxe décrit les constructions du langage, et la façon de les désigner formellement. (x : $\mathcal{L}$) et (x : $\mathcal{G}$) sont respectivement des variables locales et globales. L'écriture $x : \tau$ signifie que l'identifiant x peut être global ou local (i.e. $\tau = \mathcal{L} \mid \mathcal{G}$). Partant de ce typage explicite des variables et avec le système de typage, on type les termes (ou expressions) en termes locaux et termes globaux respectivement.

Cette syntaxe contient un mini langage fonctionnel auquel est ajouté les primitives BSML décrites informellement chapitre 2. Le terme $\overrightarrow{e}$ est manipulé par un processeur mais qui, lorsqu'on considère l'ensemble des processeurs, fait partie d'un vecteur parallèle.

Les termes locaux sont essentiellement des λ-expressions. Les termes globaux sont des termes dit "plats", c'est-à-dire, des termes qui ne permettent pas l'utilisation de la primitive **juxta** en dehors d'une primitive **sync**.

e	::=	x	(variable)
	|	c	(constante)
	|	**bsp_p**	(variable) [paramètre p du modèle BSP]
	|	**fun** $x : \tau \ \rightarrow \ e$	(fonction abstraction)
	|	op	(opérateur)
	|	$(e \ e)$	(application)
	|	$(e \ , \ e)$	(paire)
	|	**let** $x : \tau \ = \ e$ **in** e	(constructeur de liaison)
	|	**fst** e	(primitive qui renvoie le premier élément d'une paire)
	|	**snd** e	(primitive qui renvoie le second élément d'une paire)
	|	**if** e **then** e **else** e	(conditionnelle)
	|	**mkpar** e	(vecteur parallèle)
	|	**apply** e e	(application parallèle)
	|	**get** e e	(primitive de communication)
	|	**if** e **at** e **then** e **else** e	(conditionnelle globale)
	|	$\overrightarrow{e}$	(vecteur parallèle énuméré)
	|	**sync** e'	(primitive sync)

FIG. 4.1 – Syntaxe des termes

où e' représente les expressions n'ayant pas la restriction du **sync** :

$$
\begin{array}{lll}
e' & ::= & x \ \mid \ c \ \mid \ \textbf{bsp\_p} \ \mid \ \textbf{fun} \ x : \tau \ \rightarrow \ e' \ \mid \ op \ \mid \ (e' \ e') \\
& \mid & (e' \ , \ e') \ \mid \ \textbf{let} \ x : \tau \ = \ e' \ \textbf{in} \ e' \ \mid \ \textbf{fst} \ e' \ \mid \ \textbf{snd} \ e' \\
& \mid & \textbf{if} \ e' \ \textbf{then} \ e' \ \textbf{else} \ e' \ \mid \ \textbf{mkpar} \ e' \ \mid \ \textbf{apply} \ e' \ e' \ \mid \ \textbf{get} \ e' \ e' \\
& \mid & \textbf{if} \ e' \ \textbf{at} \ e' \ \textbf{then} \ e' \ \textbf{else} \ e' \ \mid \ \overrightarrow{e'} \ \mid \ \textbf{juxta} \ m \ e' \ e'
\end{array}
$$

Les valeurs sont :

Lors de l'évaluation les termes se réduisent jusqu'à atteindre une forme irréductible appelée valeur (figure-4.2). Les termes locaux et certain termes globaux (ceux

$$v \ ::= \ \textbf{fun}\ x \ \rightarrow \ e \quad | \quad c \quad | \quad op \quad | \quad (\ v\ ,\ v\ )$$

FIG. 4.2 – Syntaxe des valeurs

qui ne génèrent pas de communications) suivent des règles de réduction locale. Les autres termes globaux (**get** , **ifat**, **sync**) suivent des règles de réduction globale. Les chemins de réduction d'un terme sont multiples et ne mènent pas toujours au même résultat. Afin de maintenir la confluence du calcul lors des réductions, nous avons besoin de contextes qui nous imposent un chemin de réduction en accord avec la statégie faible d'appel par valeur employée dans Objective Caml (figure-4.3).

$$\begin{array}{rcl}
\Gamma & := & [] \quad | \quad \Gamma\ e \quad | \quad v\ \Gamma \quad | \quad \textbf{let}\ x : \tau\ =\ \Gamma\ \textbf{in}\ e \\
& | & \textbf{if}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e \quad | \quad \textbf{mkpar}\ \Gamma \quad | \quad \textbf{apply}\ \Gamma\ e \quad | \quad \textbf{apply}\ v\ \Gamma \\
& | & \textbf{if}\ e\ \textbf{at}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e \quad | \quad \textbf{if}\ \Gamma\ \textbf{at}\ v\ \textbf{then}\ e\ \textbf{else}\ e \\
& | & \textbf{get}\ \Gamma\ e \quad | \quad \textbf{get}\ v\ \Gamma \quad | \quad \textbf{juxta}\ \Gamma\ e_1\ e_2 \quad | \quad \overrightarrow{\Gamma} \quad | \quad \|\Gamma\|
\end{array}$$

FIG. 4.3 – Syntaxe des contextes

4.2 Règles

L'évaluation distibuée se définit suivant deux groupes de règles : les règles liées à l'évaluation des termes locaux et celles liées à l'évaluation globale.

Les règles pour l'évaluation locale de termes $\dot{\rightsquigarrow}$ sont composées des règles de réduction classic issues d'un mini langage BSML (de règle 4.1 à règle 4.10) et des règles de réduction parallèle issues des primitives globales de BSML (de la règle 4.11 à la règle 4.15). Elles sont présentées à la figure TAB-4.1. L'évaluation locale de termes correspond à une phase de calcul local d'un programme BSP (du calcul pure sans communications). Tous les termes locaux et toutes les applications globales ne faisant pas intervenir de communication sont évalués localement et correspondent à une phase asynchrone.

Remarque L'expression $h\mathrm{d}(\mathcal{E}_{first})$ signifie que l'on lit la valeur de la tête de la pile $\mathcal{E}_{first}$.

Les règles de réduction locale sont associées à la règle de contexte suivante :

L'évaluation globale de termes distribués autorise l'évaluation de l'opération **get**, de la conditionnelle globale ainsi que de l'opération **sync**. Ces opérations génèrent

$$\frac{(e_i, i, \mathcal{L}id_i, \mathcal{E}_{first_i}, \mathcal{E}_{bsp\_p_i}) \longrightarrow (e'_i, i, \mathcal{L}id'_i, \mathcal{E}'_{first_i}, \mathcal{E}'_{bsp\_p_i})}{(\Gamma(e), i, \mathcal{L}id_i, \mathcal{E}_{first}, \mathcal{E}_{bsp\_p}) \longrightarrow (\Gamma(e'), i, \mathcal{L}id'_i, \mathcal{E}'_{first}, \mathcal{E}'_{bsp\_p})}$$

FIG. 4.4 – Règle de contexte local

toutes des communications et des synchronisations entre les processeurs. Le traitement spécifique de ces termes est dû en partie aux contraintes imposées par le modèle BSP. Le modèle BSP impose qu'il y ait une barrière de synchronisation à chaque opération de communication. Ceci nécessite que tous les processeurs de la machine soient à l'évaluation d'une oprération de communication. Dès que cette condition est atteinte, toutes les communications s'effectuent de manière synchrone. En conséquence les règles de réductions de ces termes et leurs règles de contextes se placent au niveau du réseau (et non plus uniquement au niveau du processeur). On a donc une écriture qui présente simultanément toutes les évaluations de tous les processeurs du réseau. Ces règles sont présentées en figure TAB-4.2. Elles s'appliquent au sous réseau courant et correspondent à la phase de communication et de synchronisation d'une super-étape BSP.

Les règles de contextes associées à la réduction globale de termes se placent au niveau global. Elles prennent en compte l'ensemble des processeurs de la machine parallèle tandis que les règles de réductions globales de termes s'effectuent à l'échelle du sous réseau sollicité. Il nous faut donc un outil afin d'identifier les processeurs d'un même sous réseau au niveau global. Cet outil est la numérotation des processeurs en P_k groupes : tous les processeurs de chaque sous réseau sont regroupés dans un unique groupe P_k. La définition formelle des P_k groupe s'ennonce comme suit :

Dans une suite ordonne de processeur de la machine parallle, on identifie par j_m *le premier processeur ne faisant pas partie du sous rseau* m :
Soit j_0 *tq* $\forall i, i' \in [0, j_0[, h(\mathcal{E}^i_{first}) = h(\mathcal{E}^{i'}_{first})$ *et* $h(\mathcal{E}^i_{first}) \neq h(\mathcal{E}^{j_0}_{first})$
On definit j_m *avec* $j_{m-1} < j_m$ *tq*
$\forall i, i' \in [j_{m-1}, j_m[, h(\mathcal{E}^i_{first}) = h(\mathcal{E}^{i'}_{first})$ *et* $h(\mathcal{E}^i_{first}) \neq h(\mathcal{E}^{j_m}_{first})$
On regroupe ensuite tous les processeurs compris dans l'interval
de deux indices j *de fin de rseau successifs dans un mme groupe* P :
$P_0 = \{i \ / \ 0 \leq i < j_0\}$
$P_m = \{i \ / \ j_{m-1} \leq i < j_m\}$
On note $l_k = |P_k|$ *cardinal de l'ensemble* P_k
Et on obtient $i^k_0, \cdots, i^k_{l_k-1}$ *les elements du groupe* P_k

Ce sont les règles de contextes globaux figure-4.5 qui sont en charge de la barrière de synchronisation. En effet, pour qu'un terme mettant en oeuvre une communication se réduise, il faut que tous les processseurs de la machine soient à l'évaluation d'une communication valide. Pour ce faire, le terme doit :

– soit pouvoir se réduire en utilisant les règles de réductions globales 4.17 ou 4.18.
– soit être une primitive **sync** qui se réduit en elle même sur l'ensemble du sous réseau sollicité.

La règle 4.20 assure la compatibilité entre évaluation locale et évaluation globale. Elle retranscrit les évaluations locales à l'échelle globale et rend ainsi compte de l'aspect asynchrone des évaluations locales. C'est elle qui exprime au niveau global que chaque processeur effectue ses réductions locales indépendemment du réseau.

$$\frac{\begin{array}{l}\forall k \in [0,h] \\ < (e_0^k, \mathcal{G}id_0^k, 0,\ \mathcal{E}_{first}^k, \mathcal{E}_{bsp\_p}^k), \cdots, (e_{l_k-1}^k, \mathcal{G}id_{l_k-1}^k, l_k-1, \mathcal{E}_{first}^k,\ \mathcal{E}_{bsp\_p}^k) > \\ \rightsquigarrow \\ < (e'^k_0, \mathcal{G}id^0, 0, \mathcal{E}_{first}^k, \mathcal{E}_{bsp\_p}^k), \cdots, (e'^k_{l_k-1}, \mathcal{G}id_{l_k-1}^k, l_k-1, \mathcal{E}_{first}^k, \mathcal{E}_{bsp\_p}^k) > \\ \text{par les regles 4.17, 4.18, ou } \forall i \in P_k\ e_i = \mathbf{sync}(v)\ et\ e'_i = \mathbf{sync}(v)\end{array}}{\begin{array}{l}< \ (\Gamma_0(e_0^0), \mathcal{G}id_0^0, 0, \mathcal{E}_{first}^0, \mathcal{E}_{bsp\_p}^0), \cdots,\ (\Gamma_0(e_{l_0}^0), \mathcal{G}id_{l_0}^0, l_0, \mathcal{E}_{first}^0, \mathcal{E}_{bsp\_p}^0), \cdots, \\ (\Gamma_k(e_0^k), \mathcal{G}id_0^k, 0, \mathcal{E}_{first}^k, \mathcal{E}_{bsp\_p}^k), \cdots,\ (\Gamma_k(e_{l_k}^k), \mathcal{G}id_{l_k}^k, l_k, \mathcal{E}_{first}^k, \mathcal{E}_{bsp\_p}^k), \cdots, \\ (\Gamma_k(e_0^h), \mathcal{G}id_0^h, 0, \mathcal{E}_{first}^h, \mathcal{E}_{bsp\_p}^h), \cdots,\ (\Gamma_k(e_{l_h}^h), \mathcal{G}id_{l_h}^h, l_h, \mathcal{E}_{first}^h, \mathcal{E}_{bsp\_p}^h)\ > \\ \rightharpoonup \\ < \ (\Gamma_0(e'^0_0), \mathcal{G}id_0^0, 0, \mathcal{E}_{first}^0, \mathcal{E}_{bsp\_p}^0), \cdots,\ (\Gamma_0(e'^0_{l_0}), \mathcal{G}id_{l_0}^0, l_0, \mathcal{E}_{first}^0, \mathcal{E}_{bsp\_p}^0), \cdots, \\ (\Gamma_k(e'^k_0), \mathcal{G}id_0^k, 0, \mathcal{E}_{first}^k, \mathcal{E}_{bsp\_p}^k), \cdots,\ (\Gamma_k(e'^k_l k), \mathcal{G}id_{l_k}^k, l_k, \mathcal{E}_{first}^k, \mathcal{E}_{bsp\_p}^k), \cdots, \\ (\Gamma_l(e'^l_0), \mathcal{G}id_0^h, 0, \mathcal{E}_{first}^h, \mathcal{E}_{bsp\_p}^h), \cdots,\ (\Gamma_l(e'^l_{l_h}), \mathcal{G}id_{l_h}^h, l_h, \mathcal{E}_{first}^h, \mathcal{E}_{bsp\_p}^h)\ >\end{array}}$$

FIG. 4.5 – Règle de contexte global

$$(\mathbf{bsp\_p}\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (x \leftarrow hd(\mathcal{E}_{bsp\_p})\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.1}$$

$$((\mathbf{fun}\ x \rightarrow\ e)\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (e[x \leftarrow v]\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.2}$$

$$((\mathbf{let}\ x : \tau\ \mathbf{in}\ e)\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (e[x \leftarrow v])\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.3}$$

$$(+(n_1,\ n_2)\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (n\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.4}$$

$$(\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (e_1\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.5}$$

$$(\mathbf{if}\ e\ \mathbf{then}\ e_1\ \mathbf{else}\ e_2\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (e_2\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.6}$$

$$(\mathbf{fix}(op)\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (op\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.7}$$

$$((\mathbf{fix}(\mathbf{fun}\ x \rightarrow e))\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} ((e[x \leftarrow \mathbf{fix}(\mathbf{fun}\ x \rightarrow e)])\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.8}$$

$$(\mathbf{fst}(v_1,\ v_2)\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (v_1\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.9}$$

$$(\mathbf{snd}(v_1,\ v_2)\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (v_2\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.10}$$

$$(\mathbf{mkpar}\ v\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (\overrightarrow{v\mathcal{L}id})\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.11}$$

$$(\mathbf{apply}\ \overrightarrow{v_1}\ \overrightarrow{v_2}\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (\overrightarrow{v_1\ v_2}\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.12}$$

$$(\mathbf{juxta}\ m\ e_1\ e_2\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (\|e_1\|\ ,\mathcal{G}id\ ,\mathcal{L}id,\ hd(\mathcal{E}_{first}) :: \mathcal{E}_{first}, m :: \mathcal{E}_{bsp\_p})\ si\ \mathcal{L}id < m \tag{4.13}$$

$$(\mathbf{juxta}\ m\ e_1\ e_2\ ,\mathcal{G}id\ ,\mathcal{L}id,\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \dot{\rightsquigarrow} (\|e_2\|\ ,\mathcal{G}id\ ,\mathcal{L}id - m,\ (hd(\mathcal{E}_{first} + m)) :: \mathcal{E}_{first}, (hd(\mathcal{E}_{bsp\_p} - m) :: \mathcal{E}_{bsp\_p})\ si\ \mathcal{L}id \geq m \tag{4.14}$$

$$(\|v\|\ ,\mathcal{G}id\ ,\mathcal{L}id,\ (f' :: \mathcal{E}_{first}), (b' :: \mathcal{E}_{bsp\_p})) \dot{\rightsquigarrow} (v\ ,\mathcal{G}id\ ,\ ,\mathcal{G}id - hd(\mathcal{E}_{first}),\ \mathcal{E}_{first},\mathcal{E}_{bsp\_p}) \tag{4.15}$$

$$\tag{4.16}$$

TAB. 4.1 – Règles de réduction locale

$$
\begin{array}{c}
< ((\textbf{get } \overrightarrow{v_0}\ \overrightarrow{n_0}), \mathcal{G}id_0, 0, \mathcal{E}^0_{first}, \mathcal{E}^0_{bsp\_p}), \cdots, ((\textbf{get } \overrightarrow{v_i}\ \overrightarrow{n_i}), \mathcal{G}id_i, i, \mathcal{E}^i_{first}, \mathcal{E}^i_{bsp\_p}), \cdots, ((\textbf{get } \overrightarrow{v_{p'-1}}\ \overrightarrow{n_{p'-1}}), \mathcal{G}id_{p'-1}, p'-1, \mathcal{E}^{p'-1}_{first}, \mathcal{E}^{p'-1}_{bsp\_p}) > \\
\rightsquigarrow \\
< (\overrightarrow{v_{n_0}}, \mathcal{G}id_0, 0, \mathcal{E}^0_{first}, \mathcal{E}^0_{bsp\_p}), \cdots, (\overrightarrow{v_{n_i}}, \mathcal{G}id_i, i, \mathcal{E}^i_{first}, \mathcal{E}^i_{bsp\_p}), \cdots, (\overrightarrow{v_{n_{p'-1}}}), \mathcal{G}id_{p'-1}, p'-1, \mathcal{E}^{p'-1}_{first}, \mathcal{E}^{p'-1}_{bsp\_p}) >\ avec\ p' \ \leq\ p
\end{array}
\tag{4.17}
$$

$$
\begin{array}{l}
< ((\textbf{if } \overrightarrow{v_0} \textbf{ at } n \textbf{ then } e_0 \textbf{ else } e'_0), \mathcal{G}id_0, 0, \mathcal{E}^0_{first}, \mathcal{E}^0_{bsp\_p}), \cdots, ((\textbf{if } \overrightarrow{v_i} \textbf{ at } n \textbf{ then } e_i \textbf{ else } e'_i), \mathcal{G}id_i, i, \mathcal{E}^i_{first}, \mathcal{E}^i_{bsp\_p}), \\
\cdots, ((\textbf{if } \overrightarrow{v_{p'-1}} \textbf{ at } n \textbf{ then } e_{p'-1} \textbf{ else } e'_{p'-1}), \mathcal{G}id_{p'-1}, p'-1, \mathcal{E}^{p'-1}_{first}, \mathcal{E}^{p'-1}_{bsp\_p}) > \qquad \rightsquigarrow \\
< (a_0, \mathcal{G}id_0, 0, \mathcal{E}^0_{first}, \mathcal{E}^0_{bsp\_p}), \cdots, (a_i, \mathcal{G}id_i, i, \mathcal{E}^i_{first}, \mathcal{E}^i_{bsp\_p}), \cdots, (a_{p'-1}, \mathcal{G}id_{p'-1}, p'-1, \mathcal{E}^{p'-1}_{first}, \mathcal{E}^{p'-1}_{bsp\_p})) > \\
avec(\overrightarrow{v_n} \equiv T\ et\ a = e)\ ou\ (\overrightarrow{v_n} \equiv F\ et\ a = e')
\end{array}
\tag{4.18}
$$

$$
\begin{array}{c}
< (\textbf{sync}(v_0), 0, \mathcal{L}id^0, \mathcal{E}^0_{first}, \mathcal{E}^0_{bsp\_p}), \cdots, (\textbf{sync}(v_{p'-1}), p-1, \mathcal{L}id^{p-1}, \mathcal{E}^{p-1}_{first}, \mathcal{E}^{p-1}_{bsp\_p}) > \\
\rightsquigarrow \\
< (v_0, 0, \mathcal{L}id^0, \mathcal{E}^0_{first}, \mathcal{E}^0_{bsp\_p}), \cdots, (v_{p-1}, p-1, \mathcal{L}id^{p-1}, \mathcal{E}^{p-1}_{first}, \mathcal{E}^{p-1}_{bsp\_p}) >
\end{array}
\tag{4.19}
$$

TAB. 4.2 – Règles de réduction fonctionnelle globale

$$
\frac{(e^i,\ \mathcal{G}id^i,\ i,\ \mathcal{E}^i_{first},\ \mathcal{E}^i_{bsp\_p}) \dot{\rightsquigarrow} (e'^i\ ,\ \mathcal{G}id^i,\ i',\ \mathcal{E}'^i_{first},\ \mathcal{E}'^i_{bsp\_p})}{
\begin{array}{c}
< (e^0\ ,\ \mathcal{G}id^0,\ 0,\ \mathcal{E}^0_{first},\ \mathcal{E}^0_{bsp\_p}),\ \cdots,\ (e^i\ ,\ \mathcal{G}id^i,\ i,\ \mathcal{E}^i_{first},\ \mathcal{E}^i_{bsp\_p}),\ \cdots,\ (e^{p-1},\ \mathcal{G}id^{p-1},\ p-1, \mathcal{E}^{p-1}_{first},\ \mathcal{E}^{p-1}_{bsp\_p}) > \\
\rightsquigarrow \\
< (e^0\ \mathcal{G}id^0,\ 0,\ \mathcal{E}^0_{first},\ \mathcal{E}^0_{bsp\_p}),\ \cdots,\ (e'^i\ ,\ \mathcal{G}id^i,\ i',\ \mathcal{E}'^{\ i}_{first},\ \mathcal{E}'^{\ i}_{bsp\_p}),\ \cdots,\ (e^{p-1},\ \mathcal{G}id^{p-1},\ p-1,\ \mathcal{E}^{p-1}_{first},\ \mathcal{E}^{p-1}_{bsp\_p}) >
\end{array}
}
\tag{4.20}
$$

TAB. 4.3 – Permet la compatibilité entre évaluation locale et évaluation globale

Chapitre 5

Transformation

La juxtaposition parallèle brise la sémantique purement fonctionnelle de BSML notamment en transformant la constante p du modèle BSP en variable. Pour faire des preuves de programmes BSML avec juxtaposition parallèle en Coq, il faut donc passer par une phase de transformation du programme contenant la juxtaposition parallèle en un programme sémantiquement équivalent du point de vue fonctionnel (mais avec des performances diffèrentes) qui ne contient pas de juxtaposition parallèle. Le principal objectif de cette transformation est d'obtenir un calcul équivalent à la juxtaposition tout en maintenant un nombre constant de processeurs lors de l'évaluation.

La recherche de cette transformation s'est effectuée en deux étapes. La première étape, présentée à la section 5.1, consiste à trouver une façon de réaliser les même calculs en se passant de la primitive **juxta**. La seconde étape, présentée à la section 5.2, consite à développer une transformation purement fonctionnelle sur cette base.

5.1 Transformation avec effet de bord

La transformation présentée dans cette section n'apporte aucun avantage pour la certification à l'aide de l'assistant de preuve Coq, mais a le mértite de supprimer la la primitive **juxta**.

5.1.1 Principe de la transformation

Le principe de la juxtaposition est de séparer la machine parallèle en sous machines indépendantes, d'où le nombre variable de processeurs. Afin de concilié le principe de la juxtaposition avec les objectifs de la transformation, on dissocie le paramètre p du modèle BSP auquel le programmeur à acces et celui de la machine parallèle. Le programmeur travail maintenant sur un sous réseau virtuel.

Les paramètres du réseau virtuel sont le pid relatif et le nombre de processeurs du réseau.

Le renommage des numéros de processeurs dans la sous machine s'effectue par rapport au pid absolu du premier processeur (**first**) de la sous machine. Le pid relatif s'obtient donc par une soustraction de la valeur du **first** de la valeur du pid absolue. La donnée **first** contient à tout moment la valeur du pid absolue du premier processeur du réseau virtuel dont on fait l'évaluation. Lors de l'initialisation du programme **first** est initialisée à 0 ce qui nous donne un pid relatif identique au pid absolu tant qu'il n'y a pas de **parjux'**. A l'évaluation d'un **parjux**, la valeur du **first** est mise à jours (la même valeur pour la totalité de la machine parallèle).

La fonction renommée **bsp_p**' permet de communiquer au programmeur le nombre virtuel de processeurs auquel son sous réseau à acces. Au début de tout programme la variable **bsp_p**' est initialisée à la taille de la machine parallèle. A l'évaluation d'un **parjux'** la valeur du **bsp_p**' est mise à jours la même valeur pour la totalité de la machine parallèle.

La mise à jours des valeurs est faite à l'aide de piles. A chaque appel de la fonction **parjux'**, la nouvelle valeur est calculée puis empilée sur la pile correspondant. Une fois l'évaluation de la fonction **parjux** terminée, on dépile ; ainsi l'ancienne valeur est restorée. L'acces à la variable, se fait par lecture de la tête de pile.

Nous renommons les primitives globales associées au sous réseau virtuel.

mkpar': (int $\rightarrow$ α) $\rightarrow$ α Bsmllib.**par**
apply': (α $\rightarrow$ β) Bsmllib.**par** $\rightarrow$ α Bsmllib.**par** $\rightarrow$ β Bsmllib.**par**
put': (int $\rightarrow$ α option) Bsmllib.**par** $\rightarrow$ (int $\rightarrow$ α option) Bsmllib.**par**
at': α Bsmllib.**par** $\rightarrow$ int $\rightarrow$ α
parjux': int $\rightarrow$ (unit $\rightarrow$ α Bsmllib.**par**) $\rightarrow$ (unit $\rightarrow$ α Bsmllib.**par**)
$\rightarrow$ α Bsmllib.**par**

FIG. 5.1 – Primitives de la transformation

- **mkpar'** est l'application mkpar renommée. Elle applique la fonction placé en paramètre au pid relatif sur l'ensemble des processeurs du sous réseau virtuel et réduit les calculs extérieurs au sous réseau virtuel en None (let nc = Obj.magic None).
- **apply'** est l'application renommée de apply. Elle applique la fonction placé en paramètre à la valeur placée en paramètre sur l'ensemble des processeurs du sous réseau virtuel et réduit les calculs extérieurs au sous réseau virtuel en None.
- **ifat'** est l'application renommée de ifat. Elle teste la valeur contenue au processeur spécifié d'un vecteur parallèle en tenant compte se son pid relatif et distribue la valeur du test à l'ensemble des processeurs du sous réseau virtuel.(Code source présentée figure **??** dans la sous section 5.1.2).

– **put'** est l'application renommée de get. Elle effectue une opération de communication sur l'ensemble des processeurs du sous réseau virtuel en tenant compte de leurs pids relatifs.(Code source présentée figure 5.5 dans la sous section
5.1.2).

La fonction $parjux'$ est l'élément principal de notre transformation. Elle évalue chacun des deux programmes placés en paramètre sur l'ensemble des processeurs en leur indiquant le reseau virtuel sur lequel les valeurs sont pertinantes. Puis elle retranscrit les valeurs pertinantes des deux évaluations dans un vecteur résultat unique.

```
let parjux' m f1 f2 =
 if (0<m)&&(m<(!jux_p())) then
  let old_p =(!jux_p()) and old_f = (!jux_f) in
  let vb= begin
                jux_f:=old_f+m;
                jux_p:=(fun ()  → old_p−m);
                Bsmllib.apply(Bsmllib.mkpar(fun i ed →  ed)) (f2())
                end in
  let va= begin
                jux_f:=old_f;
                jux_p:=(fun ()  → m);
                Bsmllib.apply(Bsmllib.mkpar(fun i ed →  ed)) (f1())
                end in
   jux_p:=(fun () → old_p);jux_f:=old_f;
   Bsmlbase.apply2 (Bsmllib.mkpar (fun pid a b  →
       if ((!jux_f)<=pid)&&(pid<(!jux_f+m)) then a else
       if (!jux_f+m<=pid)&&(pid<(!jux_f+(!jux_p())))) then b else nc)) va vb
 else raise (Parjux "m is not within_bound")
```

FIG. 5.2 – Code source du **parjux'**

Chaque primitive globale (les primitives globales liées à BSML ainsi que les primitives globales crées par le programmeur) s'exécutant sur un sous réseau virtuel doit être modifiée. En effet celles-ci n'utilisent plus les paramètres systèmes mais les paramètres du réseau virtuel.

5.1.2 Modification des primitives globales

Les primitives globales travaillent sur des vecteurs aux dimension de la machine parallèle. La difficulté est double :

– Travailler sur les éléments pertinants du vecteur.

– Effectuer les calculs avec les bonnes valeurs.

Le contrôle des éléments pertinant se fait par la fonction *inbound*.

```
let inbound pid = (!jux_f<=pid) && ( pid < ( !jux_f + (!jux_p()) ) )
```

FIG. 5.3 – Code source de *i*nbound

Elle recoit en argument le pid absolu du processeur. Après avoir comparé le pid absolu aux paramètres **bsp_p**' et **first** elle renvoie un booléen. Si True alors le processeur appartient au réseau virtuel dont on est en train de faire l'évaluation. C'est donc une valeur pertinante et elle sera calculée. Si False le processeur n'appartient pas au réseau virtuel dont on est en train de faire l'évaluation. Les calculs qui s'y trouvent seront remplacés par None.

```
let mkpar' f =
  Bsmllib.mkpar (fun pid  →
  if (inbound pid) then (f (pid−(!jux_f))) else nc)

let apply' vf vv =
  Bsmlbase.apply2 (Bsmllib.mkpar (fun pid f v  →
  if (inbound pid) then (f v) else nc)) vf vv
```

FIG. 5.4 – Code source des primitives non communicantes

Les primitives de communications sont plus compliquées à gérer. La valeur des paramètres du réseau virtuel peuvent être modifiées au cours des évaluations. Aussi il nous faut sauvegarder ces données à l'entrée de la primitve afin de pouvoir les rétablir à la sortie de celle-ci.

À l'aide de ce code la transformation d'une fonction implantée avec la juxtaposition parallèle consiste à ajouter, dans la fonction, des ' (primes) après chacune des primitives mentionnées dans la transformation et de remplacer **juxta** par **parjux'**.

+

```
let put' vf =
  let old_p =(!jux_p()) and old_f = (!jux_f) in
  let vf'=Bsmllib.put (Bsmllib.apply (Bsmllib.mkpar (fun pid f  →
          if (inbound pid) then
            (fun i → if (inbound i) then (f (i−(!jux_f))) else None)
          else (fun _  → None))) vf) in
  jux_p:=(fun () → old_p);jux_f:=old_f;
  Bsmllib.apply (Bsmllib.mkpar (fun pid f  →
     if (inbound pid) then (fun i  → (f (i+(!jux_f)))) else nc)) vf'

let at' vb n =
  Bsmllib.at vb (n− (!jux_f))
```

FIG. 5.5 – Code source des primitives communicantes

5.2 Transformation purement fonctionnelle

La section précédente établit une méthode viable afin de retirer la primitive **juxta** du programme. Le problème est que cette version utilise des références lors de l'implantation de ses piles. Or les références cassent l'aspect purement fonctionnelle de notre transformation. Cette section établie une méthode afin de contourner l'usage des références pour sauvegarder la valeur de nos paramètres systèmes.

5.2.1 Principe de la transformation

Les primitives globales s'exécutant à l'intérieur d'un sous-réseau virtuel auront toujours besion des valeurs des paramètres systèmes afin de déterminer les éléments pertinants. On a donc placé les paramètres systèmes en argument de toutes les primitives globales s'exécutant dans un sous-réseau virtuel. Ce nouveau paramètre **bound** est une paire dont le premier élément contient la valeur du first et le second celle du nombre de processeur dans le sous-réseau en cours d'évaluation. La difficulté d'une telle approche est de donner le bon paramètre **bound**.

```
let parjux'' bound m f1 f2 =
  if (0<m)&&(m<(snd bound)) then
    Bsmlbase.apply2 (Bsmllib.mkpar (fun pid a b  →
        if ((fst bound)<=pid)&&(pid<((fst bound)+m)) then a else
        if ((fst bound)+m<=pid)&&(pid<((fst bound)+(snd bound)))
         then b else nc))
        (f1 ((fst bound),m) ()) (f2 (m+(fst bound),(snd bound)−m) ())
   else raise (Parjux "m␣is␣not␣within_bound")
```

FIG. 5.6 – Code source de **parjuxt"**

5.2.2 Transformation des primitives

Comme dans la section précédente les primitives doivent être modifiées. On crée donc les primitives dites "bornées" **bsp_p"**, **inbound"**, **mkpar"**, **apply"**, **put"** et **at"** qui sont les primitives modifiées correspondantes respectivement à **bsp_p**, **inbound**, **mkpar**, **apply**, **put** et **at**. Elles sont construites sur le même principe que leurs omologues de la version avec effet de bord à la différence qu'elles acceptent un argument supplémentaire **bound** et qu'elles y récupèrent directement les paramètres **first** et **bsp_p** dont elles ont besoin.

```
let inbound bound pid = ((fst bound)<=pid)&&(pid<((fst bound)+(snd bound)))

let bsp_p'' bound () = (snd bound)

let mkpar'' bound f =
 Bsmllib.mkpar (fun pid  → if (inbound bound pid) then (f (pid−(fst bound)))
                           else nc)
let apply'' bound vf vv =
 Bsmlbase.apply2 (Bsmllib.mkpar (fun pid f v  →
  if (inbound bound pid) then (f v) else nc)) vf vv

let put'' bound vf =
 let vf'=Bsmllib.put (Bsmllib.apply (Bsmllib.mkpar (fun pid f  →
           if (inbound bound pid) then
            (fun i → if (inbound bound i) then (f (i−(fst bound))) else None)
           else (fun _  → None))) vf) in
  Bsmllib.apply (Bsmllib.mkpar (fun pid f  →
      if (inbound bound pid) then (fun i  → (f (i+(fst bound)))) else nc)) vf'

let at'' bound vb n =
  Bsmllib.at vb (n− (fst bound))
```

FIG. 5.7 – Code source des primitives modifiées

5.2.3 Méthode

Tout comme la version avec effet de bord, cette transformation nécessite des modifications de l'ensemble du programme. Supposons pour simplifier qu'un programme BSML est une suite de déclaration de fonctions plus une déclaration finale **let _ = e**

1. Pour toutes les fonctions globales du programme à transformer, sauf les primitives BSML, rajouter un paramètre bound (en premier paramètre) et lors de l'application rajouter un argument "bound".
2. Remplacer toutes les occurrences des primitives de BSML par leurs contreparties bornées.
3. Supprimer **sync**.
4. Transformer la définition principale qui devient
   let _=let bound = (0,p) in e
   qui va finir de lier les occurences libres de "bound"

5.2.4 Example

La transformation du programme utilisant un scan implanté avec juxtaposition parallèle présentée à la section 2.3.2.

```
open Bsmllib
open Bsmlbase
open Transfo

let replicate'' bound x = mkpar'' bound (fun _ → x)
let parfun'' bound f vv = apply'' bound (replicate'' bound f) vv
let parfun2'' bound f vv1 vv2 = apply'' bound (parfun'' bound f vv1) vv2


let rec scan bound op vec=
  if (bsp_p'' bound ())=1 then vec else
  let mid = (bsp_p'' bound ())/2 in
  let vec' = parjux'' bound mid (fun bound () → scan bound op vec)
      (fun bound () → scan bound op vec) in
  let msg bound vec = apply'' bound (mkpar'' bound (fun i v →
    if i=mid−1
    then fun dst →  if dst>=mid then Some v else None
    else fun dst →  None)) vec
  and parop = parfun2'' bound (fun x y → match x with None → y|Some v → op v y) in
  parop (apply'' bound (put'' bound (msg bound vec'))(mkpar'' bound (fun i → mid−1))) vec'

let _ =
  print_string "BSMLlib␣Parallel␣Juxtaposition␣\n";
  initialize();
  let bound =(0 , (Bsmllib.bsp_p())) in
  begin
    let data =
      mkpar (fun x →  1)
    in
    let print x=
      print_string "␣";
      print_int x;
      print_string "\n"
    in
    parfun print (scan bound (+) data)
  end
```

FIG. 5.8 – Example de programme transformé

Chapitre 6

Conclusions

Au travers du cursus de DEA que j'ai suivit à l'Université d'Orléans, j'ai fait la découverte des sémantiques, domaine qui se révèle être pour moi d'un grand intérêt.

Ce stage ma offert l'oportunité d'avoir une expérience très enrichissante de la conception de sémantiques et de la programmation fonctionnelle. La programmation fonctionnelle ainsi que la conception de sémantique étant des dommaines nouveaux pour moi, le stage s'est présenté sous la forme d'un apprentissage de la programmation fonctionnelle et d'une initiation à la conceptioon de sémantique.

Plusieurs objectifs de ce stage ont été atteint : une implantation de la juxtaposition parallèle, une sémantique distribué ainsi qu'une transformation purement fonctionnelle pour la juxtaposition. Il reste tout fois à effectuer des tests sur les machines parallèles du LACL et du LIFO en ce qui concerne l'implantation de la juxtaposition. Du coté de la transformation, il reste à prouver à vérifier puis prouver sa correction.

Bibliographie

[1] AITEC. Contract research projects in 1997 FY. Research Institute for Advanced Information Technology, Tokyo, Japan. `www.icot.or.jp/AITEC/FGCS/funding/itaku-H9-index-E.html`.

[2] F. Gava and F. Loulergue. Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features. Technical Report 2002-14, University of Paris Val-de-Marne, LACL, october 2002.

[3] F. Gava and F. Loulergue. Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *Parallel Computing : Software Technology, Algorithms, Architectures and Applications, Proceeding of the 10th ParCo Conference*, pages 95–102, Dresden, 2004. North Holland/Elsevier.

[4] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. Technical Report 2002-17, University of Paris Val-de-Marne, LACL, november 2002.

[5] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In H. Kosch, L. Boszormányi, and H. Hellwagner, editors, *Euro-Par 2003*, number 2790 in LNCS, pages 781–788. Springer Verlag, 2003.

[6] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics of Minimally Synchronous Parallel ML. Technical Report 2004-07, University of Paris 12, LACL, 2004. In preparation.