La machine abstraite catégorique BSP

A. Merlin & G. Hains

Laboratoire d'Informatique Fondamentale d'Orléans,
45068 Orléans CEDEX 2, France
{merlin,ghains}@lifo.univ-orleans.fr

Résumé

Bulk-synchronous ML ou BSML est une extension du langage CAML pour la programmation fonctionnelle parallèle. Ce langage utilise le modèle BSP, bulk-synchronous parallelism, qui permet un calcul de coût réaliste et des echanges explicites entre processus.

Dans le but de formaliser l'implantation de BSML, nous ajoutons à la CAM (une machine à environnement pouvant servir de base à CAML) les opérations du BSλ-calcul, notre modèle théorique de la synthèse entre BSP et la programmation fonctionnelle. Les opérations ajoutées sont un constructeur parallèle, l'application parallèle de fonctions, une conditionnelle globale et une opération de communication globale synchrone.

La CAM a été dérivée par Curien à partir de sa théorie des combinateurs catégoriques. Nous appliquons à cette dérivation le principe des deux niveaux de langage (local et global) propre à BSML. Il en résulte une description exacte du rapport entre les opérations BSML et leur implantation par une machine à environnement en mémoire répartie.

1. Introduction

La programmation des ordinateurs massivement parallèles pose le problème de l'équilibre entre l'efficacité des calculs et la simplicité des langages. Deux styles de programmation sont surtout utilisés : la programmation data-parallèle avec processus implicites et la programmation par envoi de messages. La première a l'avantage de la simplicité car si elle explicite les opérations potentiellement parallèles, elle laisse la responsabilité du placement des tâches et des données au système. Cela empêche l'établissement de tout lien entre modèle de performance et langage source, alors que le but même de la programmation parallèle est la mise en œuvre d'algorithmes pour lesquels des accélérations sont prévues. Nous préconisons plutôt l'utilisation de processus (physiques) explicites, comme c'est le cas de la programmation par envoi de messages par exemple avec la bibliothèque MPI. Malheureusement les réalisations actuelles de ce style de programmation sont beaucoup trop complexes à l'utilisation car elles plongent le problème, certes combinatoire, du placement des données dans le monde de la programmation concurrente dont la logique est d'une complexité infiniment plus grande. En particulier, il en résulte des problèmes d'interblocage et de mise au point non-déterministe des programmes.

Pour combiner la transparence des performances en présence de processus explicites avec la simplicité d'une programmation véritablement applicative, nous travaillons à la réalisation d'une extension minimale du langage CAML selon le paradigme BSP : bulk-synchronous parallelism. Ce modèle [12, 10] propose le regroupement des communication inter-processus en des opérations collectives effectuées à l'occasion de barrières de synchronisation. Il assure l'absence d'interblocage et le déterminisme des calculs, en plus de fournir un modèle simple et portable d'évaluation des performances parallèles. Nous avons mis au point une version BSP du lambda-calcul (BSλ) qui préserve la confluence [9] et vérifié que son implantation en appel par valeur préservait le réalisme du modèle de performance [1].

L'implantation présentée dans [1] est une bibliothèque appelée BSMLlib réalisée autour d'Ocaml [7] et BSPlib [6]. Ne modifiant pas le langage source Ocaml, elle ne peut réaliser les contraintes statiques nécessaires à notre modèle de programmation applicative BSP. De plus, la bibliothèque n'a pas été formellement modélisée et les constantes qu'elle engendre dans le modèle de performance ne peuvent être expliquées qu'a posteriori sur chaque programme test. Pour éliminer ces inconvénients nous avons entrepris la construction systématique d'une version BSP de CAML par la définition et le test de machines à environnement. La première [11] a bien montré que les constantes du modèle de coût pouvaient provenir d'une analyse a priori ce qui permet de prévoir la construction d'outils d'analyse statique BSP sur les programmes CAML. Mais, étant une machine SECD, elle utilise une théorie ad hoc sur les variables qui ne correspond ni à l'état actuel des recherches ni aux implantations actuelles. Nous présentons ici la dérivation systématique d'une machine abstraite catégorique (CAM) selon le raisonnement suivi par Curien [2] pour l'application des combinateurs catégoriques à l'implantation de CAML. Cette méthode de travail met en lumière plusieurs détails importants du traitement BSP des variables, environnements, piles etc. et produit une spécification claire et détaillée d'une implantation BSP de CAML. Nous illustrons le résultat par son exécution sur de petits exemples et énumérons ses nombreux avantages par rapport à une ancienne version data-parallèle de la CAM [5, 4].

2. Les algorithmes parallèles BSP et BSML

Le modèle d'exécution [12] représente un calcul parallèle sur p processeurs comme une suite de phases de calcul asynchrone alternées avec des phases de synchronsation globale (d'où l'adjectif *bulk-synchronous*) et d'échange de données. Le modèle de performance BSP estime le temps d'exécution d'une phase asynchrone par le temps de calcul local maximum et le temps d'une phase de synchronisation et communication par une formule affine dont la constante mesure la latence, la pente exprime le temps de traitement d'un message et la variable est la taille de la plus grande file de messages sur un processeur.

Dans la description de la programmation de tels algorithmes, nous qualifions de *local*, noté $\dot{x}$, ce qui est relatif à un seul processeur et de *global*, noté $\bar{x}$ ce qui implique l'ensemble des p processeurs. La version BSP de CAML que nous proposons s'appelle BSML. Elle ajoute à CAML un constructeur polymorphe de type `'a Par` qui représente des structures parallèles contenant une valeur par processeur. Il est interdit de les imbriquer et la dérivation mathématique de la BSP-CAM réalise scrupuleusement cette contrainte. BSML contient aussi un petit nombre d'opérations sur les types `Par`. Le constructeur de ce type s'appelle `make-par` et prend pour argument une fonction `int -> 'a` qu'il applique aux rangs des processeurs $0, 1, ..., p-1$ ce qui produit une valeur parallèle. Les phases de calcul asynchrones sont réalisées par `apply-par` dont le premier argument est une valeur de type `('a -> 'b) Par` qu'il applique sur chaque processeur à une valeur de type `'a Par`. Les phases de synchronisation et de communication sont réalisées par la fonction `put` qui prend une valeur parallèle contenant en chaque processeur un ensemble de paires message-destinataire. Elle achemine ces messages et rend une valeur parallèle contenant en chaque processeur un ensemble de paires message-émetteur. Comme le type `Par` ne possède pas d'observateur, il serait impossible d'orienter les calculs en fonction du contenu des valeurs parallèles si BSML ne contenait une conditionnelle parallèle `if at` (if E at e then E1 else E2) de type `bool Par * int -> 'a Par -> 'a Par -> 'a Par`. Elle évalue une de ses deux branches selon la valeur du premier argument au processeur indiqué par le second argument.

La machine abstraite que nous allons présenter est une réalisation de ces opérations obtenue par l'intermédiaire de combinateurs ajoutés à la logique combinatoire catégorique de Curien. Les notations λ-calcul pour `make-par`, `apply-par`, `if at` et `put` sont respectivement πe, $E\#E$, $E \xrightarrow{e} E, E$ et $!E$.

La machine abstraite catégorique [3] est une machine à environnement pouvant implanter CAML. Elle est dérivé des combinateurs catégoriques de Curien [2] en trois temps. On donne d'abord une sémantique dénotationnelle pour un λ-calcul augmenté de paires et de couples. Elle produit des fonctions des environnements vers les domaines des valeurs. Ensuite le passage à la notation de De Bruijn permet l'encodage des environnements par des piles elles-mêmes codées par des λ-termes de forme $(...((), d_n)..,d_0)$. La deuxième étape consiste à définir une sémantique opérationnelle du même λ-calcul par une traduction vers des combinateurs directement inspirés de la sémantique dénotationnelle. La dernière étape consiste à définir une machine à environnement dont les instructions sont les combinateurs catégoriques et à compiler les λ-termes vers cette machine de manière à réaliser la sémantique opérationnelle.

Pour dériver une BSP-CAM selon la même méthode, nous remplaçons d'abord le λ-calcul par BS-λ [9] augmenté des couples et des paires, c'est-à-dire le λ-calcul de Curien avec nos quatre opérations parallèles. On ajoute aussi un domaine sémantique de vecteurs parallèles pour chaque domaine existant. Ensuite, on ajoute aux combinateurs catégoriques trois combinateurs suffisants pour réaliser nos quatre opérations parallèles.

Dans ce qui suit, nous ne présentons que les extensions non triviales de la démarche de Curien. Toutes les équations manquantes sont soit celles de [2], soit des versions parallèles triviales. Notre démarche utilisera implicitement des axiomes sur l'interaction entre vecteurs, paires, couples et application. Ces axiomes sont donnés en annexe.

3. Equations sémantiques avec environnement

Initialement nous devions utiliser l'encodage de Church pour exprimer les instructions conditionnelles. Or, cela implique une forme "interdite" : $\bar{t} \to \dot{t}$, dans le cas $if\ \dot{x}\ then\ \bar{y}\ else\ \bar{z}$. Puisque cette expression est transformée en $(\dot{x}(\bar{y}\bar{z}))$. Nous proposons donc deux opérateurs conditionnels : $A \to B, C$ pour les cas *if* $\dot{a}$ *then* $\dot{b}$ *else* $\dot{c}$ et *if* $\dot{a}$ *then* $\bar{b}$ *else* $\bar{c}$, et $A \overset{e}{\to} B, C$ pour le cas $\bar{t} \to \bar{t}$ avec communication.

3.1. Les équations sémantiques

Les équations sémantiques du λ-calcul avec environnement ρ sont reprises de celles présentées par Curien, auxquelles nous ajoutons l'instruction conditionnelle et les instructions spécifiques du BSλ-calcul.

Equations sur la conditionnelle.

$$[\![\dot{M} \to \dot{N}, \dot{P}]\!] = [\![\dot{M}]\!] \to [\![\dot{N}]\!], [\![\dot{P}]\!]$$
$$[\![\dot{M} \to \bar{N}, \bar{P}]\!] = [\![\dot{M}]\!] \to [\![\bar{N}]\!], [\![\bar{P}]\!]$$

Opérations spécifiques du BSλ-calcul.

$$[\![\pi \dot{M}]\!]\rho = ([\![\dot{M}]\!]\rho) \ll 0\ , \ \dots\ , \ (p-1) \gg$$
$$[\![\bar{M} \# \bar{N}]\!]\rho = ([\![\bar{M}]\!]\rho)([\![\bar{N}]\!]\rho)$$
$$[\![\bar{M} \overset{e}{\to} \bar{N}, \bar{P}]\!]\rho = (\lambda x.x \to ([\![\bar{N}]\!]\rho), ([\![\bar{P}]\!]\rho))([\![\bar{M}]\!]_\rho)_{[\![e]\!]_\rho}$$
$$[\![!\bar{M}]\!]\rho =$$
$$\ll \lambda i \begin{pmatrix} i = 0 \to ([\![\bar{M}]\!]\rho)_0 0, \\ \dots \\ i = p-1 \to ([\![\bar{M}]\!]\rho)_{(p-1)} 0, nc \end{pmatrix}\ , \ \dots\ , \ \lambda i \begin{pmatrix} i = 0 \to ([\![\bar{M}]\!]\rho)_0 (p-1), \\ \dots \\ i = p-1 \to ([\![\bar{M}]\!]\rho)_{(p-1)} (p-1), nc \end{pmatrix} \gg$$

Le multi-put ou $[\![!\bar{M}]\!]$ est un point important de la communication. Chaque destinataire j reçoit des expéditeurs i, des valeurs qui sont sous forme de fonction imbriquée de termes conditionnels. Cette forme permet de lister les valeurs reçues en fonction des expéditeurs. En chaque point i (expéditeur), on calcule la valeur à envoyer à chaque j (destinataire), c'est-à-dire $([\![\bar{M}]\!]\rho)_i 0, ..., ([\![\bar{M}]\!]\rho)_i (p-1)$. Chaque destinataire j reçoit les valeurs correspondantes à ces évaluations $([\![\bar{M}]\!]\rho)_0 j, ..., ([\![\bar{M}]\!]\rho)_{(p-1)} j$. Si l'expéditeur j n'existe pas la constante nc, (no communication) remplace le message.

3.2. Récapitulatif des équations sémantiques avec environnement

$$
\begin{array}{l}
\llbracket x \rrbracket \rho = \rho(x) \\
\llbracket c \rrbracket \rho = c \\
\llbracket \dot{M} \dot{N} \rrbracket \rho = \llbracket \dot{M} \rrbracket \rho (\llbracket \dot{N} \rrbracket \rho) \\
\llbracket \bar{M} \dot{N} \rrbracket \rho = \llbracket \bar{M} \rrbracket \rho (\llbracket \dot{N} \rrbracket \rho) \\
\llbracket \bar{M} \bar{N} \rrbracket \rho = \llbracket \bar{M} \rrbracket \rho (\llbracket \bar{N} \rrbracket \rho) \\
\llbracket \lambda \dot{x}.\dot{M} \rrbracket \rho d = \llbracket \dot{M} \rrbracket (\rho[\dot{x} \leftarrow d]) \\
\llbracket \lambda \dot{x}.\bar{M} \rrbracket \rho d = \llbracket \bar{M} \rrbracket (\rho[\dot{x} \leftarrow d]) \\
\llbracket \lambda \bar{x}.\bar{M} \rrbracket \rho d = \llbracket \bar{M} \rrbracket (\rho[\bar{x} \leftarrow d]) \\
\llbracket \dot{M} \rightarrow \dot{N}, \dot{P} \rrbracket = \llbracket \dot{M} \rrbracket \rightarrow \llbracket \dot{N} \rrbracket, \llbracket \dot{P} \rrbracket \\
\llbracket \dot{M} \rightarrow \bar{N}, \bar{P} \rrbracket = \llbracket \dot{M} \rrbracket \rightarrow \llbracket \bar{N} \rrbracket, \llbracket \bar{P} \rrbracket \\
\llbracket \pi \dot{M} \rrbracket \rho = (\llbracket \dot{M} \rrbracket \rho) \ll 0 \, , \, \dots \, , \, (p-1) \gg \\
\llbracket \bar{M} \# \bar{N} \rrbracket \rho = (\llbracket \bar{M} \rrbracket \rho)(\llbracket \bar{N} \rrbracket \rho) \\
\llbracket \bar{M} \overset{e}{\rightarrow} \bar{N}, \bar{P} \rrbracket \rho = (\lambda x.x \rightarrow (\llbracket \bar{N} \rrbracket \rho), (\llbracket \bar{P} \rrbracket \rho))(\llbracket \bar{M} \rrbracket_{\rho})_{\llbracket e \rrbracket_{\rho}} \\
\llbracket ! \bar{M} \rrbracket \rho = \ll \lambda i \begin{pmatrix} i = 0 \rightarrow (\llbracket \bar{M} \rrbracket \rho)_0 0, \\ \dots \\ i = p-1 \rightarrow (\llbracket \bar{M} \rrbracket \rho)_{(p-1)} 0, nc \end{pmatrix} \, , \, \dots \, , \, \lambda i \begin{pmatrix} i = 0 \rightarrow (\llbracket \bar{M} \rrbracket \rho)_0 (p-1), \\ \dots \\ i = p-1 \rightarrow (\llbracket \bar{M} \rrbracket \rho)_{(p-1)} (p-1), nc \end{pmatrix} \gg
\end{array}
$$

4. Equations sémantiques avec la notation de De Bruijn

Les équations précédentes sont transformées à l'aide de la notation de De Bruijn et à l'aide d'environnement sous la forme $(...((), d_n).., d_0)$ où d_i est associé à i. Des notations pour différencier variables $\underline{n}$ et constantes c sont introduites.

$$\llbracket \pi \dot{M} \rrbracket \rho = (\llbracket \dot{M} \rrbracket \rho) \ll 0 \, , \, \dots \, , \, (p-1) \gg$$

$$\llbracket \bar{M} \# \bar{N} \rrbracket \rho = (\llbracket \bar{M} \rrbracket \rho)(\llbracket \bar{N} \rrbracket \rho)$$

$$\llbracket \bar{M} \overset{e}{\rightarrow} \bar{N}, \bar{P} \rrbracket \rho = (\underline{0} \rightarrow (\llbracket \bar{N} \rrbracket \rho), (\llbracket \bar{P} \rrbracket \rho))(\rho, (\llbracket \bar{M} \rrbracket \rho)_{\llbracket e \rrbracket_{\rho}})$$

$$\llbracket ! \bar{M} \rrbracket \rho =$$

$$\ll \begin{pmatrix} \underline{0} = 0 \rightarrow (\llbracket \bar{M} \rrbracket \rho)_0 0, \\ \dots \\ \underline{0} = (p-1) \rightarrow (\llbracket \bar{M} \rrbracket \rho)_{(p-1)} 0, nc \end{pmatrix} (\rho, 0), \dots, \begin{pmatrix} \underline{0} = 0 \rightarrow (\llbracket \bar{M} \rrbracket \rho)_0 (p-1), \\ \dots \\ \underline{0} = (p-1) \rightarrow (\llbracket \bar{M} \rrbracket \rho)_{(p-1)} (p-1), nc \end{pmatrix} (\rho, (p-1)) \gg$$

5. Equations syntaxiques : combinateurs catégoriques

Les combinateurs suivants proviennent de la logique de combinateurs catégoriques. S, Λ sont les combinateurs de Curien, C, $\Uparrow$ et T sont ceux ajoutés pour introduire l'opération conditionnelle et les opérations parallèles.

Le S utilisé ici, est une forme non currifiée du S de la logique combinatoire de Curry. La logique catégorique combinatoire est une logique combinatoire avec produits explicites. C est le combinateur de la conditionnelle, $\Uparrow$ combinateur du broadcast, utilisé dans l'instruction de conditionnelle globale, et T combinateur de l'échange qui ressemble à une transposition de matrice utilisé pour exprimer le multi-put.

5.1. Nouvelles règles de la logique combinatoire

(var) $\underline{0}(x,y) = y$ et $\underline{(n+1)}(x,y) = \underline{n}x$
(const) $('x)y = x$
(app) $S(x,y)z = xz(yz)$
(dΛ) $\Lambda(x)yz = x(y,z)$
(cond) $C(true,(x,y)) = x \quad C(false,(x,y)) = y$
(proj) $\Uparrow \ll (i,x_0)\ ,\ \dots\ ,\ (i,x_{p-1}) \gg = \ll x_i\ ,\ \dots\ ,\ x_i \gg$
(transp) $T \ll (x_{00},...,x_{0(p-1)})\ ,\ \dots\ ,\ (x_{(p-1)0},...,x_{(p-1)(p-1)}) \gg =$

$$\ll \Lambda \left(\begin{array}{l} C(\underline{0} =' 0, (x_{00}, \\ \dots \\ C(\underline{0} =' (p-1), (x_{(p-1)0}, nc))...)) \end{array} \right)\ ,\ \dots\ ,\ \Lambda \left(\begin{array}{l} C(\underline{0} =' 0, (x_{0(p-1)}, \\ \dots \\ C(\underline{0} =' (p-1), (x_{(p-1)(p-1)}, nc))...)) \end{array} \right) \gg$$

5.2. Traduction Termes vers Combinateurs

$\llbracket n \rrbracket = \underline{n}$
$\llbracket c \rrbracket =' c$

$$\begin{array}{l|l|l} \llbracket \dot{M}\dot{N} \rrbracket = S(\llbracket \dot{M} \rrbracket, \llbracket \dot{N} \rrbracket) & \llbracket \lambda.\dot{M} \rrbracket = \Lambda(\llbracket \dot{M} \rrbracket) & \llbracket \dot{M} \to \dot{N}, \dot{P} \rrbracket = C((\llbracket \dot{M} \rrbracket), ((\llbracket \dot{N} \rrbracket), (\llbracket \dot{P} \rrbracket))) \\ \llbracket \bar{M}\dot{N} \rrbracket = S(\llbracket \bar{M} \rrbracket, \llbracket \dot{N} \rrbracket) & \llbracket \lambda.\bar{M} \rrbracket = \Lambda(\llbracket \bar{M} \rrbracket) & \llbracket \dot{M} \to \bar{N}, \bar{P} \rrbracket = C((\llbracket \dot{M} \rrbracket), ((\llbracket \bar{N} \rrbracket), (\llbracket \bar{P} \rrbracket))) \\ \llbracket \bar{M}\bar{N} \rrbracket = S(\llbracket \bar{M} \rrbracket, \llbracket \bar{N} \rrbracket) & & \end{array}$$

Seulement deux règles pour l'abstraction existent car la sorte de la variable est sous-entendue dans l'environnement. $\lambda\dot{x}.\bar{M}$ et $\lambda\bar{x}.\bar{M}$ sont confondus à ce niveau car les environnements confondent "scalaire" et "vecteur homogène".

$\llbracket \pi\dot{M} \rrbracket = \ll S(\llbracket \dot{M} \rrbracket,' 0)\ ,\ \dots\ ,\ S(\llbracket \dot{M} \rrbracket,' (p-1)) \gg$
$\llbracket \bar{M} \# \bar{N} \rrbracket = \ll S((\llbracket \bar{M} \rrbracket)_{'0}, (\llbracket \bar{N} \rrbracket)_{'0})\ ,\ \dots\ ,\ S((\llbracket \bar{M} \rrbracket)_{'(p-1)}, (\llbracket \bar{N} \rrbracket)_{'(p-1)}) \gg$
$\llbracket \bar{M} \xrightarrow{e} \bar{N}, \bar{P} \rrbracket = S(\Lambda(\ll C(\underline{0}, (\llbracket \bar{N} \rrbracket_0, \llbracket \bar{P} \rrbracket_0))\ ,\ \dots\ ,\ C(\underline{0}, (\llbracket \bar{N} \rrbracket_{(p-1)}, \llbracket \bar{P} \rrbracket_{(p-1)})) \gg), \Uparrow (\llbracket e \rrbracket, \llbracket \bar{M} \rrbracket))$
$\llbracket !\bar{M} \rrbracket = T \ll (S(\llbracket \bar{M} \rrbracket_{'0},' 0), ..., S(\llbracket \bar{M} \rrbracket_{'0},' (p-1)))\ ,\ \dots\ ,\ (S(\llbracket \bar{M} \rrbracket_{'(p-1)},' 0), ..., S(\llbracket \bar{M} \rrbracket_{'(p-1)},' (p-1))) \gg$

5.3. Couplage

Currien introduit un combinateur de paires : $\llbracket (M,N) \rrbracket = < \llbracket M \rrbracket, \llbracket N \rrbracket >$
Et les règles qui en découlent :
$(fst) Fst(x,y) = x$
$(snd) Snd(x,y) = y$
$(dpair) < x,y > z = (xz, yz)$
Cela explicite les opérations d'applications.

$S(,)$ est considéré comme une abréviation de $App\circ <,>$ et $\underline{n}$ comme une abréviation de $Snd \circ Fst^n$. On a ainsi par exemple $\underline{0} = Snd$ et $\underline{3} = Snd \circ Fst^3 = FstFstFstSnd$. Démarche pour coder les fonctions constantes comme le + :
$4+2$ devient $S(\Lambda(Snd+), ('4,'2))$ c'est-à-dire $< \Lambda(Snd+), <' 4,' 2 >> app$.

6. Compilation et exécution

Les termes traduits en combinateurs correspondent à des opérations d'une machine à environnements, la CAM (tel qu'expliqué ci-dessous). Ils sont compilés en opérations CAM puis exécutés selon la table de transition de celle-ci (6.2). Nous avons défini une version data-parallèle des combinateurs et faisons de même pour la CAM et ses opérations selon le paradigme "SPMD" (courant en data-parallélisme). Le programme-combinateur est compilé vers une suite d'opérations CAM dont certaines réalisent nos opérations parallèles. Il est ensuite placé, en p copies, sur autant de copies de la CAM. Les transitions des CAM, correspondant à des opérations parallèles, réalisent collectivement la sémantique data-parallèle définie jusqu'ici.

6.1. BSP-CAM

Nous expliquons ici les codes de la BSP-CAM.
$<$ push copie le terme au sommet de la pile
, swap inverse le sommet de la pile avec le terme
$>$ cons construit le couple à l'aide du sommet de la pile et du terme
Fst reçoit le terme (s,t) et le remplace par s
Snd reçoit le terme (s,t) et le remplace par t
cur remplace le terme s par $C : s$ où C est le code encapsulé dans Λ
App reçoit le terme $(C : s, t)$ et le remplace par (s,t) et positionne C en tête du code.
$'$ remplace le terme par la constante correspondante.
$Branch$ effectue le branchement conditionnel entre deux suites d'instructions.
$\Uparrow$ effectue le broadcast.
T effectue la transposition des valeurs de l'échange en les ordonnant à l'aide d'une fonction d'expressions conditionnelles imbriquées.

La compilation des expressions de BSλ vers du code utilise la constante pid, qui vaut le numéro du processeur en chaque processeur. Le code résultant est répliqué sur chaque processeur selon le paradigme SPMD.
$[\![n]\!] = Fst^n Snd \qquad [\![c]\!] = (quote\ c) \qquad [\![MN]\!] =< [\![M]\!], [\![N]\!] > app$
$[\![\lambda.M]\!] = (cur[\![M]\!]) \qquad [\![M]\!] \rightarrow N, P =< [\![M]\!] branch([\![N]\!], [\![P]\!])$
$[\![M+N]\!] =< \Lambda(Snd+), < [\![M]\!], [\![N]\!] >> app$
$[\![\pi M]\!] =< [\![M]\!], (quote\ pid) > app$
$[\![M \# N]\!] =< [\![M]\!], [\![N]\!] > app$
$[\![M \xrightarrow{e} N, P]\!] =< \Lambda(< Snd\, branch([\![N]\!], [\![P]\!])), \Uparrow ([\![e]\!], [\![M]\!]) > app$
$[\![!M]\!] =$
$<< [\![M]\!]_{pid}, '0 > app, < [\![M]\!]_{pid}, '1 > app >< < [\![M]\!]_{pid}, '2 > app\, add << [\![M]\!]_{pid}, '3 > app\, add < \ldots$
$\ldots < [\![M]\!]_{pid}, '(p-1) > app\, add\, T$

Le combinateur T de la transposition utilise une structure de p-uplets qui n'existe pas dans la CAM. La construction en paires imbriquées ne permet pas d'obtenir un p-uplets de valeurs en sommet de pile. Cette structure est nécessaire pour effectuer la phase de communication. On ajoute donc une commande add qui permet d'ajouter des termes évalués au uplet en cours de construction. Dans le détail, on construit un couple avec les deux premiers termes du p-uplets, puis on y ajoute les $p-3$ termes suivants.

De plus, les $[\![M]\!]_{pid}$ sont évalués, mis au sommet de la pile, puis ramenés dans le code. Il faut donc les "encapsuler" à l'aide de $quote$ pour ne pas évaluer une deuxième fois ces termes.

6.2. Les transitions de la BSP-CAM

Terme	Code	Pile	Terme	Code	Pile
(s,t)	fst;C	S	s	C	S
(s,t)	snd;C	S	t	C	S
s	(quote c);C	S	c	C	S
s	(cur C);$C1$	S	$(C:s)$	$C1$	S
s	push;C	S	s	C	$s.S$
t	swap;C	$s.S$	s	C	$t.S$
t	cons;C	$s.S$	(s,t)	C	S
$(C:s,t)$	app;$C1$	S	(s,t)	C;$C1$	S
$true$	branch($C1$,$C2$);C	$s.S$	s	$C1$;C	S
$false$	branch($C1$,$C2$);C	$s.S$	s	$C2$;C	S
(i,v_j)	$\Uparrow$;C	S	v_i	C	S
t	add;C	$(s).S$	(s,t)	C	S
$(x_{i0}\ ,\ \dots\ ,\ x_{i(p-1)})$	T;C	S	$()$	$\Lambda\left(\begin{array}{c} << \Lambda(Snd =), < Snd, '0 >> app \\ branch((quote\ x_{0i}), \\ \dots \\ << \Lambda(Snd =), < Snd, 'p-1 >> app \\ branch((quote\ x_{(p-1)i}), 'nc)\dots) \end{array}\right)$;C	S

7. Exemples

Nous donnons ici la trace de trois programmes parallèles compilés pour notre BSP-CAM. Les traces sont les mêmes sur tous les processeurs. Elles permettent de constater le respect du modèle de coût BSP et d'estimer les paramètres constants introduits par la BSP-CAM. Ces paramètres sont des majorants pour le sur-coût induit par la programmation d'algorithmes BSP en CAML.

Le premier exemple (figure 1) construit simplement un vecteur parallèle contenant le programme $\pi(\lambda x.x + 1)$. Le nombre de transitions est indépendant de p et on constate que le coût (additif) du constructeur π est de six transitions, les six premières dans la table ci-dessous. Après ces six transitions, il ne reste qu'à appliquer $\lambda x.x + 1$ au numéro du processeur, ici désigné par i.

Le deuxième exemple (figure 2) est une conditionnelle parallèle $(\pi(\lambda x.true)) \stackrel{4}{\rightarrow} C_1, C_2$ qui utilise le booléen du quatrième processeur. Le coût de cette instruction excède celui d'une conditionnelle séquentielle par moins de dix transitions et la latence du réseau pour diffuser un bit.

Le troisième exemple (figures 3, 4 et 5) est une utilisation du $\texttt{put}(!(\pi(\lambda x.\lambda y.x*10)))\#(\pi(\lambda x.x+1))$. Chaque processeur envoie son propre numéro multiplié par 10 à tous les autres. Les fonctions encapsulant les messages reçus sont appliquées par chaque processeur au numéro de son voisin de droite. Chaque processeur est alors en possession du message venant de ce dernier, soit $(\mathrm{pid} + 1) * 10$. Le coût de l'opération est proportionnel à environ $(20 * p)+$ le coût BSP de l'échange des données dans le réseau. Le facteur 20 s'additionne ainsi au paramètre BSP g qui multiplie la taille de la plus grande file de messages. Il est encourageant de constater ainsi que la BSP-CAM ne *multiplie* pas les paramètres de coût du réseau qui sont, en pratique, prédominants.

8. Conclusion

Nous avons ajouté des combinateurs à la logique catégorielle combinatoire (CCL) correspondant aux opérations de BSλ, qui est confluent et en avons dérivé systématiquement une version BSP de la CAM : la BSP-CAM pour les programmes fonctionnels BSP. Cela améliore la machine BSP-SECD [11] en formalisant le traitement des environnements et en donnant une première description formelle d'implantation BSP pour CAML. Cela améliore l'ancienne machine, la DPCAM [5, 4] en ce que DPML était basé sur des combinateurs ad hoc et le langage résultant n'était pas fonctionnel, bien que déterministe. Finalement, la BSP-CAM permet des prévisions exactes de temps de calcul selon le modèle BSP. Ce type de dérivation formelle pourrait se faire ensuite sur d'autres machines à environnement dont la ZAM [8] qui sert actuellement d'implantation à CAML.

Remerciements

Le travail d'Armelle Merlin est financé par une allocation doctorale du Ministère de la Recherche. Quentin Miller a participé à la première phase de ce projet soit la définition d'une version BSP de la machine abstraite SECD. Ce projet poursuit les travaux de Frédéric Loulergue sur BSλ et BSMLlib.

Bibliographie

[1] O. Ballereau, F. Loulergue, and G. Hains. High level BSP programming: BSML and BSλ. In G. Michaelson, P. Trinder, and H.-W. Loidl, editors, *Trends in Functional Programming*, chapter 4, pages 29–38. Intellect Books, Bristol UK, Portland, OR, USA, August 2000. Preliminary version appeared in proceedings of SFP'99: First Scottish Functional Programming Workshop, TR RM/99/9, pp.43–52, Heriot-Watt University, September 1999.

[2] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Birkhäuser, Boston, second edition, 1993.

[3] G. Cousineau, P. L. Curien, and M. Mauny. The categorical abstract machine. In J.-P. Jouannaud, editor, *Functional Programming Languages and Computer Architecture*, pages 50–64. Springer-Verlag, Berlin, DE, 1985. Lecture Notes in Computer Science 201Proceedings of. Conference at Nancy.

[4] 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*, pages 185–214, Noirmoutier, Février 1994. INRIA.

[5] G. Hains and C. Foisy. The data-parallel categorical abstract machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, pages 56–67, Munich, June 1993. Springer.

[6] J.M.D.Hill, W.F.McColl, D.C.Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T.Suel, T.Tsantilas, and R. Bisseling. BSPlib, the BSP programming library. Technical report, BSP Worldwide, May 1997. URL=`http://www.bsp-worldwide.org/`.

[7] X. Leroy. The Objective Caml system, documentation and user's guide. Technical report, INRIA, 1998. `http://pauillac.inria.fr/caml`.

[8] X. Leroy. The ZINC Experiment: An Economical Implementation of the ML Language. Technical report, INRIA, 1991..

[9] F. Loulergue, G. Hains, and C. Foisy. A calculus of functional BSP programs. *Science of Computer Programming*, 37(1–3):253–277, May 2000.

[10] W. F. McColl. Universal computing. 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.

[11] A. Merlin, G. Hains, and F. Loulergue. An SPMD environment machine for functional BSP programs. In *SFP'2001 Scottish Functional Programming Workshop*, Stirling, August 2001.

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

$\pi(\lambda x.x+1)$				
$\ll ...$	$()$	$< \Lambda(< \Lambda(Snd+), < Snd,' 1 >> app),' i > app$	$[]$	$... \gg$
$\ll ...$	$()$	$\Lambda(< \Lambda(Snd+), < Snd,' 1 >> app),' i > app$	$[()]$	$... \gg$
$\ll ...$	$< \Lambda(Snd+), < Snd,' 1 >> app : ()$	$,' i > app$	$[()]$	$... \gg$
$\ll ...$	$()$	$'i > app$	$[< \Lambda(Snd+), < Snd,' 1 >> app : ()]$	$... \gg$
$\ll ...$	i	$> app$	$[< \Lambda(Snd+), < Snd,' 1 >> app : ()]$	$... \gg$
$\ll ...$	$(< \Lambda(Snd+), < Snd,' 1 >> app : (), i)$	app	$[]$	$... \gg$
$\ll ...$	$((), i)$	$< \Lambda(Snd+), < Snd,' 1 >> app$	$[]$	$... \gg$
$\ll ...$	$((), i)$	$\Lambda(Snd+), < Snd,' 1 >> app$	$[((), i)]$	$... \gg$
$\ll ...$	$Snd+ : ((), i)$	$, < Snd,' 1 >> app$	$[((), i)]$	$... \gg$
$\ll ...$	$((), i)$	$< Snd,' 1 >> app$	$[Snd+ : ((), i)]$	$... \gg$
$\ll ...$	$((), i)$	$Snd,' 1 >> app$	$[((), i).Snd+ : ((), i)]$	$... \gg$
$\ll ...$	i	$,' 1 >> app$	$[((), i).Snd+ : ((), i)]$	$... \gg$
$\ll ...$	$((), i)$	$'1 >> app$	$[i.Snd+ : ((), i)]$	$... \gg$
$\ll ...$	1	$>> app$	$[i.Snd+ : ((), i)]$	$... \gg$
$\ll ...$	$(i, 1)$	$> app$	$[Snd+ : ((), i)]$	$... \gg$
$\ll ...$	$(Snd+ : ((), i), (i, 1))$	app	$[]$	$... \gg$
$\ll ...$	$(((), i), (i, 1))$	$Snd+$	$[]$	$... \gg$
$\ll ...$	$(i, 1)$	$+$	$[]$	$... \gg$
$\ll ...$	$i+1$	$[]$	$[]$	$... \gg$

Figure 1: Construction parallèle

$(\pi(\lambda x.true)) \xrightarrow{4} C_1, C_2$				
$\ll ...$	$()$	$<<' 4, < \Lambda('true),' i > app >\Uparrow Branch(C_1, C_2)$	$[]$	$... \gg$
$\ll ...$	$()$	$<' 4, < \Lambda('true),' i > app >\Uparrow Branch(C_1, C_2)$	$[()]$	$... \gg$
$\ll ...$	$()$	$'4, < \Lambda('true),' i > app >\Uparrow Branch(C_1, C_2)$	$[().()]$	$... \gg$
$\ll ...$	4	$, < \Lambda('true),' i > app >\Uparrow Branch(C_1, C_2)$	$[().()]$	$... \gg$
$\ll ...$	$()$	$< \Lambda('true),' i > app >\Uparrow Branch(C_1, C_2)$	$[4.()]$	$... \gg$
$\ll ...$	$()$	$\Lambda('true),' i > app >\Uparrow Branch(C_1, C_2)$	$[().4.()]$	$... \gg$
$\ll ...$	$'true : ()$	$,' i > app >\Uparrow Branch(C_1, C_2)$	$[().4.()]$	$... \gg$
$\ll ...$	$()$	$'i > app >\Uparrow Branch(C_1, C_2)$	$['true : ().4.()]$	$... \gg$
$\ll ...$	i	$> app >\Uparrow Branch(C_1, C_2)$	$['true : ().4.()]$	$... \gg$
$\ll ...$	$('true : (), i)$	$app >\Uparrow Branch(C_1, C_2)$	$[4.()]$	$... \gg$
$\ll ...$	$((), i)$	$'true >\Uparrow Branch(C_1, C_2)$	$[4.()]$	$... \gg$
$\ll ...$	$true$	$>\Uparrow Branch(C_1, C_2)$	$[4.()]$	$... \gg$
$\ll ...$	$(4, true)$	$\Uparrow Branch(C_1, C_2)$	$[()]$	$... \gg$
$BROADCAST$				
$\ll ...$	$true$	$Branch(C_1, C_2)$	$[()]$	$... \gg$
$\ll ...$	$()$	C_1	$[]$	$... \gg$

Figure 2: Conditionnelle parallèle

		$(!(\pi(\lambda x.\lambda y.x * 10)))\#(\pi(\lambda x.x+1))$		
$\ll ...$	$()$	$< \llbracket !M \rrbracket, \llbracket N \rrbracket > app$	$[]$	$... \gg$
$\ll ...$	$()$	$<< \llbracket M \rrbracket,' 0 > app, ...T, \llbracket N \rrbracket > app$	$[()]$	$... \gg$
$\ll ...$	$()$	$< \llbracket M \rrbracket,' 0 > app, ...T, \llbracket N \rrbracket > app$	$[().()]$	$... \gg$
		`environ` 20 `transitions`		
$\ll ...$	$(i, 10)$	$*, ...T, \llbracket N \rrbracket > app$	$[().()]$	$... \gg$
$\ll ...$	$i * 10$	$, < \llbracket M \rrbracket,' 1 > app > ...T, \llbracket N \rrbracket > app$	$[().()]$	$... \gg$
$\ll ...$	$()$	$< \llbracket M \rrbracket,' 1 > app > ...T, \llbracket N \rrbracket > app$	$[i * 10.()]$	$... \gg$
		`environ` 20 `transitions`		
$\ll ...$	$(i, 10)$	$* ><< \llbracket M \rrbracket ...T, \llbracket N \rrbracket > app$	$[i * 10.()]$	$... \gg$
$\ll ...$	$i * 10$	$><< \llbracket M \rrbracket ...T, \llbracket N \rrbracket > app$	$[i * 10.()]$	$... \gg$
$\ll ...$	$(i * 10, i * 10)$	$<< \llbracket M \rrbracket ...T, \llbracket N \rrbracket > app$	$[()]$	$... \gg$
$\ll ...$	$(i * 10, i * 10)$	$< \llbracket M \rrbracket ...T, \llbracket N \rrbracket > app$	$[(i * 10, i * 10).()]$	$... \gg$
$\ll ...$	$(i * 10, i * 10)$	$\llbracket M \rrbracket,' 2 > app\ add ...T, \llbracket N \rrbracket > app$	$[(i * 10, i * 10).(i * 10, i * 10).()]$	$... \gg$
		`environ` 20 `transitions`		
$\ll ...$	$(i, 10)$	$*add << \llbracket M \rrbracket ...T, \llbracket N \rrbracket > app$	$[(i * 10, i * 10).()]$	$... \gg$
$\ll ...$	$i * 10$	$add << \llbracket M \rrbracket ...T, \llbracket N \rrbracket > app$	$[(i * 10, i * 10).()]$	$... \gg$
$\ll ...$	$(i * 10, i * 10, i * 10)$	$<< \llbracket M \rrbracket,' 3 > app\ add\ T, \llbracket N \rrbracket > app$	$[()]$	$... \gg$
$\ll ...$	$(i * 10, i * 10, i * 10)$	$< \llbracket M \rrbracket,' 3 > app\ add\ T, \llbracket N \rrbracket > app$	$[(i * 10, i * 10, i * 10).()]$	$... \gg$
$\ll ...$	$(i * 10, i * 10, i * 10)$	$\llbracket M \rrbracket,' 3 > app\ add\ T, \llbracket N \rrbracket > app$	$[(i * 10, i * 10, i * 10).(i * 10, i * 10, i * 10).()]$	$... \gg$
		`environ` 20 `transitions`		
$\ll ...$	$(i, 10)$	$*add\ T, \llbracket N \rrbracket > app$	$[(i * 10, i * 10, i * 10).()]$	$... \gg$
$\ll ...$	$i * 10$	$add\ T, \llbracket N \rrbracket > app$	$[(i * 10, i * 10, i * 10).()]$	$... \gg$
$\ll ...$	$(i * 10, i * 10, i * 10, i * 10)$	$T, \llbracket N \rrbracket > app$	$[()]$	$... \gg$

Figure 3: Multiput p=4 : Calcul de $!(\pi(\lambda x.\lambda y.x * 10))$ avant l'échange.

$ECHANGE$				
$\ll ...$	$()$	$\Lambda(<< \Lambda(Snd =), < Snd,' 0 >> app\; branch(...)), [\![N]\!] > app$	$[()]$	$... \gg$
$\ll ...$	$A : ()$	$, [\![N]\!] > app$	$[()]$	$... \gg$
$\ll ...$	$()$	$< \Lambda(< \Lambda(Snd+), < Snd,' 1 >> app), pid > app > app$	$[A : ()]$	$... \gg$
$\ll ...$	$()$	$\Lambda(< \Lambda(Snd+), < Snd,' 1 >> app), pid > app > app$	$[().A : ()]$	$... \gg$
		$...$		
$\ll ...$	$i + 1$	$> app$	$[A : ()]$	$... \gg$
		$...$		
$\ll ...$	$i + 1$	$,' 0 >> app\; branch('(i * 10), ...)$	$[((), i + 1).Snd =: ((), i + 1).((), i + 1)]$	$... \gg$
$\ll ...$	$((), i + 1)$	$'0 >> app\; branch('(i * 10), ...)$	$[i + 1.Snd =: ((), i + 1).((), i + 1)]$	$... \gg$
$\ll ...$	0	$>> app\; branch('(i * 10), ...)$	$[i + 1.Snd =: ((), i + 1).((), i + 1)]$	$... \gg$
$\ll ...$	$(i + 1, 0)$	$> app\; branch('(i * 10), ...)$	$[Snd =: ((), i + 1).((), i + 1)]$	$... \gg$
$\ll ...$	$(Snd =: ((), i + 1), (i + 1, 0))$	$app\; branch('(i * 10), ...)$	$[((), i + 1)]$	$... \gg$
$\ll ...$	$(((), i + 1), (i + 1, 0))$	$Snd = branch('(i * 10), ...)$	$[((), i + 1)]$	$... \gg$
$\ll ...$	$(i + 1, 0)$	$= branch('(i * 10), ...)$	$[((), i + 1)]$	$... \gg$

où $A =<< \Lambda(Snd =), < Snd,' 0 >> app\; branch('(i * 10), ...)$

Figure 4: Multiput p=4 : Echange et calcul de $\pi(\lambda x.x + 1)$.

`Au processeur 1 :`				
$\ll$	$false$	$branch('10, << \Lambda(Snd =), < Snd,' 1 >> app\ branch('10, ...))$	$[((), 2)]$	$... \gg$
$\ll$	$((), 2)$	$<< \Lambda(Snd =), < Snd,' 1 >> app\ branch('10, ...))$	$[]$	$... \gg$
$\ll$	$((), 2)$	$< \Lambda(Snd =), < Snd,' 1 >> app\ branch('10, ...))$	$[((), 2)]$	$... \gg$
$\ll$	$((), 2)$	$\Lambda(Snd =), < Snd,' 1 >> app\ branch('10, ...))$	$[((), 2).((), 2)]$	$... \gg$
$\ll$	$Snd =: ((), 2)$	$, < Snd,' 1 >> app\ branch('10, ...))$	$[((), 2).((), 2)]$	$... \gg$
$\ll$	$((), 2)$	$< Snd,' 1 >> app\ branch('10, ...))$	$[Snd =: ((), 2).((), 2)]$	$... \gg$
$\ll$	$((), 2)$	$Snd,' 1 >> app\ branch('10, ...))$	$[((), 2).Snd =: ((), 2).((), 2)]$	$... \gg$
$\ll$	2	$,' 1 >> app\ branch('10, ...))$	$[((), 2).Snd =: ((), 2).((), 2)]$	$... \gg$
$\ll$	$((), 2)$	$'1 >> app\ branch('10, ...))$	$[2.Snd =: ((), 2).((), 2)]$	$... \gg$
$\ll$	1	$>> app\ branch('10, ...))$	$[2.Snd =: ((), 2).((), 2)]$	$... \gg$
$\ll$	$(2, 1)$	$> app\ branch('10, ...))$	$[Snd =: ((), 2).((), 2)]$	$... \gg$
$\ll$	$(Snd =: ((), 2), (2, 1))$	$app\ branch('10, ...))$	$[((), 2)]$	$... \gg$
$\ll$	$(((), 2), (2, 1))$	$Snd = branch('10, ...))$	$[((), 2)]$	$... \gg$
$\ll$	$(2, 1)$	$= branch('10, ...))$	$[((), 2)]$	$... \gg$
$\ll$	$false$	$branch('10, << \Lambda(Snd =), < Snd,' 2 >> app\ branch('10, ...))$	$[((), 2)]$	$... \gg$
		...		
$\ll$	$(2, 2)$	$= branch('10, ...))$	$[((), 2)]$	$... \gg$
$\ll$	$true$	$branch('10, ...))$	$[((), 2)]$	$... \gg$
$\ll$	$((), 2)$	$'10$	$[]$	$... \gg$
$\ll$	10	$[]$	$[]$	$... \gg$

Figure 5: Multiput p=4 : application parallèle au processeur i.