ML Parallèle Minimalement Synchrone

Myrto ARAPINIS
myrto.arapinis@wanadoo.fr

Stage de Maîtrise 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

Mai-Juin 2003

Remerciements

Je voudrais remercier Frédéric Loulergue pour m'avoir donné l'opportunité de participer à ce projet, et pour m'avoir encadré et guidé dans mon travail.

Je tiens à remercier tout particulièrement Frédéric Gava tant pour ses commentaires que pour son soutien.

Merci aussi à Frédéric Dabrowski pour ses conseils et son aide.

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. Leur programmation demeure néanmoins plus difficile que celle des machines séquentielles. La conception de langage adaptés est un sujet de recherche actif.

Le parallélisme de données est un paradigme de programmation parallèle dans lequel un programme décrit une séquence d'actions sur des tableaux à accès parallèle. Le modèle BSP [1] vise à maximiser la portabilité des performances en ajoutant une notion de processus explicites au 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.

De nombreux travaux ont étudié la question du mélange de la programmation fonctionnelle et du parallélisme. On peut les classer en deux catégories :

1. les extensions explicitement parallèles des langages fonctionnels,
2. les implantations parallèles avec sémantique fonctionnelle.

Dans la première catégorie, Concurrent ML [2] par exemple ajoute à ML une notion de canal et a une sémantique concurrente avec les défauts qui l'accompagnent. Cette extension brise l'aspect fonctionnel pur du sous-langage fonctionnel de ML.Dans la seconde catégorie on note de nombreux travaux sur la réduction de graphes en parallèle, par exemple [3]. Ces systèmes n'expriment pas directement les algorithmes parallèles et ne permettent pas la prévision des temps d'exé-

cution car les processus physiques y sont implicites et la stratégie d'évaluation parallèle n'est pas décrite par la sémantique.

Une approche intermédiaire est celle des langages à patrons ou *algorithmic skeletons* dans lesquels seulement un ensemble fixé d'opérations (les patrons) sont exécutées en parallèle. Leur sémantique fonctionnelle est explicite mais leur sémantique opérationnelle parallèle est implicite. Du point de vue du programmeur le style de programmation associé revient à l'utilisation de combinateurs dont l'effet sur la parallélisation est défini extérieurement. Les avantages par rapport aux extensions concurrentes sont bien sûr que le déterminisme et le non-blocage sont garantis et que l'on conserve une sémantique fonctionnelle pure. Cependant l'implanteur de bibliothèques de patrons doit faire face à deux problèmes. D'une part il doit proposer un ensemble le plus complet possible de patrons et cet ensemble s'avère dépendant du domaine d'application. D'autre part il lui faut implanter efficacement pour chaque architecture chaque patron.

Le projet BSλ/BSML approfondit la position intermédiaire que le paradigme des patrons occupe avec deux objectifs : 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 réseau statique de processeurs de la machine.

Le BSλ-calcul [4, 5] est un λ-calcul étendu par des opérations parallèle BSP qui s'avère confluent et universel pour les algorithmes BSP. La `BSMLlib` [6, 7] est une implantation partielle de ces opérations sous forme d'une bibliothèque pour le langage Objective CAML [8]. Cette bibliothèque permet d'écrire des programmes parallèles BSP sur une grande variété d'architectures allant du PC à deux processeurs au systèmes massivement parallèle Cray comprenant plusieurs centaines de processeurs, en passant par des clusters de PC.

Le mécanisme de synchronisation globale imposé par le modèle BSP soulève quelques problèmes tels que le surcoût de la barrière de synchronisation, ou par exemple la synchronisation à une étape donnée d'un processeur avec les autres alors qu'il n'est engagé dans aucune communication. De plus, dans la sémantique actuelle de BSML, l'opération de composition parallèle, telle qu'elle est définie, nécessite que deux expressions composées parallèlement s'évaluent en utilisant le même nombre de barrières de synchronisation. Cette nécessité disparait dans une sémantique d'évaluation asynchrone.

Dans un souci d'optimisation de l'exécution et afin d'avoir une souplesse supplémentaire en ce qui concerne l'opération de composition parallèle, nous nous proposons de définir une sémantique basée sur la désynchronisation des opérations de BSML. Nous devrions alors obtenir un langage de programmation fonc-

tionnelle parallèle minimalement synchrone comprenant un mécanisme semblable à la vague de Caml-Flight [9], tout en gardant notre langage source et un modèle de coût simple et réaliste.

Pour ce stage nous avons dû dans un premier temps nous documenter sur les langages de programmation fonctionnelle séquentiels (Ocaml) et parallèles (BSML). Nous avons consulté pour ceci : [5, 10, 11, 12]. Un résumé de cette phase préliminaire est exposé au chapitre suivant qui sert de préliminaires à ce rapport. Nous y présentons le modèle BSP, la BSMLlib, et un mini langage fonctionnel séquentiel.

Au chapitre 3, nous présentons la syntaxe ainsi que la sémantique de notre langage MSPML. Nous apportons ensuite, au chapitre 4, des éléments pour l'implémentation de ce langage. Enfin, au chapitre 5, nous discuterons d'un modèle de coût pour notre langage.

Chapitre 2

Préliminaires

2.1 Le modèle BSP

Le modèle de programmation parallèle BSP (*Bulk Synchronous Parallelism*) [13] décrit une architecture parallèle (abstraite), un modèle d'exécution et un modèle de coût.

2.1.1 L'architecture parallèle BSP

Un ordinateur parallèle BSP possède trois ensembles de composants :
- un ensemble homogène de paires processeur-mémoire,
- un réseau de communication permettant l'échange de messages entre chaque couple de processeurs,
- une unité de synchronisation globale qui exécute des demandes collectives de barrières de synchronisation.

De nombreuses architectures réelles peuvent être vues comme des ordinateurs parallèles BSP. Par exemple, les machines à mémoire partagée peuvent être utilisées de telle sorte que chaque processeur n'accède qu'à une partie (qui sera alors "privée") de la mémoire partagée et les communications peuvent être faites en utilisant des zones de la mémoire partagée réservées à cet usage. De plus, l'unité de synchronisation est rarement physique mais plutôt logicielle ([14] présente plusieurs algorithmes à cet effet). Les performances d'un ordinateur BSP sont caractérisées par trois paramètres (exprimés en multiples de la vitesse des processeurs, dans le cas contraire un quatrième paramètre, la vitesse des processeurs est donnée) :

– le nombre de paires processeur-mémoire 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 $h \times g$.

Ces paramètres peuvent être facilement obtenus en pratique en utilisant des tests [15].

2.1.2 Le modèle d'exécution

L'exécution d'un programme BSP est une séquence de *super-étapes*. Chaque super-étape est divisée en trois phases sucessives et logiquement disjointes (Fig. 2.1) :

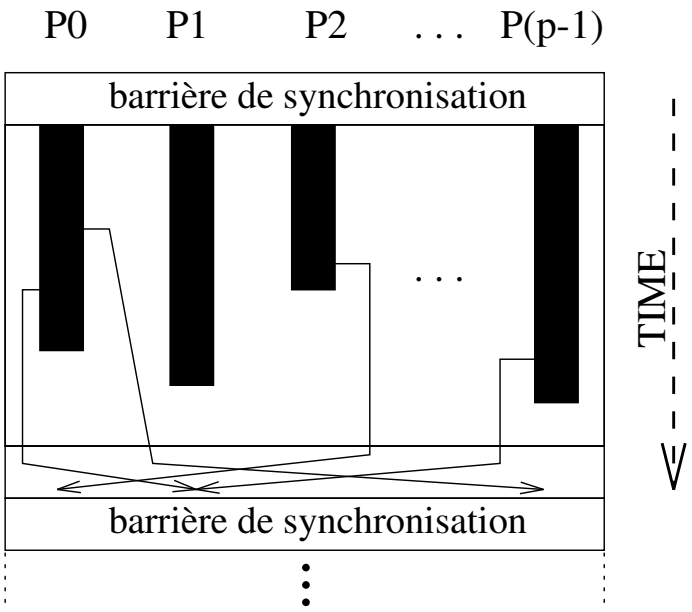

FIG. 2.1 – Une super-étape BSP

– chaque processeur utilise les données qu'il détient localement pour faire des calculs de façon séquentielle et pour demander des tranferts depuis ou vers d'autres processeurs,

– le réseau réalise les échanges de données demandés à la phase précédente,
– une barrière de synchronisation globale termine la super-étape. À l'issue de cette barrière de synchronisation globale, les données échangées sont effectivement disponibles pour la nouvelle super-étape qui commence alors.

2.1.3 Le modèle de coût

Le temps nécessaire à l'exécution d'une super-étape s est la somme :
– du maximum des temps de calculs locaux,
– du temps de la réalisation des échanges entre processeurs,
– du temps de la réalisation d'une barrière de synchronisation globale.
On l'exprime par la formule suivante :

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

où $w_i^{(s)}$ est le temps de calcul local sur le processeur i pendant la super-étape s et $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ où $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) est le nombre de mots envoyés (resp. reçus) par le processeur i durant la super-étape s. Le temps d'exécution $\sum_s \text{Time}(s)$ d'un programme BSP composé de S super-étapes est donc une somme de trois termes :

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

où $\begin{cases} W = \sum_s \max_i w_i^{(s)} \\ H = \sum_s \max_i h_i^{(s)}. \end{cases}$

En général W, H et S sont fonctions de p et de la taille des données n, ou de paramètres plus complexes tels que le déséquilibre des données (*data skew*). Afin de minimiser le temps d'exécution, un algorithme BSP doit à la fois minimiser le nombre de super-étapes, le volume total H et le déséquilibre des communications, le volume total W et le déséquilibre des calculs locaux.

2.2 La BSMLlib

Il n'y a pas pour l'instant d'implantation complète du langage Bulk Synchronous Parallel ML mais une implantation partielle en tant que bibliothèque pour Objective Caml, appelée BSMLlib.

En premier lieu cette bibliothèque donne accès aux paramètres BSP de l'architecture sur laquelle sont évalués les programmes BSML. En particulier, elle offre

la fonction `bsp_p :unit->int`. La valeur de `bsp_p()` est p, le nombre statique de processeurs de la machine parallèle. Cette valeur est constante durant l'exécution.

Les valeurs parallèles de largeur p contenant en chaque processeur une valeur de type `'a` sont représentées par le type abstrait `'a par`. L'imbrication de vecteurs parallèles est interdite. Jusqu'à présent le programmeur était responsable de l'absence d'imbrication. Le système de types et l'algorithme de synthèse de types présentés dans cet article remédient à ce défaut. Ceci constitue une amélioration par rapport à Caml Flight [16, 17] dans lequel l'imbrication de la structure parallèle globale de contrôle `sync` était interdite *dynamiquement*.

Les vecteurs parallèles sont créés par :

```
mkpar : (int -> 'a) -> 'a par
```

`(mkpar f)` s'évalue en un vecteur parallèle qui possède au processeur `i` la valeur de `(f i)`, pour tout i compris entre 0 et $(p-1)$. Nous écrivons souvent `fun pid->e` pour `f` afin de montrer que `e` peut être différente sur chaque processeur. Cette expression `e` est dite *locale*. L'expression `(mkpar f)` est un objet parallèle et est dite *globale*.

Un algorithme BSP est exprimé comme une combinaison de calculs locaux asynchrones (première phase d'une super-étape) de communications globales (seconde phase d'une super-étape) et d'une synchronisation (troisième phase d'une super-étape). Les calculs asynchrones sont programmés avec `mkpar` et avec :

```
apply : ('a -> 'b) par -> 'a par -> 'b par
```

`apply (mkpar f) (mkpar e)` s'évalue en un vecteur parallèle qui contient `(f i) (e i)` au processeur `i`. Ni l'implantation de BSMLlib, ni sa sémantique [5] ne préconisent de synchronisation entre deux appels successifs à `apply`.

Nous ignorons la distinction entre la phase de demande de communications et sa réalisation à la barrière de synchronisation. Les phases de communications et de synchronisation sont exprimées à l'aide de :

```
put :(int->'a option) par -> (int->'a option) par
```

où `'a option` est définie par `type 'a option = None | Some of 'a`.

Considérons l'expression :

$$\texttt{put(mkpar(fun i->fs}_i\texttt{))} \tag{2.1}$$

Pour envoyer une valeur `v` d'un processeur `j` vers un processeur `i`, la fonction $\texttt{fs}_j$ du processeur `j` doit être telle que $(\texttt{fs}_j$ `i)` s'évalue en `Some v`. Pour ne pas envoyer de message de `j` à `i`, $(\texttt{fs}_j$ `i)` doit s'évaluer en `None`.

L'expression (2.1) s'évalue en un vecteur parallèle contenant en chaque processeur une fonction $\texttt{fd}_i$ de messages transmis. Au processeur `i`, $(\texttt{fd}_i$ `j)` s'évalue en `None` si le processeur `j` n'a pas envoyé de message à `i` ou s'évalue en `Some v` si le processeur `j` a envoyé la valeur `v` au processeur `i`.

Il y a également une opération get duale au put :

```
get :'a par -> int par -> 'a par
```

telle que get (mkpar f_1) (mkpar f_2) s'évalue au processeur i en (f_1 ((f_2 i) mod p)).

L'opération get_list est une extension du get :

```
get_list : 'a par -> (int list) par -> ('a list) par
```

qui permet à chaque processeur de recevoir des données de plusieurs processeurs.

L'ordre des éléments dans la liste obtenue est le même que l'ordre dans lequel sont donnés les processeurs sources dans la liste passée en argument.

Le langage complet contiendra également une conditionnelle globale synchrone :

```
ifat : (bool par) * int * 'a * 'a -> 'a
```

telle que `ifat (v,i,v1,v2)` s'évaluera en `v1` ou `v2` selon que la valeur de `v` au processeur `i` est `true` ou `false`. Objective Caml étant un langage strict avec appel par valeur, cette conditionnelle ne peut être définie comme une fonction. C'est la raison pour laquelle BSMLlib contient la fonction `at :bool par -> int -> bool` qui doit être utilisée uniquement dans la construction suivante : `if (at vec pid) then... else...` où `(vec :bool par)` et `(pid :int)` et dont la sémantique est celle de `ifat`. Cette conditionnelle globale permet d'exprimer des phases de communications et synchronisation. Sans elle, il est impossible d'écrire des algorithmes ayant la forme suivante :

```
Repeat
  Parallel Iteration
Until Max of local errors < epsilon
```

2.3 Évaluation d'un mini langage fonctionnel

Avant de présenter notre sémantique asynchrone, nous nous proposons d'étudier la syntaxe et la sémantique d'un mini-langage fonctionnel. Ce chapitre nous permettra de revoir certaines notions mises en oeuvre au chapitre suivant.

2.3.1 Syntaxe

La syntaxe décrit les constructions du langage, et la façon de les désigner formellement. Pour notre mini-langage fonctionnel les expressions du langage sont essentiellement les termes du λ-calcul avec constantes et la construction de liaison **let**, d'où la syntaxe abstraite suivante :

$e ::=$	x	(variables)
	c	(constantes)
	op	(opérateurs)
	fun $x \to e$	(abstraction fonctionnelle)
	$(e\ e)$	(application)
	let $x = e$ **in** e	(liaison)
	(e, e)	(couples)

op contient les opérations booléennes, arithmétiques, la conditionnelle et l'opérateur de point fixe.

Pour définir la manière dont nos expressions sont évaluées en valeur, il nous faut avant tout définir les valeurs de notre langage. La grammaire des valeurs de nos expressions est la suivante :

$v ::=$	**fun** $x \to e$	(valeur fonctionnelle)
	c	(constantes)
	op	(opérateurs)
	(v, v)	(couples)

2.3.2 Sémantique naturelle

La sémantique dynamique décrit la façon dont se déroule le calcul. Elle met en relation des programmes avec un ou plusieurs résultats possibles. Elle consiste à définir une relation e ▷ v entre les programmes et les valeurs. La relation d'évaluation est définie par un ensemble de règles d'inférence qui décrivent l'évaluation

d'un programme à partir de sa structure et en s'appuyant sur l'évaluation de sous-programmes.

On notera $e_1[x \leftarrow e_2]$ la substitution de e_2 à toutes les occurences libres de x dans e_1.

Les règles d'évaluation de notre mini-langage fonctionnel sont données ci-dessous :

Les axiomes :

$$c \triangleright c$$

$$op \triangleright op$$

$$(\textbf{fun}\ x \rightarrow e) \triangleright (\textbf{fun}\ x \rightarrow e)$$

Les règles pour l'application, la liaison et le couple :

$$\frac{e_1 \triangleright (\textbf{fun}\ x \rightarrow e) \quad e_2 \triangleright v_2 \quad e[x \leftarrow v_2] \triangleright v}{(e_1\ e_2) \triangleright v}$$

$$\frac{e_1 \triangleright v_1 \quad e_2[x \leftarrow v_1] \triangleright v}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \triangleright v}$$

$$\frac{e_1 \triangleright v_1 \quad e_2 \triangleright v_2}{(e_1, e_2) \triangleright (v_1, v_2)}$$

Les règles pour la conditionnelle, la projection et les opérateurs arithmétiques et de point fixe :

$$\frac{e_1 \triangleright \textbf{+} \quad e_2 \triangleright n_1 \quad e_3 \triangleright n_2 \quad n_1 \text{ et } n_2 \text{ entiers et } n = n_1 + n_2}{e_1\ (e_2, e_3)}$$

$$\frac{e_1 \triangleright (\textbf{fun}\ x \rightarrow e_2) \quad e_2[x \leftarrow \textbf{fix}(e_1)] \triangleright v}{\textbf{fix}(e_1) \triangleright v}$$

$$\textbf{fix}(op) \triangleright op$$

$$\frac{e_1 \triangleright \textbf{true} \quad e_2 \triangleright v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \triangleright v}$$

$$\frac{e_1 \triangleright \textbf{false} \quad e_3 \triangleright v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \triangleright v}$$

$$\frac{e_1 \triangleright \textbf{fst} \quad e_2 \triangleright (v_1, v_2)}{(e_1\ e_2) \triangleright v_1}$$

$$\frac{e_1 \triangleright \textbf{snd} \quad e_2 \triangleright (v_1, v_2)}{(e_1\ e_2) \triangleright v_2}$$

2.3.3 Sémantique à petits pas

La sémantique naturelle, donnée ci-dessus, s'intéresse seulement aux résultats de l'évaluation. Pour décrire le calcul il nous faut définir la sémantique à petits pas du langage. En effet, la sémantique à petits pas s'intéresse au calcul proprement dit en décrivant chaque étape élémentaire ; elle consiste à définir une relation $e_1 \rightharpoonup e_2$ de réécriture des programmes sur eux-mêmes. Ainsi, un programme se réduit pas à pas, soit indéfiniment, soit jusqu'à l'obtention d'une forme normale.

Les réductions sont d'abord définies pour chaque radical. Nous écrirons $e_1 \overset{\varepsilon}{\rightharpoonup} e_2$ pour les réductions en tête, i.e. les réductions de radicaux, et plus généralement $e_1 \rightharpoonup e_2$ pour une réduction à une position quelconque dans le terme.

Nous notons $\overset{*}{\rightharpoonup}$, la fermeture transitive de $\rightharpoonup$ et nous notons $e_0 \overset{*}{\rightharpoonup} v$ pour $e_0 \rightharpoonup e_1 \rightharpoonup e_2 \rightharpoonup \ldots \rightharpoonup v$. Pour définir la relation, $\rightharpoonup$, nous commençons par les règles de réduction de tête :

$$\begin{array}{lcll} (\textbf{fun}\ x \to e)\ v & \overset{\varepsilon}{\rightharpoonup} & e[x \leftarrow v] & (\beta_{fun}) \\ (\textbf{let}\ x = v\ \textbf{in}\ e) & \overset{\varepsilon}{\rightharpoonup} & e[x \leftarrow v] & (\beta_{let}) \end{array}$$

Nous ajoutons également un ensemble de δ-règles qui décrivent la façon de réduire les primitives :

$$\begin{array}{lcll} +(n_1, n_2) & \overset{\varepsilon}{\rightharpoonup} & n \text{ avec } n = n_1 + n_2 & (\delta_+) \\ \textbf{fst}(v_1, v_2) & \overset{\varepsilon}{\rightharpoonup} & v_1 & (\delta_{fst}) \\ \textbf{snd}(v_1, v_2) & \overset{\varepsilon}{\rightharpoonup} & v_2 & (\delta_{snd}) \\ \textbf{fix}(\textbf{fun}\ x \to\ e) & \overset{\varepsilon}{\rightharpoonup} & e[x \leftarrow \textbf{fix}(\textbf{fun}\ x \to\ e)] & (\delta_{fix}) \\ \textbf{fix}(\textbf{op}) & \overset{\varepsilon}{\rightharpoonup} & \textbf{op} & \\ \textbf{if true then}\ v_1\ \textbf{else}\ v_2 & \overset{\varepsilon}{\rightharpoonup} & v_1 & (\delta_{ifthenelse}) \\ \textbf{if false then}\ v_1\ \textbf{else}\ v_2 & \overset{\varepsilon}{\rightharpoonup} & v_2 & \end{array}$$

Il faut maintenant décrire les endroits où il est possible de réduire un radical. Pour cela, on définit un ensemble de contextes d'évaluation. Un contexte d'évaluation est une expression contenant un trou $\bullet$. On note E[e] pour le résultat de la

substitution de $\bullet$ par e dans le contexte d'évaluation E[].

$$
\begin{array}{rcl}
\Gamma & ::= & \bullet \\
& | & \Gamma\ e \\
& | & v\ \Gamma \\
& | & \textbf{let}\ x = \Gamma\ \textbf{in}\ e \\
& | & (\Gamma, e) \\
& | & (v, \Gamma)
\end{array}
$$

Puis, on permet de réduire dans un contexte d'évaluation en ajoutant la règle d'évaluation suivante qui exprime que la réduction peut se produire également à l'intérieur d'un terme :

$$\frac{e \xrightarrow{\varepsilon} e'}{\Gamma(e) \rightharpoonup \Gamma(e')}$$

De la définition de nos contextes découle le fait que la sémantique de notre langage est déterministe, i.e. il n'existe qu'un chemin de réduction possible.

Chapitre 3

MSPML : Minimaly Synchronous Parallel ML

Nous sommes à présent en mesure de donner la sémantique minimalement synchrone, MSPML. Ce chapitre chapitre introduit la syntaxe du noyau fonctionnel du langage BSML ainsi que sa sémantique naturelle, et la syntaxe et la sémantique distribuée des expressions de MSPML.

3.1 Mini-BSML

3.1.1 Syntaxe

La syntaxe de notre langage reprend celle présentée au chapitre précédent, en l'étendant pour prendre en compte les opérateurs parallèles :

$e ::=$	x	(variables)
	c	(constantes)
	op	(opérateurs)
	fun $x \rightarrow e$	(abstraction fonctionnelle)
	$(e\ e)$	(application)
	let $x = e$ **in** e	(liaison)
	(e, e)	(couples)
	mkpar e	(constructeur parallèle)
	apply $e\ e$	(application parallèle)
	get $e\ e$	(opérateur de communication)
	if e **at** e **then** e **else** e	(conditionnelle globale)

Dans cette grammaire, x appartient à l'ensemble des identifiants. L'expression (ee') correspond à l'application de la fonction ou de l'opérateur e à un argument e'. Le terme **fun** $\rightarrow e$ correspond à l'abstraction du λ-calcul, soit à la fonction dont le paramètre est x et le résultat la valeur de e. Les constantes c sont les entiers et les booléens. L'ensemble des opérateurs *op* contient les opérateurs arithmétiques (+,-,...), l'opérateur de point fixe(**fix**) et la conditionnelle.**mkpar**, **apply**, **get** et **ifat** sont les opérateurs parallèles de BSML présentés dans la section sur la BSMLlib.

Exemple 1 (La Diffusion)

$$\begin{array}{rcl} \textbf{\textit{let}}\ bcast\ i\ v & = & \textbf{\textit{get}}\ v\ (\textbf{\textit{mkpar}}(\textbf{\textit{fun}}\ x\ \rightarrow\ i)) \\ bcast & : & \textbf{\textit{int}} \rightarrow \alpha\ \textbf{\textit{par}} \rightarrow \alpha\ \textbf{\textit{par}} \end{array}$$

3.1.2 Sémantique naturelle

Il existe une sémantique suivant la valeur de p, avec p le nombre de processus de la machine parallèle. Par la suite $\forall$i signifie $\forall i \in \{0, 1, \ldots, p-1\}$. Aussi nous étendons la grammaire de départ avec les vecteurs parallèles d'expressions :

$e' ::=$	x	(variables)
	c	(constantes)
	op	(opérateurs)
	fun $x \rightarrow e'$	(abstraction fonctionnelle)
	$(e'\ e')$	(application)
	let $x = e'$ **in** e'	(liaison)
	(e', e')	(couples)
	mkpar e'	(constructeur parallèle)
	apply $e'\ e'$	(application parallèle)
	get $e'\ e'$	(opérateur de communication)
	if e' **at** e' **then** e' **else** e'	(conditionnelle globale)
	$\langle e', e', \ldots, e' \rangle$	(vecteur parallèle de taille p)

Le programmeur n'utilise pas cette syntaxe, les vecteurs parallèles énumérés sont générés au cours de l'évaluation. Dans cette syntaxe, on ne distingue pas expressions locales et globales comme dans BSλ, car on s'appuie sur le système de types [11] pour éviter les problèmes comme l'emboîtement de vecteurs. Les valeurs de

MSPML sont définies par la grammaire suivante :

$$
\begin{array}{lll}
v ::= & \textbf{fun}\ x \rightarrow e' & \text{(valeur fonctionnelle)} \\
| & c & \text{(constantes)} \\
| & op & \text{(opérateurs)} \\
| & (v, v) & \text{(couples)} \\
| & \langle v, v, \ldots, v \rangle & \text{(vecteur parallèle de taille p)}
\end{array}
$$

On notera $e_1[x \leftarrow e_2]$ la substitution de e_2 à toutes les occurences libres de x dans e_1.

Les règles d'évaluation pour les axiomes :

$$c \triangleright c$$

$$op \triangleright op$$

$$(\textbf{fun}\ x \rightarrow e) \triangleright (\textbf{fun}\ x \rightarrow e)$$

pour l'application, la liaison et le couple :

$$\frac{e_1 \triangleright (\textbf{fun}\ x \rightarrow e) \quad e_2 \triangleright v_2 \quad e[x \leftarrow v_2] \triangleright v}{(e_1\ e_2) \triangleright v}$$

$$\frac{e_1 \triangleright v_1 \quad e_2[x \leftarrow v_1] \triangleright v}{\textbf{let}\ x = e_1\ \textbf{in}\ e_2 \triangleright v}$$

$$\frac{e_1 \triangleright v_1 \quad e_2 \triangleright v_2}{(e_1, e_2) \triangleright (v_1, v_2)}$$

pour la conditionnelle, la projection et les opérateurs arithmétiques et de point fixe :

$$\frac{e_1 \triangleright \textbf{+} \quad e_2 \triangleright n_1 \quad e_3 \triangleright n_2 \quad n_1 \text{ et } n_2 \text{ entiers et } n = n_1 + n_2}{e_1\ (e_2, e_3)}$$

$$\frac{e_1 \triangleright (\textbf{fun}\ x \rightarrow e_2) \quad e_2[x \leftarrow \textbf{fix}(e_1)] \triangleright v}{\textbf{fix}(e_1) \triangleright v}$$

$$\textbf{fix}(op) \triangleright op$$

$$\frac{e_1 \triangleright \textbf{true} \quad e_2 \triangleright v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \triangleright v}$$

$$\frac{e_1 \triangleright \textbf{false} \quad e_3 \triangleright v}{\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3 \triangleright v}$$

$$\frac{e_1 \triangleright \textbf{fst} \quad e_2 \triangleright (v_1, v_2)}{(a_1\ e_2) \triangleright v_1}$$

$$\frac{e_1 \triangleright \textbf{snd} \quad e_2 \triangleright (v_1, v_2)}{(e_1\ e_2) \triangleright v_2}$$

Enfin, les règles d'évaluation pour les opérateurs parallèles :

$$\frac{e_1 \triangleright v \quad \forall i\ (v\ i) \triangleright v_i}{\textbf{mkpar}\ e_1 \triangleright \langle v_0, \ldots, v_{p-1} \rangle}$$

$$\frac{e_1 \triangleright \langle \textbf{fun}\ v_1', v_2', \ldots, v_{p-1}' \rangle \quad e_2 \triangleright \langle v_0'', v_1'', \ldots, v_{p-1}'' \rangle \quad \forall i (v_i'\ v_i'') \triangleright v_i)}{\textbf{apply}\ e_1\ e_2 \triangleright \langle v_0, v_1, \ldots, v_{p-1} \rangle}$$

$$\frac{e_1 \triangleright \langle v_0, v_1, \ldots, v_{p-1} \rangle \quad e_2 \triangleright \langle i_0, i_1, \ldots, i_{p-1} \rangle}{\textbf{get}\ e_1\ e_2 \triangleright \langle v_{i_0}, v_{i_1}, \ldots, v_{i_{p-1}} \rangle}$$

$$\frac{e_1 \triangleright \langle \ldots, \overbrace{\textbf{true}}^{n}, \ldots \rangle \quad e_2 \triangleright n \quad e_3 \triangleright v_3}{\textbf{if}\ e_1\ \textbf{at}\ e_2\ \textbf{else}\ e_3\ \textbf{then}\ e_4 \triangleright v_3}$$

$$\frac{e_1 \triangleright \langle \ldots, \overbrace{\textbf{false}}^{n}, \ldots \rangle \quad e_2 \triangleright n \quad e_4 \triangleright v_4}{\textbf{if}\ e_1\ \textbf{at}\ e_2\ \textbf{else}\ e_3\ \textbf{then}\ e_4 \triangleright v_4}$$

Exemple 2 *Nous reprenons l'exemple ci-dessus de diffusion. On suppose que v s'évalue en* $\langle v_0, v_1, \ldots v_{p-1} \rangle$

$$\frac{\dfrac{\ldots}{v\ \triangleright \langle v_0, v_1, \ldots v_{p-1} \rangle} \qquad \dfrac{\textit{fun}\ x \to i\ \triangleright \textit{fun}\ x \to i \qquad \forall j \dfrac{i[x \leftarrow j]\ \triangleright\ i}{(\textit{fun}\ x \to i)\ j \triangleright i}}{\textit{mkpar}(\textit{fun}\ x \to i)\ \triangleright \langle i, i, \ldots, i \rangle}}{\dfrac{\textit{get}\ v\ (\textit{mkpar}(\textit{fun}\ x \to i)\ \triangleright \langle v_i, v_i, \ldots v_i \rangle}{\textit{bcast}\ i\ v\ \triangleright \langle v_i, v_i, \ldots, v_i \rangle}}$$

3.2 Sémantique distribuée

La sémantique naturelle ne donne pas les étapes du calcul (uniquement le résultat) et le traitement de la désynchronisation n'est donc pas visible. Pour montrer comment est effectuée la désynchronisation, il nous faut donner la sémantique distribuée de notre langage, qui donne tou les pas du calcul de l'expression vers un résultat donné.

L'évaluation distribuée $\rightarrow$ peut être d'éfinie en deux temps :

1. l'évaluation locale des expressions $\rightharpoonup_i$.
2. l'évaluation globale des termes distribués qui permet l'évaluation des requêtes de communication (**resquest**).

3.2.1 Syntaxe

Pour le programmeur, la syntaxe est la même que celle définie à la section 3.1.1 mais nous avons encore besoin de définir les termes générés au cours de l'évaluation, nous avons en plus le request :

$$
\begin{array}{rl}
e_d ::= & x \\
& \mid c \\
& \mid op \\
& \mid \textbf{fun}\ x \rightarrow e_d \\
& \mid (e_d\ e_d) \\
& \mid \textbf{let}\ x = e_d\ \textbf{in}\ e_d \\
& \mid (e_d, e_d) \\
& \mid \textbf{mkpar}\ e_d \\
& \mid \textbf{apply}\ e_d\ e_d \\
& \mid \textbf{get}\ e_d\ e_d \\
& \mid \textbf{if}\ e_d\ \textbf{at}\ e_d\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\
& \mid \textbf{request}\ e_d\ e_d
\end{array}
$$

et les valeurs pour l'évaluation locale (attention **request** n'est pas une valeur) :

$$
\begin{array}{rl}
v_d ::= & \textbf{fun}\ x \rightarrow e_d \\
& \mid c \\
& \mid op \\
& \mid (v_d, v_d)
\end{array}
$$

Pour désynchroniser notre langage, nous avons besoin d'un moyen de stocker les valeurs de chaque super-étape pour pouvoir les transmettre à un processeur qui pourrait en avoir besoin ultérieurement. Pour cela nous introduisons un environnement de communication $\mathcal{E}_C$, dans lequel chaque processeur stocke la valeur à transmettre à chaque super-étape. Les éléments stockés dans $\mathcal{E}_C$ sont de la forme (n, v_d), avec v_d la valeur à transmettre à la superétape n. On définit aussi une fonction **mstep** qui permet de récupérer le numéro de la dernière super-étape :

$$
\begin{cases}
\textbf{mste}p([]) = 0 \\
\textbf{mstep}((n, v_d) :: \mathcal{E}_C) = n.
\end{cases}
$$

3.2.2 Evaluation locale

Nous donnons les axiomes pour la relation, $(e, \mathcal{E}_C) \stackrel{\xi}{\rightarrow}_i (e', \mathcal{E}'_C)$, de réduction en tête. L'expression e dans l'environnement de communication $\mathcal{E}_C$ se réécrit en l'expression e' dans l'environnement $\mathcal{E}'_C$.

$$\begin{array}{llll} ((\textbf{fun}\ x \rightarrow e_d)\ v_d, \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (e_d[x \leftarrow v_d], \mathcal{E}_C) & (\beta_{fun}) \\ ((\textbf{let}\ x = v_d\ \textbf{in}\ e_d), \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (e_d[x \leftarrow v_d], \mathcal{E}_C) & (\beta_{let}) \end{array}$$

les δ-règles pour les opérateurs :

$$\begin{array}{llll} (+(n_1, n_2), \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (n, \mathcal{E}_C) \text{ avec } n = n_1 + n_2 & (\delta_+) \\ (\textbf{fst}(v_{d_1}, v_{d_2}), \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (v_{d_1}, \mathcal{E}_C) & (\delta_{fst}) \\ (\textbf{snd}(v_{d_1}, v_{d_2}), \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (v_{d_2}, \mathcal{E}_C) & (\delta_{snd}) \\ (\textbf{fix}(\textbf{fun}\ x \rightarrow\ e_d), \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (e_d[x \leftarrow \textbf{fix}(\textbf{fun}\ x \rightarrow\ e_d)], \mathcal{E}_C) & (\delta_{fix}) \\ (\textbf{fix}(\textbf{op}), \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (\textbf{op}, \mathcal{E}_C) & \\ (\textbf{if true then}\ e_1\ \textbf{else}\ e_2), \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (e_1, \mathcal{E}_C) & (\delta_{ifthenelse}) \\ (\textbf{if false then}\ e_1\ \textbf{else}\ e_2), \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (e_2, \mathcal{E}_C) & (\delta_{ifthenelse}) \end{array}$$

et les δ-règles les constructions parallèles :

$$\begin{array}{llll} (\textbf{mkpar}\ v_d, \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (v_d\ i, \mathcal{E}_C) & (\delta_{mkpar}) \\ (\textbf{apply}\ v_{d_1}\ v_{d_2}, \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (v_{d_1}\ v_{d_2}, \mathcal{E}_C) & (\delta_{apply}) \\ (\textbf{get}\ v_d\ j, \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (\textbf{request}\ (\textbf{mstep}(\mathcal{E}_C) + 1)\ j, & (\delta^{dst}_{get}) \\ & & (\textbf{mstep}(\mathcal{E}_C) + 1, v_d) :: \mathcal{E}_C) \text{ si } j \neq i & \\ (\textbf{get}\ v_d\ i, \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (v_d, (\textbf{mstep}(\mathcal{E}_C) + 1, v_d) :: \mathcal{E}_C) & (\delta^{loc}_{get}) \\ (\textbf{if}\ b\ \textbf{at}\ n\ \textbf{then}\ v_1\ \textbf{else}\ v_2, \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (\textbf{if}\ (\textbf{request}\ (\textbf{mstep}(\mathcal{E}_C) + 1)\ n) & (\delta^{dst}_{ifat}) \\ & & \textbf{then}\ v_1\ \textbf{else}\ v_2, (\textbf{mstep}(\mathcal{E}_C) + 1, b) :: \mathcal{E}_C) & \\ & & \text{si } n \neq i & \\ (\textbf{if}\ b\ \textbf{at}\ i\ \textbf{then}\ v_1\ \textbf{else}\ v_2, \mathcal{E}_C) & \stackrel{\xi}{\rightarrow}_i & (\textbf{if}\ b\ \textbf{then}\ v_1\ \textbf{else}\ v_2, & \\ & & (\textbf{mstep}(\mathcal{E}_C) + 1, b) :: \mathcal{E}_C) & (\delta^{loc}_{ifat}) \end{array}$$

les contextes :

$$
\begin{array}{rcl}
\Gamma & ::= & \bullet \\
& | & \Gamma\ e_d \\
& | & v_d\ \Gamma \\
& | & \textbf{let}\ x = \Gamma\ \textbf{in}\ e_d \\
& | & (\Gamma, e_d) \\
& | & (v_d, \Gamma) \\
& | & \textbf{mkpar}\ \Gamma \\
& | & \textbf{apply}\ \Gamma\ e_d \\
& | & \textbf{apply}\ v_d\ \Gamma \\
& | & \textbf{get}\ \Gamma\ e_d \\
& | & \textbf{get}\ v_d\ \Gamma \\
& | & \textbf{if}\ \Gamma\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\
& | & \textbf{if}\ \Gamma\ \textbf{at}\ e_d\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\
& | & \textbf{if}\ v_d\ \textbf{at}\ \Gamma\ \textbf{then}\ e_d\ \textbf{else}\ e_d
\end{array}
$$

et la règle de contexte :

$$\frac{(e, \mathcal{E}_C) \stackrel{\varepsilon}{\rightharpoonup}_i (e', \mathcal{E}'_C)}{(\Gamma[e], \mathcal{E}_C) \rightharpoonup_i (\Gamma[e'], \mathcal{E}'_C)}$$

3.2.3 Évaluation globale

Les expressions distribuées sont notées :

$$\langle\langle (e_{d_0}, \mathcal{E}_{C_0}), (e_{d_1}, \mathcal{E}_{C_1}), \ldots, (e_{d_{p-1}}, \mathcal{E}_{C_{p-1}}) \rangle\rangle$$

et les valeurs associées à ces expressions sont :

$$\langle\langle (v'_0, \mathcal{E}_{C_0}), (v'_1, \mathcal{E}_{C_1}), \ldots, (v'_{p-1}, \mathcal{E}_{C_{p-1}}) \rangle\rangle$$

avec :

$$
\begin{array}{rcl}
v' & ::= & \textbf{fun}\ x \rightarrow e_d \\
& | & c \\
& | & op \\
& | & (v', v')
\end{array}
$$

Pour l'évaluation globale nous ajoutons :

$$\forall i \frac{(e_{d_i}, \mathcal{E}_{C_i}) \rightharpoonup_i (e'_{d_i}, \mathcal{E}'_{C_i})}{\begin{array}{c}\langle\langle (e_{d_0}, \mathcal{E}_{C_0}), \ldots, (e_{d_i}, \mathcal{E}_{C_i}), \ldots, (e_{d_{p-1}}, \mathcal{E}_{C_{p-1}}) \rangle\rangle \\ \rightarrow \\ \langle\langle (e_{d_0}, \mathcal{E}_{C_0}), \ldots, (e'_{d_i}, \mathcal{E}'_{C_i}), \ldots, (e_{d_{p-1}}, \mathcal{E}_{C_{p-1}}) \rangle\rangle\end{array}}$$

et pour la communication nous ajoutons :

$$\forall i \frac{(e_{d_i} = \Gamma[\textbf{request}\ n\ j]) \wedge ((n, v_d) \in \mathcal{E}_{C_j})}{\begin{array}{c}\langle\langle (e_{d_0}, \mathcal{E}_{C_0}), \ldots, (e_{d_i}, \mathcal{E}_{C_i}), \ldots, (e_{d_{p-1}}, \mathcal{E}_{C_{p-1}})\rangle\rangle \\ \rightarrow \\ \langle\langle (e_{d_0}, \mathcal{E}_{C_0}), \ldots, (\Gamma[v_d], \mathcal{E}_{C_i}), \ldots, (e_{d_{p-1}}, \mathcal{E}_{C_{p-1}})\rangle\rangle\end{array}}$$

Exemple 3 *Toujours sur l'exemple de la diffusion, avec*

$$\left\{ \begin{array}{rcl} p & = & 3 \\ i & = & 2 \\ v & = & \textbf{\textit{mkpar}}(\textbf{\textit{fun}}\ x \rightarrow\ 2 \times x) \end{array} \right.$$

L'évaluation distribuée de

$$\mathit{bcast}\ 2\ (\textbf{\textit{mkpar}}(\textbf{\textit{fun}}\ x \rightarrow\ 2 \times x))$$

débute par l'évaluation locale à chaque processeur. Au processeur i cette réduction est :

$$\begin{array}{rl} & \big(\textbf{\textit{get}}\ (\textbf{\textit{mkpar}}(\textbf{\textit{fun}}\ x \rightarrow\ 2))\ (\textbf{\textit{mkpar}}(\textbf{\textit{fun}}\ x \rightarrow\ 2 \times x))\ ,\ []\big) \\ \rightharpoonup_i & \big(\textbf{\textit{get}}\ ((\textbf{\textit{fun}}\ x \rightarrow\ 2)\ i)\ (\textbf{\textit{mkpar}}(\textbf{\textit{fun}}\ x \rightarrow\ 2 \times x))\ ,\ []\big) \\ \rightharpoonup_i & \big(\textbf{\textit{get}}\ 2\ (\textbf{\textit{mkpar}}(\textbf{\textit{fun}}\ x \rightarrow\ 2 \times x))\ ,\ []\big) \\ \rightharpoonup_i & \big(\textbf{\textit{get}}\ 2\ ((\textbf{\textit{fun}}\ x \rightarrow\ 2 \times x)\ i)\ ,\ []\big) \\ \rightharpoonup_i & \big(\textbf{\textit{get}}\ 2\ 2i\ ,\ []\big) \\ \rightharpoonup_i & \big(\textbf{\textit{request}}\ 0\ 2\ ,\ [(0, 2i)]\big) \end{array}$$

Puis, on réduit globalement :

$$\begin{array}{rl} & \langle\langle (\textbf{\textit{request}}\ 0\ 2, [(0,0)]), (\textbf{\textit{request}}\ 0\ 2, [(0,2)]), (\textbf{\textit{request}}\ 0\ 2, [(0,4)])\rangle\rangle \\ \stackrel{3}{\rightarrow} & \langle\langle (4, [(0,0)]), (4, [(0,2)]), (4, [(0,4)]),)\rangle\rangle \end{array}$$

Chapitre 4

Vers une implémentation

MSPML, tout comme BSML actuellement, sera implémenté sous forme de bibliothèque, la MSPML. Etant donné que nous conservons le même langage source, elle devra proposer les mêmes primitives que la BSMLlib présentée au chapitre 2, sauf que la fonction get devient une primitive. Bien entendu, la différence réside en l'implémentation de ces primitives. Les autres fonctions seront implémentée dans ce qu'on appelle la bibliothèque standard.

4.1 Principes d'implémentation

Afin d'implémenter la sémantique présentée au chapitre précédent, il faudra que nos instances de programmes soient composées de deux processus légers :

– un pour les calculs séquentiels (qui correspond à $e \xrightarrow{\varepsilon}_i e'$)

– un pour les communications (réalise $\rightarrow$)

A chaque requête reçue, le processus léger de communication lancera un nouveau processus léger qui la prendra en charge, renverra la valeur au demandeur puis s'arrêtera.

Le module Marshall de Objective Caml sera utilisé pour linéariser les messages à l'aide de la fonction to_string, et la fonction from_string pour l'opération inverse.

Pour l'environnement de communication, il sera implémenté à l'aide un tableau. Afin de libérer l'espace associé aux environnements de communication, le programmeur entrera sur la ligne de commande un entier s qui imposera une synchronisation des processeurs toutes les s super-étapes. Cet entier correspond à ce qui est définit dans Caml-Flight comme la profondeur d'asynchronisme [9]. Notre

implémentation au dessus de TCP/IP reprend des parties de l'implémentation récente de Caml-Flight [9].

4.2 Le module MSPML

Nous ne représentons pas ici les primitives **mkpar**, **apply**, **get** et **ifat**.

- **initialize** sera la fonction qui se chargera de la création des deux processus légers qui constituent une instance d'un programme MSPML.
- **finalize** s'occupera d'arrêter ces deux processus légers à la fin du programme.
- **start_timing**, **stop_timing** et **get_cost** seront implémentées pour avoir accès au temps d'exécution. Elles nous serviront entre autres lors de la phase de tests à valider notre modèle de coûts.

Chapitre 5

Modèle de coûts

Au chapitre 2, nous avons présenté le modèle BSP et son modèle de coûts. Ce dernier n'est pas applicable à MSPML contrairement à BSML. En effet, notre ordinateur parallèle ne possède plus que deux composants : un ensemble homogène de paires processeurs-mémoires, et un réseau de communication permettant l'échange de messages entre chaque couple de processeurs. Dans ce chapitre, nous exposons trois modèles de coûts suceptibles de convenir à notre langage : BSPWB et MPM introduits dans [18], et LogP introduit dans [19]. Suite à cela nous, déterminerons lequel semble le plus réaliste et le plus proche de notre langage.

5.1 BSPWB : BSP Without Barrier

BSPWB, pour *BSP Without Barrier*, est un modèle de coûts directement inspiré de celui de BSP. Il propose de remplacer la notion de superétape par celle de M-step, pour *Message Step*, qu'il définit comme suit : lors d'une M-step, chaque processeur effectue une phase de calcul séquentiel, puis une phase de communication pendant laquelle les processeurs s'échangent les données dont ils auront besoin à la M-step suivante. Il s'agit en fait d'une généralisation directe de la super-étape.

Comme pour le modèle BSP, l'ordinateur parallèle est caractérisé par trois paramètres :

- le nombre de paires processeur-mémoire p,
- le temps nécessaire L (latence)pour l'accès au réseau,
- le temps g que prend l'acheminement d'un mot de un processeur à un autre.

Le temps nécessaire au processeur i à l'exécution d'une M-step s, $t_{s,i}$ est borné

par T_s le temps d'exécution de la M-step s. T_s est définit inductivement comme suit :

$$T_1 = max\{w_{1,i}\} + \max\{g * h_{1,i} + L\} \text{ pour } i = 1, 2, \ldots, p-1$$

$$T_s = T_\{s-1\} + max\{w_{s,i}\} + max\{g * h_{s,i} + L\}$$

pour $i = 0, 1, \ldots, p-1$ et $s = 2, 3, \ldots, R$,
où R est le nombre de M-step du programme.
avec $w_{s,i}$ et $h_{s,i}$ respectivement dénotent le temps de calcul local sur le processeur i pendant la M-step s et le $max\{h^+_{s,i}, h^-_{s,i}\}$ où $h^+_{s,i}$ (respectivement $h^-_{s,i}$) est le nombre de mots envoyés (respectivement reçus) par le processeur i durant la M-step s. Le temps d'exécution du programme MSPML est donc borné par T_R.

Il apparait assez clairement qu'utiliser ce modèle de coût pour prédire le temps d'exécution d'un programme MSPML donnerait des résultats trop grossiers. On se propose donc d'examiner un autre modèle du coût dont les résultats devraient approcher plus précisément le temps d'exécution d'un programme MSPML.

5.2 MPM : Message Passing Machine

Une borne $\Phi_{s,i}$ plus proche de $t_{s,i}$ calculée par le modèle BSPWB est donnée par le modèle MPM présenté dans cette section.
Dans ce modèle, l'ordinateur parallèle est caractérisé par les même paramètres p, L et g que dans le modèle BSPWB.

Il introduit l'ensemble $\Omega_{s,i}$ pour un processeur donné i à une M-step s définit tel que :

$$\Omega_{s,i} = \{j/le\ processeur\ j\ envoie\ un\ message\ au\ processeur\ i \\ la\ M step\ s\} \bigcup \{i\}$$

Les processeurs inclus dans $\Omega_{s,i}$ sont définis comme les "partenaires entrants" du processeur i à la M-step s. Puis $\Phi_{s,i}$ est calculé inductivement grâce aux formules suivantes :

$$\Phi_{1,i} = max\{w_{1,j}/j \ \in \Omega_{1,i}\} + (g * h_{1,i} + L)\ pour\ i = 1, 2, \ldots, p-1$$

$$\Phi_{s,i} = max\{\Phi_{s-1,j} + w_{s-1,j}/j \ \in \Omega_{s,i}\} + (g * h_{s,i} + L)$$

$$pour\ i = 0, 1, \ldots, p-1\ et\ s = 2, 3, \ldots, R$$

et

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

Le temps d'exécution du programme MSPML est donc borné par :

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

Le modèle MPM traduit bien le fait que un processeur ne se synchronise plus que avec ses "partenaires entrants". En effet, pour calculer le temps $\Phi_{s,i}$ d'exécution pour le processeur i de la M-step s, on calcule le temps d'exécution de tous ses partenaires entrant pour la M-step s et on se synchronise avec.

5.3 LogP

LogP propose un modèle d'architecture tenant compte des caractéristiques des machines actuelles. Il repose sur une machine abstraite composée d'unités de calcul asynchrone possédant chacune une mémoire locale. Ces unités sont reliées entre elles par un réseau d'interconnexion. Elles communiquent par des messages point par point.

Quatre paramètres caractérisent l'architecture :

- Le nombre p de processeurs
- La bande passante g pour les communications. Elle correspond à l'intervalle de temps minimum entre deux transmissions consécutives par un même processeur.
- Le délai de communication L. Il correspond au temps maximum nécessaire pour acheminer un message d'un mot du processeur source au processeur cible.
- Le surcoût o dû aux communications. Il correspond au temps où un processeur est occupé à la transmission ou à la réception d'un message.

Le paramètre o correspond au paramètre $n_{1/2}$ de BSP, sauf que ce dernier peut être négligé pour la plupart des programmes. De plus, on peut négliger sur certaines architectures le paramètre g par rapport à o, ou même augmenter o de telle sorte à ce que o soit égal à g et g pouvant alors être négligé. Si on garde

le paramètre g de LogP et on néglige le o, on peut réécrire les formules de coût proposée dans la section précédente, pour le modèle LogP. On aura :

$$g_{MPM} = 4 \times o_{LogP} + 2 \times L_{LogP}$$

et

$$L_{MPM} = g_{LogP}$$

Il apparaît donc que changer de modèle en faveur du modèle LogP n'apporte pas un modèle de coût plus réaliste. Pour garder donc un modèle de coût simple et assez proche de celui de BSML, nous proposons d'adopter le modèle MPM et son modèle de coûts pour notre langage.

Chapitre 6

Conclusion

En partant de la sémantique distribuée de [5] nous avons défini une sémantique minimalement synchrone de mini-BSML très proche de l'implémentation.

Reste maintenant à finaliser l'implémentation comme il est prévu dans le chapitre 5, et valider son modèle de coûts par des tests de l'implantation. On pourrait comparer les performances ainsi obtenues à celles de la BSMLlib et de Caml-Flight, et vérifier que les motivations d'optimisation pour ce travail étaient bien justifiées.

Il faudrait ensuite prouver le théorème suivant :

$$\forall\, e, v \quad (e \triangleright v \quad \textbf{ssi} \quad \langle\langle e, e, \ldots, e\rangle\rangle \rightarrow \langle\langle \mathcal{T}_0(v), \mathcal{T}_1(v), \ldots, \mathcal{T}_{p-1}(v)\rangle\rangle)$$

$$\begin{array}{rcl}
\mathcal{T}_i(x) & = & x \\
\mathcal{T}_i(c) & = & c \\
\mathcal{T}_i(op) & = & op \\
\mathcal{T}_i(\textbf{fun}\ x \rightarrow e) & = & \textbf{fun}\ x \rightarrow \mathcal{T}_i(e) \\
\mathcal{T}_i(e_1\ e_2) & = & (\mathcal{T}_i(e_1)\ \mathcal{T}_i(e_2)) \\
\mathcal{T}_i(\textbf{let}\ x = e_1\ \textbf{in}\ e_2) & = & \textbf{let}\ x = \mathcal{T}_i(e_1)\ \textbf{in}\ \mathcal{T}_i(e_2) \\
\mathcal{T}_i((e_1, e_2)) & = & (\mathcal{T}_i(e_1), \mathcal{T}_i(e_2)) \\
\mathcal{T}_i(\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3) & = & \textbf{if}\ \mathcal{T}_i(e_1)\ \textbf{then}\ \mathcal{T}_i(e_2)\ \textbf{else}\ \mathcal{T}_i(e_3) \\
\mathcal{T}_i(\textbf{mkpar}\ e) & = & \textbf{mkpar}\ \mathcal{T}_i(e) \\
\mathcal{T}_i(\textbf{apply}\ e_1\ e_2) & = & \textbf{apply}\ \mathcal{T}_i(e_1)\ \mathcal{T}_i(e_2) \\
\mathcal{T}_i(\textbf{get}\ e_1\ e_2) & = & \textbf{get}\ \mathcal{T}_i(e_1)\ \mathcal{T}_i(e_2) \\
\mathcal{T}_i(\textbf{if}\ e_1\ \textbf{at}\ e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4) & = & \textbf{if}\ \mathcal{T}_i(e_1)\ \textbf{at}\ \mathcal{T}_i(e_2)\ \textbf{then}\ \mathcal{T}_i(e_3)\ \textbf{else}\ \mathcal{T}_i(e_4) \\
\mathcal{T}_i(\langle e_0, ..., e_i, ..., e_{p-1}\rangle) & = & e_i
\end{array}$$

Bibliographie

[1] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8) :103, August 1990.

[2] P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science. Springer, 1996.

[3] G. Akerholt, K. Hammond, S. Peyton-Jones, and P. Trinder. Processing transactions on GRIP, a parallel graph reducer. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, Munich, June 1993. Springer.

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

[5] F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4) :423–437, 2001.

[6] G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation : Theory and Practice, pages 165–178. Nova Science Publishers, august 2002.

[7] F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In Stephen Gilmore, editor, *Proceedings of the second Scottish Functional Programming Workshop*, St Andrews, July 2000.

[8] Xavier Leroy. The Objective Caml System 3.06, 2002. web pages at www.ocaml.org.

[9] E. Chailloux and C. Foisy. Caml-Flight alpha : Implantation et applications. In C. Queinnec, V. V. Donzeau-Gouge, and P. Weis, editors, *Journées Fran-*

cophones des Langages Applicatifs, number 13 in Collection Didactique. INRIA, Janvier 1995.

[10] G. Cousineau and M. Mauny. *Approche Fonctionnelle de la Programmation*. Ediscience International, 1995.

[11] Frédéric Gava. A Polymorphic Type System for BSML. Technical Report 2002-12, Master Thesis, University of Paris Val-de-Marne, LACL, 2002.

[12] Armelle Merlin. Bslambda simplement typé : typage et sémantique naturelle. Master's thesis, LIFO, Université d'Orléans, 2000.

[13] W. F. McColl. Universal computing. In L. Bouge and al., editors, *Proc. Euro-Par '96*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[14] Jonathan M. D. Hill and David B. Skillicorn. Practical Barrier Synchronisation. In *6th EuroMicro Workshop on Parallel and Distributed Processing (PDP'98)*. IEEE Computer Society Press, January 1998.

[15] J.M.D. Hill, W.F. McColl, and al. BSPlib : The BSP Programming Library. *Parallel Computing*, 24 :1947–1980, 1998.

[16] G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, number 694 in LNCS, pages 56–67. Springer, 1993.

[17] C. Foisy and E. Chailloux. Caml Flight : a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.

[18] 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.

[19] D. Culler, R. Karp, D. Patterson, A. Sahay, K. E. Schauser, E. Santos, R. Subramonian, and T. von Eicken. LogP : Towards a realistic model of parallel computation. In *Fourth Symposium on Principles and Practice of Parallel Programming*, San Diego, California, May 1993. ACM SIGPLAN.