Implantation et prédiction des performances de squelettes data-parallèles en utilisant un langage BSP de haut niveau

Frédéric Gava, Sovanna Tan

Laboratoire d'Algorithmique, Complexité et Logique (LACL)
Université Paris-Est
Créteil-Paris, France
frederic.gava@univ-paris-est.fr, sovanna.tan@univ-paris-est.fr

Résumé. *Ecrire des programmes parallèles est une tâche difficile. Souvent, les programmeurs ne veulent pas avoir à raisonner au niveau de l'envoi et de la réception des données mais pouvoir combiner des schémas de parallélisation de haut niveau définis dans des squelettes algorithmiques. Mais ces derniers ne permettent pas de représenter des schémas de communication compliqués. C'est pourquoi, disposer d'une bibliothèque de squelettes pour un langage de bas niveau plus général, nous semble un choix pertinent. Nous présentons dans cet article une implantation* BSML *(OCaml+BSP) d'un ensemble de squelettes data-parallèles avec leurs coût BSP, complétée par des mesures de performance.*

Mots-Clés : *BSP, BSML, Squelettes*

1. Introduction

1.1. *Généralités*

Dans le contexte de « Penser Parallèle ou Périr [Rei09] » le code parallèle serait la norme. Mais beaucoup de programmeurs ne sont pas capables de manipuler des routines de bas niveau [Gor04] ou des fils d'exécution [Lee06] sans introduire de bogues comme des verrous mortels et du non-déterminisme. En fait, beaucoup de programmes parallèles ne sont pas aussi peu structurés qu'ils ne paraissent et pourraient

être considérés comme des enchaînements de modèles/structures parallèle : cette idée fondamentale est utilisée dans la programmation BSP [Bis04] et dans la conception des squelettes algorithmiques [Col04]. La programmation par squelette (resp. BSP) propose que de tels modèles soient abstraits et fournis en tant que trousse à outils au programmeur. Les spécifications de la trousse à outils doivent dépasser les variations des architectures des calculateurs tandis que les implantations doivent identifier et tirer parti de ces dernières pour améliorer les performances. Mais ces squelettes sont rarement utilisés car ils n'offrent pas un jeu de structures suffisamment large pour une programmation pratique et efficace. De ce fait, la conception de nouveaux langages de programmation parallèle robustes est un domaine de recherche important. La création d'un tel langage demande un compromis entre la possibilité d'écrire des programmes efficaces dont on peut prédire les performances et l'abstraction des squelettes algorithmiques qui rendent la programmation plus facile et plus sûre.

1.2. *BSML*

Le langage Bulk-Synchronous Parallel (BSP) ML (BSML) est un compromis intéressant. C'est une extension de ML qui permet de coder des algorithmes BSP [Bis04]. Il combine le haut niveau d'abstraction de ML sans nuire aux performances (les programmes ML sont souvent aussi efficaces que les programmes écrits en C) et les propriétés des algorithmes BSP dont on peut prédire les performances pour différents degrés de parallélisme.

BSML utilise un ensemble de primitives restreint qui sont implantées en tant que bibliothèque (`http://bsmllib.free.fr`) pour le langage polyvalent Objective Caml (*i.e.* OCaml). BSML suit le paradigme BSP qui structure les calculs et les communications entre les processeurs de manière data-parallèle. En BSML, toutes les communications sont collectives (elles concernent tous les processeurs) et la stricte distinction entre les calculs locaux et globaux empêche les interblocages.

1.3. *Contribution de l'article*

On constate que beaucoup d'algorithmes parallèles peuvent être caractérisés et classés par leur adhérence à un petit nombre de schémas génériques de calcul — ferme, pipeline, *etc.* La programmation fondée sur les squelettes rend ces schémas abstraits et fournit au programmeur une trousse à outils dans lequel les spécifications s'affranchissent des variations d'architecture mais dont tient compte l'implantation pour améliorer les performances [Col04]. Le concept de programmation à base de squelettes est simple. Sa simplicité fait sa force.

L'inconvénient majeur des langages à base de squelettes est qu'ils ne permettent que les calculs parallèles définis dans les squelettes alors que de nombreuses applications parallèles ne sont pas représentables si on se limite à ceux-ci. Les langages utilisant les squelettes doivent donc permettre l'intégration des squelettes et du parallélisme *ad hoc* [Col04].

C'est pourquoi, l'ajout de squelettes en BSML permet de profiter du modèle de communication BSP et du pouvoir d'expression des squelettes. Cet article décrit l'implantation des squelettes data-parallèles. Et si cette dernière est moins efficace que celle d'un langage dédié (ou d'une implantation MPI à base de envoyer/recevoir), le programmeur peut composer les squelettes quand ça lui paraît naturel et utiliser un style de programmation BSP quand c'est nécessaire.

2. La programmation en BSP ML

2.1. *Le modèle BSP*

Un calculateur BSP est un ensemble uniforme de couples processeur-mémoire interconnectés par un réseau de communication qui permet aux processeurs d'échanger des messages [Bis04, SHM97]. Les super-calculateurs, les grappes de PCs, les machines multi-cœurs [GSF$^+$07] et les GPUs, *etc.* sont des calculateurs BSP.

L'exécution d'un programme BSP est une séquence de super étapes (voir Fig. 1, partie gauche). Chaque super étape est divisée en trois phases disjointes qui s'exécutent l'une après l'autre. Dans la première, chaque processeur utilise ses données locales pour effectuer des calculs

séquentiels et pour demander des transferts de données vers les autres processeurs. Dans la seconde phase, le réseau achemine les données. Pour finir, une barrière de synchronisation est établie afin que les données transférées soient disponibles pour la super étape suivante.

Les performances d'une machine BSP se caractérisent par quatre paramètres constants que l'on peut mesurer expérimentalement [Bis04] et qui permettent de déterminer le temps d'exécution des programmes BSP. Le premier est la vitesse du processeur $\mathbf{r}$, le second est le nombre de processeurs $\mathbf{p}$, le troisième est le temps $\mathbf{L}$ nécessaire pour réaliser une barrière de synchronisation et le dernier $\mathbf{g}$ est le temps requis pour acheminer collectivement une 1-relation, c'est à dire une phase de communication dans laquelle chaque processeur envoie et reçoit au plus un mot. On déduit ce paramètre que le temps d'acheminement d'une h-relation (chaque processeur envoie et reçoit au plus h mots) est $\mathbf{g} \times h$.

Le temps d'exécution (coût) d'une super étape est la somme des maxima des temps de calcul local, d'acheminement des données et de la barrière de synchronisation. Le coût d'un programme est la somme des coûts des super étapes. Sur la plupart des architectures distribuées de faible coût, le temps requis pour la barrière devient de plus en plus important lorsque le nombre de processeurs augmente (les barrières sont logicielles). Cependant sur les architectures dédiées, le matériel accélère grandement la réalisation des barrières. Ces dernières présentent de plus de nombreux avantages. Elles réduisent le risque d'interblocage car elles empêchent les dépendances circulaires de données. Elles permettent aussi des formes nouvelles de résistance aux pannes [SHM97].

Le modèle BSP considère les communications *en masse*. Il est moins flexible que l'échange de messages asynchrone mais plus facile à déboguer. L'envoi groupé de données améliore les performances car il diminue le temps de latence dû au réseau.

2.2. *BSML*

2.2.1. *Description générale*

BSML est une bibliothèque fondée sur le langage Objective Caml (OCaml). Ce dernier a été choisi parmi les différentes variantes de ML

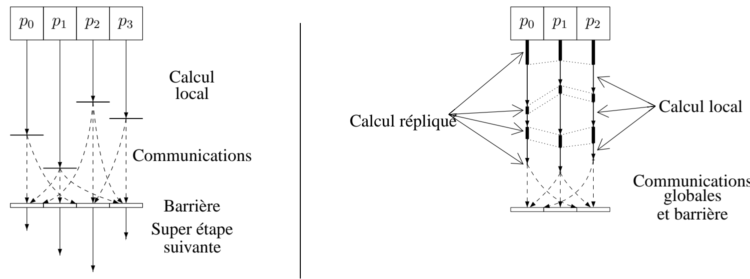


Figure 1 – Une super étape et un modèle d'exécution

pour des raisons d'efficacité. En outre, OCaml offre de nombreux outils et bibliothèques. Nous implémentons le langage BSML en entier en générant du code OCaml. Le cœur de la syntaxe de BSML est celle de OCaml – avec quelques restrictions, par exemple, on ne peut pas définir des modules locaux. Les programmes BSML se lisent comme des programmes Ocaml. En particulier, même si le programme est parallèle, l'ordre dans lequel sont exécutées les instructions est familier au programmeur habitué à OCaml. En outre, la plupart des programmes Ocaml peuvent être considérés comme des programmes BSML qui n'utilisent pas le parallélisme. Les programmes s'exécutent séquentiellement sur chaque processeur de la machine parallèle et renvoient leurs résultats normalement. Cela permet de paralléliser les programmes de manière incrémentale à partir d'un programme séquentiel. On peut trouver des exemples typiques de programmes BSML dans [Gav08, GGLD09, Ges09].

Le parallélisme nécessite peu de points d'entrée. BSML repose sur un type de donnée appelé vecteur parallèle qui, parmi tous les autres types de OCaml, permet le parallélisme. Un vecteur a pour type `'a par` et contient $\mathbf{p}$ valeurs de type `'a`, une sur chacun des $\mathbf{p}$ processeurs de la machine BSP. Le nombre de processeurs $\mathbf{p}$ est défini comme une constante **bsp_p** pendant toute l'exécution du programme. On décrit un vecteur parallèle avec la notation suivante : $\langle x_0, x_1, \ldots, x_{\mathbf{p}-1} \rangle$.

2.2.2. *Le modèle d'exécution*

Le fait que les différentes valeurs *locales* du vecteur ne se voient pas les unes les autres distingue cette structure d'un vecteur usuel de taille $\mathbf{p}$. Les deux seules manières d'accéder à la valeur locale x_i sont soit

localement sur le processeur i (en utilisant une primitive spécifique), soit après avoir reçu la valeur après une phase de communication. Ces restrictions qui sont inhérentes au calcul parallèle sur les machines à mémoire distribuée, sont ici renforcées par l'usage d'un type opaque.

Etant donné qu'un programme BSML s'applique simultanément à une machine parallèle entière et à chaque processeur individuellement, il faut distinguer les différents niveaux d'exécution (voir schéma de droite Fig. 1) :

– **La réplication** est le mode par défaut. Le code qui n'utilise pas de primitive BSML (ni par conséquent de vecteur parallèle) est exécuté par la machine parallèle comme elle le serait sur un seul processeur. Cela est utilisé pour coordonner le travail des processeurs.

– **L'exécution locale** se déroule au sein d'un vecteur sur chaque composante : le processeur utilise les données locales pour effectuer des calculs qui varient d'un processeur à l'autre.

Les exécutions répliquées et locales sont strictement disjointes et typiquement un processeur alterne entre les deux. La distinction entre ces niveaux est stricte [Ges09].

2.2.3. *Les primitives parallèles*

Les vecteurs parallèles sont traités en utilisant différentes primitives de communication qui constituent le cœur de BSML. Leur implantation repose soit sur MPI, BSPlib, TCP/IP, soit sur une implantation séquentielle. L'utilisateur peut choisir entre ces différentes implantations. Le tableau suivant résume l'utilisation des différentes primitives :

primitive	**type**	**description informelle**
$\ll$ x $\gg$	t par (if $x : t$)	$\langle x, \ldots, x \rangle$
\$pid\$	int	valeur i sur le processeur i
\$v\$	t (if $v : t$ par)	v_i sur le processeur i if $v = \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$
proj	'a par→ (int→ 'a)	$\langle x_0, \ldots, x_{\mathbf{p}-1} \rangle \mapsto (\text{fun}\ i \rightarrow x_i)$
put	(int→ 'a)par → (int→ 'a)par	$\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto$ $\langle (\text{fun}\ i \rightarrow f_i\ 0), \ldots$ $\ldots, (\text{fun}\ i \rightarrow f_i\ (\mathbf{p}-1)) \rangle$
super	(unit→ 'a)→ (unit→ 'a)→ 'a∗'b	$f_a \mapsto f_b \mapsto (f_a\ (), f_b\ ())$

Soit $\ll$ x $\gg$ le vecteur contenant x partout — sur chaque processeur. Les symboles $\ll\gg$ indiquent l'entrée dans une section de calcul local et donc le passage au niveau local. L'information répliquée est disponible à l'intérieur du vecteur. Maintenant, pour accéder à l'information locale, ajoutons la syntaxe x pour accéder au vecteur x et obtenir les valeurs locales qu'il contient. Ceci ne peut s'effectuer que dans une section de calcul local, c'est-à-dire un vecteur.

La primitive **proj** est la seule manière d'extraire une valeur locale d'un vecteur. Etant donné un vecteur, elle renvoie une fonction qui, appliquée au numéro d'un processeur, renvoie la valeur du vecteur sur ce processeur. La primitive **proj** effectue des communications de manière à ce qu'un résultat local soit disponible globalement à l'intérieur de la fonction renvoyée. Elle établit alors un point de rencontre pour tous les processeurs et donc en terme BSP termine la super étape courante.

La primitive **put** est la primitive de communication universelle. Elle permet de transférer n'importe quelle valeur locale à un autre processeur. De ce fait, elle est plus flexible que **proj** mais les valeurs restent locales. Elle est également synchrone et termine la super étape courante. Le paramètre de **put** est un vecteur qui contient pour chaque processeur, une fonction de type (int $\rightarrow$ 'a) qui renvoie quand on l'applique à i, les données à envoyer au processeur i. Le résultat de **put** est un autre vecteur de fonctions. Chaque fonction appliquée à i renvoie les données *reçues* du processeur i.

La simplicité et l'élégance du paradigme BSP a un prix : le modèle de coût BSP ne permet pas de synchroniser indépendamment un sous-ensemble de processeurs. Or la synchronisation d'un sous-ensemble de processeurs permet de décomposer les calculs récursivement en tâches indépendantes — en utilisant le paradigme « diviser pour régner ». En BSML, on implante ce type d'algorithme à l'aide de la primitive **super** qui permet l'évaluation de deux expressions BSML E_1 et E_2 en tant que super fils d'exécution. Du point de vue du programmeur, la sémantique de **super** correspond à l'appariement *i.e.*, à la construction de la paire (E_1, E_2). Mais l'évaluation de **super** $E_1\ E_2$ est moins coûteuse car elle fusionne les phases de synchronisation et de communication de E_1 et E_2.

2.3. *Exemples*

La définition de bibliothèques utiles simplifie le codage des algorithmes. Dans cette section, on définit des fonctions typiques pour la programmation BSML. La fonction suivante répartit les éléments d'une liste sur les $\mathbf{p}$ processeurs en découpant la liste en autant de sous-listes.

```
(∗ select_list:'a list → 'a list par ∗)
let select_list l = let len = List.length l
            in ≪ cut_list l ($pid$ ∗ len / bsp_p) (($pid$ + 1) ∗ len / bsp_p))≫
```

L'appel `cut_list l a b` renvoie la sous liste des éléments de `l` à partir de l'indice `a` inclus jusqu'à l'indice `b` exclu. Effectuer un map en parallèle sur une liste distribuée comme ci dessus (avec le squelette data-parallèle classique) est simple :

```
(∗ parmap : ('a→ 'b)→ 'a list par → 'b list par ∗)
let parmap f parlist = ≪ (List.map f) $parlist$≫
```

La primitive **proj** est souvent utilisée à la fin d'un calcul parallèle pour rassembler les résultats du calcul. Par exemple, convertir un vecteur parallèle en liste s'écrit :

```
(∗ proj_list : 'a par → 'a list ∗)
let proj_list v = List.map (proj v) procs_list
```

`procs_list` désigne ici la liste des numéros des processeurs $[0; 1; \ldots \mathbf{p}-1]$. Une boucle systolique doit décaler les valeurs de chaque processeur vers son voisin de droite. Cela peut s'écrire :

```
(∗ shift: 'a par → 'a par ∗)
let shift data =
 let comm = put ≪ fun dst → if dst=($pid$+1) mod p then Some $data$
                                  else None ≫ in
  ≪ noSome ($comm$ (($pid$−1) mod p)) ≫
```

Ici `noSome` efface le constructeur `Some`.

On peut généraliser et écrire un scan qui calcule $\oplus_{k=0}^{\mathbf{p}-1} v_k$ sur un processeur à partir du vecteur parallèle $\langle v_0, v_1, \ldots, v_{\mathbf{p}-1} \rangle$) :

```
(∗ scan': int→ ('a→ 'a→ 'a)→ 'a→ 'a par→ 'a par ∗)
let rec scan' step op neutral v =
 if step >= bsp_p then v else
  let comm = put ≪ fun j → if (j mod (2∗step) = 0) && ($pid$ = j + step)
                                   then $v$ else neutral ≫
   in let v' = ≪ if $pid$ mod (2∗step) = 0
                     then if $pid$ + step < bsp_p
```

```
                    then op $v$ ($comm$ ($pid$ + step))
                    else $v$
                else neutral ≫
 in scan' (step∗2) op neutral v'

(∗ scan_log: ('a→ 'a→ 'a)→ 'a→ 'a par→ 'a par ∗)
let scan_log op neutral v = scan' 1 op neutral v
```

La réduction s'effectue en nombre logarithmique de super-étapes. A chaque super-étape $step$, on combine localement les valeurs des processeurs i et $i+2^{step}$ sur le processeur i et pour $step$ variant de 0 à $\lceil \log_2 p \rceil$. Le programme (scan' 1), étant donné un opérateur associatif op ($\oplus$) et un élément neutre e regroupe donc les données pour chaque processeur dont le numéro est pair puis ensuite pour les processeurs dont les numéros sont multiples de 4, 8 *etc.* A chaque super-étape, on effectue préalablement une communication : l'argument de put renvoie donc la valeur de communication unit (c'est-à-dire rien) sauf lorsqu'on envoie du processeur $(2 \times step + 1) \times i$ au processeur $2 \times step \times i$ pour chaque i. Ensuite le calcul est effectué avec op sur les processeurs $2 \times step \times i$. A la dernière étape, la réduction complète est effectuée sur le processeur $\mathbf{p}-1$. Cette réduction a donc demandé $\log(\mathbf{p})$ super-étapes. Notons que les conditions sur $step$ permettent de ne pas avoir de phénomène de bord et que cette fonction n'est pas limitée au cas où $\mathbf{p}$ est une puissance de deux.

Cela peut ensuite s'appliquer à un vecteur de tableaux pour lequel chaque processeur contient un tableau :

```
(∗ scan_array: ('a→ 'a→ 'a)→ 'a→ 'a array par→ 'a array par ∗)
let scan_array op neutral v_arr =
 let tmp_arr = ≪ Array.scan op $va$≫ in
 let exch_v=scan_log op neutral ≪ Array.last_elt $tmp_arr$≫ in
  ≪ if $pid$<>0 then Array.map (op $exch_v$) tmp_arr else $tmp_arr$ ≫
```

Chaque processeur effectue un scan local puis le scan parallèle est effectué et les résultats sont ajoutés dans les tableaux locaux.

2.4. *Mesures des paramètres BSP*

Un des principaux avantages du modèle BSP est son modèle de coût : il est très simple et cependant assez précis. Nous avons utilisé un programme provenant de [Bis04] qui calcule les paramètres BSP de notre

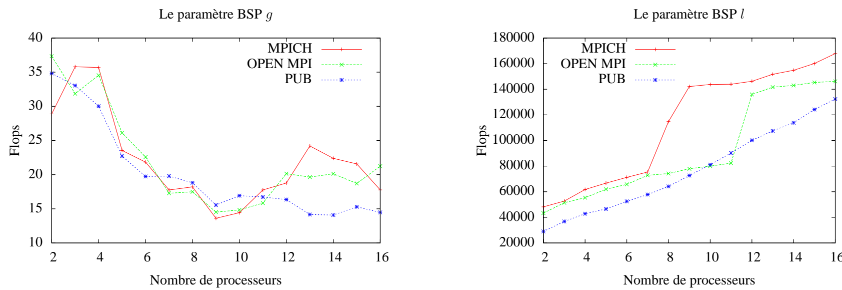


Figure 2 – Les paramètres BSP de notre machine.

machine — une grappe de 16 nœuds Pentium IV à 2.8 Ghz avec 1 GB RAM interconnectés avec un commutateur Ethernet Gigabit. Nous avons calculé ces paramètres pour trois bibliothèques : MPICH, OPEN-MPI et PUB [BJOR03].

La Fig. 2 résume les mesures de temps pour un nombre croissant de processeurs. $\mathbf{r}$ vaut 330 Mflops/s pour chaque bibliothèque. On remarque que le paramètre $\mathbf{l}$ croît de manière quasi-linéaire pour la bibliothèque PUB. Cependant pour les bibliothèques MPI deux sauts apparaissent. Ce phénomène semble être du à la gestion des synchronisations inter-processeurs : suivant les performances réseaux détectées par les bibliothèques au démarrage des calculs, différents algorithmes avec un nombre d'étapes de synchronisation proportionnel au nombre de processeurs sont utilisés. Pour le paramètre $\mathbf{g}$, il est surprenant de voir que sa valeur est élevée quand il y a peu de processeurs et qu'il se stabilise ensuite — il représente le vrai paramètre d'échange du réseau. Cela est sûrement dû à la façon dont sont gérés les tampons dans le système d'exploitation pour les protocoles de communication : quand le nombre de processeurs augmente, les tampons se remplissent plus vite et les messages sont transmis immédiatement.

3. Les squelettes data-parallèles

3.1. *Notre ensemble de squelettes data-parallèles*

Nous décrivons ici la sémantique fonctionnelle d'un ensemble de squelettes data-parallèles [Col04, Alt07]. On peut le lire comme une

implantation séquentielle naïve (qui utilise les listes de ML) ou comme des spécifications formelles.

3.1.1. *Map*

Le squelette **map** reçoit en entrée une liste de valeurs et une fonction séquentielle unaire. Il applique f à chaque élément de la liste.

$$\mathbf{map}\ f\ [x_1, \ldots, x_n] = [(f\, x_1), \ldots, (f\, x_n)]$$

Du point de vue sémantique, ce squelette est équivalent au style de programmation parallèle *Single-Program-Multiple-Data* (SPMD) où un unique programme f est appliqué à différentes données en parallèle. On obtient l'exécution parallèle en distribuant une partie de la liste des valeurs sur chaque processeur. Le squelette **mapidx** est une variante qui applique une fonction binaire g à chaque élément de la liste d'entrée en appelant g avec l'élément de la liste et son indice comme paramètres :

$$\mathbf{mapidx}\ g\ [x_1, \ldots, x_n] = [(g\, 1\, x_1), \ldots, (g\, n\, x_n)]$$

La parallélisation s'effectue comme pour **map**.

3.1.2. *Zip*

Le squelette **zip** combine les éléments de deux listes de même longueur avec un opérateur binaire. On peut le voir comme un map avec une fonction binaire.

$$\mathbf{zip}\ \oplus\ [x_1, \ldots, x_n]\ [y_1, \ldots, y_n] = [x_1 \oplus y_1, \ldots, x_n \oplus y_n]$$

La parallélisation s'effectue de la même manière que pour **map**.

3.1.3. *Réduction*

Le squelette **reduce** est tel que **reduce** $\oplus\ e\, l$ calcule la « somme » de tous les éléments d'une liste l en utilisant un opérateur binaire $\oplus$ et un élément neutre e. La réduction est une opération très prisée dans le calcul parallèle. Elle est fournie sous forme d'une opération collective `MPI_Reduce` dans le standard MPI. Formellement **reduce** se définit de la manière suivante :

$$\mathbf{reduce}\ \oplus\ e\, [x_1, \ldots, x_n] = x_1 \oplus \cdots \oplus x_n$$

Il est à noter que l'opérateur $\oplus$ peut être lui même consommateur de temps de calcul. Pour paralléliser le squelette $\mathbf{reduce}$, la liste d'entrée est divisée en sous-listes qui sont distribuées aux processeurs. Les processeurs calculent localement la « somme » de leurs éléments en parallèle. Les résultats locaux sont ensuite combinés soit sur un seul processeur en une seule super-étape, soit en utilisant un schéma logarithmique de calculs et de communications en utilisant l'associativité de l'opérateur binaire.

3.1.4. *Scan*

Le squelette $scan$ est similaire à $\mathbf{reduce}$ (et est fourni en tant qu'opération collective `MPI_Scan` dans MPI). Il utilise aussi un opérateur binaire associatif $\oplus$ et un élément neutre e pour combiner les éléments d'une liste d'entrées l. Mais au lieu de calculer simplement la « somme » comme dans $\mathbf{reduce}$, $\mathbf{scan}$ calcule les sommes partielles (préfixes) de tous les éléments de la liste :

$$\mathbf{scan}\ \oplus\ e\,[x_1, \ldots, x_n] = [x_1, (x_1 \oplus x_2), \cdots, ((x_1 \oplus x_2) \cdots \oplus x_n)]$$

Une implantation parallèle efficace commence par calculer le $\mathbf{scan}$ localement sur la partie de la liste qui a été distribuée au processeur. Les résultats de la réduction (c'est-à-dire, le dernier élément du résultat du calcul local de $\mathbf{scan}$) sont ensuite échangés entre les processeurs comme pour $\mathbf{reduce}$. Cela permet de calculer un ensemble de sommes partielles globales. Dans la dernière étape, ces sommes partielles sont ajoutées aux sommes partielles locales.

3.1.5. *Homomorphisme distribuable*

Les squelettes $\mathbf{reduce}$ et $\mathbf{scan}$ sont des squelettes de base qui sont contenus dans le standard MPI en tant qu'opérations collectives. On présente ici un squelette data-parallèle plus complexe, le squelette homomorphisme distribuable, noté $\mathbf{dh}$ et présenté dans [Alt07]. Il est utilisé pour représenter une classe spécifique d'algorithmes de type diviser pour régner. $dh\ \oplus\ \otimes\, l$ transforme une liste $l = [x_1, \cdots, x_n]$ de taille $n = 2^m$ en une liste résultat $r = [y_1, \cdots, y_n]$ de même longueur dont les éléments sont calculés récursivement de la manière suivante :

$$y_i = \begin{cases} u_i \oplus v_i & \text{si } i \leq \frac{n}{2} \\ u_{i-\frac{n}{2}} \otimes v_{i-\frac{n}{2}} & \text{sinon} \end{cases}$$

où $u = \mathbf{dh} \oplus \otimes [x_1, \ldots, x_{\frac{n}{2}}]$, *i.e.* $\mathbf{dh}$ appliqué à la moitié gauche de la liste d'entrée et $v = \mathbf{dh} \oplus \otimes [x_{\frac{n}{2}+1}, \ldots, x_n]$, *i.e.* $\mathbf{dh}$ appliqué à la moitié droite de l. Le squelette $\mathbf{dh}$ fournit le schéma d'échange de données papillon qui a de nombreuses utilisations comme la résolution d'un système tri-diagonal ou la transformée de Fourier rapide.

Par exemple :

```
# dh (^) (fun x y → y^x) ["a";"b"];;
− : string list = ["ab"; "ba"]
# dh (^) (fun x y → y^x) ["a";"b";"c";"d"];;
− : string list = ["abcd"; "badc"; "cdab"; "dcba"]
```

3.1.6. *Créer des données*

Le squelette $\mathbf{repl}$ crée une nouvelle liste contenant n fois l'élément x :

$$\mathbf{repl}\, x\, n = [x, \ldots, x]$$

Ici, on utilise le mot liste comme spécification, mais les implantations parallèles utilisent des structures de données plus efficaces comme des tableaux (dans cet article) ou des flots (dans des environnements client/-serveur ou de grille) étant donné que la taille des listes reste constante.

4. Implantation en BSML

4.1. *Calculs asynchrones*

Les squelettes manipulent des listes qui sont en fait répartis sur les processeurs. Nous appelons ces listes parallèles, des « flots ». Notez que les flots sont persistants (aucun squelette ne modifie en place un flot). Cela nous autorise d'implanter un flot sous la forme d'un couple comprenant tout d'abord un vecteur parallèle de tableaux polymorphes (un tableau par processeur) puis la taille de ce flot :

```
type 'a flow = 'a array par ∗ int

(∗ repl : 'a → int → 'a flow ∗)
let repl a n = let p=bsp_p in
 (≪ if pid<n mod p
      then Array.create (n/p+1) a
     else Array.create (n/p) a≫ ,n)
```

Ainsi, le squelette $\mathbf{repl}$ créé $\mathbf{p}$ tableaux de même longueur (modulo 1). Avec cette distribution des éléments, la plupart des squelettes sont faciles à implanter :

```
(∗ map : ('a → 'b) → 'a flow → 'b flow ∗)
let map f (fl,n) = (≪ (Array.map f) $fl$≫ ,n)

(∗ mapidx : (int → 'a → 'b) → 'a flow → 'b flow ∗)
let mapidx f (fl,n) = (≪ (Array.mapi f) $fl$≫ ,n)

(∗ zip : ('a → 'b → 'c) → 'a flow → 'b flow → 'c flow ∗)
let zip op (fl1,n1) (fl2,n2) = (≪ (Array.zip n1 op) $fl1$ $fl2$≫ ,n1)
```

Dans les codes présentés ci-dessus, nous ne montrons pas les cas d'utilisation dégénérés des squelettes, comme par exemple des flots de tailles différentes. Des exceptions (que nous ne présentons pas ici) sont utilisées dans ces cas. Les coûts BSP pour ces squelettes sont tout simplement le temps maximal d'exécution locale de l'application des fonctions.

Le squelette $\mathbf{scan}$ est naturellement l'appel à la réduction pour les tableaux présenté à la section 2 :

```
(∗ scan : ('a → 'a → 'a) → 'a → 'a flow → 'a flow ∗)
let scan oplus e (fl,n) = (scan_array oplus e fl, n)
```

Comme nous utilisons une réduction logarithmique classique (Section 2), si nous supposons une liste de taille n, une taille constante s pour les données du flot, un temps d'exécution constant $c_\oplus$ pour l'opérateur $\oplus$ alors le coût BSP est $\frac{n}{\mathbf{p}} \times c_\oplus + \log(\mathbf{p})(2 \times s \times \mathbf{g} + \mathbf{L})$. C'est-à-dire la somme des temps pour effectuer les réductions locales de tableaux et pour calculer la réduction parallèle où au plus deux éléments sont envoyés ou reçus par chaque processeur.

L'implantation de $\mathbf{reduce}$ est similaire.

4.2. *Calculer le papillon*

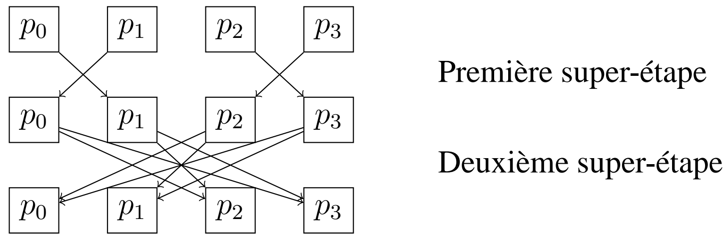
L'implantation du squelette $\mathbf{dh}$ est la plus difficile. Dans [Alt07], l'auteur propose une implantation pour une machine parallèle à mémoire partagée, en rassemblant préalablement les éléments d'unités de calcul sur un serveur et en utilisant des fils d'exécution sur ce serveur. Sans perte de généralité et afin de simplifier la présentation, nous avons

supposé que $\mathbf{p} = 2^q$ (autrement, des arrondis sur la taille des données à échanger sont utilisés). Le code suivant donne une implantation distribuée en BSML de $\mathbf{dh}$:

```
(∗ super_mix: 'a par∗'a par → 'a par ∗)
let super_mix m (v1,v2) = ≪ if $pid$<=m then $v1$ else $v2$≫

(∗ dh : ('a → 'a → 'a) → ('a → 'a → 'a) → 'a flow → 'a flow ∗)
let dh oplus omult (fl,len) =
 let rec tmp n1 n2 n vec =
  if n=1 then vec else
   let n'=n/2 in
   let n1'=n1+n' and n2'=n1+n'−1 in
   let vec'= super_mix (n1'−1) (super (fun () → tmp n1 n2' n' vec)
                                      (fun () → tmp n1' n2 n' vec)) in
   let send= put ≪ if pid<n1' then (fun dst → if dst=($pid$+n') then Some $vec'$ else None)
                   else (fun dst → if dst=($pid$−n') then Some $vec'$ else None))
 in ≪ if $pid$<n1' then match ($send$ ($pid$+n')) with
                                    Some b → Array.map2 oplus $vec'$ b
                                  | None → $vec'$
                               else match ($send$ ($pid$−n')) with
                                    Some b → Array.map2 omult b $vec'$
                                  | None → $vec'$≫
 in (tmp 0 (bsp_p−1) bsp_p ≪ local_dh oplus omult (Array.copy fl)≫ ,len)
```

Tout d'abord, chaque processeur calcule localement un « papillon » partiel avec ses propres données. C'est le rôle de la fonction local_dh. Ensuite, nous avons utilisé une fonction récursive qui calcule le papillon complet par la fusion des éléments en un nombre logarithmique de super-étapes. Si nous imaginons les processeurs comme un tableau, à chaque super-étape s, les processeurs sont séparés en $\mathbf{p} - 2^s$ parties où chacun envoie ses éléments à tous les autres de la partie qui suit (dans l'ordre des parties) comme schématisé ici pour quatre processeurs :



Si nous supposons que l'opérateur $\oplus$ (resp. $\otimes$) a un coût d'exécution constant noté $c_{\oplus}$ (resp. $c_{\otimes}$) et que la taille du flot initial est de $2^m = \mathbf{p} \times 2^n$ (*i.e.* $m = n + q$), chacun des éléments ayant une taille constante

s, alors nous pouvons déduire le coût BSP suivant : $\log(\mathbf{p}) \times (2^n \times s \times \mathbf{g} + \mathbf{L} + n \times 2^n(c_\oplus + c_\otimes))$

5. Tests de performances d'applications numériques

5.1. *Exemples classiques de calculs numériques*

5.1.1. *Résolution d'un système tri-diagonal*

Comme premier exemple, nous considérons la résolution d'un système d'équations tri-diagonal [Alt07] : $A.x = b$ où A est une matrice creuse de taille $n \times n$ qui représente les coefficients, x un vecteur d'inconnues et b un vecteur. Les seules valeurs de la matrice A non nulles se trouvent sur la diagonale, ainsi que directement au-dessus et au-dessous. Nous appelons respectivement ces dernières diagonales supérieure et inférieure.

Nous représentons les systèmes tri-diagonaux avec une liste de lignes. Chaque ligne est elle-même constituée de quatre valeurs (a_1, a_2, a_3, a_4) : la valeur a_1 appartient à la diagonale inférieure de la matrice A, la valeur a_2 appartient à la diagonale principale, la valeur a_3 appartient à la diagonale supérieure, et la valeur a_4 fait partie du vecteur b. La première et la dernière ligne de A ne contiennent que deux valeurs, mais dans le but d'obtenir une représentation cohérente nous avons mis $a_1 = 0$ pour le premier et $a_3 = 0$ pour la dernière. Cela correspond à l'ajout d'une colonne de zéros à gauche et à droite de la matrice A.

Nous utilisons maintenant le squelette $\mathbf{dh}$ pour paralléliser ce problème avec une technique « diviser pour régner ». Dans la phase dite de « conquête », deux sous-systèmes peuvent être combinés à l'aide de la première et la dernière ligne du système. Notre implantation travaille sur des triplets (a, f, l) de lignes, contenant chacun la « ligne effective » a, la première ligne f et la dernière ligne l du sous-système. En utilisant cette représentation sous forme de listes, l'algorithme peut être exprimé comme suit : $tds\, m = \mathbf{map}\, \pi_1\, (\mathbf{dh}\ \oplus\ \otimes\, (\mathbf{map}\, triple\, m))$ avec :

$$\begin{pmatrix} a_1 \\ f_1 \\ l_1 \end{pmatrix} \oplus \begin{pmatrix} a_2 \\ f_2 \\ l_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} a_1 \diamond (l_1 \star f_2) \\ f_1 \diamond (l_1 \star f_2) \\ (l_1 \circ f_2) \bullet l_2 \end{pmatrix}$$

$$\begin{pmatrix} a_1 \\ f_1 \\ l_1 \end{pmatrix} \otimes \begin{pmatrix} a_2 \\ f_2 \\ l_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} (l_1 \circ f_2) \bullet a_2 \\ (f_1 \diamond (l_1 \star f_2) \\ (l_1 \circ f_2) \bullet l_2 \end{pmatrix}$$

où si $a = (a_1, a_2, a_3, a_4)$ et $b = (b_1, b_2, b_3, b_4)$:

$$\begin{array}{rcl} a \star b & = & (a_1, a_3 - (\frac{a_2}{b_1}) \times b_2, b_3 \times (-\frac{a_2}{b_1}), a_4 - (\frac{a_2}{b_1}) \times b_4) \\ a \diamond b & = & (a_1 - (\frac{a_3}{b_2}) \times b_1, a_2, (-\frac{a_3}{b_2}) \times b_3, a_4 - (\frac{a_3}{b_2}) \times b_4) \\ a \circ b & = & (a_1, a_2 - (b_1 \times \frac{a_3}{b_2}), (-b_3 \times a_3/b_2), a_4 - (\frac{a_3}{b_2}) \times b_4) \\ a \bullet b & = & (a_1, -(\frac{a_2}{b_1}) \times b_2, a_3 - (b_3 \times \frac{a_2}{b_1}), a_4 - \frac{b_4 \times a_2}{b_1}) \end{array}$$

$\star, \diamond, \circ$ et $\bullet$ sont des opérations classiques dans un algorithme utilisant une élimination gaussienne.

La méthode utilisant dh fonctionne comme suit. Dans la phase dite de « division », la matrice est subdivisée en lignes simples. La phase de « conquête » commence en combinant les lignes voisines, en appliquant $\star$ (resp. $\circ$) puis $\diamond$ (resp. $\bullet$), ce qui produit des systèmes de deux équations chacun, avec des éléments non nuls dans la première colonne, la diagonale principale et la dernière colonne. Les sous-matrices sont alors combinées dans des matrices de quatre lignes avec la même structure, c'est-à-dire où tous les éléments non nuls se trouvent sur la diagonale, dans la première ou dans la dernière colonne. Ce processus se poursuit jusqu'à ce que, finalement, l'ensemble du système d'équations ait cette forme. Notez que la première et la dernière colonne de la matrice restent à zéro tout au long du processus, ce qui garantit la solution pour le système d'équations initial.

La combinaison de deux sous-systèmes est réalisée en utilisant une ligne spéciale, obtenue à partir de la dernière ligne l du premier système et de la première ligne f du second, à l'aide de l'opérateur $\star$ (resp. $\circ$). Cette règle est appliquée en utilisant l'opérateur $\diamond$. (resp. $\bullet$) à chaque ligne du premier système. De même, les lignes du deuxième sous-système sont ajustées à l'aide d'une autre ligne spéciale obtenue par les premières et dernières lignes.

Le problème du système tri-diagonal est principalement implanté à l'aide du squelette $\mathbf{dh}$. La complexité séquentielle de l'opérateur $\oplus$ (resp. $\otimes$) est une constante (relativement élevée) notée $c_\oplus$ (resp. $c_\otimes$). Si la matrice initiale A contient $\mathbf{p} \times 2^n$ données où chaque donnée a une taille constante s, on déduit le coût BSP suivant : $\log(\mathbf{p}) \times (2^n \times s \times \mathbf{g} + \mathbf{L} + n \times 2^n(c_\oplus + c_\otimes))$.

5.1.2. *La transformée de Fourier rapide*

Le problème de la transformée de Fourier rapide notée FFT pour Fast Fourier Transform, consiste, pour une liste $x = [x_0, \ldots, x_{n-1}]$ de taille $n = 2^m$ à calculer une liste dont le i-ème élément est défini ainsi :

$$(FFT\,x)_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$$

où ω_n désigne la n-ème racine de l'unité $e^{2\pi\sqrt{-1}/n}$.

Le problème de la FFT peut être exprimé sous la forme d'un problème « diviser pour régner » de la manière suivante :

$$(FFT\,x)_i = \begin{cases} (FFT\,u)_i \oplus_{i,n} (FFT\,v)_i & \text{si } i < \frac{n}{2} \\ (FFT\,u)_j \otimes_{j,n} (FFT\,v)_j & \text{sinon} \end{cases}$$

où $u = [x_0, x_2, \ldots, x_{n-2}]$, $v = [x_1, x_3, \ldots, x_{n-1}]$, $j = i - \frac{n}{2}$, $a \oplus_{i,n} b = a + \omega_n^i b$ et $a \otimes_{j,n} b = a - \omega_n^j b$. Cette formulation est très proche du squelette $\mathbf{dh}$ excepté $\oplus_{i,n}$ et $\otimes_{j,n}$ qui sont paramétrés par i and n. Ces opérateurs calculent itérativement les racines. Au lieu de les recalculer à chaque appel, les racines peuvent être calculées une bonne fois pour toute et contenues dans une liste $\Omega = [\omega_n^1, \ldots, \omega_n^{\frac{n}{2}}]$ accessible par chaque opérateur. Pour cela, nous utilisons un $\mathbf{scan}$ et le problème de la FFT peut être exprimé ainsi :

$$\begin{array}{rcl} (FFT\,l) & = & \mathbf{let}\,\Omega = \mathbf{scan}\,+\,1\,(\mathbf{repl}\,(\omega\,n)\,\frac{n}{2}) \\ & & \quad \mathbf{in\ map}\,\pi_1\,(\mathbf{dh} \oplus \otimes\,(\mathbf{mapidx}\,\mathit{triple}\,l)) \end{array}$$

où $\begin{pmatrix} x_1 \\ i_1 \\ n_1 \end{pmatrix} \oplus \begin{pmatrix} x_2 \\ i_2 \\ n_2 \\ t_2 \end{pmatrix} = \begin{pmatrix} x_1 \oplus_{n_1}^{i_1} x_2 \\ i_1 \\ 2n_1 \end{pmatrix}$ et où $\otimes$ est défini de la même manière.

Le premier élément de chaque triplet contient la valeur d'entrée, le second, sa position et le dernier, la taille de la liste courante. A chaque étape du calcul de $\mathbf{dh}$, ces opérateurs sont appliqués aux éléments de deux listes de taille $n_1 = n_2$, ce qui produit une liste de taille $2n_1$. Le terme $x_1 \oplus_{n_1}^{i_1} x_2$ (resp. $\otimes$) est défini par $x_1 + x_2 \times (th\,(n \times \frac{i_1}{n_2})\,\Omega)$ où $th\,n\,[x_1, \ldots, x_n, \ldots] = x_n$.

Le calcul de la transformée de Fourier rapide est principalement implanté à l'aide des squelettes $\mathbf{reduce}$ et $\mathbf{dh}$. La complexité des opérateurs $\oplus$ (resp. $\otimes$) est constante et notée $c_\oplus$ (resp. $c_\otimes$). Notons c_ω^n la complexité linaire pour calculer ω^n. Si la liste initiale contient $\mathbf{p} \times 2^n$ nombres complexes, (chacun ayant une taille constante s), nous déduisons le coût BSP suivant qui correspond à la somme des coûts de chacun des squelettes : $\log(\mathbf{p}) \times (s \times \mathbf{g} + \mathbf{L} + c_\omega^n + 2^n \times s \times \mathbf{g} + \mathbf{L} + n \times 2^n (c_\oplus + c_\otimes))$.

5.2. *Performances*

Dans la Fig. 5.2 (resp. Fig. 4), nous donnons les prédictions des performances et celles réellement mesurées (en utilisant une implantation PUB de BSML) du problème de résolution de système tri-diagonal, (resp. du calcul de la transformée de Fourier rapide) en utilisant notre implantation des squelettes. Dans chacune des figures, nous donnons le temps d'exécution mesuré pour une implantation non-optimisée (algorithmiquement parlant) et séquentielle des problèmes ainsi que pour les programmes à squelettes avec $\mathbf{p}$ variant de 2 à 16. Les temps estimés l'ont été en insérant les paramètres BSP mesurés $\mathbf{g}$ et $\mathbf{L}$ dans les formules de coûts BSP présentés dans la section précédente. La difficulté, ici, ne fut pas de mesurer la taille des données (ici clairement constantes et trivialement mesurable) mais les temps d'exécutions (théoriquement constants) des opérateurs. Nous avons ici pris une moyenne des temps d'exécution pour de grand nombres d'exécutions des opérateurs et pour des données aléatoires[1]. Nous donnons également l'accélération pour une valeur de n particulière.

1. Nous nous rendons bien compte que cette méthode est très empirique et repose uniquement sur la relative stabilité des temps d'exécution des opérations de calculs numériques en OCaml. Pour des opérateurs plus abstraits, une méthode plus systématique devra être mise en oeuvre et peut être devra-t-elle tenir compte du ramasse-miettes de OCaml.

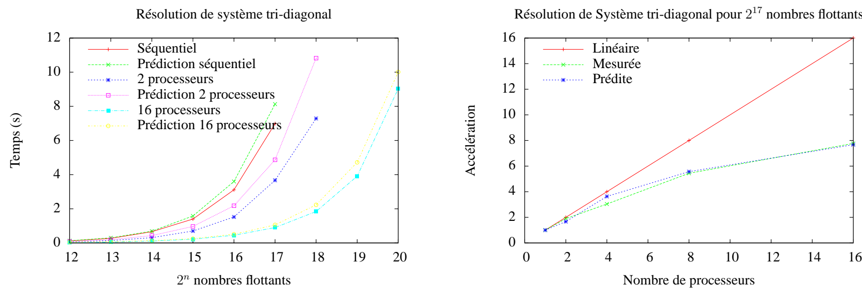


Figure 3 – Tests de performances du programme à squelette de résolution de système tri-diagonal

Pour les deux problèmes, les temps mesurés et prédits sont relativement proches. Cela vient du fait que nos programmes ne font essentiellement que du calcul numérique et que OCaml est connu pour avoir des temps d'exécution très stables si le ramasse-miettes n'est pas trop sollicité : nous avons seulement utilisé des tableaux de nombres flottants et de nombres complexes, et donc aucune structure de données complexe. Mais cela est caché aux programmeurs qui ne voient que des compositions de haut niveau de squelettes.

L'accélération est très bonne pour le calcul de la transformée de Fourier rapide. Cela n'est pas une surprise, puisqu'il s'agit d'un problème connu pour se paralléliser efficacement. Pour la résolution de système tri-diagonal, le surcoût des communications/synchronisations réduit classiquement l'accélération lorsque nous augmentons le nombre de processeurs. Mais les deux problèmes ont tout de même une bonne accélération, comme le prédit le coût BSP. Cela est très encourageant, car les programmes ML ont la réputation absurde, dans le monde du calcul haute performance, d'être lents et inefficaces.

Notez que nous n'avons pas effectué de comparaison entre ces programmes et des programmes écrits en C à la main, ces derniers seraient bien évidemment plus efficaces mais au prix d'une source plus grande d'erreurs.

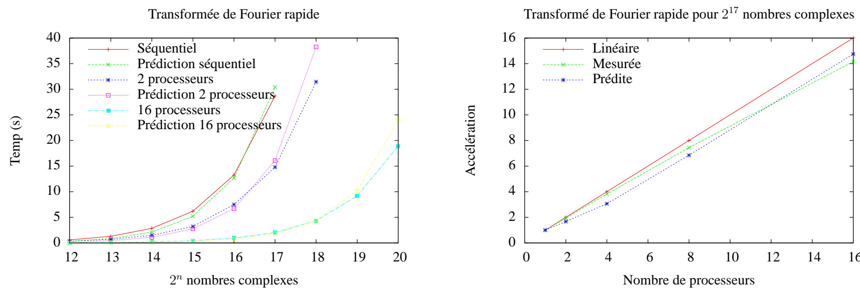


Figure 4 – Tests de performances du programme à squelette de calcul de transformée de Fourier rapide

6. Travaux connexes

Dans [GG09], les auteurs présentent l'implantation d'un ensemble de squelettes de flot de données en BSML. [Col04] décrit comment ajouter des squelettes dans MPI ainsi que certaines expériences effectuées — c'est la bibliothèque Eskel. Il donne aussi des arguments convaincants et pragmatiques pour une programmation en C qui mixerait squelettes algorithmiques et passage de messages même si ces deux approches sont souvent en dichotomie. Nous pensons que l'utilisation de OCaml pour la programmation parallèle (applications de haute performance) n'est pas un mauvais choix car le code généré est souvent très compétitif par rapport à ceux provenant de programmes C avec les avantages bien connus au JFLA de la programmation de haut niveau (OCaml). Quelques tests de performances sur une implantation en OCaml de squelettes de flot de données pour un problème numérique sont décrits dans [CMV+06] mais cette implantation des squelettes est à notre connaissance toujours uniquement à base de connexion TCP/IP alors que BSML est implanté dans différentes bibliothèques de passages de messages.

L'implantation en BSP de squelettes data-parallèles n'est pas une nouveauté. Des premiers travaux en C sont décrits dans [Zav01] et la prévision des coûts BSP a également été faite dans [HC02]. Dans les deux travaux, une bibliothèque C a été utilisée mais moins de squelettes ont été implantés que dans notre travail. Aussi, en utilisant OCaml, notre travail d'implantation nous semble-t-il plus aisé et plus sûr.

[SF08] décrit l'implantation d'un ensemble de squelettes de flots de données en utilisant OCaml +MPI. Il faut noter l'existence de OCaml-Float [2] qui est une interface OCaml pour les bibliothèques LAPACK et Blas, et qui vise à améliorer la clarté et l'efficacité des algorithmes numériques.

7. Conclusion

Dans cet article, nous avons implanté quelques squelettes de données parallèles en utilisant une extension de OCaml pour la programmation BSP, et effectué quelques tests de performances. Le modèle BSP est intéressant car il dispose d'un modèle de coût pour une estimation du temps d'exécution de ses programmes. Nous avons donc donné le coût BSP de chacun des squelettes et nous avons ensuite comparé les performances prédites avec celles mesurées de quelques applications numériques programmées avec notre série de squelettes. Nous avons constaté de bonnes performances et de bonnes prédictions. En fait, de nombreux algorithmes scientifiques (calculs numériques, tels que ceux pour les matrices) n'ont pas une complexité trop difficile à évaluer (souvent un nombre polynomial ou logarithmique de super-étapes). Dans le cas de « cloud computing » [AFGa09], on pourrait imaginer un serveur de répartition de charge qui distribue les programmes parallèles en fonction du coût afin d'optimiser les temps d'exécution et la consommation d'énergie.

Les travaux futurs seront principalement la vérification formelle des ces implantations et leur adaptation à d'autres squelettes (ainsi que leur fusion avec les squelettes de flot de données). Nous pensons également à un outil permettant de formellement prouver que les combinaisons de squelettes donnent bien les résultats escomptés (comme pour la preuve formelle de programmes BSML [Gav03] ou avec une autre méthode). En outre, il est théoriquement possible d'imbriquer les squelettes[3], (d' imbriquer du code parallèle dans du code parallèle) par exemple remplacer le f du map par n'importe quelle fonction ou squelette. Actuellement, notre implantation l'interdit car la fonction f du map est appliquée lo-

2. `http://www.mirrorsky.com/ocaml/`
3. `http://en.wikipedia.org/wiki/Algorithmic_skeleton`

calement en chaque processeur sur les éléments distribués. Cette fonction f est locale (c'est-à-dire dans un vecteur parallèle) et BSML interdit pour l'instant à f de contenir du code parallèle, elle ne doit être qu'une simple fonction ML séquentielle.

Aplatir ce parallélisme imbriqué est théoriquement possible, mais aucun outil formel fourni ne le fait. Enfin, la prédiction de coût est pour l'instant (dans cet article), faite de manière complètement empirique et par jeu de tests (à la main) successifs, afin de déduire au mieux les temps d'exécution. Il faudrait automatiser ce travail mais la plus grande des difficultés provient non pas du parallélisme mais de l'estimation des temps séquentiels. Concevoir et implanter tous ces outils formels nous semble être un grand défi.

Références

[AFGa09] M. Armbrust, A. Fox, R. Griffith, and al. Above the clouds : A berkeley view of cloud computing. Technical Report UCB/EECS-2009-28, EECS Department, University of California, Berkeley, 2009.

[Alt07] M. Alt. *Using Algorithmic Skeletons for Efficient Grid Computing with Predictable Performance*. PhD thesis, Universität Münster, 2007.

[Bis04] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

[BJOR03] O. Bonorden, B. Juurlink, I. Von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2) :187–207, 2003.

[CMV+06] F. Clément, V. Martin, A. Vodicka, R. Di Cosmo, and P. Weis. Domain Decomposition and Skeleton Programming with OCamlP3l. *Parallel Computing*, 32 :539–550, 2006.

[Col04] M. Cole. Bringing Skeletons out of the Closet : A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3) :389–406, 2004.

[Gav03] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3) :365–376, 2003.

[Gav08] F. Gava. A modular implementation of data structures in bulk-synchronous parallel ml. *Parallel Processing Letters*, 18(1) :39–53, 2008.

[Ges09] L. Gesbert. *Développement systématique et sureté d'exécution en programmation paralléle structurée*. PhD thesis, University of Paris-East, 2009.

[GG09] F. Gava and I. Garnier. New Implementation of a BSP Composition Primitive with Application to the Implementation of Algorithmic Skeletons. In *Workshop APDCM, part of IPDPS'2009*. IEEE Computer Society, 2009.

[GGLD09] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski. Bulk Synchronous Parallel ML with Exceptions. *Future Generation Computer Systems*, 2009. to appear.

[Gor04] Sergei Gorlatch. Send-receive considered harmful : Myths and realities of message passing. *ACM TOPLAS*, 26(1) :47–56, 2004.

[GSF+07] A. Ghuloum, E. Sprangle, J. Fang, G. Wu, and X. Zhou. Ct : A Flexible Parallel Programming Model for Tera-scale Architectures. Technical report, Intel Research, 2007.

[HC02] Y. Hayashi and M. Cole. Automated bsp cost analysis of a parallel maximum segment sum program derivation. *Parallel Processing Letters*, 12(1) :95–112, 2002.

[Lee06] E. A. Lee. The Problem with Threads. Technical Report UCB/EECS-2006-1, Electrical Engineering and Computer Sciences University of California at Berkeley, 2006.

[Rei09] James Reinders. Think parallel or perish, 2009. `http://software.intel.com/sites/products/parallelmag/parallel_mag_issue1.pdf`.

[SF08] J. Sérot and J. Falcou. Functional meta-programming for parallel skeletons. In *8th International Conference Computational Science (ICCS)*, volume 5101 of *LNCS*, pages 154–163. Springer, 2008.

[SHM97] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3) :249–274, 1997.

[Zav01] A. Zavanella. Skeletons, bsp and performance portability. *Parallel Processing Letters*, 11(4) :393–407, 2001.