Une implantation de la juxtaposition parallèle

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

Résumé

1. Introduction

Programmation fonctionnelle BSP. Bulk Synchronous Parallel ML ou BSML [17] est une extension de ML pour la programmation fonctionnelle *data-parallel* d'algorithmes BSP (*Bulk Synchronous Parallelism*), un modèle de programmation parallèle [3, 21] qui offre à la fois un haut degré d'abstraction (tout en étant portable) et permettant la prévision réaliste de performances sur une grande variété d'architectures.

L'exécution d'un programme BSP est une séquence de *super-étapes*. Chaque super-étape est divisée en trois phases successives et logiquement disjointes : calcul locals asynchrones en chaque processeur, échanges de données entre les processeurs 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.

Les programmes BSML sont écrits à l'aide d'un petit ensemble de primitives travaillant sur des vecteurs parallèles : un constructeur de ses vecteurs, une application parallèle asynchrone, une opération de communication globale synchrone et enfin une projection globale synchrone.

Problématique. La sous-synchronisation, c'est-à-dire la synchronisation que d'une partie des processeurs de la machine, est une fonctionnalité d'une grande variété de modèles de programmation parallèle (indépendants ou dérivés du modèle BSP [6, 5]). La présence et l'utilisation de la sous-synchronisation sont justifiées par la nécessité de pouvoir décomposer récursivement un problème en sous-problèmes indépendants les uns des autres. Cette technique algorithmique est appelée «diviser-pour-régner», et dans le cas de la programmation parallèle, simplifie l'écriture de certains programmes.

Le modèle BSP n'autorise pas la sous-synchronisation : les barrières sont collectives. C'est souvent considéré comme un obstacle pour la programmation parallèle d'algorithmes diviser-pour-régner en BSP (et dans notre cas, en BSML). Dans l'article [21], les auteurs ont noté que pour de grandes applications, la perte d'efficacité et d'expressivité due à la contrainte des barrières globales, est compensée par les avantages des communications BSP (notamment l'efficacité accrue des transferts de données sur certaines architectures) qui rendent la programmation parallèle plus simple.

Pourtant, la technique dite diviser-pour-régner est une méthode naturelle pour la conception et l'implantation de nombreux algorithmes. La transformation manuelle d'un programme utilisant des techniques diviser-pour-régner en un programme sans ces techniques est un travail fastidieux où il est facile de se tromper [12]. Les auteurs de la BSPlib [14] font remarquer que le désavantage principal du partionnement par groupe de processeurs (c'est-à-dire la sous-synchronisation) est une perte de la prévision des performances du modèle de coût ([10] donne des arguments, sur des cas pratiques, en défaveur de la sous-synchronisation).

Précédent travaux. Pour exprimer aisément les algorithmes diviser-pour-régner en BSML, [16] a tout d'abord proposé une nouvelle primitive appelée «composition parallèle». Mais celle-ci était limitée à la composition de deux programmes BSML dont les évaluations nécessitaient le même nombre de super-étapes. C'était évidemment restrictif et le respect de cette contrainte échoyait au programmeur. Cette restriction a été levée avec la «juxtaposition parallèle» de [18]. Mais aucune implantation de cette primitive n'a été donnée.

Dans [24], l'auteur avance l'hypothèse que le paradigme diviser-pour-régner peut s'insérer dans le modèle BSP sans utiliser la sous-synchronisation. Il propose une méhode pour la programmation diviser-pour-régner qui est en adéquation avec le modèle des barrières de synchronisations globales du modèle BSP. Cette méthode est basée sur un entrelacement de processus légers, appelés *super-threads*[1], chacun d'entre eux étant un processus de calcul BSP.

L'article [19] présente une nouvelle primitive BSML, appelée *superposition parallèle* et qui permet d'écrire de manière fonctionnelle ce type d'algorithmes. Dans une sémantique de haut niveau, la superposition peut être simplement décrite sous la forme de la création d'une paire. Dans [7], une nouvelle sémantique de plus bas niveau, explicitant directement les super-threads, a permis une implantation de cette nouvelle primitive de composition parallèle.

Présent travail. Dans cet article, nous allons détailler comment implanter, avec la superposition parallèle, la juxtaposition parallèle (primitive permettant une décomposition récursive et parallèle plus aisée, d'un problème en sous-problèmes indépendants).

Dans un premier temps, nous introduirons (de manière plus conséquente) le modèle BSP (section 2.1), le langage BSML ((section 2.2) et la superposition parallèle (section 2.3). Puis, nous présenterons la juxtaposition parallèle (section 3) et son implantation. Enfin, nous donnerons un exemple de programme et quelques expériences préliminaires de cette implantation (section 4). Nous terminerons par quelques travaux connexes (section 5) et les travaux futurs (section 6).

2. Programmation fonctionnelle BSP

2.1. Le modèle BSP

Le modèle de programmation parallèle BSP (*Bulk Synchronous Parallelism*) [21] 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 grappes de PC ou les machines à mémoire partagée). De plus, l'unité de synchronisation est rarement physique mais plutôt logicielle.

Les performances d'un ordinateur BSP sont caractérisées par trois paramètres (exprimés en multiples de la vitesse des processeurs, dans le cas contraire un quatrième paramètre, la vitesse des processeurs est donnée) : le nombre de paires processeur-mémoire p, le temps l nécessaire à la réalisation d'une barrière de synchronisation et pour finir, le temps g pour un échange collectif

[1] Nous préférons le terme de *super-thread* et non son anglicisme «super processus léger» qui est un peu pompeux.

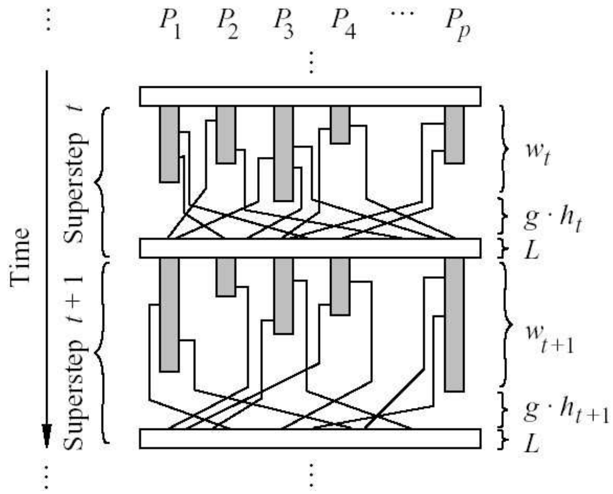

FIG. 1 – Une super-étape BSP

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 [14, 3].

L'exécution d'un programme BSP est une séquence de *super-étapes*. Chaque super-étape est divisée en trois phases successives et logiquement disjointes (figure 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 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.

2.2. La bibliothèque BSMLLIB

Les programmes BSML sont des programmes OCaml manipulant une structure de données parallèle. 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 OCaml, appelée BSMLLIB[2].

Cette bibliothèque est basée sur les primitives données par la figure 2. En premier lieu, cette bibliothèque donne accès aux paramètres BSP de l'architecture sur laquelle sont évalués les programmes BSML. La valeur de **bsp_p**() est p, le nombre statique de processeurs de la machine parallèle.

Les valeurs parallèles de largeur p contenant en chaque processeur une valeur de type α , appelées vecteurs parallèles, sont représentées par le type abstrait α **par**. L'imbrication de vecteurs parallèles est interdite. Jusqu'à présent, le programmeur était responsable de l'absence d'imbrication. Le système de types présenté dans [9] remédie à ce défaut.

[2] page web à `http://bsmllib.free.fr/`

bsp_p: unit→int **bsp_l**: unit→float **bsp_g**: unit→float
mkpar: (int→α)→α **par**
apply: (α →β)**par**→α **par**→β **par**
put: (int→α option)**par**→(int→α option)**par**
proj: α option **par**→int→α option
avec **type** α option = None | Some **of** α

FIG. 2 – Les primitives de la BSMLLIB

Les vecteurs parallèles sont créés par la primitive **mkpar** tel que (**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 $(p-1)$:

$$\textbf{mkpar}\ \mathsf{f} = \boxed{(\mathsf{f}\ 0)\ |\ (\mathsf{f}\ 1)\ |\ \cdots\ |\ (\mathsf{f}\ i)\ |\ \cdots\ |\ (\mathsf{f}\ (p-1))}$$

Les calculs asynchrones sont programmés avec les primitives **mkpar** et **apply** tels que (**apply** (**mkpar** f) (**mkpar** e)) calcule ((f i) (e i)) au processeur i :

$$\textbf{apply}\ \boxed{f_0\ |\ f_1\ |\ \cdots\ |\ f_i\ |\ \cdots\ |\ f_{p-1}}\ \boxed{v_0\ |\ v_1\ |\ \cdots\ |\ v_i\ |\ \cdots\ |\ v_{p-1}}$$
$$= \boxed{(f_0\ v_0)\ |\ (f_1\ v_1)\ |\ \cdots\ |\ (f_i\ v_i)\ |\ \cdots\ |\ (f_{p-1}\ v_{p-1})}$$

Les phases de communication et de synchronisation sont exprimées à l'aide des primitives **put** et **proj**.

Considérons l'expression suivante : **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 doit s'évaluer en None.

L'expression s'évalue en un vecteur parallèle contenant en chaque processeur une fonction fd_i des 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.

La bibliothèque contient également une primitive de projection globale synchrone appelée **proj**. L'expression (**proj** vec) calcule une fonction f telle que (f n) retourne la $n$ième valeur du vecteur vec (valeur contenue *que* par le processeur n). Si cette valeur est la valeur vide None alors le processeur n n'a rien transmis aux autres processeurs. Autrement, celle-ci est Some v, alors v est diffusée aux autres processeurs. C'est donc une primitive de multi-diffusion permettant de «sortir» des valeurs d'un vecteur parallèle, c'est-à-dire de les passer du contexte local à celui global. Ainsi, sans cette primitive, le contrôle global ne pourrait pas tenir compte des données calculées localement. Cette projection est nécessaire pour exprimer des algorithmes ayant la forme suivante :

Repeat Iteration Parallèle **Until** Max des erreurs locales $< \epsilon$.

2.3. La superposition parallèle

Le principe des super-threads est le suivant : au lieu de diviser la machine en sous-machines effectuant des sous-synchronisations, on peut diviser la puissance de la machine parallèle en des processus BSP. Les resources de la machine BSP sont partagées par les processus. Ces processus sont appelés *super-threads* car chacun de ces processus légers utilise l'ensemble des processeurs. Les communications entre les processus légers sont fusionnées et les synchronisations globales du modèle BSP sont alors partagées par les super-threads : on n'a qu'une seule synchronisation pour plusieurs super-threads effectuant des communications. Les méthodes optimisants les transferts de données BSP peuvent alors être conservées. Cette nouvelle primitive appelée *superposition* permet de cette manière l'évaluation de deux expressions BSML. La figure 3 illustre ce propos avec deux super-threads qui sont le calcul de deux expressions BSML indépendantes. Sans la superposition et les super-threads, les deux expressions sont évaluées séquentiellement. Dans le cas d'une exécution avec super-threads, les communications et les synchronisations sont fusionnées, ce qui diminue le temps d'exécution globale Notons que le calcul local, lui, n'est pas affecté. Seul l'ordre d'évaluation change.

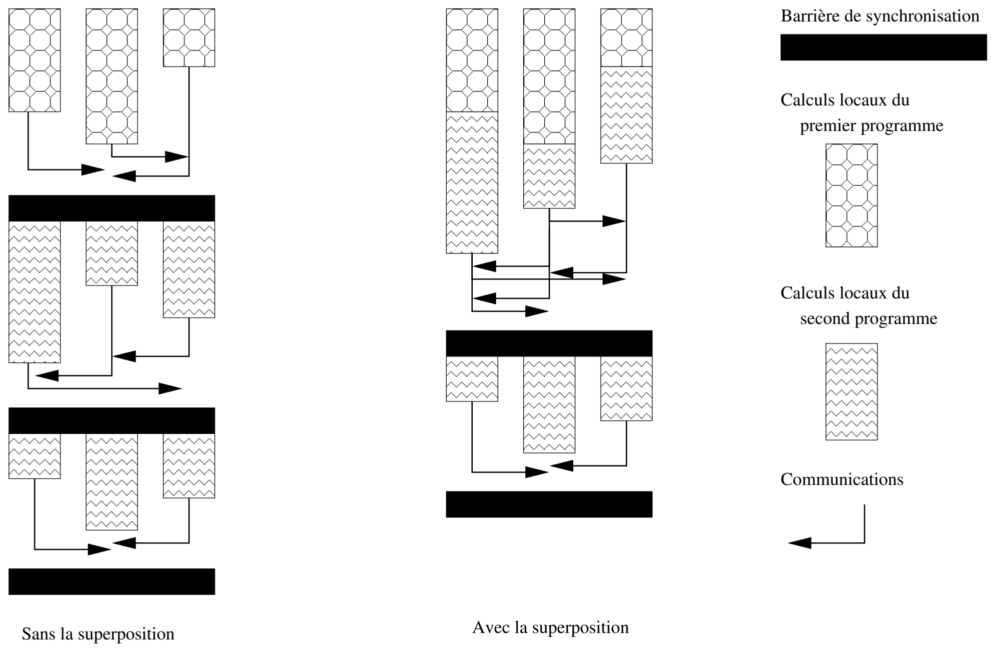

FIG. 3 – Superposition de deux programmes

De point de vue du programmeur, la sémantique parallèle de la superposition est la même que la paire. Dans la BSMLLIB, la primitive a le type suivant :

super: (unit$\to\alpha$)$\to$(unit$\to\beta$)$\to\alpha * \beta$

Bien évidemment l'évaluation de **super** E_1 E_2 est différente de celle de $((E_1\ ()), (E_2\ ()))$. Elle se déroule ainsi (voir [7] pour une sémantique formelle) :

1. La première phase de calcul asynchrone de E_1 et celle de E_2 sont exécutées ;
2. Ensuite, la première phase de communication de E_1 est mise en commun avec celle de E_2. Les messages sont la «concaténation» des messages de la première phase de communication de E_1 et de E_2 ;
3. Enfin, une seule barrière de synchronisation termine la première super-étape de E_1 *et* celle de E_2. Les super-étapes suivantes des deux *super-threads* peuvent alors être exécutées.

Si l'évaluation de E_1, par exemple, nécessite moins de super-étapes que celle de E_2 alors l'évaluation de E_2 se termine comme s'il n'y avait pas de superposition de deux *super-threads* (et vice-versa).

La superposition parallèle de E_1 et E_2 est moins coûteuse que l'évaluation de E_1 suivie de l'évaluation de E_2. Par exemple le h résultant de la superposition peut être égal au plus grand des deux et donc inférieur à la somme. C'est bien sûr toujours le cas pour le nombre de barrières de synchronisation.

L'ordre d'évaluation (la stratégie) d'expressions fonctionnelles n'a pas d'importance : la sémantique est confluente. Mais, dans la familles des langages ML, il est facile d'ajouter des fonctionnalités impératives (voir [8] pour BSML). Dans ce cas, l'ordre d'évaluation est nécessaire pour garder une sémantique déterministe (ou confluente dans le cas de BSML). La présence d'expressions impératives dans les super-threads et de données partagées forcent à l'utilisation d'une stratégie pour les super-threads : chacun des super-threads peut affecter une valeur différente à une variable. Sans une stratégie bien prédéfinie pour les super-threads (en tant que processus légers), il est impossible de prévoir lequel fera en dernier l'affectation. Le résultat de l'expression sera alors non déterministe. Pire même, évaluée

```
(* scan_super: (α →α →α )→α →α par→α par *)
let scan_super op e vec =
  let rec scan' fst lst op vec =
    if fst>=lst then vec
    else let mid = (fst+lst)/2 in
    let vec'= mix mid (super (fun()→scan' fst mid op vec)
                             (fun()→scan'(mid+1) lst op vec)) in
    let msg vec = apply (mkpar(fun i v→
      if i=mid then fun dst→if inbounds (mid+1) lst dst then Some v else None
      else fun dst→ None)) vec
    and parop = parfun2(fun x y→match x with None→y|Some v→op v y) in
    parop (apply(put(msg vec'))(mkpar(fun i→mid))) vec' in
  applyat 0 (fun _ →e) (fun x→x) (scan' 0 (bsp_p()−1) op vec)
```

FIG. 4 – Code de la «version superposition» du calcul des préfixes

avec un **proj**, nous pouvons avoir une erreur d'exécution car cet ordre sera généralement différent en chaque processeur, ce qui donnera des valeurs différentes au niveau global.

Notre exemple utilisant directement la superposition est une version diviser-pour-régner du calcul des préfixes d'un opérateur sur un vecteur parallèle de valeurs. Ce calcul peut être schématisé ainsi :

$$\text{scan} \oplus \boxed{v_0 \mid \cdots \mid v_{p-1}} = \boxed{\imath_\oplus \mid v_0 \oplus v_1 \mid \cdots \mid \oplus_{k=0}^{p-1} v_k}$$

où $\imath_\oplus$ est l'élément neutre de l'opérateur $\oplus$.

Dans cette version de calcul, le réseau est divisé en deux parties et le calcul des préfixes est récursivement appelé sur ces deux parties. La valeur calculée par le dernier processeur de la première partie est diffusée aux processeurs de la seconde partie. Ensuite, en chaque processeur de cette seconde partie,cette valeur et la valeur calculée localement sont combinées à l'aide de l'opérateur donné en paramètre op. La figure 4 donne le code de cette version du calcul des préfixes. Le code utilise les fonctions suivantes :

```
(* replicate: α →α par *)
let replicate e = mkpar (fun _ →e)
(* parfun: (α →β )→α par→β par *)
let parfun f v = apply (replicate f) v
(* applyat: int→(α →β )→(α →β )→α par→β par *)
let applyat n f1 f2 v = apply (mkpar (fun pid →if pid=n then f1 else f2)) v
(* inbounds: int→int→int→bool *)
let inbounds first last n = (n>=first)&&(n<=last)
(* mix: int→α par * α par→α par *)
let mix m (v1,v2) = let f pid v1 v2 = if pid<=m then v1 else v2 in apply(apply(mkpar f) v1) v2
```

qui teste si un numéro de processeur est compris entre le premier et dernier des processeurs de la partie courante. La seconde fonction permet de combiner les résultats de la première et de la seconde partie (sous forme de vecteur parallèles) en un seul vecteur parallèle.

3. Implantation de la juxtaposition parallèle

La juxtaposition parallèle est une primitive qui a été proposée dans [18] pour faciliter la programmation des algorithmes diviser-pour-régner en BSML : deux programmes sont évalués sur la même machine découpée en deux sous-parties disjointes mais tout en conservant une exécution BSP (figure 5). Cette section propose une implantation de cette primitive avec la superposition.

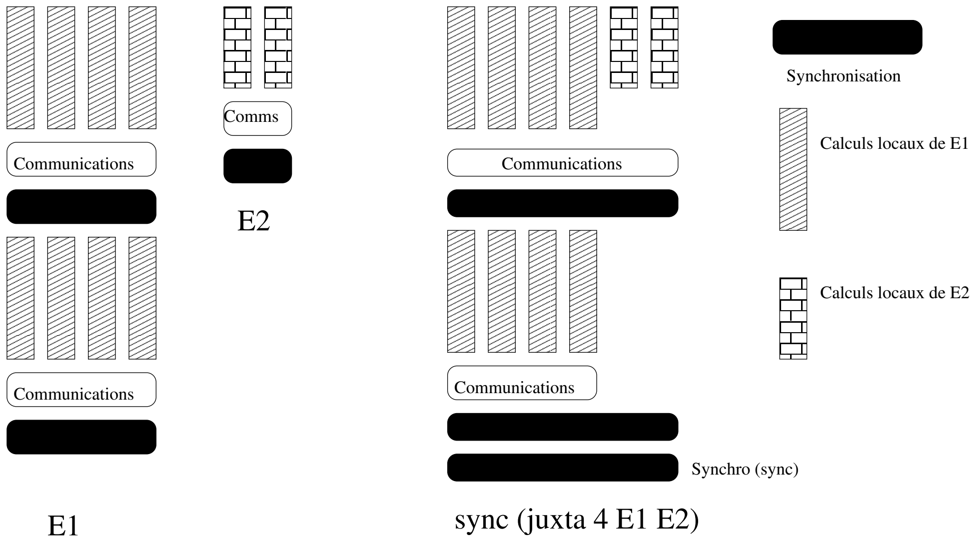

FIG. 5 – Juxtaposition de deux programmes

Pour comprendre cette implantation, nous allons d'abord décrire les modifications qu'ajoute la juxtaposition.

3.1. Présentation informelle et exemple

Pour évaluer deux programmes parallèles sur la même machine, on peut la partitionner en deux et évaluer indépendamment chacun des programmes sur chacune des partitions. Toutefois, en procédant ainsi le modèle BSP est perdu puisqu'alors la synchronisation globale de chaque sous-machine ne coûtera plus l. Pour conserver le modèle BSP, ce qui est souhaitable [10], il faut donc que les barrières de synchronisation concernent toute la machine.

Considérons le terme **juxta** m E_1 E_2. L'évaluation de ce terme se déroule comme suit : les m premiers processeurs évaluent le terme E_1 et les $p-m$ restant évaluent E_2. Ces $p-m$ processeurs sont toutefois renommés, le processeur m devenant 0 et le $(p-1)$ème devenant $(p-1)-m$. En dehors de ce renommage et du changement de la valeur **bsp_p**() sur chaque sous-machine, l'évaluation est la même que précédemment, à ceci près que l'évaluation de **put** ou de **proj** met en jeu tout le réseau au moment de la synchronisation globale. Un problème arrive toutefois si le nombre de super-étapes nécessaires à l'évaluation de E_1 et E_2 est différent. C'est pourquoi une autre primitive est nécessaire. Il s'agit de la primitive **sync**. Celle-ci appelle en boucle des barrières de synchronisation jusqu'à ce que l'un des appels concerne tout le réseau. La figure 5 illustre cette méthode.

Ainsi dans le cas où l'évaluation de E_1 nécessite une super-étape de plus que celle de E_2, l'évaluation de **sync**(**juxta** m E_1 E_2) peut être décrite ainsi :

- Au début, chaque appel de synchronisation globale pour l'évaluation de E_1 correspond à un appel de synchronisation globale pour l'évaluation de E_2 ;
- Ensuite, l'évaluation de E_2 se termine. E_1 fait une demande de synchronisation globale supplémentaire pour sa dernière super-étape. La seconde sous-machine qui a fini d'évaluer E_2 évalue alors le **sync** : c'est un appel à une synchronisation globale qui va être en correspondance avec l'appel de la première sous-machine ;
- Chaque sous-machine a fini d'évaluer son terme et les deux font une demande de synchronisation émanant d'un **sync**. Cet appel concernant tout le réseau, l'évaluation du **sync** se termine.

```
let rec scan_juxta op vec =
 if bsp_p'()=1 then vec else
  let mid = bsp_p'()/2 in
  let vec'=juxta' mid (fun ()→scan_juxta op vec) (fun ()→scan_juxta op vec) in
  let msg vec=apply'(mkpar'(fun i v→ if i<>mid−1 then (fun dst→None)
                                  else fun dst→if dst>=mid then Some v else None)) vec
  and parop=parfun2'(fun x y→match x with None→y|Some v→op v y) in
   parop (apply' (put' (msg vec')) (replicate' (mid−1))) vec'
```

FIG. 6 – Code du calcul parallèle des préfixes avec la juxtaposition

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

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

De point de vue fonctionnel, la fonction **sync** est l'identité. Au niveau de la bibliothèque BSML et sachant qu'OCaml est un langage dont la stratégie d'évaluation est une stratégie faible d'appel par valeur, il faut éviter que les deux derniers arguments de la fonction **juxta** et l'argument de la fonction **sync** ne soient évalués. Il faut qu'ils soient des fonctions :

juxta: int→(unit→α **par**)→(unit→α **par**)→α **par**
sync: (unit→α **par**)→α **par**

[18] décrit une sémantique à grands pas d'un mini-BSML avec juxtaposition parallèle. Si l'on ajoute naïvement la notion de juxtaposition parallèle à BSML, la confluence est perdue. Plusieurs raisons en sont la cause. En particulier, le nombre de processeurs, qui est aussi la taille des vecteurs, n'est plus constant et dépend du contexte dans lequel est évalué le terme. Ainsi, selon la stratégie de réduction, on obtient des résultats différents. Comme dans les chapitres précédents, une stratégie faible d'appel par valeur a été utilisée afin de conserver la confluence du langage.

La figure 6 donne le code d'une version avec juxtaposition du calcul des préfixes (les primitives sont notées avec un «'» car ce sont les primitives simulées à la figure 7). Le réseau est divisé en deux parties et la fonction scan_juxta y est appliquée récursivement. La valeur au dernier processeur de la première partie est diffusée à tous les processeurs de la seconde partie. Puis cette valeur et la valeur locale calculée par l'appel récursif sont combinées avec l'opération op sur chaque processeur de la seconde partie.

3.2. Implantation avec la superposition

Pour implanter la juxtaposition, nous pouvons utiliser la superposition (figure 7) afin de couper le réseau en deux. Chaque super-thread s'occupera d'une sous-partie du réseau. A cause de l'effet de bord sur le renommage des processeurs, nous redéfinissons les primitives BSML afin qu'elles ne fonctionnent que sur une sous-partie du vecteur parallèle. Pour cela, nous utilisons deux «variables» jux_p': unit→int **ref** et jux_f: int **ref** qui sont respectivement la taille du vecteur (nombre de processeurs dans la sous-machine) et le numéro de processeur «réel» du premier processeur de la sous-machine (quand on divise le réseau en deux, le processeur 0 de la seconde sous-machine a en réalité un autre numéro «réel»). Ce sont donc les bornes de la sous-machine.

Nous insérons une valeur nulle nc sur les processeurs qui ne prennent pas part au calcul. Ces sous-parties sont définies en utilisant les bornes (nom abstrait des processeurs de la sous-machine). La juxtaposition parallèle est, dans ce cas, l'appel à la superposition parallèle avec chaque sous-machine sur chaque super-thread. Les bornes sont alors modifiées (et préalablement sauvegardées afin d'être restaurées à la fin de l'appel de la primitive) en chaque sous-machine (deux super-threads sont alors exécutés). Notons que la valeur nulle nc n'est présente que dans les parties non-utilisées d'un vecteur.

```
let nc = Obj.magic None
let jux_p' = ref bsp_p and jux_f = ref 0
let inbound pid = (!jux_f<=pid)&&(pid<(!jux_f+(!jux_p'())))

let bsp_p' () = !jux_p'()
let mkpar' f = mkpar (fun pid→if (inbound pid) then (f (pid−(!jux_f))) else nc)
let apply' vf vv = apply2 (mkpar (fun pid f v→if (inbound pid) then (f v) else nc)) vf vv

let put' vf =
 let old_p' =(!jux_p'()) and old_f = (!jux_f) in
 let vf'=put (apply (mkpar (fun pid f→
           if (inbound pid) then (fun i→if (inbound i) then (f (i−(!jux_f))) else None)
           else (fun _→None))) vf) in
  jux_p':=(fun ()→old_p');
  jux_f:=old_f;
  apply (mkpar (fun pid f→if (inbound pid) then (fun i→(f (i+(!jux_f)))) else nc)) vf'

let proj' vv =
  let old_p' =(!jux_p'()) and old_f = (!jux_f) in
  let f=proj (apply (mkpar (fun pid v →if (inbound pid) then v else None)) vv) in
   jux_p':=(fun () →old_p');
   jux_f:=old_f;
   (fun pid →if (old_f<=pid)&&(pid<(old_f+(old_p'))) then (f pid) else None)

let juxta' m f1 f2 =
 if (0<m)&&(m<(!jux_p'())) then
  let old_p' =(!jux_p'()) and old_f = (!jux_f) in
  let (va,vb) = super (fun ()→jux_p':=(fun ()→m); f1())
                      (fun ()→jux_p':=(fun ()→old_p'−m); jux_f:=m+old_f; f2()) in
   jux_p':=(fun ()→old_p');jux_f:=old_f;
   apply2 (mkpar (fun pid a b →
      if ((!jux_f)<=pid)&&(pid<(!jux_f+m)) then a
       else if (!jux_f+m<=pid)&&(pid<(!jux_f+(!jux_p'()))) then b else nc)) va vb
 else raise (Parjux "m␣is␣not␣within␣bound")

let sync v = put (fun _ →None);v
```

FIG. 7 – Code de l'implantation de la juxtaposition

Les nouvelles primitives de communication sont implantées comme celles qui sont plates mais en redéfinissant leurs fonctions résultantes avec les nouvelles bornes de la sous-machine qui l'utilise. Á Chaque primitive de communication, on sauvegarde ces bornes puis on les restaure. Notons que la primitive **sync** est juste la fonction d'identité puisque la gestion des barrières est affectée aux super-threads. Pour respecter le modèle de coûts de la juxtaposition, une barrière de synchronisation (**put** pour des fonctions retournant toujours None) est ajoutée.

4. Exemples et expériences

Dans nos expériences, nous allons faire des comparaisons de performances entre les versions récursives du calcul des préfixes (avec superposition et avec juxtaposition) et une version directe (au fonctionnement proche d'une version séquentielle) présentée à la figure 8.

Des tests de performances ont été effectués sur une grappe de 10 nœuds ayant chacun 1Go de RAM. Les nœuds sont des Intel pentium IV 2.8 Ghz avec des cartes Gigabit Ethernet et interconnectés par

```
(* scan_direct: (α →β →α )→α →β par→α par *)
let scan_direct op e vv =
 let mkmsg pid v dst=if dst<pid then None else Some v in
 let procs_lists=mkpar(fun pid→from_to 0 pid) in
 let rcv_msgs=put(apply(mkpar mkmsg) vv) in
 let values_lists= parfun2 List.map (parfun (compose noSome) rcv_msgs) procs_lists in
  applyat 0 (fun _ →e) (List.fold_left op e) values_lists
```

FIG. 8 – Code de l'algorithme directe pour la réduction parallèle

un réseau Gigabit Ethernet (10/100/1000). Une Mandrake clic 2.0 a été utilisée comme système d'exploitation et les programmes ont été compilés avec OCaml 3.08.02 en mode *natif*. Chacun des programmes comporte 100 exécutions du calcul des préfixes. Les programmes ont été exécutés 5 fois et la moyenne des exécution a été prise pour les graphiques. Les versions de la BSMLLIB utilisant MPI [22] et TCP/IP (seules les fonctionnalités TCP/IP de OCaml ont été utilisées) ont été utilisées. L'opération employée pour le calcul des préfixes est la somme de deux polynômes. Les polynômes sont générés de manière aléatoire à chaque calcul des préfixes. Les degrés (identiques en chaque processeur) des polynômes vont croissant.

Sur la figures 9, la version MPI (graphiques de gauches) et la version TCP/IP (graphiques de droite) de la BSMLLIB ont été utilisées. Nous notons «avec superposition», la version du calcul des préfixes qui utilise la superposition et «avec juxtaposition», la version utilisant la juxtaposition simulée par la superposition.

Nous constatons que la version directe est plus efficace dans le cas de petits polynômes. Par contre, dans le cas de polynômes plus importants, les versions avec superposition et juxtaposition sont plus rapides. Ceci est normal d'après l'analyse de coût BSP de ces méthodes [3]. La version TCP/IP est dans tous les cas, meilleure que celle MPI. La version «juxtaposition» est plus rapide que la version «superposition» dans le cas de petits polynômes mais plus lente quand leurs degrés augmentent. Nous n'avons pour l'instant pas d'explication à ce phénomène.

5. Travaux connexes

Modèles pour les algorithmes «diviser-pour-régner» et la sous-synchronisation [24] présente une façon d'implanter des algorithmes diviser-pour-règner BSP dans le cadre d'un langage à objets. Il n'y a pas de sémantique formelle ni d'implantation pour l'instant. Le principe des opérations des objets est très proche de notre superposition parallèle. Le même auteur propose dans [20] une nouvelle extension du modèle BSP qui permet d'aborder facilement les algorithmes diviser-pour-régner, en ajoutant un niveau supplémentaire au modèle BSP et de nouveaux paramètres.

[25] présente un langage à patrons qui offre des patrons diviser-pour-régner. Toutefois le modèle de coût n'est plus le modèle BSP mais le modèle D-BSP [6] qui permet la synchronisation de sous-réseaux. [10] présente des arguments pour rejeter une telle possibilité.

Dans la bibliothèque BSPlib [14] la synchronisation de sous-réseaux n'est pas autorisée comme expliqué dans [21, 10]. La bibliothèque PUB [4] offre des caractéristiques supplémentaires par rapport à la proposition de standard BSPlib, suivant le modèle BSP* [2], notamment la synchronisation de sous-réseaux. [5] est un exemple d'application implantée en utilisant ces possibilités.

Méthodes sémantiques pour les algorithmes «diviser-pour-régner» Une méthodologie (et un langage dans [12]) permettant une décomposition parallèle des programmes diviser-pour-régner est présentée dans [11]. Elle utilise un modèle de calcul géométrique basé sur la transformation de coordonnées où le temps (l'ordonnanceur) et l'espace (les processeurs) sont explicites. Cette technique

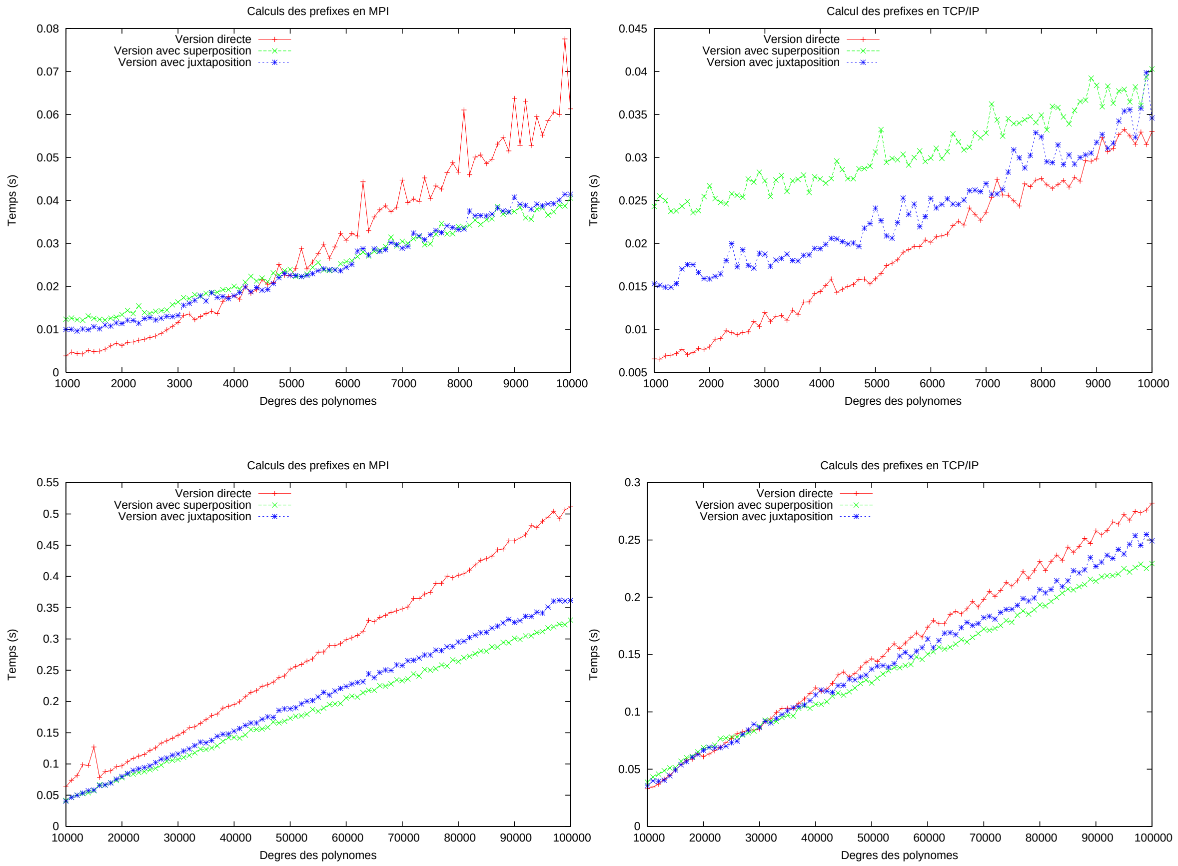

FIG. 9 – Calcul des préfixes

peut être appliquée à une classe de fonctions récursives «diviser-pour-régner» et donne des programmes fonctionnels à patrons (implantés avec MPI). Dans cet environnement à patrons, la prédiction des performances y est difficile et les optimisations algorithmiques (comme dans BSP) encore plus difficiles.

Une formulation générale *data-parallel*, pour une classe de problèmes diviser-pour-régner, a été évaluée dans [1]. Une combinaison de différentes techniques a été employée pour réorganiser les flots de données entre les processeurs, réduisant ainsi les communications et permettant une meilleur utilisation du réseau et de la mémoire. Mais ces techniques n'ont été définies que pour un langage de bas niveau qu'est *High Performance Fortran*.

Dans l'article [13], les auteurs proposent une approche distinguant trois niveaux d'abstractions et leurs instanciations :

1. Un petit langage, étendant ML et définissant les parties (statiques) parallèles d'un programme. Le langage est évalué avec un transformateur de programmes (basé sur le modèle des multi-stages de [23]) qu'est MetaOCaml. Ce langage ne doit être utilisé que par les programmeurs spécialistes en parallélisme afin de fournir des bibliothèques parallèles aux non spécialistes ;

2. Une implantation d'un patron parallèle diviser-pour-régner démontre comment la méta-programmation apportée par MetaOCaml permet de générer un ensemble approprié de communications pour un processus particulier défini à l'aide d'une spécification abstraite ;

3. Une application utilisant ce patron a été imaginée comme écrite par un non-spécialiste. Le but étant de tester si un non-spécialiste pouvait écrire du code parallèle sans avoir à considérer tous les détails de la programmation parallèle.

Le code fourni par MetaOCaml a été expérimenté sur un cluster. Malheureusement, la prédictions des coûts y est encore impossible. De plus, du code natif (permettant de bonnes performances comparé à C+MPI) ne peut être généré de manière portable car MetaOCaml a été employé (il n'existe un compilateur natif que pour les architectures Intel).

6. Conclusion et futurs travaux

La superposition parallèle est une nouvelle primitive de BSML. Elle permet l'évaluation de deux expressions par deux *super-threads* qui partagent leurs phases de communication et synchronisation. Cette primitive (purement fonctionnelle) peut être très utile pour la fusion des calculs, comme par exemple dans l'implantation de structures de données parallèles avec rebalancement automatique des données entre les processeurs [7].

La superposition permet de simuler efficacement une autre opération de multi-traitement : la juxtaposition parallèle [18]. Celle-ci permet de diviser le réseau en sous-réseaux (composition parallèle qui permet d'évaluer deux expressions chacune sur un sous-ensemble différent de la machine parallèle) 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 (le code avec la juxtaposition est clairement plus facile à comprendre que celui avec superposition). De plus le modèle de coûts BSP n'est pas perdu, ce qui n'était pas le cas de la transformation des programmes BSML avec juxtaposition en des programmes sans juxtaposition [15].

Les travaux futurs comprendront une preuve formelle de cette implantation et de la persistance des coûts BSP. En effet, [19] et [18] proposent une analyse informelle des coûts BSP des primitives de composition parallèle ici présentées. [7] donne une analyse formelle des coûts BSP pour la superposition. Il nous faudrait donc une preuve formelle que les coûts BSP de la juxtaposition simulée par la superposition soient ceux attendus (et ce quel que soit le programme).

Références

[1] M. Aumor, F. Arguello, J. Lopez, O. Plata, and L. Zapata. A data-parallel formulation for divide-and-conquer algorithms. *The Computer Journal*, 44(4) :303–320, 2001.

[2] W. Bäumker, A. Dittrich and F. Meyer auf der Heide. Truly efficient parallel algorithms : c-optimal multisearch for an extension of the BSP model. In 3^{rd} *European Symposium on Algorithms (ESA)*, pages 17–30, 1995.

[3] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

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

[5] O. Bonorden, F. Meyer auf der Heide, and R. Wanka. Composition of Efficient Nested BSP Algorithms : Minimum Spanning Tree Computation as an Instructive Example. In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2002.

[6] P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. 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.

[7] F. Gava. *Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs ; Sémantiques, implantation et certification.* PhD thesis, Université Paris XII Val-de-Marne, Décembre 2005.

[8] F. Gava and F. Loulergue. Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *Parallel Computing : Software Technology, Algorithms, Architectures and Applications, Proceeding of the 10th ParCo Conference*, Dresden, 2004. North Holland/Elsevier.

[9] F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5) :665–671, 2005.

[10] G. Hains. Subset synchronization in BSP computing. In H.R Arabnia, editor, *PDPTA'98, International Conference on Parallel and Distributed Processing Techniques and Applications*, volume 1, pages 242–246, Las Vegas, July 1998. CSREA Press.

[11] C. A. Hermann and C. Lengauer. On the space-time mapping of a class of divide-and-conquer recursions. *Parallel Processing Letter*, 6 :525–537, 1996.

[12] C. A. Hermann and C. Lengauer. Hdc : A high-order language for divide-and-conquer. *Parallel Processing Letters*, 10(2-3) :239–250, 2000.

[13] C. A. Herrmann. Functional meta-programming in the construction of parallel programs. *Parallel Processing Letters*, 2005. to appear.

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

[15] D. Louis-Régis. Certification de programmes bsml avec juxtaposition parallèle. Master's thesis, Université d'Orléans (LIFO), 2005.

[16] F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.

[17] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14*th IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.

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

[19] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part III*, number 2659 in LNCS, pages 223–232. Springer Verlag, june 2003.

[20] J. M. R. Martin and A. Tiskin. BSP modelling a two-tiered parallel architectures. In B. M. Cook, editor, *WoTUG'90*, pages 47–55, 1999.

[21] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.

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

[23] W. Taha. A gentle introduction to multi-stage programming. In C. Lengauer, D. Batory, C. Consel, and M. Odersky, editors, *Domain-Specific Program Generation*, number 3016 in LNCS, pages 30–50. Springer-Verlag, 2004.

[24] A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, 11(4) :409–422, 2001.

[25] A. Zavanella. *Skeletons and BSP : Performance Portability for Parallel Programming*. PhD thesis, Universita degli studi di Pisa, 1999.