Une bibliothèque certifiée de programmes fonctionnels BSP

Frédéric Gava[1]

[1] *Laboratoire d'Algorithmique, Complexité et Logique*
Université Parix XII, Val-de-Marne
61, avenue du Général de Gaulle
94010 Créteil cedex – France
`gava@univ-paris12.fr`

1. Introduction

Bulk Synchronous Parallel ML ou BSML est une extension de ML pour la programmation fonctionnelle en mode direct d'algorithmes parallèles BSP (*Bulk Synchronous Parallelism*), un modèle de programmation introduit par Valiant [28] afin d'offrir à la fois un haut degré d'abstraction comme les modèles PRAM [7] tout en étant portable et en permettant la prévision réaliste de performances sur une grande variété d'architectures. Un algorithme BSP est dit en en mode direct [12] lorsque la structure physique de ses processus est explicite. Les programmes BSML sont écrits à l'aide d'un petit ensemble de primitives issues d'un calcul *confluent*, le BSλ-calcul [20] : un constructeur de vecteurs parallèles, une application parallèle asynchrone, une opération de communication globale synchrone et enfin une conditionnelle globale synchrone. Ceci est donc un bon cadre de travail pour l'écriture et la preuve de programmes parallèles car nous pouvons concevoir des programmes parallèles purement fonctionnels. Sans effet de bords, il est plus simple de prouver leur correction et de les réutiliser en tenant compte de leurs propriétés formelles (la sémantique est dite *compositionnelle*).

La bibliothèque BSMLlib [1] implante les primitives BSML en Objective Caml [2] avec la bibliothèque MPI (*Message Passing Interface*) [26]. Elle est efficace et ses performances suivent les estimations faites par le *modèle de coût* BSP (qui permet d'estimer les temps d'exécution parallèle). Elle est aussi utilisée comme base du projet Caraml [3], qui a pour objectif l'utilisation d'Objective Caml pour la programmation parallèle sur la grille de calcul, avec des applications par exemple aux bases de données parallèles ou à la simulation moléculaire.

Un système de types [10] (dont l'inférence est possible) a été conçu afin d'assurer la sûreté du langage, notamment en prohibant l'emboîtement du parallélisme (imbrication de vecteurs parallèles) afin d'avoir à la fois une implantation simple et un modèle de coût qui suit le modèle BSP. Mais ceci n'est pas suffisant pour les programmes *critiques* qui demandent des preuves formelles de leurs algorithmes ou pour la bibliothèque standard qui devrait être certifiée, c'est-à-dire ne comporter que des programmes qui calculent réellement ce que l'on attend d'eux. Tester les programmes sur une grande variété d'entrées permet de détecter un certain nombre d'erreurs. Malgré tout, seules les méthodes formelles peuvent en garantir la correction.

Le système **Coq** [4] est un environnement (et un langage logique) pour le développement de preuves basé sur le Calcul des Constructions Inductives [3], un λ-calcul typé étendu avec des *définitions*

1. http://bsmllib.free.fr/
2. http://ocaml.org/
3. http://caraml.org/
4. http://coq.inria.fr/

inductives. La théorie des types est un bon cadre pour le développement de preuves de programmes (notamment les programmes fonctionnels [22]) car il fournit une grande expressivité logique. Dans l'assistant de preuves **Coq** il existe une interprétation constructive des preuves, c'est-à-dire, prouver une formule logique implique la construction d'un λ-terme typable. Ainsi, dans un tel formalisme et grâce à l'isomorphisme de Curry-Howard, une preuve de la formule: $\forall x.P(x) \Rightarrow \exists y.Q(y,x)$ (appelée une *spécification*) permet d'obtenir un programme correct qui vérifie en entrée la propriété P et fournit un résultat qui vérifie Q. Le programme extrait de la preuve (en oubliant les parties logiques de la preuve [23]) est donc garanti par **Coq** pour bien réaliser la spécification donnée (on parle alors de *programme certifié*).

Nous allons maintenant nous intéresser à la spécification de programmes BSML et à leur développement (réalisation) en **Coq**. Tout d'abord, nous présentons rapidement le modèle BSP (section 2.1), donnons une présentation informelle de BSML (section 2.2) puis du système **Coq** (section 3). Nous pouvons alors expliquer comment intégrer ce langage parallèle dans le système **Coq** (section 4) pour certifier des éléments de la bibliothèque BSMLlib et ainsi obtenir une bibliothèque certifiée (section 5). Enfin, nous parlerons brièvement d'autres travaux sur la preuve de programmes parallèles (section 6) puis nous conclurons (section 7).

2. Programmation fonctionnelle BSP

2.1. Le modèle BSP

Le modèle de programmation parallèle BSP (*Bulk Synchronous Parallelism*) [25] décrit une architecture parallèle (abstraite), un modèle d'exécution et un modèle de coût.

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 et enfin 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). Par la suite, nous parlerons indifféremment de processus ou de processeurs. 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 nécessaire à la réalisation d'une barrière de synchronisation et pour finir, le temps 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 [13].

L'exécution d'un programme BSP est une séquence de *super-étapes*. Chaque super-étape est divisée en trois phases successives et logiquement disjointes (Fig. 1) : 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, puis le réseau réalise les échanges de données demandés à la phase précédente et enfin 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.

Le temps nécessaire à l'exécution d'une super-étape est la somme du maximum des temps de calculs locaux, du temps de la réalisation des échanges entre processeurs (la h-relation) et du temps de la

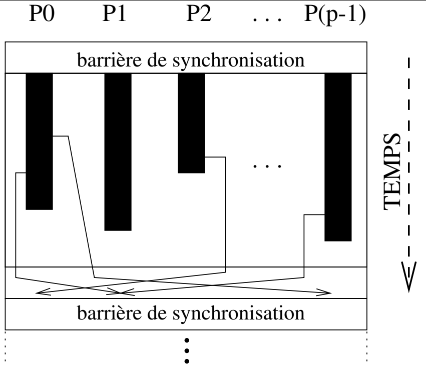


FIG. 1: *Une super-étape BSP*

réalisation d'une barrière de synchronisation globale. Le temps d'exécution d'un programme BSP est donc la somme du temps d'exécution de ces super-étapes. Afin de minimiser le temps d'exécution, un algorithme BSP doit donc à la fois minimiser le nombre de super-étapes, le déséquilibre des communications et le déséquilibre des calculs locaux. Nous nous référons à [25] pour de plus amples informations sur le modèle de coût et à [12] pour la description de nombreux algorithmes BSP.

2.2. La bibliothèque `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. Notre système de type [10] permet d'éviter cette imbrication. Ceci constitue une amélioration par rapport à Caml Flight [2] dans lequel l'imbrication de la structure parallèle globale de contrôle `sync` était interdite *dynamiquement*. Contrairement à la programmation SPMD (*Single Program Multiple Date*), qui est pratiquée en utilisant un langage séquentiel et une bibliothèque de communication par passage de messages comme MPI (*Message Passing Interface*) [26], BSML ne contient pas de deadlock, est confluent et possède le modèle de coût BSP. 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 $(p-1)$.

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`. 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 :

put(mkpar(fun i->fs$_i$)). Pour envoyer une valeur v d'un processeur j vers un processeur i, la fonction fs$_j$ du processeur j doit être telle que (fs$_j$ i) s'évalue en Some v. Pour ne pas envoyer de message de j à i, (fs$_j$ i) doit s'évaluer en None. L'expression précédente s'évalue en un vecteur parallèle contenant en chaque processeur une fonction fd$_i$ de messages transmis. Au processeur i, (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. Le langage complet contiendra également une projection globale synchrone [5] :

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

telle que proj v n s'évaluera en v_n la valeur du n$^{\text{ième}}$ processeur (la n$^{\text{ième}}$ composante du vecteur parallèle). Cette projection globale permet d'exprimer une phase de communications et une de synchronisation (et est implémentée comme un "broadcast"). Sans elle, le contrôle global ne pourrait tenir compte des valeurs locales et il serait impossible d'écrire des algorithmes ayant la forme suivante :

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

L'annexe A présente des exemples de programmes BSML de la bibliothèque certifiée (générés par l'extracteur du système **Coq**).

3. L'assistant de preuve Coq

Le système **Coq** est un assistant de preuves. Il fournit des mécanismes pour écrire des définitions et pour faire des preuves formelles. D'après l'isomorphisme de Curry-Howard, les types sont vus comme des propositions (spécifications des structures de données) et les termes comme des preuves. **Coq** construit ses termes au moyen de *tactiques* données par l'utilisateur. La *sûreté* du système **Coq** (principe de certification) est une conséquence de la *consistance* (confluence et normalisation forte) du cadre logique et aussi du contrôle des types une fois la preuve réalisée. Nous présenterons brièvement les principes de **Coq**, afin d'expliquer comment nous avons réalisé un procédé de décision pour la spécification et la réalisation de programmes de la bibliothèque BSMLLIB.

Les types peuvent être définis par leurs aritées. Par exemple, le type des nombres naturels peut être défini comme nat:Set avec le type inductif suivant :

```
Inductive nat : Set := O : nat | S : nat->nat
```

Ainsi, la fonction successeur peut être définie par succ:nat ->nat. Des prédicats sur les objets nat peuvent alors être définis, comme par exemple, le prédicat $\leq$ qui peut être défini par le:nat ->nat ->Prop. Maintenant, prédicats et objets peuvent être mélangés dans les spécifications. Par exemple, (n:nat) {m:nat | m=n*2} est la spécification pour exprimer $\forall n \Rightarrow \exists m\ m = n \times 2$. Une preuve d'une telle spécification est habitée par une paire composée d'un nombre naturel et d'une preuve que ce nombre satisfait la spécification. Le programme extrait à partir de la preuve (une réalisation) de cette spécification est naturellement la fonction double. Nous nous référons au manuel de référence de la page web du système **Coq** pour les notations et les différentes formations possibles des termes et des types de données (types inductifs).

Quand la spécification a été donnée, nous devons la prouver. Pour ceci, **Coq** entre dans un mode spécial et interactif où l'utilisateur doit prouver les différents buts de la spécification dans le contexte auquel il fait la preuve (c'est à dire les spécifications déjà prouvées) à l'aide de tactiques (des commandes qui appliquent une ou plusieurs règles logiques au but courant). Quelques autres systèmes comme HOL ou PVS offrent des possibilités semblables. Chaque tactique est associée à une validation qui permet, une fois que la preuve est faite, de vérifier que les règles de déduction ont été correctement employées. Comme nous l'avons expliqué précédemment, des preuves de spécifications peuvent être développées (avec un contenu algorithmique qui peut différer) et des programmes peuvent alors être extraits de ces preuves. Un tel programme s'appelle une *réalisation* d'une spécification. L'extraction permet l'élimination des parties non algorithmiques des preuves, y compris les types dépendants.

5. en réalité, cette opérateur s'appelle at dans la BSMLLIB mais est utilisé autrement dans cette article (section 4)

```
Parameters bsp_p:unit -> Z; Vector : Set -> Set;
           mkpar: (T:Set) (Z->T) -> (Vector T).

Axiom good_bsp_p:`0 < (bsp_p tt)`.

Axiom at: (T:Set) (Vector T) -> (i:Z) `0<=i<(bsp_p tt)` -> T.

Axiom mkpar_def: (T:Set) (f:Z -> T)(i:Z) (H:`0 <= i < (bsp_p tt)`)
                    (at T (mkpar T f) i H)=(f i).

Axiom apply_def: (T1,T2:Set) (V1:(Vector (T1 -> T2)))
                 (V2:(Vector T1)) (i:Z) (H:`0 <= i < (bsp_p tt)`)
(at T2 (apply T1 T2 V1 V2)i H)=((at (T1->T2) V1 i H)(at T1 V2 i H)).

Axiom put_def: (T:Set)(Vf:(Vector (Z -> (option T))))  (i:Z) (H:`0 <= i < (bsp_p tt)`)
((at (Z->(option T)) (put T Vf) i H)=([j:Z] if (within_bound j)
    then [H1]((at (Z-> (option T)) Vf j H1) i)  else [H2](None T))).

Axiom proj_def: (T:Set)(V:(Vector T)) (n:Z) (Hyp_n: `0 <= n < (bsp_p tt)`)
                ((proj T V n)= (at T V n Hyp_n)).
```

FIG. 2: *Axiomatisation des opérateurs BSML*

Coq est donc un bon cadre pour établir une bibliothèque de programmes certifiés provenant de l'extraction des preuves de nos spécifications. Ces programmes peuvent être donnés dans deux langages de programmation différents: Objective Caml et Haskell [6] (ses extracteurs font partis de la distribution standard). Ainsi, dans le premier cas, nous pourrons employer la bibliothèque BSMLLIB pour compiler notre bibliothèque de programmes extraits et dans le second cas, le développement de [21].

4. Formalisation des opérateurs BSML

Pour représenter notre langage parallèle et pour avoir une spécification-réalisation des programmes BSML, nous avons choisi une approche classique: une *axiomatisation* de nos opérateurs. Les axiomes sont basés sur le modèle de programmation du BSλ_p-calcul [17] (qui est un BSλ-calcul avec une énumération des vecteurs) et sur la sémantique intrinsèque de BSML. Les opérateurs de BSML sont donnés via des *paramètres* (par souci de concision, nous ne donnons que le paramètre `mkpar`) et donc, ne dépendent pas de leurs implémentations (séquentielles ou parallèles).

Axiomes et paramètres Le nombre de processus, `bsp_p()`, est naturellement un nombre entier supposé supérieur à 0. Les vecteurs parallèles sont indexés sur le type `Z` (les nombres entiers de **Coq**). Ils sont représentés dans le "monde logique" par un type dépendant (via une fonction abstraite): `Vector T` où `T` est le type des éléments du vecteur. Ce type abstrait est bien entendu uniquement manipulé par nos paramètres. Notre axiomatisation des opérateurs parallèles BSML est donnée dans la figure 2. Celle-ci est simple mais suffisante pour notre approche (l'extraction des preuves de nos spécifications pour avoir une bibliothèque certifiée). `at` est une *fonction abstraite* d'"accès" pour les vecteurs parallèles. Elle donne la valeur locale contenue dans un processus et sera employée par la suite dans les spécifications des programmes pour donner les valeurs contenues dans les vecteurs parallèles.

6. http://www.haskell.org

Elle contient un type dépendant pour vérifier que l'entier `i` est bel et bien un nom de processus.

Les opérateurs asynchrones, le constructeur `mkpar` et l'application globale `apply` sont axiomatisés avec `mkpar_def` et `apply_def`. Pour une fonction `f`, `mkpar_def` applique `(f i)` sur le processus `i` en employant la fonction d'accès `at`. De la même manière, `apply_def` permet l'application des composantes au processus `i`. Le vecteur parallèle résultant est alors décrit avec une égalité qui est donc une proposition logique. Ce résultat est donné pour un paramètre `i` qui doit être prouvé comme un nom valide de processus.

L'opérateur `put` est axiomatisé avec `put_def`. Il transforme un vecteur fonctionnel vers un autre vecteur fonctionnel qui permet la communication en utilisant le paramètre `j` pour lire les valeurs des processus distants (dans le vecteur `Vf`). Le paramètre `j` est testé avec la fonction `within_bound` de type: `(i:Z) {'0<=i<(bsp_p tt)'}+{~'0<=i<(bsp_p tt)'}` qui indique si un entier est un nom de processus valide ou non et en fournit une preuve. En effet, pour des preuves de programmes, il est plus aisé de manipuler directement des preuves que des constantes primitives. Si `i` est bien un nom de processus valide, la valeur sur le processus `i` est lue sur le processus `j` avec la fonction d'accès `at` sur la $\mathtt{j}^{\text{ième}}$ composante du vecteur `Vf` (j est un nom valide de processus; la preuve est donnée par `within_bound`). Autrement, une constante vide est retournée. Dans une véritable implémentation, et pour des raisons évidentes d'optimisation, les valeurs à émettre sont tout d'abord calculées et ensuite échangées.

L'axiomatisation complète contient également la projection globale synchrone. Elle est facilement exprimable dans le système **Coq** en utilisant la fonction d'`"accès"` `at` pour "lire" la valeur au processeur `n` (dans la $\mathtt{n}^{\text{ième}}$ composante du vecteur parallèle `V`). Ce paramètre `n` doit être un nom valide de processus et cette preuve est donnée à la fonction d'`"accès"` `at` par un type dépendant.

Nous allons maintenant illustrer cette axiomatisation en employant nos axiomes pour obtenir une bibliothèque certifiée par l'assistant de preuve **Coq**. Nous pouvons voir l'ajout de ces axiomes comme le passage de **Coq** à un "BS-Coq" (comme le passage de ML à BSML par l'utilisation de nos opérateur parallèles). Bien entendu, il est possible d'extraire un programme ML non parallèle (extraction classique du système **Coq**) si on n'utilise pas ces paramètres (et leurs axiomes associés). D'ailleurs une partie de notre bibliothèque certifiée provient directement de l'extraction des librairies standard de **Coq** (notamment la bibliothèque sur les entiers relatifs).

Cohérence Le BSλ_p-calcul a été utilisé pour tester la consistance de nos axiomes : pour accepter un axiome comme une règle d'inférence, nous avons préalablement et informellement vérifié que celui-ci est satisfait par les régles du BSλ_p-calcul qui est confluent. En effet, [18] propose deux exécutions possibles et équivalentes des BSλ_p termes : une version séquentielle et la version distribuée (parallèle) associée. Pour prouver la cohérence de nos axiomes dans le système **Coq** (impossibilité de prouver $\perp$), nous envisageons une implémentation en **Coq** (sans axiome) à l'aide de listes (de tailles "fixes") : les axiomes serait alors des lemmes et la cohérence viendrait de celle de **Coq**. Ainsi, nos axiomes et nos paramètres (dont on pourrait donner une implémentation certifiée) correspondraient à l'exécution séquentielle mais au moment de la compilation (des fonctions extraites), rien n'empêcherait d'utiliser l'implémentation parallèle (qui a été prouvée équivalente).

5. Création d'une bibliothèque `BSMLlib` certifiée

Nous présentons uniquement ici, par souci de clarté et de concision, le développement de trois fonctions parallèles de la bibliothèque BSMLlib dans le système **Coq** qui sont très communes dans un algorithme BSP. Cette étude de cas démontre la pertinence de l'utilisation du système **Coq** dans la preuve de programmes parallèles (les fonctions n'ont pas tous été spécifiées dans un autre formulisme et encore moins implémentées).

Recevoir des valeurs d'autres processeurs Notre premier cas est le développement d'un dual de l'opérateur `put`: `get_list`. En effet, échanger des valeurs est l'un des points critiques d'un algorithme BSP. Cette fonction permet à un processus de recevoir les valeurs d'autres processus. Le type ML est donc: `(vec:'a par) ->(lpids:z list par) ->'a list par` avec :

$$\texttt{get\_list}\ \langle v_0, \ldots, v_{p-1}\rangle\ \langle l_0, \ldots, l_{p-1}\rangle\ =\ \langle l'_0, \ldots, l'_{p-1}\rangle$$

tels que si $(\texttt{nth}\ l_i) = j$ alors $(\texttt{nth}\ l_i)' = v_j$ pour $0 \leq i < p$. Pour une simple raison, cette fonction est implémentée avec deux `put` : les processus, à la première super-étape, envoient des requêtes aux processus désirés pour obtenir leurs valeurs puis, à la seconde super-étape, ceux-ci envoient leurs valeurs aux processus qui leur ont envoyé une requête. Nous avons alors la spécification logique suivante (nous ne donnons pas le type quand celui-ci est évident et [.] est un sucre syntaxique pour l'"accés" aux éléments d'un vecteur parallèle) :

$$\begin{gathered}\forall \mathrm{Data}\ \forall vec\!:\!(Vector\ Data)\ \forall lpids\!:\!(Vector\ (list\ Z))\\ (\forall i\ H\!:\!(0\leq i<p)(lpids[i]=nil)\vee(\forall n\ H_n\!:\!(1\leq n<length(lpids[i]))\to(0\leq(nth\ lpids[i]\ n\ H_n)<p))\\ \Rightarrow \exists res \text{ tels que } \forall i\ H\!:\!(0\leq i<p)\ length(res[i])=length(lpids[i])\wedge\\ (\forall n\ H_n\!:\!(1\leq n<length(lpids[i]))\to(nth\ res[i]\ n\ H_n)=vec[(nth\ lpids[i]\ n\ H_n)]\end{gathered}$$

où nous donnons le vecteur des éléments puis le vecteur des listes de noms de processus. Ces listes, en chaque processus, sont vides ou contenant des noms valides de processeurs. Le résultat est un vecteur de listes (d'éléments) qui sont de même longueur que la liste de départ et qui contiennent les éléments désirés. Cette spécification est prouvée en utilisant les propriétés formelles des listes (*nth* nécessite une preuve que l'entier soit compris entre 1 et la taille de la liste) et par cas sur les numéros de processus des listes données en paramètres (si celui existe, d'où l'utilisation du $\vee$ logique). On peut remarquer que la fonction est *non-totale* puisqu'elle ne peut être appliquée (utilisée) que si ces numéros de processus sont valides. Ceci permet de toujours fournir une preuve pour l'"accés" aux élements du vecteur.

Rassemblement d'un vecteur parallèle `gather_list` (de type `'a par -> z ->'a list par`) est une fonction qui rassemble les valeurs $v_0, \ldots, v_{p-1}$ (une par processeur) dans une liste sur un processeur donné en paramètre ; (`gather_list` *root* $\langle v_0, \ldots, v_{p-1}\rangle$) s'évalue en un vecteur parallèle dont la valeur au processeur *root* est la liste $[v_0; \ldots; v_{p-1}]$. Pour cela, nous utilisons une fonction intermédiaire, `gather`, qui donne un vecteur parallèle de fonctions, dont celle au processeur *root* donne la valeur d'un autre processeur *dest*, si celui-ci est un numéro de processeur valide (autrement, elle retourne `None`), et les fonctions aux processeurs $i \neq root$ retournent toujours `None` pour toute valeur de *dest*. Ainsi, au processeur *root*, nous pouvons appliquer cette fonction à une liste contenant les numéros de processeur dans le cas du processeur *root* et une liste vide dans le cas des autres processeurs. Nous avons alors le vecteur parallèle désiré. La spécification formelle de `gather_list` est :

$$\begin{gathered}\forall \mathrm{Data}\ \forall root\ (0\leq root<p)\ \forall vect \Rightarrow \exists res \text{ tels que}\\ \exists l\!:\!list \text{ tels que } l=res[root] \text{ et } H\!:\!(length\ l)=p \text{ et } \forall j\ (0\leq j<p)\ (nth\ l\ H\ j)=vect[j]\\ \text{et } \forall i(\ 0\leq i<p)\ (i\neq root)\ res[i]=nil\end{gathered}$$

où `nth` est une fonction de la $n^{\text{ième}}$ valeur de la liste l (utilisant la preuve que $j \leq (length\ l)$) et l est la liste au processeur *root* de longueur p (le nombre de processeurs) et contenant les éléments désirés (sur les autres processeurs, la liste est vide). Cette spécification peut être prouvée en utilisant la propriété de `gather`, par cas sur $i = root$ et avec un lemme technique qui donne une propriété sur la longueur des listes de chaque processus (celle-ci est de longueur p au processeur *root*, 0 autrement)

Réduction directe La fonction "fold" (de type `('a ->'a ->'a) ->'a par ->'a par`) est un algorithme parallèle classique. L'algorithme utilise un opérateur binaire `R` et informellement, (`fold` R

$\langle x_0,\ldots,x_{p-1}\rangle)$ s'évalue en $\langle s,\ldots,s\rangle$ où $s = R_{k\leq p}x_k$. On peut donc en déduire la spécification suivante:

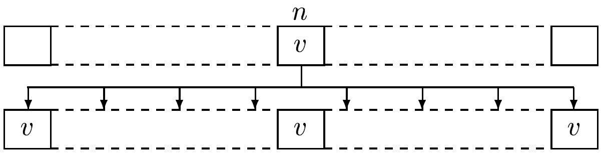
$$\forall \text{Data } \forall vect\ \forall R : Data \to Data \to Data \Rightarrow$$
$$\exists res \text{ tels que } \forall i\ H : (0 \leq i < p)$$
$$(k\_first\_R\ T\ R\ res\ (p-1)\ (0 \leq (p-1) < p)\ res[i])$$

qui utilise le type inductif suivant :

```
Inductive  k_first_R  [T:Set][R:T->T->T] [v:(Vector T)]
              : (k:Z) `0<=k<(bsp_p tt)` -> (res:T) Prop:=
   k_zero: (k_first_R T R v `0` H_0 (at T v `0` H_0))         (* cas de départ *)
 | k_rec : (k:Z) (H_r:`0<=k<(bsp_p tt)`)  (a:T) (H_sub:`k>0`)
                 (k_first_R T R v `k-1` (p_sub k H_r H_sub)  a) (* cas récursif *)
             -> (k_first_R T R v `k` H_r (R a (at T v `k` H_r))).
```

qui donne l'application de l'opérateur `R` aux k premiers éléments du vecteur parallèle (`p_sub` est un lemme technique qui transforme une preuve de $0 \leq k < p$ et $k > 0$ en $0 \leq (k-1) < p$). Étant un inductif logique, il n'apparaît donc pas dans le programme ML extrait. Il est seulement employé ici pour vérifier les propriétés formelles. Ainsi nous pouvons utiliser la fonction d'"accès" sans problème. Pour construire notre réalisation de cette spécification, nous avons dû préalablement développer une fonction d'"échange total" entre les processus (qui retourne, en chaque processus, une liste des valeurs des vecteurs parallèles) puis, faire une démonstration par cas sur le nom du processus : le premier processus n'effectue pas de calcul tandis que les autres vont utiliser une fonction séquentielle (`fold_left`) sur les listes. Nous utilisons alors les propriétés (données par la bibliothèque standard `PolyList`) de cette fonction (interactions entre les éléments de la liste des valeurs du vecteur et les k premiers éléments) pour terminer la preuve.

Diffusion d'une valeur Notre dernière fonction certifiée est un algorithme BSP classique : l'émission ("broadcast") d'une valeur v d'un processeur n vers les autres processeurs. Ceci peut être représenter par la figure suivante :



Ceci peut être effectué de manière directe (une super-étape) et pourrait être spécifié par :

$$\forall \text{Data } \forall root\ (0 \leq root < p)\ \forall vv \Rightarrow \exists res \text{ tels que } \forall i(\ 0 \leq i < p)\ (\text{res}[i] = \text{vv}[root])$$

qui indique que chaque composante (chaque processeur) a la valeur désirée. Mais pour les données de grandes tailles, [12] propose un algorithme plus fin (et plus efficace), l'émission en deux super-étapes (illustrée par la figure 3) qui procéde ainsi : le processeur de départ "découpe" son message en p messages et envoie chacun d'eux aux $p-1$ autres processeurs (première super-étape). Ensuite, chaque processeur envoie son bout de message aux autres processeurs (un échange total) et pour finir chaque processeur "recolle" les morceaux reçus. Nous avons alors la fonction : `bcast_twophases: ('a -> z ->'b option) ->((z ->'b) ->'a) -> z ->'a par ->'a par` et la spécification suivante :

$$\forall \text{Data1,Data2 } \forall decoupe : (Data1 \to z \to (Data2\ option))\ \forall recolle : (z \to Data2) \to Data1$$
$$(H1 : (\forall x\ \forall i\ \neg((partition\ x\ i) = None)))\ (\forall x\ (recolle\ (\lambda i.(noSome\ (H_1\ x\ i))\ (decoupe\ x\ i))) = x)$$
$$\forall root\ (0 \leq root < p)\ \forall vect \Rightarrow \exists res \text{ tels que } \forall i(\ 0 \leq i < p)\ (\text{res}[i] = \text{vect}[root])$$

où nous avons les fonctions *decoupe* et *recolle* tels que *decoupe* donne toujours un éléments (éventuellement vide) à envoyer à un processeur et *recolle*, pour chaque morceau du message, retourne

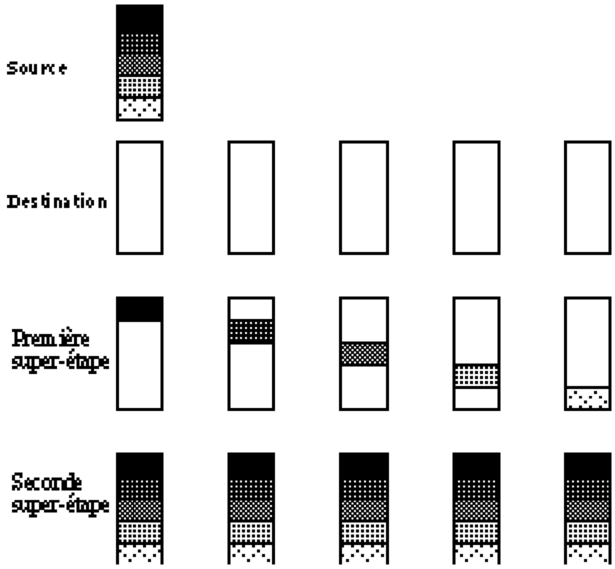


FIG. 3: *Diffusion d'une valeur en 2 super-étapes*

le message d'origine (l'utilisation du *noSome* et donc de $H1$ vient du fait que l'on manipule des objets de type `option`). Pour construire notre réalisation de cette spécification, nous avons dû préalablement développer une fonction d'"éparpillement" qui effectue la première super-étape puis cette spécification est prouvée par cas sur les processeurs (s'il font l'éparpillement ou non) et en utilisant les propriétés des paramètres pour reconstruire le message (après le "total-exchange", tous les bouts de messages sont disponibles pour pouvoir appliquer la fonction *recolle*).

Extraction de code Pour obtenir notre bibliothèque BSMLLIB certifiée, il suffit d'utiliser l'extracteur du système **Coq** sur les preuves constructives de nos spécifications. Naturellement, les fonctions données en paramètres devant être extraites vers les opérateurs pré-définis du langage BSML, nous devons préalablement préciser ce choix à l'extracteur du système. Les développements formels décrits ci-dessus (ainsi que les nombreux autres éléments la bibliothèque BSMLLIB certifiée) sont librement disponibles à la page web de l'auteur [7]. Ainsi, on peut programmer d'autres algorithmes BSP plus complexes mais dont les éléments standard (notamment les fonctions de communication) sont sûrs et par la suite, exécuter ces algorithmes sur de véritables machines parallèles avec les opérateurs parallèles de la bibliothèque BSMLLIB.

Pour illustrer ce travail, nous donnons en annexe le code CAML obtenu par l'extraction des réalisations de nos précédentes spécifications (avec quelques mise en formes). Pour aider la lecture de ces programmes nous donnons le type ML des principales fonctions, ainsi que leurs descriptions informelles.

`replicate: 'a ->'a par` permet de répliquer une même valeur sur tous les composants d'un même vecteur.

`parfun: ('a ->'b) ->'a par ->'b par` et `parfun2: ('a ->'b ->'c) ->'a par ->'b par -> 'c par` sont l'application d'une fonction aux éléments de vecteurs parallèles.

`procs : unit -> z list` retourne une liste des numéros de processus. `z` est le type des entiers relatifs dans le système **Coq**; cette fonction utilise `from_to: z ->z list` qui construit une liste d'entiers de 0 au paramètre ; dans ce cas, celui-ci est `last: unit ->z` qui donne le numéro du dernier

7. http://www.univ-paris12.fr/lacl/gava/

processus; l'égalité des entiers est donnée par la fonction `z_eq_dec: z ->z ->bool` (bibliothèque standard de l'assistant de preuves **Coq**); `in_dec: 'a list ->'a list ->bool` est la fonction de la bibliothèque standard de test d'inclusion de listes.

`totex: 'a par -> (z->'a) par` appliqué au vecteur parallèle $\langle v_0, v_1, \ldots, v_{p-1}\rangle$ s'évalue en un vecteur $\langle f_0, f_1, \ldots, f_{p-1}\rangle$ tels que $(f_i\ j) = v_j$; `total_exchange: 'a par ->'a list par` appliqué au même vecteur s'évalue en $\langle l_0, l_1, \ldots, l_{p-1}\rangle$ tels que $\mathrm{j}^{\text{ième}}$ éléments de l_i est v_j.

`gather: z ->'a par -> (z ->'a option) par` rassemble les valeurs $v_0, \ldots, v_{p-1}$ sur le processeur donné en premier paramètre; le résultat est en chaque processeur, une fonction f tels que $(f\ i) = v_i$ si i est un numéro de processeur valide.

`scatter: ('a -> z ->'b option) -> z ->'a par ->'b par`. (`scatter partition root` $\langle v_0, ..., v_{p-1}\rangle$) "éparpille" la valeur v_{root} (qui est découpé avec `partition`) sur les autres processeurs. (partition v pid) indique la partie du message `v` qui sera envoyée au processeur de nom `pid`.

Applications possibles [8] propose un langage fonctionnel parallèle : ML étendu avec des squelettes de programmation parallèle sur des vecteurs qui peuvent être imbriqués et qui peuvent être regroupés en quatre classes : calcul, ré-organisation, communication et regroupements. La compilation est décrite en une série de transformations (prouvées correctes) vers un langage fonctionnel de type SPMD. Celui-ci utilise des opérations parallèles qui sont en réalité celles que nous venons de décrire (et certifier). Nous pouvons de ce fait envisager l'utilisation de notre bibliothèque pour une implémentation certifiée de ce langage. [4] propose un calcul de distribution dont le but est de décrire des stratégies pour la distribution de tableaux (de multiples dimensions) sur un ensemble de processeurs. Les auteurs donnent une sémantique formelle permettant de prouver différentes équations de distributions, montrent comment associer à ces équations le modèle de coûts BSP autorisant ainsi de choisir une stratégie appropriée et distinguent ainsi un certains nombres d'opérateurs pour ces distributions. Ces opérateurs sont encore ceux que nous venons de certifier. En utilisant notre bibliothèque, les auteurs pourraient alors valider leurs systèmes d'équations de stratégies dans un environnement certifié.

6. Travaux antérieurs

La première sémantique formelle du modèle BSP était une logique axiomatique "à la Hoare" avec un parallélisme explicite pour un petit langage impératif avec une mémoire partagée [16]. Les programmes étaient donnés avec leurs spécifications et chaque pas du calcul devait être justifié par un raisonnement mathématique. Ceux-ci étaient basés en aplatissant chaque processus en une séquence de traces qui peuvent être combinées pour déterminer le fonctionnement de chaque composant. Malheureusement, aucun programme complet n'a été certifié avec ces règles algébriques et aucune implémentation de ce modèle de certification n'a été faite. Mais l'idée suggère la possibilité de prouver des algorithmes parallèles BSP avec des règles formelles tout en préservant le modèle de coût. Dans une autre approche [27], le raisonnement est fait en utilisant une séquence de transformations (globales) d'états parallèles rendant le raisonnement plus facile car les opérations parallèles peuvent être décrite par ces transformations. Le même papier montre l'utilisation de ces règles sur un algorithme BSP du plus court chemin d'un graphe (extention aisée de l'algorithme de Floyd). [24] présente une extension du "Refinement calculus" (un calcul de pré-condition faible) pour permettre la dérivation de programme BSP impératif. Toutes ces approches sont basées sur des langages impératifs avec des preuves faites "à la main" sans assistant de preuves. Un premier travail avait été fait dans [11] pour vérifier les propriétés de programme BSML de [20] en utilisant les règles de la théorie équationnelle du BSλ-calcul mais l'aide d'un assistant de preuves, les auteurs avaient fait quelques erreurs.

Notre approche a les avantages suivants. Elle est basée sur un langage purement fonctionnel (sans effets de bords) avec parallélisme explicite (les vecteurs parallèles) doté d'une sémantique de haut niveau. Ceci constitue une amélioration par rapport à Caml-Flight où le parallélisme est un effet. Le raisonnement en est ainsi grandement simplifié, ce qui permet de faire des preuves sur les programmes (le modèle BSP, en séparant calcul et communication, rend impossible les inter-blocages). En utilisant un assistant de preuve comme **Coq**, nos preuves sont partiellement automatisées ce qui n'est le cas dans aucune des approches précédentes. De plus, après les preuves, nous pouvons générer des programmes certifiés portables (grâce au modèle BSP) et bénéficier des développements ultérieurs faits en **Coq** (bibliothèque standard et contribution des autres utilisateurs).

7. Conclusion et futurs travaux

Cet article a démontré l'adéquation de l'assistant de preuves **Coq** à la spécification et à la réalisation de programmes parallèles. Nous avons formalisé les opérations parallèles du langage BSML et, en utilisant cette axiomatisation, nous avons pu développer des programmes BSML certifiés qui sont fréquemment utilisés et qui constituent un sous-ensemble important de la bibliothèque BSMLLIB. Nos axiomes sont informellement similaires aux opérateurs du BSλ_p-calcul mais une version (séquentielle) consistante qui réalise ses axiomes semble possible à réaliser, et fera partie de nos prochaines études.

Nos futurs travaux vont suivre les orientations suivantes. Tout d'abord, les fonctions ici certifiées sont simples à comprendre mais sont d'une variété suffisante pour se convaincre que le traitement d'un grand nombre d'algorithmes BSP pourra être effectué. Nous pensons notamment à la validation de programmes BSP plus complexes tels que le calcul scientifique [1] ou de squelettes algorithmiques [15]. Notre but est d'avoir une bibliothèque BSMLLIB certifiée (sa bibliothèque standard y compris) plus importante. Nous ensisageons aussi d'étudier la possibilité de prouver les formules de coût des algorithmes BSP (par des annotations ou des indications de coûts dans le langage source ?), l'utilisation d'autre environnements de certification de bibliothèque tels que `FoC`[8], d'ajouter des traits impératifs, en utilisant le travail déjà effectué sur les programmes séquentiels [6] et ainsi obtenir un logiciel pour la certification de programmes BSML (comme dans [6] et son logiciel WHY[9]).

Nous continuerons également notre travail sur la certification de l'environnement, en particulier, une preuve de correction de la machine abstraite parallèle [9] par rapport à la sémantique d'évaluation et une implémentation parallèle sûre (en ADA) des instructions parallèles de la machine en utilisant la méthodologie proposée par le logiciel Spark[10]. Ainsi, la bibliothèque BSMLLIB certifiée pourra être utilisée de concert avec cette machine abstraite et notre système de types pour programmer, avec un langage fonctionnel, des algorithmes BSP dans un environnement sûr et certifié.

Nos travaux futurs considèreront également des extensions de BSML comme la juxtaposition parallèle [19] qui permet de diviser le réseau en sous-réseaux et qui suit toujours le modèle BSP (qui interdit la synchronisation des sous-réseaux). Cette nouvelle construction est particulièrement intéressante pour la multiprogrammation. Elle permet d'écrire facilement des algorithmes parallèles diviser-pour-régner et est, ainsi, une étape importante vers l'utilisation de BSML pour les grilles de calculs (GRID). Par conséquent, de futurs "algorithmes GRID" pourront être développés et automatiquement extraits de manière certifiée. Ce type de travail pourrait aussi être appliqué à d'autres extensions parallèle des langages fonctionnels, notamment les langages à squelettes (comme par exemple [5]) en donnant ceux-ci comme des paramètres, leurs axiomes associés et une implémentation séquentielle équivalente pour la cohérence pour valider d'autres types d'algorithmes parallèles.

Remerciements Ce travail a été financé par le projet CARAML (ACI Grid) du Ministère de la Recherche. Merci à Hélène pour ses corrections, Frédéric Loulergue et les re-lecteurs anonymes pour leurs commentaires.

8. http://www-spi.lip6.fr/foc
9. http://why.lri.fr/
10. http://www.sparkada.com

Références

[1] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51, pages 509–514. Elsevier, 1994.

[2] E. Chailloux and C. Foisy. A portable implementation for objective caml flight. *Parallel Processing Letters*, 13(2):425–436, 2003.

[3] Th. Coquand and G. Huet. The Calculus of Constructions. *Information and Computation*, 37(2-3), 1988.

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

[5] M Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Parallel functional programming with skeletons: the OcamlP3L experiments. In *The ML Workshop*, 1998.

[6] F.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4), 2003.

[7] S. Fortune and J. Wyllie. Parallelism in random access machines. In *Proceedings of the Tenth Annual Symposium on Theory of Computing (STOC)*. ACM, May 1978.

[8] P. Fradet and J. Mallet. Compilation of a Specialized Functional Language for Massively Parallel Computers. *Journal of Functional Programming*, 10(6):561–605, 2000.

[9] F. Gava and F. Loulergue. A Parallel Virtual Machine for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *ICCS 2003, Part I*, number 2657 in LNCS, pages 155–164. Springer Verlag, june 2003.

[10] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In *PaCT 2003*, LNCS. Springer Verlag, 2003. to appear.

[11] F. Gava and F. Loulergue. Verifying Functional Bulk Synchronous Parallel Programs Using the Coq System. Technical report, University of Paris 12, LACL, 2003.

[12] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

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

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

[15] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *European Symposium on Programming*, number 2035 in LNCS, pages 83–97. Springer, 2002.

[16] H. Jifeng, Q. Miller, and L. Chen. Algebraic Laws for BSP Programming. In L. Bouge and Y. Robert, editors, *Euro-Par'96 Parallel Processing*, number 1124 in LNCS, pages 359–368, Lyon, France, August 1996. Springer.

[17] F. Loulergue. BSλ_p: Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, number 1940 in Lecture Notes in Computer Science, pages 355–363. Springer, October 2000.

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

[19] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In Harald Kosch.et. al, editor, *Euro-Par 2003*, LNCS. Springer Verlag, 2003. to appear.

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

[21] Q. Miller. BSP in a Lazy Functional Context. In S. Gilmore, editor, *Trends in Functional Programming, Volume 3*, pages 1–13. Intellect Books, 2002.

[22] C. Parent. Developing certified programs in the system coq: The program tactic. In H. Barendregt and T. Nipkow, editors, *Types for Proofs and Programs*, pages 291–312. Springer, Berlin, Heidelberg, 1993.

[23] C. Paulin-Mohring and B. Werner. Synthesis of ML programs in the system Coq. *Journal of Symbolic Computation*, 15:607–640, 1993.

[24] D. B. Skillicorn. Building BSP Progams Using the Refinement Calculus . In *Formal Methods for Parallel Programming and Applications workshop at IPPS/SPDP'98*, 1998.

[25] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3), 1997.

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

[27] A. Stewart, M. Clint, and J. Gabarro. Algebraic Rules for Reasoning about BSP Programs. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*. Nova Science Publishers, august 2002.

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

A. Code de l'extraction

Quelques fonctions utilitaires

```
let replicate a = mkpar (fun pid -> a)

let parfun f v = apply (replicate f) v

let parfun2 f v1 v2 = apply (parfun f v1) v2
```

Code de `get_list`

```
let get_list datas lpids = parfun2 (fun x x0 -> map x x0)
    (put (parfun2 (fun f d dst -> match f dst with
                                     | Some u -> Some d
                                     | None -> None)
        (put (parfun (fun l dst -> match (in_dec z_eq_dec dst l) with
                                            | true -> Some ()
                                            | false -> None) lpids)) datas)) lpids
```

Code de la réduction directe

```
let totex_tmp vec = put (parfun (fun data dst -> Some data) vec)

let totex vec = parfun (fun f src -> noSome (f src)) (totex_tmp vec)

let procs h = from_to (last ())

let total_exchange vec = parfun2 map (totex vec) (replicate (procs ()))

let tl_hd_vect v = parfun (fun l -> Pair ((hd l), (tl l))) v

let fold_left_direct op0 vec =
  parfun (fun hd_tl -> fold_left op0 (snd hd_tl) (fst hd_tl))
         (tl_hd_vect (total_exchange vec))
```

Code du rassemblement

```
let procs_lists_gather root = match within_bound root with
  | true -> mkpar (fun pid ->  match z_eq_dec root pid with
                                | true -> from_to (zminus (bsp_p ()) (POS XH))
                                | false -> Nil)
  | false -> assert false

let gather root vv = match within_bound root with
  | true -> put (apply (mkpar (fun pid v dest -> match z_eq_dec dest root with
                                                    | true -> Some v
     | false -> None)) vv)
  | false -> assert false

let gather_list vec root = match within_bound root with
  | true -> parfun2 (fun x x0 -> map x x0)
                    (parfun (fun x x0 -> (fun x1 -> noSome x1) (x x0))
                    (gather root vec)) (procs_lists_gather root)
  | false -> assert false
```

Code de la diffusion en deux phases

```
let scatter partition root v =  match within_bound root with
   | true ->  parfun noSome
                 (apply (put (apply (mkpar(fun pid->if match z_eq_dec pid root with
                                                  | true -> partition
                                                  | false -> (fun x a -> None))) v))
                        (replicate root))
   | false -> assert false

let bcast_twophases partition paste root vv =
                    match within_bound root with
                       | true -> parfun paste (totex (scatter partition root vv))
                       | false -> assert false
```