Composition parallèle pour MSPML

Radia Benheddi[1] & Frédéric Loulergue[2]

Laboratoire d'Informatique Fondamentale d'Orléans,
Bâtiment IIIA, rue Léonard de Vinci, B.P. 6759
F-45067 ORLEANS CEDEX 2, France
1: benheddi@lifo.univ-orleans.fr
2: loulergu@lifo.univ-orleans.fr

Résumé

Cet article présente l'ajout, à un langage fonctionnel parallèle appelé *Minimally Synchronous Parallel ML* (MSPML), d'une primitive pour la composition parallèle, appelée juxtaposition.

MSPML est un langage fonctionnel parallèle basé sur un petit nombre de primitives sur une structure de données parallèle. Les programmes sont écrits comme des programmes ML usuels en utilisant ce petit ensemble de fonctions. MSPML est déterministe et sans inter-blocage. Le temps d'exécution des programmes peut être estimé. Il a une sémantique asynchrone, c'est-à-dire sans barrière de synchronisation globale. Cette propriété s'avère intéressante pour des programmes non équilibrés à chaque étape.

Toutefois il ne permettait pas d'écrire des algorithmes diviser-pour-règner parallèles qui sont courants dans la littérature. La juxtaposition remédie à cette limitation.

Mots clés

Parallélisme, asynchronisme, sémantique, composition parallèle.

1. Introduction

Le modèle de programmation parallèle *Bulk Synchronous Parallel* (BSP) tel qu'il a été introduit par Valiant en 1990 [28] décrit une architecture parallèle (abstraite), un modèle d'exécution et un modèle de coût. Il consiste en un calcul parallèle de p processeurs dans une séquence de super-étapes.

Une super-étape consiste en:

- une phase de calcul local où chaque processeur travaille sur ses données locales ;
- une phase de communication qui permet l'échange de données entre processeurs ;
- une barrière de synchronisation globale.

Les données échangées ne sont disponibles pour le calcul local qu'après la barrière de synchronisation (donc pour la super-étape suivante).

Dans un but de simplification du modèle de coût, une machine parallèle BSP est caractérisée par seulement trois paramètres (exprimés en multiples de la vitesse des processeurs, dans le cas contraire un quatrième paramètre, la vitesse des processeurs et donnée) qui sont:

- le nombre de processeurs p,
- le temps L nécessaire à la réalisation d'une barrière de synchronisation,

- le temps g pour un échange collectif de messages, appelé 1-relation entre les différentes paires processeur-mémoire dans laquelle chaque processeur envoie et/ou reçoit au plus un mot; le réseau peut réaliser un échange, appelé h-relation (chaque processeur envoie et/ou reçoit au plus h-mots) en temps $h \times g$.

Les principaux avantages du modèle BSP sont:

- l'absence de blocage ainsi que l'absence ou la restriction de l'indéterminisme par la séparation entre la synchronisation et les communications et l'obligation que ces dernières soient des opérations collectives ;
- les preuves de programmes BSP [16] sont proches en complexité du cas séquentiel ;
- la portabilité des performances en ajoutant une notion de processus explicites au parallélisme de données.

Bulk Synchronous Parallel ML (BSML) est une bibliothèque développée pour Objective Caml [18, 8]. Elle permet la programmation data-parallèle basée sur une structure de données parallèle polymorphe. Les programmes sont des fonctions (séquentielles) que l'on peut programmer en Caml mais manipulent cette structure de données parallèle à l'aide d'opérations dédiées. En particulier, l'ordre de lecture et d'exécution BSP sont identiques. Les difficultés de la programmation *Single Program Multiple Data* ou SPMD (comme par exemple en MPI [26]) sont donc supprimées. Puisque BSML suit le modèle d'exécution BSP et est basé sur un calcul confluent [22], les inter-blocages sont impossibles et le déterminisme est garanti. La complexité des preuves des programmes BSML est identique à celle du cas séquentiel [12].

Minimally synchronous Parallel ML (MSPML) [21] est un langage fonctionnel parallèle qui a la même sémantique de haut niveau (modèle de programmation) que BSML mais une sémantique de bas niveau (modèle d'exécution) et une implantation complètement différentes. MSPML possède ainsi, une sémantique asynchrone, c'est-à-dire sans les barrières de synchronisation globale de BSP. Cette propriété s'avère intéressante notamment pour des programmes non équilibrés à chaque étape.

Une autre motivation de la conception de MSPML est la programmation de méta-ordinateurs. Dans [13], MSPML et BSML sont mélangés pour obtenir un nouveau langage parallèle fonctionnel nommé *Departemental Metacomputing ML* (DMML) pour la programmation de grappes de machines parallèles (qui sont elles-mêmes souvent des grappes de PC) qu'on appelle méta-ordinateurs. Plusieurs programmes BSML s'exécutent sur chaque noeud parallèle et sont coordonnés par un programme MSPML. En effet la coordination par un programme BSML s'avèrerait coûteuse : ici les noeuds ne sont plus supposés identiques et le réseau qui les relie est souvent d'efficacité médiocre par rapport aux réseaux internes des noeuds.

Nous nous intéressons ici à la composition de programmes MSPML. On peut bien sûr utiliser la composition de fonctions en MSPML. On ne peut toutefois écrire simplement des programmes parallèles MSPML de type diviser-pour-règner dans lesquels le réseau est partitionné et où chaque partition reçoit un sous-problème à traiter. On peut voir aussi ceci du point de vue de la composition. On ne peut dans MSPML, évaluer deux programmes parallèles sur la même machine en la partitionnant en deux et en évaluant chacun des programmes indépendamment de l'autre sur chaque partition. Nous présentons dans cet article l'ajout d'une primitive de composition parallèle à MSPML.

Cet article est organisé comme suit: on commence par donner une idée du modèle de coût et d'exécution *Message Passing Machine* (MPM) (section 2), puis on donne une présentation informelle de la bibliothèque MSPML sans composition parallèle, de la juxtaposition parallèle et des mécanismes sous-jacents (section 3). Ensuite, on présente la sémantique distribuée de MSPML avec juxtaposition parallèle (section 4). Après des comparaisons avec des travaux connexes (section 5), on termine par des conclusions et perspectives (section 6).

2. Modèle de coûts et d'exécution

Il y a de nombreux programmes parallèles implémentés, en particulier en C et MPI [26], qui ne suivent pas le modèle BSP mais pour lesquels on souhaite pouvoir raisonner sur le coût (si leur structure n'est pas trop complexe). C'est ce qui a conduit au modèle *BSP without barrier* [25] (BSPWB) puis au modèle *Message Passing Machine* (MPM) [24].

BSPWB est un modèle directement inspiré du modèle BSP. Il propose de remplacer la notion de super-étape par la notion de m-étape définie comme suit. À chaque m-étape, chaque processeur effectue une phase de calcul suivie par une phase de communication. Durant la phase de communication, les processeurs échangent les données dont ils ont besoin pour la m-étape suivante. Il n' y a plus de barrière de synchronisation.

La machine parallèle est caractérisée par les trois paramètres suivants (les deux derniers sont exprimés comme multiples de la puissance de calcul des processeurs) : le nombre de processeurs p, la latence L du réseau, le temps g pour échanger un mot entre deux processeurs.

Le temps nécessaire à un processeur i pour exécuter une m-étape s est $t_{s,i}$ borné par T_s le temps nécessaire à l'exécution de la m-étape s par la machine parallèle.

T_s est défini inductivement par :

$$\begin{cases} T_1 = max\{w_{1,i}\} + \max\{g \times h_{1,i} + L\} \\ T_s = T_{s-1} + max\{w_{s,i}\} + max\{g \times h_{s,i} + L\} \end{cases}$$

où $i \in \{0, \ldots, p-1\}$ et $s \in \{2, \ldots, R\}$ où R est le nombre de m-étapes du programme et $w_{s,i}$ et $h_{s,i}$ sont respectivement le temps de calcul local au processeur i durant la m-étape s et $h_{s,i} = max\{h^+_{s,i}, h^-_{s,i}\}$ où $h^+_{s,i}$ (respectivement $h^-_{s,i}$) est le nombre de mots reçus (respectivement envoyés) par le processeur i durant la m-étape s.

Dans ce modèle il y a toutefois une barrière implicite à chaque étape, le coût de la barrière elle-même étant nul. De ce fait ce modèle est une approximation trop grossière. Une meilleure borne $\Phi_{s,i}$ est donnée par le modèle MPM. Les paramètres de ce modèle sont identiques à ceux du modèle BSPWB.

On utilise l'ensemble $\Omega_{s,i}$ pour un processeur i et une m-étape s définie par :

$$\Omega_{s,i} \;=\; \{j|\text{processeur } j \text{ envoie un message au processeur } i \text{ à la m-étape } s\} \bigcup \{i\}$$

Les processeurs de l'ensemble $\Omega_{s,i}$ sont appelés “partenaires entrants” du processeur i à la m-étape s. La borne $\Phi_{s,i}$ est définie inductivement par :

$$\begin{cases} \Phi_{1,i} = \max\{w_{1,j} | j \in \Omega_{1,i}\} + (g \times h_{1,i} + L) \\ \Phi_{s,i} = \max\{\Phi_{s-1,j} + w_{s-1,j} | j \in \Omega_{s,i}\} + (g \times h_{s,i} + L) \end{cases}$$

où $h_{s,i} = \max\{h^+_{s,i}, h^-_{s,i}\}$ pour $i \in \{0, \ldots, p-1\}$ et $s \in \{2, \ldots, R\}$.

Le temps d'exécution pour un programme est donc borné par :

$$\Psi = \max\{\Phi_{R,j} | j \in \{0, 1, \ldots, p-1\}\}$$

Le modèle MPM prend en compte le fait qu'un processeur ne se synchronise qu'avec chacun de ses partenaires entrants et est donc plus précis que BSPWB. Les expériences menées montrent que ce modèle s'applique bien à MSPML [21].

3. Présentation informelle de MSPML

Dans cette section, on présente brièvement le noyau de la bibliothèque MSPML sans juxtaposition et la nouvelle primitive appelée juxtaposition, illustrés par quelques exemples. On présente ensuite le mécanisme de communication dans les environnements de communication. Enfin on termine par l'explication de la numérotation des m-étapes dans MSPML avec juxtaposition.

3.1. Noyau de la bibliothèque

La bibliothèque MSPML est basée sur les primitives données dans la figure 1. Elles donnent l'accès aux paramètres du modèle *Message Passing Machine* de la machine utilisée. En particulier, la fonction **p** retourne le nombre statique de processeurs de la machine parallèle. Cette valeur ne change pas durant l'exécution, tant que la juxtaposition n'est pas introduite. Il y a aussi un type abstrait polymorphe α **par** qui représente le type des vecteurs parallèles de longueur p d'objets de type α, un seul objet par processeur. Le non emboîtement des expressions parallèles peut être assurée par un système de types [21].

p : unit → int
g : unit → float
l : unit → float
mkpar : (int → α) → α **par**
apply : (α → β) **par** → α **par** → β **par**
get : α **par** → int **par** → α **par**
mget : (int → α) **par** → (int → bool) **par** → (int → α option) **par**
at : α **par** → int → α

Figure 1: Le noyau de la bibliothèque MSPML

Les constructeurs parallèles opèrent sur les vecteurs parallèles. Ces vecteurs parallèles sont crées par la primitive **mkpar**, ainsi, (**mkpar** f) stocke $(f i)$ dans le processeur i pour i entre 0 et $(p-1)$. On écrit habituellement **fun** $pid \rightarrow e$ pour f afin de montrer que l'expression e peut être différente sur chaque processeur. Cette expression est appelée locale : elle est à l'intérieur de la fonction **mkpar** et sa valeur dépend du processeur local sur lequel elle se trouve. L'expression (**mkpar** f) est un objet parallèle global. Par exemple l'expression **mkpar**(**fun** $pid \rightarrow pid$) sera évaluée en un vecteur parallèle $\langle\, 0 \,, \ldots ,\, p-1 \,\rangle$.

Dans le modèle MPM, un algorithme est écrit comme étant une combinaison entre des calculs locaux asynchrones et des phases de communication. Les phases asynchrones sont programmées avec **mkpar** et **apply**. Par exemple, l'expression **apply** (**mkpar** f) (**mkpar** e) stocke $(f i)$ $(e i)$ dans le processeur i.

Les phases de communication sont réalisées par **get** et **mget**. La sémantique du **get** est donnée par : **get**$\langle\, v_0 \,, \ldots ,\, v_{p-1} \,\rangle\langle\, i_0 \,, \ldots ,\, i_{p-1} \,\rangle = \langle\, v_{i0\%p} \,, \ldots ,\, v_{ip-1\%p-1} \,\rangle$ où % est le modulo.

La fonction **mget** est une généralisation qui permet d'avoir les données à partir de différents processeurs durant la même m-étape et de délivrer différents messages à de différents processeurs. La sémantique de la fonction **mget** est: **mget**$\langle\, f_0 \,, \ldots ,\, f_{p-1} \,\rangle\langle\, b_0 \,, \ldots ,\, b_{p-1} \,\rangle = \langle\, g_0 \,, \ldots ,\, g_{p-1} \,\rangle$ où $g_i =$ **fun** $j \rightarrow$ **if** $b_i\ j$ **then** Some $(f_j\ i)$ **else** None

Le langage complet contient aussi une conditionnelle globale : **if** e **at** n **then** e_1 **else** e_2. Selon la valeur du vecteur parallèle au processeur donné par la valeur n l'expression est évaluée en v_1 (la valeur obtenue par l'évaluation de e_1) ou en v_2 (la valeur obtenue par l'évaluation de e_2). Mais Objective Caml est un langage strict[1] ainsi cette opération conditionnelle globale ne peut être définie comme

[1]Objective Caml est un langage strict utilisant la stratégie faible d'appel par valeur, c'est-à-dire que tous les arguments

une fonction[2]. Pour cela, la bibliothèque MSPML contient la fonction **at**[3] pour être utilisée dans les constructions suivantes : **if at** e n **then** ... **else** ..., **match**(**at** e n) **with**...

at exprime la phase de communication. La conditionnelle globale est nécessaire pour exprimer des algorithmes tels que: **Repeat** Itération Parallèle **Until** erreur locale maximale $< \epsilon$. Sans le **at**, le contrôle global ne peut prendre en compte les données calculées localement.

3.2. Coûts

Le temps d'exécution d'un programme MSPML est représenté par un vecteur du temps d'exécution sur chaque processeur :$\langle\ c_0\ , \ldots,\ c_{p-1}\ \rangle$. Si l'évaluation d'une expression ML e en dehors d'un **mkpar**, nécessite (séquentiel) le temps w alors son évaluation en tant que programme MSPML ajoutera w à chaque composante du vecteur de coûts : $\langle c_0 + w, \ldots, c_{p-1} + w\rangle$

L'évaluation d'un **mkpar** nécessite w_i au processeur i, le temps nécessaire pour l'évaluation de $(f\,i)$.

Pour **apply** le temps nécessaire est celui de l'évaluation de ces arguments donnant les vecteurs $\langle\ f_0\ , \ldots,\ f_{p-1}\ \rangle$ et $\langle\ v_0\ , \ldots,\ v_{p-1}\ \rangle$ puis au processeur i, le temps w_i d'évaluation de $(f_i v_i)$.

Enfin pour un **get**, si le vecteur de coûts après évaluation de ses arguments est : $\langle\ c_0\ , \ldots,\ c_{p-1}\ \rangle$, on obtiendra le vecteur de coûts $\langle\ c'_0\ , \ldots,\ c'_{p-1}\ \rangle$ définie par :

- soit $\langle\ v_0\ , \ldots,\ v_{p-1}\ \rangle$ le premier argument, on note $\#v_i$ la taille d'une valeur,
- soit $\langle\ i_0\ , \ldots,\ i_{p-1}\ \rangle$ le second argument du **get**,
- soit Ω_k les partenaires entrants du processeur k, $\Omega_k = \{j | i_j = k\} \cup \{k\}$, et
- $c'_k = max\{c_i | i \in \Omega_k\} + max\{\sum_{j \in \Omega_k \setminus \{k\}} \#v_j, \#v_{j_k} \text{ si } j_k <> k\} \times g + L$.

3.3. Exemples

On présente maintenant quelques exemples qui font partie de la bibliothèque standard de MSPML.

3.3.1. Fonctions usuelles

Les fonctions usuelles sont définies en utilisant uniquement les primitives. Par exemple la fonction **replicate** crée des vecteurs parallèles qui contiennent la même valeur partout. La primitive **apply** ne peut être utilisée que dans le cas des vecteurs parallèles de fonctions qui prennent un seul argument. Pour les fonctions à deux arguments on a besoin de définir la fonction **apply2**.

```
(* val replicate: α → α par *)
let replicate x = mkpar(fun pid→ x)

(* val apply2:( α → β → γ ) par → α par → β par → γ par *)
let apply2 f v1 v2 = apply(apply f v1) v2
```

Il est aussi très commode d'appliquer la même fonction séquentielle à chaque processeur. Cela peut être fait en utilisant les fonctions **parfun**. Elles diffèrent seulement dans le nombre de leurs arguments :

des fonctions doivent tous être évalués avant l'évaluation des fonctions

[2] Compte tenu de la stratégie d'évaluation, si **ifat** a été défini comme étant une fonction les deux branches pour le **true** et le **false** devaient être évaluées.

[3] Une autre possibilité aurait pu être choisie, c'est de définir **ifat** comme étant une fonction ainsi que ses deux derniers arguments afin d'éviter leur évaluation, comme nous le faisons pour la juxtaposition.

```
(* val parfun: ( α → β )par→ α par→ β par *)
let parfun f v = apply (replicate f) v

(* val parfun2: ( α → β → γ ) par → α par → β par → γ par *)
let parfun2 f v1 v2 = apply(parfun f v1) v2
```

3.3.2. Fonctions de communication

La sémantique de la fonction d'échange totale est donnée par:

$$\mathsf{totex}\langle\, v_0 \,, \ldots,\, v_{p-1} \,\rangle = \langle\, f \,, \ldots,\, f \,, \ldots,\, f \,\rangle \text{ où } \forall i.(0 \le i < p-1) \Rightarrow\ (f\ i) = v_i$$

Le code est présenté ci-dessous où, **noSome** enlève le constructeur **Some** et **compose** est la composition de fonctions :

```
(* val totex: α par → (int → α ) par *)
let totex vv = (parfun compose noSome)(mget (parfun(fun v i→ v)vv)(replicate(fun i → true)))
```

Son coût parallèle est $(p-1) \times s \times g + L$, où s dénote la taille en mots de la plus grande valeur v qui se trouve sur un certain processeur n. À partir de la fonction d'échange total, on peut obtenir une version qui retourne un vecteur parallèle de listes, où procs() = [0;. . . ; **p**()-1] :

```
(* val totex_list: α par → α list par *)
let totex_list v = (parfun2 List.map(totex v))(replicate(procs()))
```

La sémantique de la diffusion est: bcast $\langle\, v_0 \,, \ldots,\, v_{p-1} \,\rangle\ r \;=\; \langle\, v_{r\%p} \,, \ldots,\, v_{r\%p} \,\rangle$

La fonction broadcast_direct qui réalise la diffusion peut être écrite comme suit :

```
(* bcast_direct: int → α par → α par *)
let bcast_direct root vv = get vv (replicate root)
```

Son coût parallèle est $(p-1) \times s \times g\ +\ L$, où s dénote la taille en mots de la valeur v_n qui se trouve sur le processeur n.

La bibliothèque standard de MSPML contient une collection de ces fonctions qui facilitent l'écriture de programmes. Ainsi, il est similaire d'écrire des programmes MSPML ou d'écrire des programmes en utilisant des patrons data-parallèles, mais, avec MSPML il est possible d'écrire ses propres patrons comme étant des fonctions de haut niveau si la bibliothèque standard ne fournit pas les fonctions nécessaires.

Quelques fonctions de la bibliothèque standard sont récursives. Par exemple, il existe une fonction de diffusion qui est évaluée en $\log p$ m-étapes au lieu d'une m-étape :

```
let bcast_logp root vv =
  let from n = mkpar(fun i→ let j=natmod (i+(p())−root) (p()) in
                     if (n/2<=j)&&(j<n) then i−(n/2) else i) in
  let rec aux n vv = if n<1 then vv else get (aux (n/2) vv) (from n)
  in aux (p()) vv
```

3.4. Une opération de composition parallèle : la juxtaposition

La composition parallèle spatiale que l'on appelle *juxtaposition* permet de subdiviser la machine en deux sous-machines, ce qui permet l'évaluation de deux programmes parallèles indépendemment l'un de l'autre sur une même machine. L'évaluation du terme (**juxta** m E_1 E_2), se déroule comme suit : les

m premiers processeurs évaluent le terme E_1 et les $p-m$ restant évaluent E_2. Ces $p-m$ processeurs sont toutefois renommés, le processeur m devenant 0 et le $(p-1)$-ième devenant $(p-1)-m$.

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

$$\textsf{\textbf{juxta}}\ m\ \langle\ v_0\ ,\ldots,\ v_{m-1}\ \rangle\ \langle\ v'_0\ ,\ldots,\ v'_{p-1-m}\ \rangle = \langle\ v_0\ ,\ldots,\ v_{m-1}, v'_0\ ,\ldots,\ v'_{p-1-m}\ \rangle$$

Au niveau de la bibliothèque MSPML, Objectif Caml étant un langage dont la stratégie d'évaluation est une stratégie faible d'appel par valeur, pour éviter que les deux derniers arguments de la fonction **juxta** ne soient évalués il faut qu'ils soient des fonctions :

$$\textsf{\textbf{juxta}}\textsf{: int} \to (\textsf{unit} \to\ \alpha\ \textsf{\textbf{par}}) \to (\textsf{unit} \to\ \alpha\ \textsf{\textbf{par}}) \to\ \alpha\ \textsf{\textbf{par}}.$$

L'ajout de l'opération de juxtaposition ne modifie en rien le modèle du coût MPM présenté dans la section 2, et le coût de la juxtaposition sera le coût d'évaluation des deux derniers arguments de la juxtaposition. Sachant que le numéro de m-étape cette fois-ci ne sera plus un entier naturel mais suit la grammaire "step" décrite dans la section 3.5.1 L'exemple suivant (Figure 2) est une version diviser-pour-règner du calcul des préfixes en parallèle, appelé scan. Sa sémantique est définie par :

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

où $\oplus$ est une opération binaire associative.

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

Figure 2: Calcul des préfixes avec la juxtaposition parallèle

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.

3.5. Le mécanisme de communication

L'asynchronisme de MSPML a conduit à baser le mécanisme de communication sur un moyen de stockage des valeurs de chaque m-étape dans le cas où un processeur distant pourrait en avoir besoin ultérieurement. C'est pour cela que les environnements de communication ont été introduits. Un environnement de communication peut être vu comme une liste d'association qui relie les nombres de m-étapes avec les valeurs détenues par ces processus à ces m-étapes.

Durant l'exécution d'un programme MSPML, pour chaque processus i, le système possède une variable $mstep_i$ contenant un entier désignant le nombre actuel de m-étapes. À chaque fois une expression **(get vv vi)**, est évaluée à un processus i donné :

1. mstep_i est incrémentée;
2. le couple contenant: la valeur que tient ce processus dans le vecteur parallèle vv et la valeur de mstep_i est sauvegardée dans l'environnement de communication;

3. la valeur j que tient ce processus dans le vecteur parallèle vi désigne le numéro de processus duquel le processus i veut recevoir une valeur. Ainsi, le processus i envoie une requête au processus j : il demande la valeur à la m-étape mstep_i. Quand le processus j reçoit la requête (des *threads* sont dédiés au traitement de telles requêtes, donc, le travail du processus j n'est pas interrompu), deux cas se présentent:

   - $\mathrm{mstep}_j \geq \mathrm{mstep}_i$: cela veut dire que le processus j a déjà atteint la même m-étape que le processus i. Ainsi, le processus j accède dans son environnement de communication à la valeur associée à la m-étape mstep_i et l'envoie au processus i.
   - $\mathrm{mstep}_j < \mathrm{mstep}_i$: rien ne peut se faire jusqu'à ce que le processus j atteigne la même m-étape que le processus i.

Si $i = j$, l'étape 3 n'est pas exécutée.

Sans juxtaposition tous les processus exécutent le même nombre de m-étapes ce qui n'est pas le cas avec la juxtaposition. Par exemple, dans l'expression suivante :

$$\textsf{\textbf{let}}\ \mathit{this} = \textsf{\textbf{mkpar}}\,(\textsf{\textbf{fun}}\ i \to\ i)\ \textsf{\textbf{in juxta}}\ 2\ (\textsf{\textbf{get}}\ \mathit{this}\ \mathit{this})\ \mathit{this}$$

Les deux premiers processeurs du réseaux sont les seuls à incrémenter leurs numéros de m-étape en réalisant la communication à l'évaluation du **get**, d'où la nouvelle numérotation présentée dans la section suivante. Le mécanisme de communication dans MSPML avec juxtaposition a le même principe que celui présenté ci-dessus. L'ordre sur les nouveaux numéros est simplement plus compliqué.

3.5.1. Numérotation des m-étapes

Le numéro de m-étape entier ne suffit plus pour distinguer entre les différents messages des différentes sous machines, d'où la numérotation donnée par la grammaire suivante :

$$\mathrm{step} ::= (n, m)\ \mid\ \mathrm{step.R}(n, m)\ \mid\ \mathrm{step.L}(n, m) \qquad \text{où } n \text{ et } m \text{ sont des naturels.}$$

Les appels à la juxtaposition peuvent former une hiérarchie, ce qui apparaît dans la numérotation avec la notation pointée suivie de R (resp. L) pour indiquer que l'on se place dans le sous-réseau gauche (resp. droit) :

$$(\textsf{\textbf{juxta}}\ m\ (\textsf{\textbf{juxta}}\ m'\ (\textsf{\textbf{juxta}}\ m''\ \ldots\ \ldots)\ \ldots)\ \ldots)$$

Mais on peut avoir aussi des appels successifs à des juxtapositions :

$$\textsf{\textbf{let}}\ e_1 = (\textsf{\textbf{juxta}}\ m\ \ldots\ \ldots)\ \textsf{\textbf{in let}}\ e_1 = (\textsf{\textbf{juxta}}\ m\ \ldots\ \ldots)\ \textsf{\textbf{in let}}\ e_1 = (\textsf{\textbf{juxta}}\ m\ \ldots\ \ldots)\ \textsf{\textbf{in}}\ \ldots$$

Ainsi on a besoin des couples (n, m) qui sont composés :

- du nombre n d'étapes de communication, incrémenté à chaque appel d'un **get**, **mget** ou **at**
- du nombre de juxtaposition effectuées m

pour le niveau hiérarchique considéré. L'ordre est lexicographique sur la suite des couples et lexicographique inverse sur chaque couple. Les m-étapes de sous-réseaux différents ne sont pas comparables.

La figure 3 présente un exemple illustrant la description de la numérotation des m-étapes.

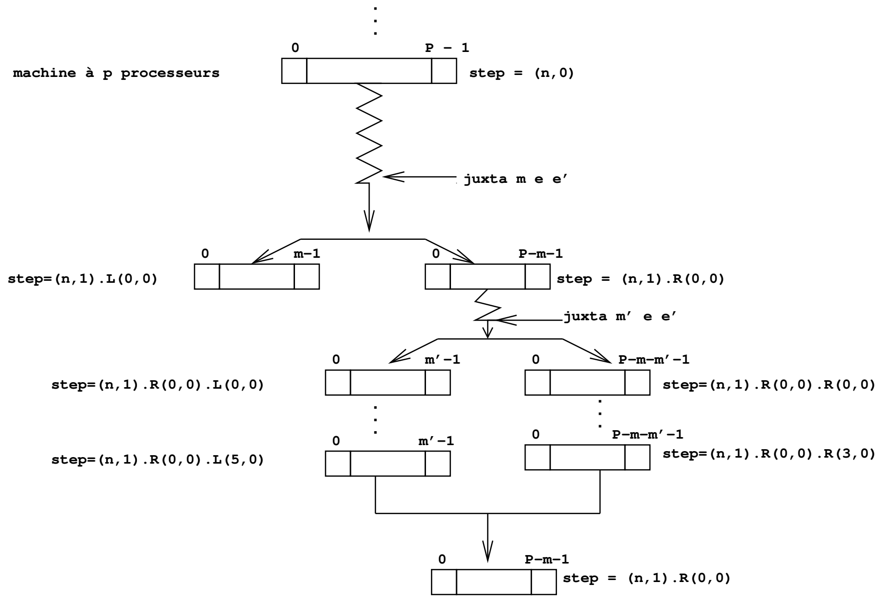

Figure 3: Numérotation des m-étapes dans une juxtaposition

4. Sémantique distribuée de MSPML avec juxtaposition

MSPML à deux sémantiques. La première est la sémantique à grands pas qui correspond au modèle de programmation, elle ne donne pas les détails de calcul mais juste le résultat. Par conséquent, tous les opérateurs parallèles semblent être synchrones dans cette sémantique. Un programme MSPML est vu comme un programme séquentiel Objective Caml classique avec l'utilisation de temps en temps d'opérations sur une structure de données parallèle. De ce point de vue, le modèle de programmation de MSPML est très semblable à celui de BSML, la différence essentielle étant finalement le coût parallèle que l'on attribue à chacune des primitives. Pour montrer comment la désynchronisation est manipulée dans MSPML, une sémantique distribuée, qui donne les étapes de la réduction vers une valeur est donc nécessaire. Cette sémantique est proche de l'implantation et formalise le modèle d'exécution de MSPML. Un programme MSPML est dans ce cas vu comme un programme SPMD : c'est en fait p copies d'un même programme, une par processeur de la machine parallèle, qui travaillent chacun sur "une tranche" de la structure parallèle, ici une valeur du vecteur parallèle. Ces copies communiquent par passage de message. Les primitives décrites dans la section 3.1, se trouve dans un module appelé MSPML qui utilise le module " Tcpip" qui offre un petit ensemble de fonctions similaires à celles de MPI, implémenté en utilisant le module Unix d'Objective Caml. Par conséquent, le module MSPML est écrit dans le style SPMD. Les deux sémantiques, à grands pas et distribuée sont équivalente. On se contente ici de présenter la sémantique distribuée.

4.1. Syntaxe

Dans la syntaxe des deux sémantiques de MSPML, nous ne distinguons pas expressions locales et globales de la même façon que dans le BSλ-calcul [19]. Nous distinguons simplement les variables locales des variables globales. Grâce à un système de typage, omis ici mais présenté au complet dans [4], on retrouve la distinction entre expressions locales et expressions globales. Ce système permettant d'éviter l'emboîtement parallèle.

$$
\begin{array}{lllll}
e ::= & x & \mid c & \mid \textbf{fun}\ x:\tau \rightarrow e & \mid \textbf{let}\ x:\tau\ \textbf{in}\ e \\
& \mid op & \mid (e\ e) & \mid (e, e) & \mid \textbf{if}\ e\ \textbf{then}\ e\ \textbf{else}\ e \\
& \mid \textbf{fst}\ e & \mid \textbf{snd}\ e & \mid \textbf{mkpar}\ e & \mid \textbf{apply}\ e\ e \\
& \mid \textbf{get}\ e\ e & \mid \textbf{request}\ e\ e & \mid \textbf{juxta}\ e\ e\ e & \mid \overrightarrow{e} \\
& \mid \|e_d\| & \mid \textbf{if}\ e\ \textbf{at}\ e\ \textbf{then}\ e\ \textbf{else}\ e & &
\end{array}
$$

Figure 4: La syntaxe du noyau du langage

La syntaxe est celle présentée dans la figure 4, où on trouve :

- les expressions fonctionnelles classiques ;
- les opérations parallèles présentées à la section précédente, dont **juxta** $e\ e\ e$ la juxtaposition parallèle ;
- $\overrightarrow{e}$ qui représente l'expression locale d'un vecteur parallèle au processeur qui la contient ("la tranche" de ce processeur), car il est important de savoir à un moment donné qu'un terme provient d'un vecteur et non pas d'une expression séquentielle, pour éviter notamment l'emboîtement d'expressions parallèles.
- **request** $e\ e$ qui représente la requête d'une valeur d'un processeur donné (premier argument) à une m-étape donnée (second argument) ;
- $\|e_d\|$ représente l'expression e dans une sous-machine, indiquant que cette expression est évaluée dans le cadre d'une juxtaposition.

4.2. Évaluations

Les valeurs de la réduction locale sont les suivantes : $v \ ::= \ \textbf{fun}\ x \rightarrow \ e \mid c \mid op \mid (v,\ v) \mid \overrightarrow{v}$
Il faut bien noter que les expressions **request** (qui sont le résultat d'un **get**) et $\|v_d\|$*ne sont pas des valeurs.*

4.2.1. Évaluation locale

L'ajout de la juxtaposition parallèle rend le nombre de processeurs et les identifiants de processeurs variables. Pour gérer ce dynamisme chaque processeur contient deux piles : $\mathcal{E}_p$ et $\mathcal{E}_{pid}$ contenant respectivement le nombre de processus et l'identifiant du processus du réseau auquel appartient le processus. L'environnement $\mathcal{E}_p$ (resp.$\mathcal{E}_{pid}$) est initialisé au début par le nombre de processeurs (resp. l'identifiant absolu).

Les règles sont de la forme $(e, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \ \rightharpoonup_{\mathrm{i}} \ (e', step', \mathcal{E}'_c, \mathcal{E}'_p, \mathcal{E}'_{pid})$ et peuvent être lues comme suit :

> "Au processeur hd($\mathcal{E}_{pid}$)[4]dans le réseau à hd($\mathcal{E}_p$) processeurs, dont le *pid* absolu est i, et à la m-étape step l'expression e dans l'environnement de communication $\mathcal{E}_c$ est réduite *localement* à l'expression e' avec des changements possibles dans le numéro de m-étape, les diverses piles et l'environnement de communication $\mathcal{E}'_c$".

[4]L'environnement $\mathcal{E}_{pid}$ (resp.$\mathcal{E}_p$) est une pile dont le sommet est désigné par hd($\mathcal{E}_{pid}$) (resp. hd($\mathcal{E}_p$))

$$
\begin{array}{lllll}
\Gamma & ::= & [] & \mid \; \Gamma\ e & \mid \; v\ \Gamma & \mid \; \textsf{fst}\ \Gamma \\
& \mid & \textsf{snd}\ \Gamma & \mid \; \textbf{let}\ x:\tau = \Gamma\ \textbf{in}\ e & \mid \; \overrightarrow{\Gamma} & \mid \; \textsf{if}\ \Gamma\ \textsf{then}\ e\ \textsf{else}\ e \\
& \mid & \textsf{mkpar}\,\Gamma & \mid \; \textsf{apply}\,\Gamma\, e & \mid \; \textsf{apply}\, v\, \Gamma & \mid \; \textsf{get}\,\Gamma\, e \\
& \mid & \textsf{get}\, v\, \Gamma & \mid \; \textsf{if}\,\Gamma\,\textsf{at}\, e\,\textsf{then}\, e\,\textsf{else}\, e & \mid \; \textsf{if}\, v\,\textsf{at}\,\Gamma\,\textsf{then}\, e\,\textsf{else}\, e & \mid \; \textsf{juxta}\ \Gamma\ e\ e \\
& \mid & \|\Gamma\| & & &
\end{array}
$$

Figure 5: Les contextes d'évaluation de la sémantique distribuée

La réduction locale est subdivisée en trois groupes de règles : les contextes et la règle de contexte, la réduction fonctionnelle qui correspond à une sémantique à petits pas classique et enfin, la réduction pour les opérations parallèles spécifiques à MSPML.

La réduction de tête ne peut être appliquée dans n'importe quel contexte. On rappelle que la stratégie choisie est la stratégie faible d'appel par valeur. Donc, Les contextes présentés dans ci-dessus définissent l'ordre d'évaluation des arguments pour chaque terme. Les contextes forcent l'évaluation des arguments (de gauche à droite) avant de permettre la réduction. Les contextes sont appliqués en utilisant la règle de contexte (1), afin d'évaluer tous les arguments d'une expression avant toute réduction locale. [] est un trou qui peut être rempli par n'importe quelle expression e[5].

$$
\frac{(e_i, step_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}) \ \rightharpoonup_{\mathrm{i}} \ (e'_i, step'_i, \mathcal{E}'_{c_i}, \mathcal{E}'_{p_i}, \mathcal{E}'_{pid_i})}{(\Gamma(e_i), step_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}) \ \rightharpoonup_{\mathrm{i}} \ (\Gamma(e'_i), step'_i, \mathcal{E}'_{c_i}, \mathcal{E}'_{p_i}, \mathcal{E}'_{pid_i})} \tag{1}
$$

Les règles de la réduction fonctionnelle, ne changent que le premier composant du tuple et sont finalement des règles à petits pas très classiques. L'ensemble des règles est présenté en détails dans [4]. Soit par exemple la règle de réduction pour la liaison :

$$
((\textsf{let}\ x:\tau = v\ \textsf{in}\ e), step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup_{\mathrm{i}} \quad (e[x \leftarrow v], step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \tag{2}
$$

La règle (3) permet la création de vecteurs parallèles énumérés. $(\overrightarrow{v\ i'})$ est un morceau de vecteur parallèle énuméré qui contient v dans le processeur i.

$$
(\textsf{mkpar}\, v, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup_{\mathrm{i}} \quad ((\overrightarrow{v\ i'}), step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \text{ avec } i' = hd(\mathcal{E}_{pid}) \tag{3}
$$

La règle (4) est une règle parallèle classique :

$$
\left(\textsf{apply}\,\overrightarrow{v_1}\,\overrightarrow{v_2}, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}\right) \quad \rightharpoonup_{\mathrm{i}} \quad \left(\overrightarrow{v_1\ v_2}, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}\right) \tag{4}
$$

Les règles de communication concernent le **get** et **at**, on donne le fonctionnement du **get** dans ce qui suit. Lors de l'évaluation d'un **get**, la valeur est mise dans l'environnement de communication et le **get** devient un **request**, si la communication n'est pas à destination du processus qui évalue :

$$
(\textsf{get}\,\overrightarrow{v}\,\overrightarrow{j}, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup_{\mathrm{i}} \quad \begin{array}{l} (\overrightarrow{\textsf{request}\ step'\ \ abs\_pid_j}, step', (step', v) :: \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \\ \text{si } hd(\mathcal{E}_{pid}) \neq j \\ \text{avec } abs\_pid_j = (i - hd(\mathcal{E}_{pid})) + (j\% hd(\mathcal{E}_p)) \\ \text{et } step' = \mathrm{inc}_c(step) \end{array} \tag{5}
$$

Sinon il y a simplement mise de la valeur dans l'environnement de communication :

$$
(\textsf{get}\,\overrightarrow{v}\,\overrightarrow{i}, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup_{\mathrm{i}} \quad (\overrightarrow{v}, \mathrm{inc}_c(step), (\mathrm{inc}_c(step), v) :: \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \tag{6}
$$

[5]Un contexte n'a qu'une occurrence de []

avec inc_c définie par :

$$\begin{cases}\mathrm{inc}_c((n,m)) & = (n+1,m)\\ \mathrm{inc}_c(step.\mathrm{D}(n,m)) & = step.\mathrm{D}(n+1,m) \text{ où } D = L \text{ ou } D = R\end{cases}$$

Pour que les échanges se fassent, il faut considérer l'évaluation globalement et non plus localement, ce qui est fait à la sous-section suivante.

À l'évaluation du terme **juxta** $m\ e1\ e2$, la machine absolue est subdivisée en deux sous-machines comme suit. Soit le numéro de processeur dans le contexte du sous-réseau est inférieur au premier argument de la juxtaposition :

$$(\textbf{juxta}\ m\ e_1\ e_2, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup_{\mathrm{i}} \quad \begin{array}{l}\left(\|e_1\|, \mathrm{inc}_j(step).L(0,0), \mathcal{E}_c, m :: \mathcal{E}_p, pid :: \mathcal{E}_{pid}\right)\\ \text{avec } pid = hd(\mathcal{E}_{pid}) \text{ et si } pid < m\end{array} \tag{7}$$

soit il est supérieur :

$$(\textbf{juxta}\ m\ e_1\ e_2, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup_{\mathrm{i}} \quad \begin{array}{l}\left(\|e_2\|, \mathrm{inc}_j(step).R(0,0), \mathcal{E}_c, (p-m) :: \mathcal{E}_p, (pid-m) :: \mathcal{E}_{pid}\right)\\ \text{avec } pid = hd(\mathcal{E}_{pid}) \text{ et si } pid \geq m\end{array} \tag{8}$$

où inc_j a le même effet que inc_c mais sur la seconde composante du couple.

Dans les deux cas on incrémente la seconde composante du dernière couple du numéro de m-étape et on empile un nouveau couple dans le numéro de m-étape précédé de l'indication du sous-réseau. On empile le nombre de processeurs du sous-réseau et le nom du processeur. On place également l'expression à évaluer dans $\|\cdot\|$ pour indiquer que cette expression s'évalue dans le contexte d'une juxtaposition.

Lorsque l'évaluation de l'expression est terminée, celle de la juxtaposition aussi et on “dépile” le dernier couple du numéro de m-étape, on dépile également les piles des nombres de processeurs et des noms de processeur :

$$(\|v\|, step.D(n,m), \mathcal{E}_c, p' :: \mathcal{E}_p, pid :: \mathcal{E}_{pid}) \quad \rightharpoonup_{\mathrm{i}} \quad (v, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \text{ où D=L ou D=R} \tag{9}$$

4.2.2. Évaluation globale

La réduction globale $\rightarrow$ est une relation sur les vecteurs distribués[6]. Le passage au contexte global se fait par application de la réduction locale réalisée par la règle suivante :

$$\frac{(e_{d_i}, step_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}) \quad \rightharpoonup_{\mathrm{i}} \quad (e'_{d_i}, step'_i, \mathcal{E'}_{c_i}, \mathcal{E'}_{p_i}, \mathcal{E'}_{pid_i})}{\begin{array}{c}\ll (e_{d_0}, step_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}), \ldots, (e_{d_i}, step_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}), \ldots,\\ (e_{d_{p-1}}, step_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}) \gg\\ \rightarrow\\ \ll (e_{d_0}, step_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}), \ldots, (e'_{d_i}, step'_i, \mathcal{E'}_{c_i}, \mathcal{E'}_{p_i}, \mathcal{E'}_{pid_i}), \ldots,\\ (e_{d_{p-1}}, step_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}) \gg\end{array}} \tag{10}$$

L'échange de messages entre deux processeurs est modélisé donc par la règle suivante :

$$\frac{(e_{d_i} = \Gamma[\textbf{request}\, n\ \ j]) \text{ et } ((n,v) \in \mathcal{E}_{c_j}}{\begin{array}{c}\ll (e_{d_0}, step_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}), \ldots, (e_{d_i}, step_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}), \ldots,\\ (e_{d_{p-1}}, step_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}) \gg\\ \rightarrow\\ \ll (e_{d_0}, step_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}), \ldots, (\Gamma[v], step_i, \mathcal{E}_{c_i}, \mathcal{E}_{p_i}, \mathcal{E}_{pid_i}), \ldots,\\ (e_{d_{p-1}}, step_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}) \gg\end{array}} \tag{11}$$

[6]Les vecteurs distribués sont notées :
$\ll (e_{d_0}, step_0, \mathcal{E}_{c_0}, \mathcal{E}_{p_0}, \mathcal{E}_{pid_0}), \ldots, (e_{d_{p-1}}, step_{p-1}, \mathcal{E}_{c_{p-1}}, \mathcal{E}_{p_{p-1}}, \mathcal{E}_{pid_{p-1}}) \gg$

La règle (11) signifie que si un processeur i demande la valeur détenue par un processeur j à la m-étape n **request** et que l'environnement de communication $\mathcal{E}_{c_j}$ du processeur j contient la valeur v à la m-étape n alors la valeur v est envoyé au processeur i. Sinon la règle ne peut pas être appliquée : si le processeur j n'a pas atteint la m-étape numéro n, alors le processeur i doit attendre.

Pour cette sémantique on a obtenu les résultats suivants :

Proposition 1 (Sûreté du typage) *Soit e une expression MSPML close et bien formée. Si $E \vdash e : \tau$ et $(e, step, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \rightharpoonup_i (v, step', \mathcal{E}'_c, \mathcal{E}'_p, \mathcal{E}'_{pid})$ alors $E \vdash v : \tau$.*

Proposition 2 (Confluence de la réduction globale) *Soit E un vecteur MSPML distribuée. Si $E \rightarrow E'$ et $E \rightarrow E''$, alors il existe E''' tel que : $E' \rightarrow E'''$ et $E'' \rightarrow E'''$.*

Les preuves sont données en détails dans [4].

5. Travaux connexes

MSPML peut être utilisé pour implémenter des patrons data-parallèles. Nous avons réalisé une implantation du patron “Diffusion” mais d'autres comme [9] peuvent être également considérés. [2] a montré que NESL [5] est plus efficace lorsque la taille des vecteurs est constante. Même si ce n'est pas le cas, la plupart des opérations de NESL peuvent être implémentées en MSPML. En particulier les listes emboîtées peuvent être implémentées comme dans [15]. De ce point de vue MSPML peut paraître plus bas niveau que NESL. Mais MSPML offre les fonctions d'ordre supérieur alors que ce n'est pas le cas pour NESL.

Il y a de nombreux travaux sur la désynchronisation de barrières BSP qui se basent sur différentes méthodes de comptage de messages [10, 1, 17]. À notre connaissance la seule extension qui a donné lieu à une implantation disponible est celle que l'on trouve dans la bibliothèque PUB [6]. La synchronisation **bsp_oblsync** prend en argument le nombre de messages devant être reçus par le processeur à une super-étape donnée. Lorsque ce nombre de messages a été reçu le processeur passe à la super-étape d'après sans prendre part à une synchronisation globale.

Le langage fonctionnel parallèle Caml-Flight [7] est basé sur le mécanisme de *vague*. La primitive sync est utilisée pour indiquer quels processeurs peuvent échanger des messages durant l'utilisation de la primitive **get** qui est très différente de la notre : il s'agit ici de demander l'évaluation distante d'une expression. Ce mécanisme est plus complexe que celui de MSPML et il n'y a pas de sémantique purement fonctionnelle de Caml-Flight [11]. De plus les programmes Caml-Flight sont SPMD et donc plus difficile à écrire et lire que les programmes MSPML. Le système de type détectant le parallélisme emboîté incorrect est également complexe [27].

[23] décrit le mécanisme des *horloges structurelles* qui permet l'exécution de programmes data-parallèles écrit dans un petit langage impératif SPMD. La difficulté dans ce cadre est que le nombre de phases de communication peut être différent sur chaque processeur car il y a un opérateur de composition parallèle. Nous avons également besoin d'un mécanisme plus complexe que dans le cas de MSPML sans juxtaposition dès que nous ajoutons une juxtaposition parallèle à MSPML.

Si on introduit une notion de synchronisation de sous-réseau dans le modèle BSP, on perd la simplicité du modèle puisqu'alors la synchronisation globale de chaque sous-machine ne coûtera plus L. Pour conserver le modèle BSP, ce qui est souhaitable [14], il faut donc que les barrières de synchronisation concernent toute la machine. Dans la sémantique de BSML avec composition parallèle, l'opération de composition parallèle nécessite que deux expressions composées parallèlement s'évaluent en utilisant le même nombre de barrières de synchronisation. Cette condition peut être levée avec un surcoût d'une barrière de synchronisation pour une hiérarchie d'appels à la composition parallèle [20]. Cette nécessité disparaît dans une la sémantique d'évaluation asynchrone de MSPML.

6. Conclusions et travaux futurs

L'ajout à Minimally Synchronous Parallel ML de la juxtaposition permet d'écrire facilement des programmes diviser-pour-règner. Une implantation a été développée, prenant la sémantique présentée dans cet article comme spécification. Il nous reste à expérimenter ces nouvelles possibilités, aussi bien au niveau de l'expressivité que de la prévision de performances. L'introduction de la juxtaposition ne simplifie pas l'écriture des formules de coûts MPM des programmes MSPML. Il faut également vérifier expérimentalement la validité de ce modèle de prévision de performances.

Au niveau sémantique il nous reste à prouver l'équivalence de la sémantique à grands pas et de la sémantique distribuée. On pourra alors avoir confiance en la correspondance entre le modèle de programmation et le modèle d'exécution de MSPML.

Dans l'implantation actuelle, la gestion des environnements de communications nécessite de temps en temps une synchronisation globale. Dans le cas de MSPML sans composition parallèle, un nouveau mécanisme a été proposé qui supprime ces synchronisations globales [3]. Il faudra l'adapter pour MSPML avec juxtaposition.

Remerciements

Ce travail bénéficie d'un soutien du ministère de la recherche sous la forme d'un projet ACI jeunes chercheuses et chercheurs intitulé “Programmation parallèle certifiée” (http://wwwpropac.free.fr). Les auteurs souhaitent remercier les relecteurs anonymes pour leurs commentaires.

Bibliographie

[1] R. Alpert and J. Philbin. cBSP: Zero-Cost Synchronization in a Modified BSP Model. Technical Report 97-054, NEC Research Institute, 1997.

[2] M. Bamha. L'implémentation d'un langage portable à parallélisme emboîté en processus statiques. Mémoire de DEA d'informatique, LIFO, Université d'Orléans, Septembre 1996.

[3] A. Belbekkouche. MSPML : Environnements de communication et tolérance aux pannes. Mémoire de master de recherche en informatique, LIFO, Université d'Orléans, Septembre 2005.

[4] R. Benheddi. Composition parallèle pour MSPML: sémantique et implantation. Mémoire de master de recherche en informatique, LIFO, Université d'Orléans, Septembre 2005.

[5] G.E. Blelloch. NESL: A Nested Data-Parallel Language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

[6] O. Bonorden, B. Juurlink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[7] E. Chailloux and C. Foisy. A Portable Implementation for Objective Caml Flight. *Parallel Processing Letters*, 13(3):425–436, 2003.

[8] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly, 2000.

[9] R. Di Cosmo and S. Pelagatti. A Calculus for Dense Array Distributions. *Parallel Processing Letters*, 13(3):377–388, 2003.

[10] A. Fahmy and A. Heddaya. Communicable Memory and Lazy Barriers for Bulk Synchronous Parallelism in BSPk. Technical Report BU-CS-96-012, Boston University, 1996.

[11] C. Foisy, J. Vachon, and G. Hains. DPML: de la sémantique à l'implantation. In P. Cointe, C. Queinnec, and B. Serpette, editors, *Journées Francophones des Langages Applicatifs*, volume 11 of *Collection Didactique*, Noirmoutier, Février 1994. INRIA.

[12] F. Gava. Une bibliothèque certifiée de programmes fonctionnels BSP. In Ménissier-Morain, V., editor, *Journées Francophones des Langages Applicatif, JFLA*, pages 55–68. INRIA, january 2004.

[13] F. Gava and F. Loulergue. A Functional Language for Departmental Metacomputing. *Parallel Processing Letters*, 2005. est paru.

[14] G. Hains. Subset synchronization in BSP computing. In H.R.Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 242–246, Las Vegas, July 1998. CSREA Press.

[15] Z. Hu, T. Takahashi, H. Iwasaki, and M. Takeichi. Segmented Diffusion Theorem. In *IEEE International Conference on Systems, Man and Cybernetics (SMC 02)*. IEEE Press, October 6-9 2002.

[16] H. Jifeng, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, number 1123–1124 in Lecture Notes in Computer Science, Lyon, August 1996. LIP-ENSL, Springer.

[17] Jin-Soo Kim, Soonhoi Ha, and Chu Shik Jhon. Relaxed barrier synchronization for the BSP model of computation on message-passing architectures. *Information Processing Letters*, 66(5):247–253, 1998.

[18] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.08, 2004. web pages at www.ocaml.org.

[19] F. Loulergue. Extension du BSλ-calcul. In P. Weis, editor, *JFLA'99 : Journées Francophones des Langages Applicatifs*, pages 93–112, Morzine-Avoriaz, February 1999.

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

[21] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics and Implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, 5(3):182–199, 2004. W. Dosch, editor, special issue on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing.

[22] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.

[23] X. Rebeuf. *Un modèle de coût symbolique pour les programmes parallèles asynchrones à dépendances structurées*. PhD thesis, Université d'Orléans, LIFO, 2000.

[24] J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the execution time of message passing models. *Concurrency: Practice and Experience*, 11(9):461–477, 1999.

[25] C. Rodriguez, J.L. Roda, F. Sande, D.G. Morales, and F. Almeida. A new parallel model for the analysis of asynchronous algorithms. *Parallel Computing*, 26:753–767, 2000.

[26] M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.

[27] J. Vachon. Une analyse statique pour le contrôle des effets de bords en Caml-Flight beta. In C. Queinnec, V. V. Donzeau-Gouge, and P. Weis, editors, *JFLA*, number 13. INRIA, Janvier 1995.

[28] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, pages 103–111, August 1990.