Extension du BSλ-calcul

Frédéric Loulergue
loulergu@lifo.univ-orleans.fr

Laboratoire d'Informatique Fondamentale d'Orléans
BP 6759
45069 ORLEANS Cedex 9, France
loulergu@lifo.univ-orleans.fr

Résumé

Après une présentation du calcul confluent BSλ, extension du λ-calcul par des opérations parallèles permettant l'expression d'algorithmes BSP, nous proposons trois nouvelles opérations parallèles étendant le calcul. Le calcul et cette nouvelle extension sont illustrés par différentes algorithmes de diffusion donnés dans une syntaxe à la ML.

1. Introduction

Pour que la programmation parallèle devienne aussi répandue que la programmation séquentielle, elle se doit de posséder tous les mécanismes d'abstraction standards tels que les fonctions d'ordre supérieur et la récursion. Mais si ces langages se veulent utilisables pour la programmation *extensible*, l'abstraction ne doit pas être obtenue au prix de coûts d'exécution prohibitifs ou imprévisibles. Par exemple, de nombreux langages parallèles ne décrivent pas le placement des données de telle sorte que les performances n'y sont pas prévisibles à partir du programme source (les performances dépendent de l'implantation, non de la sémantique).

Au contraire le paradigme *Bulk Synchronous Parallelism* de Valiant[McC96] démontre que des programmes écrits explicitement pour un nombre p de processeurs peuvent avoir des coûts d'exécution prévisibles sur un grand nombre d'architectures. Mais la combinaison d'algorithmes BSP avec des caractéristiques de langages de haut niveau n'est pas bien comprise et freine le développement de langages BSP. Nous étudions cette question du point de vue de la programmation fonctionnelle et avons défini une extension du λ-calcul avec des opérations BSP, appelée BSλ [LHF98, LH98], et qui est confluente[LHF].

Ce calcul est inspiré par le paradigme des squelettes algorithmiques de Cole[Col89]. C'est une formalisation de la proposition d'extension de ML par des opérations BSP présentée dans [Lou98]. On l'obtient en ajoutant des constantes et des règles spéciales et en étant plus restrictif que d'ordinaire : les termes sont divisés en deux sortes. L'une des sortes décrit les objets globaux et l'autre les objets "séquentiels" habituels qui ne peuvent contenir de termes globaux. Cette absence d'imbrication de sorte réalise le concept de *processus explicites* en rendant explicite la réduction parallèle de termes. Les processus explicites servent à faciliter la construction de mesures de coûts d'exécution BSP.

Nous présentons d'abord le BSλ-calcul. Dans ce calcul une seule opération de communication, get, est disponible et est unaire, c'est-à-dire qu'un processeur ne peut recevoir un message que d'au plus un autre processeur et différents processeurs ne peuvent recevoir d'un même processeur que le même message. Or chaque get provoque une barrière de synchronisation. Afin d'éviter de nombreux get, nous introduisons ici deux nouvelles opérations de communications :

- le multi-get, c'est-à-dire une opération de communication où chaque processeur peut recevoir des messages de plusieurs autres processeurs.

- le put, c'est-à-dire une opération de communication où chaque processeur peut envoyer un message différent à plusieurs autres processeurs.

De même la conditionnelle globale synchrone du BSλ-calcul induit une barrière de synchronisation car la décision se fait à partir d'un champ de données booléen (objet parallèle). Or dans de nombreux cas ce champ pourrait être remplacé par un scalaire booléen. Ceci nous amène à proposer une conditionnelle globale mais dont la condition est un terme local et qui est asynchrone.

Le calcul étendu par ces nouvelles opérations est confluent. Un aperçu de la preuve est donné en annexe. Enfin, nous illustrons le calcul par plusieurs versions d'algorithmes de diffusion.

2. Le calcul

Après une rapide introduction au modèle BSP, nous décrivons la syntaxe BSλ , donnons sa théorie équationnelle ainsi que des motivations dénotationnelles et/ou opérationnelles.

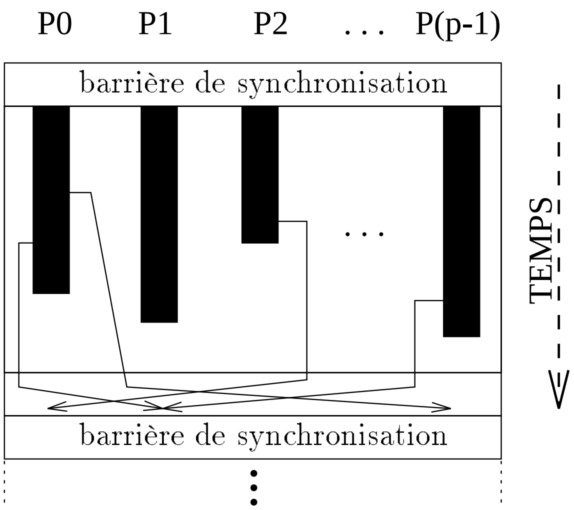

Figure 1: Super-étapes BSP

2.1. Le modèle BSP

Le modèle *Bulk-Synchronous Parallelism* (BSP) est un modèle de programmation parallèle introduit par Valiant [Val90] pour offrir un niveau d'abstraction comparable aux modèles PRAM tout en permettant des performances prévisibles et portables sur une large variété d'architectures. Un ordinateur BSP contient un ensemble de paires processeur-mémoire, un réseau de communication permettant l'échange de messages inter-processeur et une unité de synchronisation globale qui exécute des demandes collectives de barrière de synchronisation. Ses performances sont caractérisées par trois paramètres : le nombre p de paires processeur-mémoire, le temps l nécessaire à une barrière de synchronisation et le temps g nécessaire à une 1-relation (phase de communication où chaque processeur envoie ou reçoit au plus un mot). Pour n'importe quel h le réseau peut réaliser une h-relation, c'est-à-dire une phase de communication où chaque processeur envoie ou reçoit au plus h mots, en temps gh.

Un programme BSP est exécuté comme une séquence de *super-étapes*, chacune étant au plus divisée en trois phases successives et logiquement disjointes (Figure 1).

Pendant la première phase, chaque processeur utilise ses données locales pour du calcul séquentiel et pour demander des transferts de données vers ou depuis d'autres nœuds. Pendant la seconde phase, le réseau effectue les transferts de données demandés. Pendant la troisième phase, une barrière de synchronisation se produit, rendant disponibles pour la super-étape suivante les données transférées. Le temps d'exécution d'une super-étape s est ainsi la somme du maximum des temps de calcul locaux, du temps de communication des données et du temps de synchronisation globale :

$$\mathrm{Time}(s) = \max_{i:processeur} w_i^{(s)} + \max_{i:processeur} h_i^{(s)} * g + l$$

où $w_i^{(s)}$ = temps de calcul local du processeur i durant 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 transmis (resp. reçus) par le processeur i durant la super-étape s.

Le temps d'exécution $\sum_s \mathrm{Time}(s)$ d'un programme BSP composé de S super-étapes est la somme de trois termes : $W + H * g + S * l$ où $W = \sum_s \max_i w_i^{(s)}$ et $H = \sum_s \max_i h_i^{(s)}$. En général W, H et S sont fonctions de p et de la taille des données n, ou de paramètres plus complexes. Pour minimiser le temps d'exécution, un algorithme BSP doit minimiser conjointement le nombre de super-étapes, le volume total H (resp. W) et les déséquilibres $h^{(s)}$ (resp. $w^{(s)}$) de communication (resp. de calcul local).

L'utilité de BSλ est de permettre l'écriture déclarative d'algorithmes BSP sans recourir à des primitives impératives comme l'envoi et la réception de messages.

2.2. Syntaxe

Nous considérons l'ensemble $\dot{V}$ des variables *locales* et l'ensemble $\bar{V}$ des variables *globales*. Nous notons $\dot{x}, \dot{y}, \ldots$ les variables locales et $\bar{x}, \bar{y}, \ldots$ les variables globales. x dénote une variable soit locale soit globale.

La syntaxe de BSλ commence avec les termes *locaux* e : ce sont des λ-termes représentant des valeurs ou des programmes stockés dans la mémoire locale d'un processeur. L'ensemble $\dot{\Lambda}$ des termes locaux est donné par la grammaire suivante :

$$e \quad ::= \quad \dot{x} \quad | \quad e\, e \quad | \quad \lambda \dot{x}.\, e \quad | \quad c$$

où $\dot{x}$ dénote une variable locale arbitraire. Nous notons $(e_1 \rightarrow e_2, e_3)$ la conditionnelle $e_1\ e_2\ e_3$. L'opération booléenne $\Rightarrow$ est aussi supposée être encodée sous forme de λ-terme[Bar86].

Les termes principaux E de BSλ sont appelés termes *globaux* et représentent des *champs de données*, c'est-à-dire des fonctions d'un ensemble fixé de processeurs vers des valeurs. Les noms des processeurs ne sont pas spécifiés mais l'on suppose que ce sont des λ-termes clos de telle sorte que les champs de données puissent être exprimés intensionnellement par une constante spéciale appelée π (pour objet *parallèle*). Le terme $\pi\, e$ représente un champ de données dont les valeurs sont données par la fonction e (Figure 2, pour illustration les noms de processeurs seront toujours $0, \ldots, p-1$).

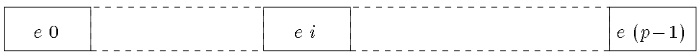

Figure 2: Le terme πe

L'ensemble $\bar{\Lambda}$ des termes globaux est donné par la grammaire suivante :

$$\begin{array}{rcl} E & ::= & \bar{x} \mid E\,E \mid E\,e \mid \lambda\bar{x}.\,E \mid \lambda\dot{x}.\,E \\ & \mid & \pi\,e \mid E \# E \mid E\,?\,E \mid (E \stackrel{e}{\rightarrow} E, E) \end{array}$$

et a la signification suivante. Soit p le nombre fini des noms de processeurs $\mathcal{N} = \{n_i \mid 0 \leq i < p\}$, chacun étant un λ-terme clos. Les termes globaux dénotent des fonctions de $\mathcal{N}$ vers des valeurs locales, des fonctions entre elles ou des fonctions des termes locaux vers de telles fonctions. En particulier, la dénotation de $\pi\, e$ a, au processeur n_i, la valeur (forme normale) de $e\, n_i$. Les formes $E_1 \# E_2$ et $E_1\,?\,E_2$ sont appelées application parallèle (*apply-par*) et *get* respectivement. Apply-par représente l'application point à point d'un champ de fonctions à un champ de valeurs (phase de calcul pur d'une super-étape BSP). Get représente la phase de communication d'une super-étape BSP : un échange collectif de données avec une barrière de synchronisation. Dans $E_1\,?\,E_2$, le champ de données résultant contient les valeurs de E_1 prises aux noms de processeurs définis dans E_2. La signification exacte de apply-par et get est définie par les équations de BSλ. La dernière forme de terme global définit la conditionnelle globale synchrone. La signification de $(E_1 \stackrel{e}{\rightarrow} E_2, E_3)$ est celle de E_2 (resp. E_3) si le champ de données dénoté par E_1 a la valeur T (resp. F) au processeur de nom dénoté par e.

2.3. Sémantique équationnelle

L'égalité des termes locaux est simplement la β-équivalence : la fermeture réflexive symétrique et transitive de

$$(\beta) \qquad (\lambda\dot{x}.\, e)e' = e[\dot{x} \leftarrow e']$$

appliquée à n'importe quel sous-terme.

L'égalité des termes globaux est définie par des règles basées sur la syntaxe et des règles de contexte qui déterminent l'applicabilité des premières. Nous donnons maintenant les règles basées sur la syntaxe :

Il y a d'abord les axiomes concernant la beta-équivalence globale.

$$\begin{array}{llcl} (B) & (\lambda\bar{x}.\, E)E' & = & E[\bar{x} \leftarrow E'] \\ (B') & (\lambda\dot{x}.\, E)e' & = & E[\dot{x} \leftarrow e'] \end{array}$$

Il y a également des axiomes concernant l'interaction du constructeur de champ π avec les autres opérations BSP.

Les équations ($?\pi$) (Figure 3) et ($\#\pi$) (Figure 4) encodent la signification dénotationelle des opérations BSP sur les champs. En particulier, get est la composition fonctionnelle à l'intérieur du π. La valeur de $\pi\, e_1\, ?\, \pi\, e_2$ au processeur de nom n_i est $e_1\,(e_2\, n_i)$, i.e. la valeur de $\pi\, e_1$ au processeur de nom $e_2\, n_i$. On notera qu'en pratique ceci représente une opération où chaque processeur reçoit une (et une seule) valeur d'un et d'un seul autre processeur. La section suivante présente de nouveaux opérateurs de communications permettant de se libérer de cette contrainte.

$$(?\pi) \quad (\pi\ e_1)\,?\,(\pi\ e_2) \quad = \quad \pi(\lambda\dot{x}.\ e_1(e_2\ \dot{x}))$$

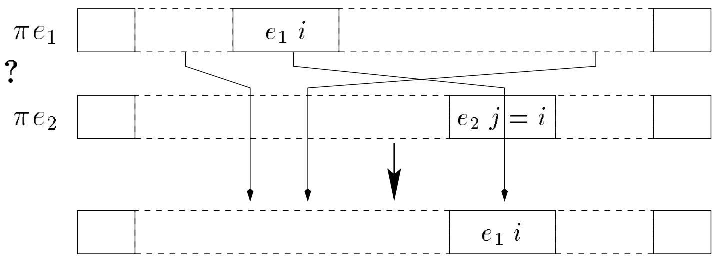

Figure 3: La règle ($?\pi$)

$$(\#\pi) \quad (\pi\ e_1)\#(\pi\ e_2) \quad = \quad \pi(\lambda\dot{x}.\ (e_1\ \dot{x})(e_2\ \dot{x}))$$

La conditionnelle globale synchrone est définie par deux règles dont les numérateurs font référence à des calculs locaux.

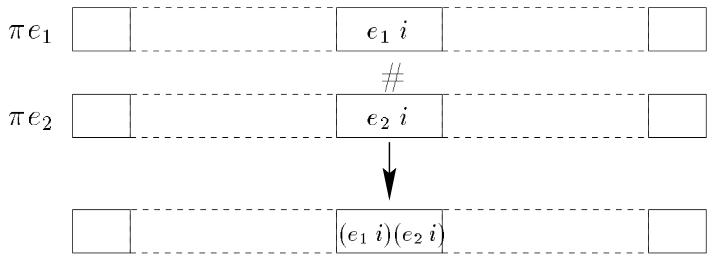

Figure 4: La règle ($\#\pi$)

$$(\overset{e}{\rightarrow} T)\ \frac{e\ e' \doteq T \quad e' \in \mathcal{N}}{((\pi\ e) \overset{e'}{\rightarrow} E_1, E_2)\ =\ E_1}$$

$$(\overset{e}{\rightarrow} F)\ \frac{e\ e' \doteq F \quad e' \in \mathcal{N}}{((\pi\ e) \overset{e'}{\rightarrow} E_1, E_2)\ =\ E_2}$$

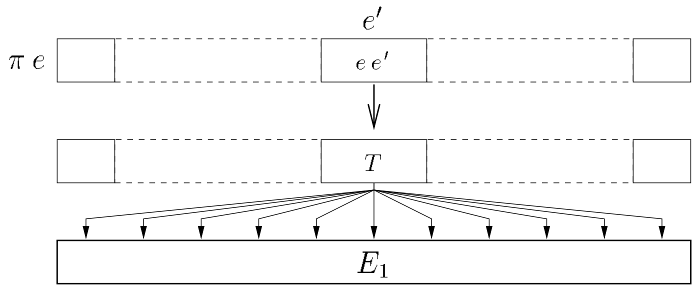

Figure 5: La règle ($\overset{e}{\rightarrow} T$)

Les deux règles précédentes (Figure 5 pour le premier cas) génèrent le calcul BSP suivant : premièrement une phase de calcul pur où tous les processeurs évaluent le terme local e' conduisant à la valeur v. Alors le processeur v prend l'initiative d'évaluer $e\,v$ donnant la valeur v'. Si $v' = T$ (resp. F) alors le processeur v diffuse l'ordre d'évaluation globale de E_1 (resp. E_2); sinon le calcul échoue.

Règles de contexte

Les règles de contexte de BSλ sont nombreuses (voir annexe A). Toutes les équations précédentes peuvent être appliquées dans n'importe quel contexte.

3. Nouvelles opérations

3.1. Multi-get

Nous présentons ici l'opérateur multi-get qui permet que les processeurs puissent recevoir plusieurs messages, chaque message provenant d'un processeur différent.

Un terme global peut désormais être de la forme $E_1 \;?\; E_2$. Un terme local peut maintenant être la constante $\mathbf{nc}$ qui est utilisée dans le terme résultant d'une communication pour indiquer qu'il n'y a pas eu de communication entre les deux processeurs concernés.

Le sens de la règle $(?\pi)$ est le suivant. Le terme $\pi\, e_2$ est un champ dont la valeur à chaque processeur est une fonction des noms de processeurs vers les valeurs F ou T.

Si $e_2\ i\ j = T$ alors le processeur i recevra le terme $e\ j$. Sinon le processeur i ne recevra aucun message du processeur j et la valeur du champ résultat au processeur i sera une fonction qui appliquée à j donnera la valeur $\mathbf{nc}$ (Figure 6).

$$(?\pi) \quad \pi\, e_1 \;?\; \pi\, e_2 \;=\; \pi(\lambda\dot{x}\,\dot{y}.(e_2\,\dot{x}\,\dot{y} \to e_1\,\dot{y}, \mathbf{nc}))$$

3.2. Put

Le multi-get permet aux processeurs de recevoir plusieurs messages différents mais il manque encore un opérateur permettant à un processeur *d'envoyer* plusieurs messages différents à différents processeurs en une seule super-étape. C'est ici qu'intervient le put. En effet, la simulation d'une opération put par des opérations get aurait pour effet dans le pire cas de multipler le nombre de super-étapes par p. Il est donc utile d'ajouter put au calcul.

Un terme global peut désormais être de la forme $E!E$. Le sens de la règle $(!\pi)$ est le suivant. Si au processeur i, la fonction dénotée par le champ $\pi\, e_2$ appliquée à j donne la valeur F, il n'y a pas de communication. Sinon la valeur $e_1\, i\, j$ est envoyée au processeur j par le processeur i.

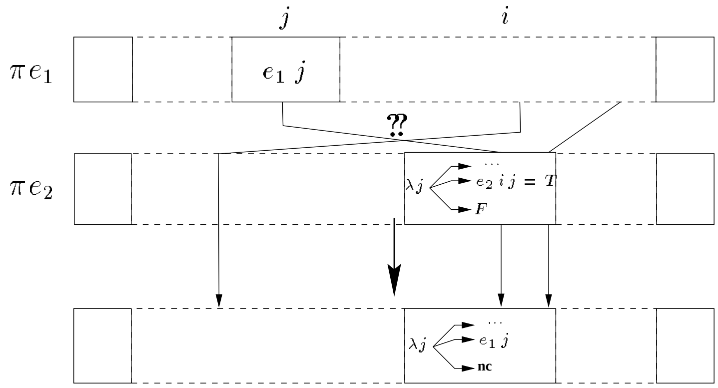

Figure 6: La règle $(??\pi)$

Le champ résultant est un champ de fonctions. Au processeur j, la fonction appliquée à i vaut $e_1\, i\, j$ dans le cas où une communication a été réalisée, **nc** sinon.

$$(!\pi) \quad (\pi e_1)!(\pi e_2) \;=\; \pi(\lambda \dot{x}\dot{y}.(e_2\,\dot{x}\,\dot{y} \to (e_1\,\dot{y}\,\dot{x}), \mathbf{nc}))$$

3.3. Conditionnelle globale asynchrone

La conditionnelle globale synchrone induit une synchronisation et prend sa décision à partir d'un champ de booléens. Or il peut être utile d'avoir une conditionnelle dont la condition est terme local booléen mais dont la décision influe au niveau global. Cette conditionnelle globale n'induit pas de synchronisation.

Tous les processeurs calculent la valeur de la condition. Cette condition étant un terme local, sa valeur ne dépend pas du processeur où elle est évaluée. Tous les processeurs se retrouvent donc avec la même valeur et choisissent tous la même branche à évaluer.

$$(\to T)\ \frac{e \doteq T}{(e \to E_1, E_2) = E_1} \qquad\qquad (\to F)\ \frac{e \doteq F}{(e \to E_1, E_2) = E_2}$$

3.4. Confluence

Les opérations ajoutées à BSλ ne nuisent pas à la confluence du calcul.

Théorème 1 *La relation de réduction obtenue en orientant de gauche à droite les règles de la théorie équationnelle de BSλ étendu est confluente.*

Preuve : Des éléments de la démonstration sont donnés en annexe B.□

4. Exemples

Afin d'illustrer l'expressivité de BSλ , nous présentons deux versions d'algorithmes de diffusion. Une diffusion binaire nécessitant $\lg(p)$ super-étapes, et un second algorithme dû à l'équipe BSP d'Oxford et appelé *dispersion*, nécessitant deux super-étapes.

Dans les exemples nous supposons avoir p processeurs. Les noms de processeurs sont les entiers de 0 à $p-1$. Ces exemples sont donnés en syntaxe à la ML (Figure 7). Pour des raisons de simplicité les algorithmes présentés diffusent la donnée située au processeur $\frac{p-1}{2}$. On peut facilement généraliser au cas où l'on précise le lieu de la donnée à diffuser.

L'algorithme naif en une super-étape consiste en une communication où tous les processeurs demandent la valeur du processeur $\frac{p-1}{2}$. Le terme BSλ correspondant est :

```
let diffusion X = get X from (fun i-> (nprocs-1)/2);;
```

Cependant la communication réalise dans ce cas une $(p-1)$-relation. Le coût de communication et de synchronisation est alors de

$$s \times (p-1) \times g + l$$

où s est la taille de la donnée diffusée. Le programme de diffusion binaire est réalisé en $\lg(p)$ super-étapes. À chaque super-étape les processeurs ayant demandé à recevoir un message se font demander un message par deux autres processeurs, ceci en partant du processeur $\frac{p-1}{2}$. Une 2-relation est alors réalisée. Le coût de communication et synchronisation est alors de

$$2 \times s \times \lg(p) \times g + \lg(p) \times l$$

où s est la taille de la donnée diffusée. Les valeurs habituelles de g et l sont telles que les algorithmes binaires sont plus efficaces pour des valeurs de p et s relativement faibles.

La diffusion (Figure 8) est une récursion linéaire : `diffusion_aux` n'apparait qu'une seule fois dans l'expression définissant la fonction `diffusion_aux`.

L'exécution de cette diffusion est schématisée figure 9.

$\lambda\bar{x}.\,E$	`fun` $X \to E$
$\lambda\dot{x}.\,E$	`fun` $x \to E$
$\lambda\dot{x}.\,e$	`fun` $x \to e$
$\pi(\lambda x.\,e)$	`par` (`fun` $x \to e$)
$E_1 \# E_2$	`apply` E_1 `to` E_2
$E_1 \; ? \; E_2$	`get` E_1 `from` E_2
$E_1 \; ?\!? \; E_2$	`mget` E_1 `from` E_2
$E_1 ! E_2$	`put` E_1 `at` E_2
$(E \xrightarrow{e} E_1, E_2)$	`if` E `at` e `then` E_1 `else` E_2
$(e \to E_1, E_2)$	`if` e `then` E_1 `else` E_2
p	`nprocs`

Figure 7: Syntaxe à la ML

```
let diffusion_binaire X =
 let diffusion_aux X t =
  let rotate = fun i -> ((i+(nprocs-((nprocs-1)/2))) mod nprocs) in
   let rotate_back= fun i->((i+((nprocs-1)/2)) mod nprocs) in
    if t<0
    then X
    else
     let from i =
      let deux_puiss_t = pow 2 t in
       if i < deux_puiss_t then i else i-deux_puiss_t in
       let rotated_from i = rotate_back(from (rotate i)) in
        get  (par(fun i-> rotated_from i))
        from (diffusion_aux X (t-1)) in
 diffusion_aux X (ln nprocs);;
```

Figure 8: Diffusion binaire plate

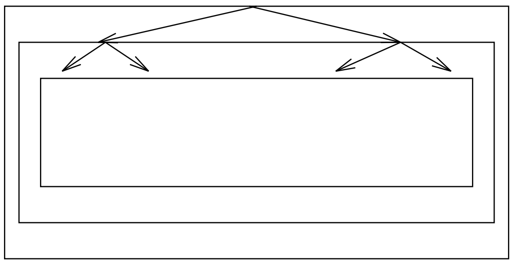

Figure 9: Diffusion binaire plate. p=7

La dispersion est une façon économique de réaliser une diffusion lorsque la donnée à diffuser est de taille suffisante. La dispersion procède en deux phases de communication et synchronisation.

Dans la première super-étape la liste au processeur $\frac{p-1}{2}$ est découpée en p morceaux, le $i^{\text{ième}}$ morceau étant envoyé au processeur i. On obtient le champ appelé ici `morceaux_dispersés`. Le coût de communication et synchronisation de cette super-étape est de

$$\frac{s}{p-1} \times (p-1) \times g + l$$

où s est la taille de la donnée diffusée. On extrait ensuite le terme qui nous intéresse du résultat du `put` c'est-à-dire en chaque processeur la valeur de la fonction obtenue appliquée à $\frac{p-1}{2}$.

La deuxième phase de communication en un échange total et est suivie d'une phase de calcul qui a pour but de "recoller" les morceaux issus de tous les processeurs. Le coût de communication et synchronisation de cette super-étape est de

$$\frac{s}{p-1} \times (p-1) \times g + l$$

où s est la taille de la donnée diffusée.

Pour définir la dispersion, nous utilisons deux fonctions auxiliaires que nous décrivons informellement :

`decouper` est une fonction qui prend en entrée un entier n et un champ x de listes et qui renvoie un champ de fonctions. Au processeur $\frac{p-1}{2}$ le résultat est une fonction des noms de processeurs vers des listes, qui associe à chaque nom de processeur i, le $i^{\text{ième}}$ morceau de la liste du champ x au processeur i, tous les morceaux ayant la même longueur à un élément près.

`recoller` est une fonction qui prend en entrée une fonction f des noms de processeurs vers des listes et qui renvoie la liste qui est la concaténation des $f\ i$ pour i allant de 0 à $p-1$.

Le terme suivant réalise la dispersion :

```
let dispersion X =
 let Morceaux_dispersés = apply par (put (decouper X)
                                    at  par(fun i j->i=(nprocs-1)/2))
                          to    par (fun i -> (nprocs-1)/2) in
  let Echange_total = mget Morceaux_dispersés
                      from par (fun j -> fun j -> true) in
 apply (par fun i -> recoller ) to Echange_total;;
```

L'exécution est schématisée à la figure 10.

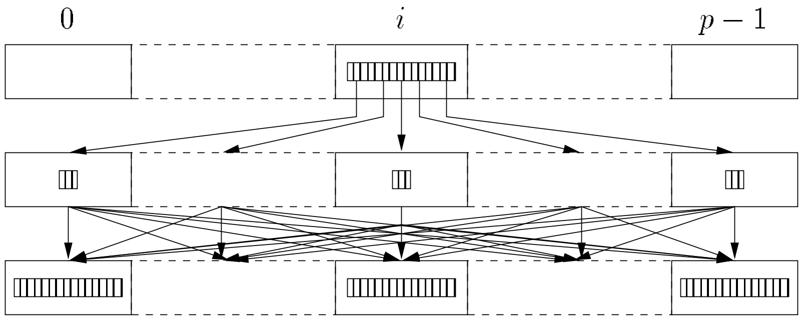

Figure 10: Dispersion

5. Travaux connexes

Le BSλ-calcul adapte la notion de xappings[SH86] ou champs de données [CCL91, VP92, VP93] à BSP.

Skil[BK96] et Caml Flight[CF95] sont deux langages apparentés à BSλ. Le premier est plongé dans un langage impératif et combiné avec une compilation efficace des fonctions d'ordre supérieur, atteint de très hautes performances. Le second transforme le sous-ensemble fonctionnel de Caml en un langage déterministe avec un possible entrelacement asynchrone d'opérations data-parallèles. Cependant Skil place des contraintes sur les fonctions d'ordre supérieur qui peuvent être appliquées. Caml Flight quant à lui nécessite un mécanisme qui empêche dynamiquement l'imbriquement du constructeur sync qui sert à préserver l'environnement. BSλ n'impose aucune restriction sur les fonctions et l'imbrication de π est empêchée statiquement (Ici c'est le découpage de la syntaxe en deux sortes, locale et globale, qui empêche d'écrire des imbrications de π, dans un langage à la ML ceci se fera par typage).

6. Conclusions

Il y a eu de nombreuses tentatives pour mêler la programmation fonctionnelle et les architectures parallèles mais l'intégration du paradigme BSP et ses processus explicites avec la programmation fonctionnelle n'a jamais, à notre connaissance, été réalisée.

Étant une théorie équationnelle, BSλ est une base pour des preuves d'équivalence de programmes BSP.

Les travaux en cours sur BSλ concernent une sémantique d'évaluation.

Suivrons une sémantique d'exécutions distribuée dérivée de celle-ci, un modèle de coût et une implantation.

Les extensions proposées et validées ici nous assurent de pouvoir construire un langage BSP fonctionnel réaliste sur la base de BSλ.

A. Règles de passage au contexte

Les règles complètes définissent deux égalités. La première est la β-équivalence classique sur les termes locaux. La seconde relation est la BSλ-équivalence sur les termes globaux.

Nous ne présentons ici que les règles n'ayant pas été présentées précédemment. Nous commençons par les règles du BSλ-calcul. Nous donnons à la fin les règles relatives aux nouvelles opérations.

Tout d'abord pour la relation $\doteq$ sur les termes locaux :

$$\frac{m \doteq n}{m\,z \doteq n\,z} \qquad \frac{m \doteq n}{z\,m \doteq z\,n} \qquad \frac{m \doteq n}{\lambda\dot{x}.m \doteq \lambda\dot{x}.n}$$

Ensuite pour la relation $=$ sur les termes globaux :

$$\frac{(M,N) \in \left(\begin{array}{l} (=_{(B)} \cup =_{(B')} \cup =_{(?\pi)} \cup =_{(\#\pi)} \cup =_{(\stackrel{e}{\rightarrow}T)} \\ \cup =_{(\stackrel{e}{\rightarrow}F)}) =_{(??\pi)} \cup =_{(!\pi)} \cup =_{(\rightarrow T)} \cup =_{(\rightarrow F)} \end{array}\right)}{M \;=\; N}$$

$$\frac{m \doteq n}{\pi\,m \;=\; \pi\,n} \qquad \frac{M \;=\; N}{\lambda x.M \;=\; \lambda x.N}$$

$$\frac{M \;=\; N}{M\,Z \;=\; N\,Z} \qquad \frac{M \;=\; N}{Z\,M \;=\; Z\,N}$$

$$\frac{m \doteq n}{Z\,m \;=\; Z\,n} \qquad \frac{M \doteq N}{M\,z \;=\; N\,z}$$

$$\frac{M \;=\; N}{Z\,?\,M \;=\; Z\,?\,N} \qquad \frac{M \;=\; N}{M\,?\,Z \;=\; N\,?\,Z}$$

$$\frac{M \;=\; N}{Z\,\#\,M \;=\; Z\,\#\,N} \qquad \frac{M \;=\; N}{M\,\#\,Z \;=\; N\,\#\,Z}$$

$$\frac{M \;=\; N}{(M \stackrel{z}{\rightarrow} Z_1, Z_2) \;=\; (N \stackrel{z}{\rightarrow} Z_1, Z_2)} \qquad \frac{M_1 \;=\; N_1}{(Z \stackrel{z}{\rightarrow} M_1, Z_2) \;=\; (Z \stackrel{z}{\rightarrow} N_1, Z_2)}$$

$$\frac{M_2 \;=\; N_2}{(Z \stackrel{z}{\rightarrow} Z_1, M_2) \;=\; (Z \stackrel{z}{\rightarrow} Z_1, N_2)} \qquad \frac{m \doteq n}{(Z \stackrel{m}{\rightarrow} Z_1, Z_2) \;=\; (Z \stackrel{n}{\rightarrow} Z_1, Z_2)}$$

$$\frac{M_1 \;=\; N_1}{(z \rightarrow M_1, Z_2) \;=\; (z \rightarrow N_1, Z_2)} \qquad \frac{M_2 \;=\; N_2}{(z \rightarrow Z_1, M_2) \;=\; (z \rightarrow Z_1, N_2)}$$

$$\frac{m \doteq n}{(m \rightarrow Z_1, Z_2) \;=\; (n \rightarrow Z_1, Z_2)}$$

B. Confluence du calcul étendu

En orientant de gauche à droite les conclusions et les hypothèses des règles de la théorie équationnelle de BSλ on obtient deux relations de réduction, $\dot{\rightarrow}$ et $\rightarrow$ correspondant respectivement à $\doteq$ et $=$. La preuve de confluence de $\rightarrow$ est organisée comme suit :

1. on définit des relations $\dot{\triangleright}$ et $\triangleright$ qui contiennent les deux relations de réduction mentionnées précédemment et qui sont contenues dans leur fermeture transitive.
2. on prouve que ces deux relations sont fortement confluentes. La confluence forte est démontrée pour $\dot{\triangleright}$ puis $\triangleright$.
3. $\triangleright$ étant confluente, sa fermeture transitive l'est aussi.
4. comme $(\rightarrow \cup \mathcal{D}(\bar{\Lambda})) \subseteq \triangleright \subseteq \rightarrow^*$ (avec $\mathcal{D}(\bar{\Lambda}) = \{(M, M) | M \in \bar{\Lambda}\}$) la relation $\rightarrow^*$ est la fermeture transitive de $\triangleright$ et est donc fortement confluente.

Nous donnons ici les deux relations fortement confluentes et une idée de la preuve de la confluence forte.

$$(i)\ m \mathrel{\dot{\triangleright}} m \qquad (ii)\ \frac{m \mathrel{\dot{\triangleright}} n}{\lambda x.m \mathrel{\dot{\triangleright}} \lambda x.n}$$

$$(iii)\ \frac{m \mathrel{\dot{\triangleright}} m' \quad n \mathrel{\dot{\triangleright}} n'}{m\, n \mathrel{\dot{\triangleright}} m'\, n'} \qquad (iv)\ \frac{m \mathrel{\dot{\triangleright}} m' \quad n \mathrel{\dot{\triangleright}} n'}{(\lambda \dot{x}.m)n \mathrel{\dot{\triangleright}} m'[\dot{x} := n']}$$

$$(I)\ M \triangleright M \qquad (II)\ \frac{m \mathrel{\dot{\triangleright}} m'}{\pi\, m \triangleright \pi\, m'} \qquad (III)\ \frac{M \triangleright M'}{\lambda x.M \triangleright \lambda x.M'}$$

$$(IV)\ \frac{M \triangleright M' \quad N \triangleright N'}{M\, N \triangleright M'\, N'} \qquad (V)\ \frac{M \triangleright M' \quad n \mathrel{\dot{\triangleright}} n'}{M\, n \triangleright M'\, n'}$$

$$(VI)\ \frac{M \triangleright M' \quad N \triangleright N'}{M \# N \triangleright M' \# N'} \qquad (VII)\ \frac{M \triangleright M' \quad N \triangleright N'}{M \,?\, N \triangleright M' \,?\, N'}$$

$$(VIII)\ \frac{M \triangleright M' \quad N \triangleright N'}{M \mathbin{\ddot{?}} N \triangleright M' \mathbin{\ddot{?}} N'} \qquad (IX)\ \frac{M \triangleright M' \quad N \triangleright N'}{M \,!\, N \triangleright M' \,!\, N'}$$

$$(X)\ \frac{M \triangleright M' \quad m \mathrel{\dot{\triangleright}} m' \quad M_1 \triangleright M_1' \quad M_2 \triangleright M_2'}{(M \overset{m}{\rightarrow} M_1, M_2) \triangleright (M' \overset{m'}{\rightarrow} M_1', M_2')}$$

$$(XI)\ \frac{m \mathrel{\dot{\triangleright}} m' \quad M_1 \triangleright M_1' \quad M_2 \triangleright M_2'}{(m \rightarrow M_1, M_2) \triangleright (m' \rightarrow M_1', M_2')}$$

$$(XII)\ \frac{M \ \triangleright\ M' \quad N \ \triangleright\ N'}{(\lambda\bar{x}.M)N \ \triangleright\ M'[\bar{x} := N']} \qquad (XIII)\ \frac{M \ \triangleright\ M' \quad n \mathrel{\dot{\triangleright}} n'}{(\lambda\dot{x}.M)n \ \triangleright\ M'[\dot{x} := n']}$$

$$(XIV)\ \frac{m \mathrel{\dot{\triangleright}} m' \quad n \mathrel{\dot{\triangleright}} n'}{\pi\, m\, ?\, \pi\, n \ \triangleright\ \pi(\lambda\dot{x}.m'(n'\,\dot{x}))}$$

$$(XV)\ \frac{m \mathrel{\dot{\triangleright}} m' \quad n \mathrel{\dot{\triangleright}} n'}{\pi\, m \,\#\, \pi\, n \ \triangleright\ \pi(\lambda\dot{x}.(m'\,\dot{x})(n'\,\dot{x}))}$$

$$(XVI)\ \frac{m \mathrel{\dot{\triangleright}} m' \quad n \mathrel{\dot{\triangleright}} n'}{\pi\, m\, \boldsymbol{?}\, \pi\, n \ \triangleright\ \pi(\lambda\dot{x}\dot{y}.(n\,\dot{x}\,\dot{y} \to m\,\dot{y}, \mathbf{nc}))}$$

$$(XVII)\ \frac{m \mathrel{\dot{\triangleright}} m' \quad n \mathrel{\dot{\triangleright}} n'}{\pi\, m\, !\, \pi\, n \ \triangleright\ \pi(\lambda\dot{x}\dot{y}.(n\,\dot{x}\,\dot{y} \to m\,\dot{y}\,\dot{x}, \mathbf{nc}))}$$

$$(XVIII)\ \frac{M_1 \ \triangleright\ M_1' \quad m\,n \mathrel{\dot{\triangleright}} T}{(\pi\, m \xrightarrow{n} M_1, M_2) \ \triangleright\ M_1'} \qquad (XIX)\ \frac{M_2 \ \triangleright\ M_2' \quad m\,n \mathrel{\dot{\triangleright}} F}{(\pi\, m \xrightarrow{n} M_1, M_2) \ \triangleright\ M_2'}$$

$$(XX)\ \frac{M_1 \ \triangleright\ M_1' \quad m \mathrel{\dot{\triangleright}} T}{(m \to M_1, M_2) \ \triangleright\ M_1'} \qquad (XXI)\ \frac{M_2 \ \triangleright\ M_2' \quad m \mathrel{\dot{\triangleright}} F}{(m \to M_1, M_2) \ \triangleright\ M_2'}$$

La preuve de la confluence forte de $\dot{\triangleright}$ est donnée dans [Bar86]. Deux lemmes préalables sont nécessaires pour la preuve de la confluence forte de $\triangleright$.

Lemme 1 *Soit $M, N \in \bar{\Lambda}$ tels que $M \ \triangleright\ M'$ et $N \ \triangleright\ N'$. alors*

$$M[\bar{x} := N] \ \triangleright\ M'[\bar{x} := N']$$

Lemme 2 *Soit $M \in \bar{\Lambda}$ et $n \in \dot{\Lambda}$ tels que $M \ \triangleright\ M'$ et $n \mathrel{\dot{\triangleright}} n'$. alors*

$$M[\dot{x} := n] \ \triangleright\ M'[\dot{x} := n']$$

La preuve de ces lemmes se fait par induction sur la définition de $M \triangleright M'$ et utilise une adaptation du lemme de substitution pour λ-termes à nos BSλ-termes. Il reste ensuite à prouver le lemme de confluence forte :

Lemme 3 *$\forall M, M_1, M_2 \in \bar{\Lambda}$ tels que $M \ \triangleright\ M_1$ et $M \ \triangleright\ M_2$ il existe $M_3 \in \bar{\Lambda}$ tels que $M_1 \ \triangleright\ M_3$ et $M_2 \ \triangleright\ M_3$:*

$$\begin{tikzcd}
 & M \arrow[dl] \arrow[dr] & \\
M_1 \arrow[dr] & & M_2 \arrow[dl] \\
 & M_3 &
\end{tikzcd}$$

Preuve :

On raisonne par induction sur la définition de $M \rhd M_1$. Il y a ainsi autant de cas qu'il y a de règles dans la définition de $\rhd$. Pour chaque cas $M \rhd M_1$ est le dénominateur de la règle considérée. Il faut pour chaque cas de $M \rhd M_1$ considérer les cas de $M \rhd M_2$. Le choix de la règle pour $M \rhd M_1$ impose une racine syntaxique pour M. On étudie alors les règles pour $M \rhd M_2$ où M a la bonne racine syntaxique. Nous allons donner un aperçu de la preuve sur deux cas.

Cas 1 $M \rhd M_1$ est $(\lambda\bar{x}.N)P \rhd N'[\bar{x} := P']$ avec $N \rhd N'$ et $P \rhd P'$ (dénominateur de la règle (XII)). Deux sous-cas sont alors possibles.

Sous-cas 1.1 $M \rhd M_2$ est $Q\,P \rhd Q''\,P''$ avec $P \rhd P''$ et $Q \rhd Q''$. Nécessairement, on a $Q \equiv (\lambda\bar{x}.N)$, $Q'' \equiv (\lambda\bar{x}.N'')$, avec $N \rhd N''$. Par hypothèse d'induction, il existe $N''', P''' \in \bar{\Lambda}$ tels que $N' \rhd N'''$, $N'' \rhd N'''$, $P' \rhd P'''$ et $P'' \rhd P'''$. La règle (XII) entraine

$$(\lambda\bar{x}.N'')P'' \rhd N'''[\bar{x} := P''']$$

Le lemme 1 nous donne que

$$N'[\bar{x} := P'] \rhd N'''[\bar{x} := P''']$$

Ainsi on conclut que $M_3 \equiv N'''[\bar{x} := P''']$.

Sous-cas 1.2 $M \rhd M_2$ est $(\lambda\bar{x}.N)P \rhd N''[\bar{x} := P'']$ avec $N \rhd N''$ et $P \rhd P''$. Par hypothèse d'induction, il existe $N''', P''' \in \bar{\Lambda}$ tels que $N' \rhd N'''$, $N'' \rhd N'''$, $P' \rhd P'''$ et $P'' \rhd P'''$. Le lemme 1 nous donne

$$N'[\bar{x} := P'] \rhd N'''[\bar{x} := P''']$$
$$N''[\bar{x} := P''] \rhd N'''[\bar{x} := P''']$$

Ainsi on conclut que $M_3 \equiv N'''[\bar{x} := P''']$.

Cas 2 $M \rhd M_1$ est $(m \to M_1, M_2) \rhd M_1'$ avec $M_1 \rhd M_1'$ et $m \dot{\rhd} T$ (dénominateur de la règle (XX)). Deux sous-cas sont alors possibles.

Sous-cas 2.1 $M \rhd M_2$ est $(m \to M_1, M_2) \rhd M_1''$ avec $M_1 \rhd M_1''$ et $m \dot{\rhd} T$. Par hypothèse d'induction, il existe M_1''' tel que $M_1' \rhd M_1'''$ et $M_1'' \rhd M_1'''$. On prend alors $M_3 \equiv M_1'''$.

Sous-cas 2.2 $M \rhd M_2$ est $(m \to M_1, M_2) \rhd (m'' \to M_1'', M_2'')$ avec $m \dot{\rhd} m''$, $M_1 \rhd M_1''$ et $M_2 \rhd M_2''$. Par confluence forte de $\dot{\rhd}$, $m'' \dot{\rhd} T$. Par hypothèse d'induction, il existe M_1''' tel que $M_1' \rhd M_1'''$ et $M_1'' \rhd M_1'''$. Par la règle (XX), on a $(m'' \to M_1'', M_2'') \rhd M_1'''$ et on conclut que $M_3 \equiv M_1'''$.

□

Remerciements

Dans une précédente version du calcul nous proposions une opération de composition parallèle. Les relecteurs ont détecté une erreur dans cette précédente version qui impliquait la non-confluence du calcul obtenu. Qu'ils en soient remerciés. Le calcul du présent article ne contient plus cette opération de composition parallèle. Ce travail a été réalisé en collaboration avec Gaétan Hains. Christian Foisy a participé aux débuts de BSλ lors d'une visite à l'Université d'Orléans. BSλ a été inspiré par le concept miniBSP de David Skillicorn. Arnaud Lallouet a suggéré la forme actuelle du constructeur de champ π. Dans une version précédente le formalisme était moins général : des termes de la forme $\pi\, x.\, e$ où x est lié dans e étaient utilisés. Thérèse Hardin a eu la gentillesse de nous faire un passage en revue des différentes méthodes de preuves de confluence du λ-calcul. L'auteur est financé par une allocation du MENER et a été financé par le GDRP du CNRS.

Bibliographie

[Bar86] H.P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 1986.

[BFMR96] 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.

[BK96] G.-H. Botorog and H. Kuchen. Efficient parallel programming with algorithmic skeletons. In Bougé et al. [BFMR96].

[CCL91] M. Chen, Y.-Il Choo, and J. Li. Crystal: Theory and pragmatics of generating efficient parallel code. In B. K. Szymanski, editor, *Parallel Functional Languages and Compilers.* ACM Press, 1991.

[CF95] E. Chailloux and C. Foisy. Caml-Flight alpha: Implantation et applications. In C. Queinnec, V. V. Donzeau-Gouge, and P. Weis, editors, *Journées Francophones desLangages Applicatifs*, number 13 in Collection Didactique. INRIA, Janvier 1995.

[Col89] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation.* MIT Press, 1989.

[LH98] F. Loulergue and G. Hains. An Introduction to BSλ. Technical Report RR98-09, LIFO, Université d'Orléans, September 1998.

[LHF] F Loulergue, G. Hains, and C. Foisy. A Calculus of Recursive-Parallel BSP Programs. Extended version with proof of confluence. Submitted for publication.

[LHF98] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Recursive-Parallel BSP Programs. In S. Gorlatch, editor, *First International Workshop on Constructive Methods for Parallel Programming (CMPP'98)*, Research Report MIP-9805, pages 59–70. University of Passau, May 1998.

[Lou98] F. Loulergue. BSML : Programmation BSP purement fonctionnelle. In D. Méry and G.-R. Perrin, editors, *Dixième Rencontres Francophones du Parallélisme (Renpar'10)*, pages 243–246, Strasbourg, june 1998.

[McC96] W.F. McColl. Universal computing. In Bougé et al. [BFMR96].

[SH86] G.H. Steele and W.D. Hillis. Connection Machine Lisp: A Fine Grained Parallel Symbolic Processing Language. In *ACM Conf. on Lisp and Funct. Progr.*, pages 276–297, Cambridge MA, 1986.

[Val90] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[VP92] E. Violard and G.-R. Perrin. PEI: A language and its refinement calculus for parallel programming. *Parallel Computing*, 18(10):1167–1184, October 1992.

[VP93] E. Violard and G.-R. Perrin. PEI: a single unifying model to design parallel programs. 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.