Programmation parallèle fonctionnelle en C++

Dabrowski Frédéric
Université Paris XII
dabrowski.f@wanadoo.fr

Stage de Maîtrise
encadré par F. Loulergue

Table des matières

Chapitre 1

Introduction

De nombreuses architectures parallèles très performantes sont aujourd'hui disponibles et leur utilité n'est plus à démontrer, en particulier dans des domaines tels que la simulation de phénomènes physiques ou encore la gestion de bases de données importantes où seule la puissance de ces machines permet de réaliser certains calculs. Malgré cette nécessité dans ces domaines et dans bien d'autres encore, il existe toujours un manque dans le développement de logiciels adaptés à ce type d'architectures. Les raisons principales de ce manque sont la difficulté et le coût du développement associés à ce type de programmation ainsi que la complexité que représente la prévision des performances d'une architecture à une autre contrairement au modèle séquentiel où le modèle de Von neumann a permis en son temps un essor important pour les architectures correspondantes. De nombreuses recherches sont en cours afin de proposer des solutions à ces différents problèmes. Trois catégories principales peuvent être distinguées lorsque l'on considére les solutions qui peuvent être proposées au programmeur pour le developpement d'applications parallèles. La distinction entre ces trois catégories repose sur le niveau d'abstraction qu'elles offrent à ce dernier.

La première catégorie, celle proposant le niveau d'abstraction le plus élevé, libère totalement le programmeur du concept de parallélisme. Le développement d'application se fait selon un mode séquentiel et un compilateur se charge alors de la détection automatique du parallélisme. Cependant la détection du parallèlisme dans un algorithme séquentiel peut s'avérer trés ardue et même impossible en général alors même qu'une solution efficace existe. Considérons un exemple simple donné par Cole [2] concernant le calcul de la factorielle. Une définition classique d'une telle fonction dans un programme séquentiel serait

$$\begin{cases} factorielle\ 0 &= 1 \\ factorielle\ n &= n * factorielle(n-1) \end{cases}$$

Cette version ne semble pas présenter de parallèlisme évident et pourtant une implémentation de cette fonction telle que celle présenté ci-dessous est tout à fait adapté à une parallélisation. Cette version oblige toute fois le programmeur à prendre en compte le travail du compilateur et remet donc en cause l'intérêt de cette approche.

$$\begin{cases} factorielle\ 0 &= 1 \\ factorielle\ n &= produit\ 1\ n \\ produit\ a\ a &= a \\ produit\ a\ b &= (produit\ a \lfloor \frac{a+b}{2} \rfloor) * (produit (a \lfloor \frac{a+b}{2} \rfloor + 1) b) \end{cases}$$

La seconde catégorie ajoute à un langage séquentiel classique un ensemble de primitives de communication, MPI[6] en est un exemple, laissant alors au programmeur l'entière responsabilité de la gestion du parallélisme dans l'écriture de ses applications avec toute la complexité, l'indéterminisme et les risques de blocages que cela entraîne. L'évaluation des temps de calculs pour ce modèle de programmation est de plus très complexes.

La troisième catégorie, celle choisie ici, présente une solution intermédiaire. Seule la décomposition du problème reste à la charge du programmeur, la sémantique parallèle opérationnelle étant implicite. Un ensemble de fonctions de haut niveau appelées skeletons (ou patrons) est proposé au programmeur qui compose son programme en précisant les fonctions séquentielles utilisées par ces skeletons. Considérons l'exemple simple de la fonction de haut niveau

$$Map : ('a \rightarrow' b) \rightarrow ['a] \rightarrow ['b].$$

Le programmeur spécifie la fonction devant être appliquée à tous les éléments d'une liste et obtient ainsi une instance du patron Map qui réalisera effectivement l'application de la fonction en parallèle sur tous les éléments. Le but de ce travail est de présenter une implémentation d'un tel mécanisme de programmation parallèle. En raison de sa forte diffusion, le langage C++ a été choisi. Cependant afin d'offrir un plus haut niveau d'abstraction, ces patrons sont implémentés suivant le paradigme fonctionnel grâce à l'utilisation de la librairie FC++[4] qui permet en particulier de considérer des fonctions polymorphes en tant qu'arguments d'autres fonctions et autorise la curryfication et la composition de fonctions ainsi qu'une gestion élaborée des listes. Le modèle de programmation parallèle soutenant cet ensemble de patrons est celui de la librairie de programmation parallèle BSMLlib[1] une extension du langage Objective Caml. Nous présentons ici la première partie de ce travail qui consiste principalement à développer une version C++ (FC++) de cette librairie et à étudier les possibilités fonctionnelles du langage C++ augmenté de FC++. L'implémentation de la BSMLlib présentée repose sur les primitives de communications MPI, ce qui permet d'obtenir une portabilité du code sur une grande variété d'architectures. Un modèle de coût reposant sur le modèle BSP hérité

ici de la BSMLlib permettra d'offrir une prévision des coûts d'exécution fiable pour la réalisation des patrons sur de multiples architectures. Nous présenterons en premier lieu, dans le chapitre 1 une description du modèle BSP puis de la librairie BSMLlib. Ensuite, Dans le chapitre 2 l'utilisation de la librairie FC++ sera explicitée, la compréhension de celle-ci étant nécessaire à l'écriture de programmes BSFC++ élaborées bien qu'un certain nombre de problèmes puissent être résolues à l'aide de quelques opérations seulement. Dans le chapitre 2, nous présentons les quelques opérations constituant le noyau de BSFC++. Le chapitre 3 quand à lui définit la librairie standard de BSFC++ regroupant un certain nombre d'opérations supplémentaires. Des tests achevant la partie concernant la librairie BSFC++ contituent le chapitre 4. Un état de l'art des patrons parallèles et une implémentation BSFC++ d'un ensemble de ces fonctions seront présentés ultérieurement.

Chapitre 2

Préliminaires

2.1 BSP

Le modèle *Bulk-Synchronous Parallelism* (BSP) est un modèle de programmation parallèle introduit par Valiant [7] pour offrir un niveau d'abstraction comparable aux modèles PRAM tout en permettant des performances prévisibles et portables sur une large variété d'architectures. Un ordinateur BSP contient un ensemble de paires processeur-mémoire, un réseau de communication permettant l'échange de messages inter-processeur et une unité de synchronisation globale qui exécute des demandes collectives de barrières de synchronisation. Ses performances sont caractérisées par trois paramètres : le nombre p de paires processeur-mémoire, le temps l nécessaire à une barrière de synchronisation et le temps g nécessaire à une 1-relation (phase de communication où chaque processeur envoie ou reçoit au plus un mot). Pour n'importe quel h le réseau peut réaliser une h-relation, c'est-à-dire une phase de communication où chaque processeur envoie ou reçoit au plus h mots, en temps gh.

Un programme BSP est exécuté comme une séquence de *super-étapes*, chacune étant au plus divisée en trois phases successives et logiquement disjointes (Figure 2.1).

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

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

où $w_i^{(s)}$ = temps de calcul local du processeur i durant la super-étape s et $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ où $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) est le nombre de mots transmis (resp. reçus) par le processeur i durant la super-étape s.

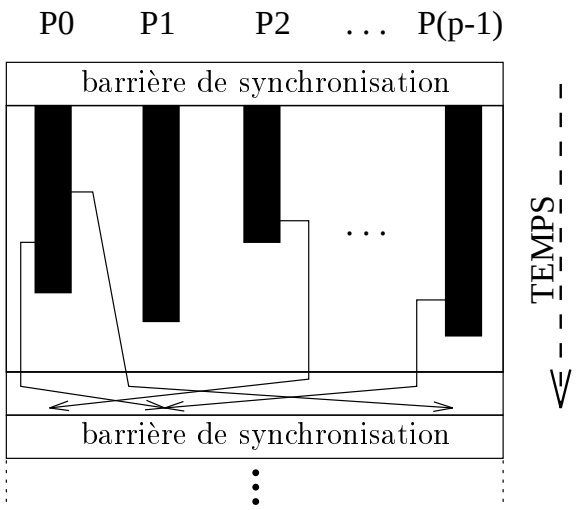


FIG. 2.1 – Super-étapes BSP

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

2.2 BSML

La BSMLlib est une librairie de programmation fonctionnelle Objective Caml pour la programmation selon le mode direct BSP. Cette librairie est développée par Frédéric Loulergue, Gaétan Hains et Olivier Ballerau.

La BSMLlib ne possède pas la variable pid des programmes SPMD, mais utilise une valeur externe `bsp_p :unit->int` telle que la valeur de `bsp_p()` est p, le nombre de processus. La valeur de cette variable ne change pas durant l'exécution. Il y a aussi un constructeur de type polymorphe `par` telle que `'a par` représente le type de vecteurs de p objets de type `'a`, un par processeur. l'imbrication de types `par` est interdite. Le système de type assure cette restriction.

Les objets parallèles sont crées par

```
mkpar : (int -> 'a) -> 'a par
```

de telle manière `(mkpar f)` place `(f i)` au processus i for $i = 0, 1, \ldots, (p-1)$.

Un algorithme BSP s'exprime comme une combinaison de phases de calculs locaux asynchrones et de phases de communications globales accompagnés de synchronisations globales. Le lecteur familier de la BSPlib observera qu'aucune distinction n'est faîte entre une demande de communication et sa réalisation à la barrière de synchronisation. Les phases asynchrones sont programmés avec

```
apply : ('a -> 'b) par -> 'a par -> 'b par
```

Dont la sémantique est celle d'un map sur la structure parallèle. En d'autres termes `apply (mkpar f) (mkpar e)` place `(f i) (e i)` au processus i. Ni l'implémentation de la BSMLlib ni sa sémantique ne requierent une barrière de synchronisation entre deux utilisations successives de `apply`[3] Les phases de communications et de synchronisation s'expriment par

```
put : (int -> 'a option) par -> (int -> 'a option) par
```

où `'a option` est définit par : `type 'a option = None|Some of 'a`.

Considérons l'expression :

$$\texttt{put(mkpar(fun i->fs}_i\texttt{))} \tag{2.1}$$

Pour envoyer une valeur `v` du processus `j` au processus `i`, la fonction $\texttt{fs}_j$ au processus `j` doit être telle que ($\texttt{fs}_j$ `i`) s'évalue à `Some v`. Pour ne pas envoyer de valeurs du processus `j` au processus `i`, ($\texttt{fs}_j$ `i`) doit s'évaluer à `None`.

l'expression (2.1) s'évalue à un vecteur parallèle contenant une fonction $\texttt{fd}_i$ de messages délivrés sur chaque processus. Au processus `i`, ($\texttt{fd}_i$ `j`) s'évalue à`None` si le processus `j` n'a pas envoyé de message au processus `i` ou s'évalue `Some v` si le processus `j` a envoyé `v` au processus `i`.

Il y a également une opération conditionnelle synchrone

```
ifat : (bool par) * int * 'a * 'a -> 'a
```

telle `ifat (v,i,v1,v2)` s'évalue `v1` ou `v2` selon que la valeur de `v` au processus `i` . Mais Objective Caml est un langage strict et cette opération conditionnelle globale ne peut pas être définie comme une fonction. C'est pourquoi le noyau de la BSMLlib contient la fonction : `at :bool par -> int -> bool` qui s'utilise uniquement dans la construction : `if (at vec pid) then... else...` où `(vec :bool par)` et `(pid :int)`.

La signification `if (at vec pid) then expr1 else expr2` est celle de `ifat(vec,pid,expr1,expr2)`.

2.3 FC++

FC++ est une librairie C++ proposée par Brian MacNamara et Yannis Smaragdakis (de l'institut des technologies de Georgie) ajoutant des possibilités de programmation fonctionnelle au langage C++. Cette librairie offre également un grand nombre d'opérations prédéfinies reprenant celles de la librairie standard Haskell[5].

2.3.1 Représentation des fonctions

Bien que C++ offre le support de fonctions polymorphes par l'intermédiaire des *templates*, Ces fonctions ne peuvent être passées en argument à d'autres fonctions. FC++ permet de résoudre ce problème en définissant un type particulier de fonctions appelées *functoids* possédant leur propre système de type parallèlement à celui du langage C++. FC++ propose en outre la curryfication des fonctions et leur composition. Le nombre d'arguments des fonctions FC++ est limité à 6 mais peut cependant être augmenté bien qu'étant nécessairement fini (la librairie doit alors être étendue par de nouvelles fonctions gérant les *functoids*).

2.3.2 Fonctions monomorphes directes

Les fonctions monomorphes directes représentent la classe la plus simple de fonctions FC++. Pour définir une fonction monomorphe directe, on déclare une structure, par exemple `Twice`, dont on définit l'opérateur () pour expliciter la fonction. Cependant Le type C++ d'une telle structure ne représente pas le type réel de la fonction ainsi définie, par exemple

Exemple 1 `twice`

```
1   struct Twice {
2
3       int operator() (int x) {
4
5           return 2*x;
6       }
7   }twice;
```

Le type C++ de cette fonction est `Twice` alors que son type réel est

$$int \to int$$

C'est pourquoi, FC++ encapsule le type réel de la fonction dans la structure même servant à la définir. Ceci ce fait en faisant hériter la structure définissant la fonction de

$$CFun\#Type < A1, A2, \ldots, An, R >$$

où
- # est le nombre d'arguments de la fonction
- $A1, \ldots, An$ sont les types des arguments
- R le type du résultat de la fonction.

Ce qui dans le cas de `Twice` nous donne :

Exemple 2

```
1   struct Twice : public CFun1Type<int,int>{
2
3       int operator() (int x) {
4
5           return 2*x;
6       }
7   }twice;
```

Remarque 1 *Une fonction monomorphe directe peut être passée en argument d'une autre fonction.*

Remarque 2 *L'utilisation du mot-clé* `const` *dans la définition de l'opérateur() est obligatoire (exemple ligne 3).*

Exemple 3

Remarque 3 *Une fonction native C++ peut être transformée en fonction monomorphe directe par l'utilisation de l'opérateur* `ptr_to_fun`.

Exemple 4

```
1   int carre (int x){
2           return x*x;
3   }
4
5   int main(){
6
7           List<int> l = enumFromTo(0,5);
8           List<int> m = map(ptr_to_fun(&carre),l);
9   }
```

2.3.3 Fonctions indirectes

Les fonctions indirectes sont un moyen de définir des variables dont les valeurs possibles couvrent l'ensemble des fonctions d'un même type. Par exemple pour une fonction à une variable de type $int \rightarrow int$ on définit une variable $Fun1 < int, int >$ f. Une telle fonction peut être définie en héritant de

$$Fun\#Impl < A1, A2, \ldots, An, R >$$

où
- # est le nombre d'arguments de la fonction
- $A1, \ldots, An$ sont les types des arguments
- R est le type du résultat de la fonction.

Ou plus simplement en convertissant une fonction monomorphe directe de manière explicite (ex,ligne 8) par l'opérateur `makeFun#` ou de manière implicite (ex,ligne 13).

Exemple 5 *fonctions indirectes*

```
1   int carre (int x){return x*x;}
2
3   int main(){
4
5           //Conversion explicite d'une fonction monomorphe
6           //directe vers une fonction indirecte
7
8           Fun1<int,int> f = makefun(ptr_to_fun(&carre));
9
10          //Conversion implicite d'une fonction monomorphe
11          //directe vers une fonction indirecte
12
13          Fun1<int,int> g = ptr_to_fun(&carre);
14  }
```

2.3.4 Fonctions polymorphes directes

Finalement le type le plus intéressant est celui des fonctions polymorphes directes. Ces fonctions utilisent le mécanisme des *templates* de C++ pour assurer le polymorphisme. Pour définir une fonction polymorphe directe on déclare une structure, disons `struct Fun`, et on définit l'opérateur () pour cette structure selon le comportement désiré pour la fonction. Dans cette structure, on déclare un membre de type `struct Sig` héritant de

$$FunType < A1, A2 \ldots, An, R >$$

où
– # est le nombre d'arguments de la fonction
– $A1, \ldots, An$ sont les types des arguments
– R est le type du résultat de la fonction.

Ce membre représente la signature réelle de la fonction. On obtient finalement la fonction en déclarant une instance, par exemple `fun` de cette structure `Fun`. La fonction sera appelée par `fun(a1,a2,...)`. Une fonction polymorphe directe peut être rendue monomorphe par application de la fonction $monomorphize\# < A_1, A_2, \ldots, A_n > (fonction)$ où les $\texttt{A}_\texttt{i}$ sont les nouveaux types.

Remarque 4 *L'utilisation du mot-clé const dans la définition de l'opérateur() est obligatoire (exemple de Map : ligne 10).*

Exemple 6 *La fonction* `map` *de la librairie FC++*

```
1   struct Map{
2
3       template <class F, class L>
4       struct Sig : public FunType<F,L,
5                           List<typename F::template Sig<
6                           typename L::ElementType>::ResultType> >{};
7
8       template <class F, class T>
9       typename Sig<F,List<T> >::ResultType operator()
10          (const F& f, const List<T>& l) const{
11
12              if (null(l))
13                  return NIL;
14              else
15                  return cons(f(head(l)),
16                              curry2(Map(), f, tail(l)));
17      }
18  }map;
```

La structure `Sig` *possède un champ* `ResultType` *qui permet grâce à la commande* `typename` *de connaître le type de retour en fonction du type des arguments (ligne*

9). A la ligne 4, on définit la structure `Sig`, `F` *représente la fonction et* `L` *la liste pour laquelle on veut appliquer la fonction à ses éléments. A la ligne 5 on définit le type de retour de la fonction qui est*

$$List < typename\ F :: template\ Sig < typename\ L :: ElementType >:: ResultType >$$

Ce qui veut dire que le type de retour est une liste de type typename F :: *template* $Sig <$ *typename* $L ::$ $ElementType >::$ $ResultType$*, ce qui correspond au type de retour d'un élément de type* `F` *dont le premier argument est de type* `typename L : :ElementType`*, où* `ElementType` *est un membre du type* `List` *qui comme son nom l'indique donne le type des éléments constituant la liste.*

2.3.5 Curryfication

FC++ propose un mécanisme de curryfication pour tous les types de fonctions précédemment définis. Il faut distinguer dans l'utilisation de ce mécanisme les classes de fonctions utilisées.

1. **fonctions indirectes :** Une fonction indirecte est directement curryfiable. Dans l'exemple ci dessous, lignes 29 à 32, on déclare simplement une fonction indirecte à partir d'une fonction native et on lui passe ses deux arguments successivement.
2. **fonctions monomorphes directes :** Soit `Fun2` une structure définissant une fonction polymorphe directe à n arguments et `_fun2` une instance de cette structure. Pour obtenir une version curryfiable de cette fonction, on doit instancier une nouvelle variable de type `Curryable#<Fun2>`, où `#` est le nombre d'aguments de la fonction, dont le constructeur prend en paramètre une instance de la structure `Fun2` (ligne 46). La curryfication peut également être utilisée pour les fonctions polymorphes directes par conversion implicite(ligne 38) ou explicite(ligne 42) vers des fonctions indirectes.
3. **fonctions polymorphes directes :** Comme les fonctions monomorphes directes, les fonctions polymorphes directes doivent être explicitement déclarées comme curryfiables (ligne 60). On peut là encore, les convertir en fonctions indirectes (ligne 53) ou les rendres monomorphes(ligne 57) (ceci étant fait implicitement lors de la conversion vers une fonction indirecte).

Remarque 5 *Si plusieurs types utilisent le mécanisme des templates dans la définition d'une fonction, ils doivent le faire en utilisant chacun leur propre type générique pour permettre au mécanisme de curryfication de fonctionner.*

```
1  #include<iostream>
2  #include<prelude.h>
3
4  // Fonction polymorphe directe
5  struct Func1{
6          template <class A, class B>
7          struct Sig : public Funtype<A,B,A>{};
8          template<class A>
9          A operator()(A a, A b)const{
10                 return a+b;
11         }
12 }_func1;
13
14 // Fonction monomorphe directe
15 struct Func2:public CFunType<int,int,int>{
16         int operator()(int a, int b){
17                 return a+b;
18         }
19 }_func2
20
21 //Fonction native
22 int sumi(int x, int y){return x + y;}
23
24 int main(){
25
26         // I - Fonctions indirectes
27         // Automatiquement curryfiables
28
29         Fun2<int,int,int> sumf = makeFun2(ptr_to_fun(&sumi));
30         cout << sumf(2)(3) << "\n";
31         Fun1<int,int> sumg = sumf(2);
32         cout << sumg(3) << "\n";
33
34         // II - Fonctions monomorphes directes
35         // II - 1 - Conversion vers une fonction
36         // indirecte
37
38         Fun2<int,int,int> g = _func2;
39
40         cout << g(1)(2) << "\n";
41         // ou
42         cout << makeFun2(_func2)(1)(3) << "\n";
43
44         // II - 2 - Declaree curryfiable
45
46         Curryable<Func2> func2 (_func2);
```

```
47          cout << func2(2)(4) << "\n";
48
49          // III Fonctions polymorphes directes
50          // III - 1 - Conversion vers une fonction
51          // indirecte
52
53          Fun2<int,int,int> h = _func1;
54          cout << h(1)(2) << "\n";
55
56          // III - 2 - vers fonction monomorphe
57          cout << monomorphize2<int,int,int>(_func1)(1)(2) << "\n";
58
59          // III - 3 - Declaree curryfiable
60          Curryable2<Func1> func1(_func1);
61          cout << func1(1)(7);
62  }
```

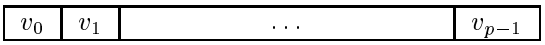
Chapitre 3

BSFC++ : Noyau

3.1 Aperçu

Nous présentons ici le noyau de BSFC++, une implémentation C++ de la BSMLlib basée sur les librairies FC++ et MPI. BSFC++ permet de programmer selon le modèle BSP à ceci près qu'aucune distinction n'est faîte entre les opérations de communications et les barrières de synchronisation les rendants effectives. L'utilisation du modèle BSP permet de définir un modèle de coût simple pour les instructions BSFC++ simplifiant la prévision des temps d'exécution des programmes écrits à l'aide de cette librairie. Les programmes écrits en BSFC++ sont déterministes et ne comporte aucun risque de blocaqe. La librairie BSFC++ hérite des possibilités fonctionnelles de FC++, à savoir le polymorphisme, la curryfication et la composition de fonctions. La portabilité des programmes BSFC++ est assurée par l'utilisation des instructions MPI comme support des opérations de communication, cette librairie étant largement diffusée sur de multiples architectures. Contrairement au modèle SPMD, la construction d'un programme BSFC++ ne repose pas sur l'utilisation de la variable externe pid dont dépend habituellement l'exécution séquentielle sur chaque processus. L'approche est plus intuitive, le parallèlisme de données s'exprimant ici par le type abstrait $Par < A >$ qui représente un vecteur parallèle de valeurs de type A (une pour chaque processus)[1]. On appelle projection sur le processus i d'un vecteur parallèle, la valeur de ce vecteur sur le processus i. Un vecteur parallèle possède un champ `ElementType` renseignant sur le type de la donnée encapsulée et qui servira au moment du typage des fonctions.

[1] Les vecteurs parallèles ne peuvent être imbriqués, cette responsabilité est laissée au programmeur



$\begin{cases} \text{p est le nombre de processus} \\ v_i \text{ est la valeur correspondant au processus i} \end{cases}$

Toute les opérations de communications reposent sur deux opérations de base, **put** qui permet aux processus d'échanger des données et **atpar** qui permet à chaque processus de connaitre la projection sur un processus particulier d'un vecteur parallèle de booléens. Un programme BSFC++ est une suite d'opérations locales et d'opérations de communications globales s'exprimant sur des vecteurs parallèles, comprises entre les instructions **bsfcpp_begin(argc, argv)** et **bsfcpp_end()** ou argc et argv sont les paramètres de la fonction main. Les librairies FC++ et MPI doivent bien sur être installées pour permettre la compilation de programmes BSFC++, le code devant être compilé par mpiCC -I"bsfcpp_dir" -o prog prog.cc -lmpi où bsfcpp_dir est l'emplacement des fichiers de la librairie BSFC++. La version de la librairie FC++ installée doit être celle livrée avec BSFC++ en raison de modification de certains éléments, les tests de la librairie ont poutr l'instant été effectués avec lamMPI.

3.2 Opérations BSFC++

Nous présentons ici les opérations de base de la librairie BSFC++. Nous présentons en premier lieu les notations utilisées. Un vecteur v dans un environnement à n processus est noté $< v_1,\ v_2, \ldots,\ v_p >$ si $\forall i \in \{1,\ 2, \ldots,\ p-1\}$ la projection sur le processus i de v est v_i. On appelle jugement un couple (e, v) où e est une expression et v une valeur. Un jugement est valide si l'expression e s'évalue v et on note alors

$$e \Rightarrow v$$

On appelle règle et on note

$$\frac{p_1\ p_2 \ldots\ p_n}{c}$$

un ensemble de jugements $\{p_1,\ p_2, \ldots,\ p_n, c\}$ tels que si $\forall i \in 1,\ 2, \ldots,\ n,\ p_i$ est valide alors p est valide. Si n=0, on note

$$\frac{}{c}$$

En ce qui concerne le type des fonctions présentées, on prendra une notation inspirée du typage FC++. On note $FunType < A_1,\ A_2,\ \ldots,\ A_n,\ R >$ pour indiquer que la fonction a n arguments de types respectifs $A_1,\ A_2,\ \ldots,\ A_n$ et que son type de retour est R. Si la fonction est polymorphe, on placera devant le type de la fonction `template` $< classC_1, classC_2, \ldots, classC_m >$ si

$C_1, C_2, \ldots, C_m$ représentant les types à préciser. $RT < A, B >:: ResultType$ est le type d'un objet de type A après avoir reçu un objet de type B. Pour des raisons relatives à l'implémentation de FC++, Si les paramètres d'une fonction polymorphe sont de type Par (ou un autre type encapsulant des données), la signature définie par la structure Sig de la fonction diffère des types apparaissant dans la définition de l'opérateur (). Considérons le cas suivant

```
1   template<class Fun, class Arg>
2   Par<RT<Fun,Arg>::ResultType> operator()(Par<Fun> f, Par<Arg> x) const{
3           (...)
4   }
5
6   template<class Funp, class Argp>
7   struct Sig :
8   public FunType< Funp,
9                   Argp,
10                  Par<RT< typename Funp::ElementType,
11                          typename Argp::ElementType>::ResultType>
```

A la ligne 2 nous définissons le premier argument comme étant de type `Par<Fun>` car notre fonction prend en paramètre un vecteur parallèle de valeurs de type `Fun`, cette définition semble donc naturelle. A la ligne 10, en revanche le type est `Funp` et le type utilisé pour calculer le type de retour est `Funp : :Elementtype` car ici `Funp` est un type vecteur parallèle et non le type des éléments du vecteur parallèle. Cette distinction est liée au fait que l'instanciation des types polymorphes par une autre fonction utilisant celle définie ici, se fait par `Sig<X,Y>` si `X` et `Y` sont les types réels passés à ce moment, le type X correspondant alors à un type de vecteur parallèle. Si on avait mis

```
1   template<class Funp, class Argp>
2   struct Sig :
3   public FunType< Par<Fun>,
4                   Par<Arg>,
5                   Par<RT<Fun,Arg>::ResultType>
```

alors `Fun` aurait été de type `X=Par<A>` et non `A`. Cependant, par soucis de simplité dans la présentation des opérations BSFC++, on choisira cette dernière notation (celle représentant le type réel de la fonction dans le dernier exemple).

3.2.1 bsp_p

bsp_p() retourne le nombre de processus alloués à l'exécution du programme parallèle.

$$\boxed{bsp\_p : FunType < int >}$$

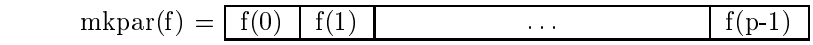
$$\overline{bsp\_p() \Rightarrow p}$$

3.2.2 mk_par

$$\boxed{mkpar : FunType < Fun,\ Par < RT < Fun, int >:: ResultType >}$$

$$\frac{\forall i \in \{1,...,p-1\} f(i) \Rightarrow v_i}{mkpar(f) \Rightarrow <v_0,v_1,...,v_{p-1}>}$$

La fonction **mkpar** permet de créer des vecteurs parallèles. L'argument f est une fonction, au sens de FC++, des indices des processus qui à chacun associe une valeur. f doit être soit une fonction indirecte, soit une fonction directe curryfiable dans le cas de la création de vecteurs parallèles de fonctions. Le vecteur parallèle résultant est le vecteur composé de chacune de ces valeurs.



Exemple 7

```
1   #include "bsfcpp_core.h"
2
3   template <class A>
4   struct Rep_Helper:CFunType<int,A>{
5           A a;
6           Rep_Helper(A b):a(b){};
7           A operator()(int pid)const{
8                   return a;
9           }
10  };
11
12  struct Replicate_par{
13          template <class A>
14          Par<A> operator()(A a)const{
15                  return mk_par(makeFun1(Rep_Helper<A>(a)));
16          }
17          template <class A>
```

```
18          struct Sig : public FunType<A,Par<A> >{};
19   }replicate_par;
```

Dans cet exemple, nous présentons la fonction replicatepar qui permet de créer un vecteur parallèle dont tous les projections sont une copie de la valeur passée en argument. Rep_Helper est une fonction auxiliaire qui permet de renvoyer, quelque soit l'argument qui lui est passé la même valeur (la valeur que l'on veut copier). La fonction replicatepar réalise un mkpar de cette fonction et donne donc le résultat désiré.

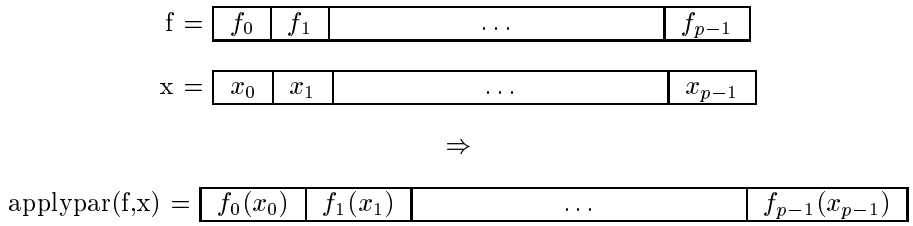
La valeur passée en argument est `makeFun1(Rep_Helper<A>(a))` *car la fonction* **mkpar** *attend une fonction indirecte.*

3.2.3 applypar

La fonction **applypar** applique en parallèle un vecteur parallèle de fonctions à un vecteur parallèle de valeurs (au sens C++ habituel ou de fonctions au sens FC++).

$$applypar : FunType < Par < Fun >, \\ Par < Arg >, \\ Par < RT < Fun, Arg >:: ResultType >$$

$$\frac{f \Rightarrow <f_0,f_1,...,f_{p-1}> \quad v \Rightarrow <v_0,v_1,...,v_{p-1}> \ \forall i \in \{1,...,p-1\} f(v_i) \Rightarrow w_i}{applypar(f,v) \ \Rightarrow \ <w_0,w_1,...,w_{p-1}>}$$



Exemple 8 *Somme de deux vecteurs*

```
1   #include "bsfcpp.h"
2
3   int fun(int i, int j, int k){
4
5           return j+k;
6   }
7
8   int vf(int j, int pid){
9
10          return j*pid;
11  }
12
13  int main(int argc, char* argv[]){
14
15          bsfcpp_begin(argc, argv);
16
17          Fun3<int,int,int,int> f = makeFun3(ptr_to_fun(&fun));
18          Fun2<int,int,int> id_par = makeFun2(ptr_to_fun(&vf));
19
20          Par<int> v = mkpar(id_par(1));
21          Par<int> w =  mkpar(id_par(1));
22
23          Par<int> x =apply_par(apply_par(mkpar(f),v),w);
24
25          bsfcpp_end();
26  }
```

Ligne 17 : récupération d'une fonction directe.
Ligne 23 : Un premier appel à applypar permet de récupérer une fonction qui à un vecteur associe la somme de ce vecteur et de celui passé précédemment (applypar est curryfiable). Un second appel à applypar renvoie la somme des deux vecteurs.

3.2.4 put

La fonction **put** permet l'échange de données entre processus. **put** prend en entrée un vecteur parallèle de fonctions des indices de processus vers des valeurs $< f_0, f_1, \ldots, f_{p-1} >$. Chacune de ces fonctions associe à un indice de processus la valeur à envoyer au processus correspondant, i.e $f_i(j)$ est la valeur que le processus i envoie au processus j. Les donnés peuvent être de n'importe quel type simple exception faite des pointeurs ou de type structure, les membres de ces structures ne devant pas non plus être de type pointeur. De manière générale, les

donnés envoyées peuvent être de n'importe quel type A dont la valeur donnée par size_of() reflète l'espace mémoire réellement occupé (cf. chapître suivant pour les types d'objet complexes). Afin de permettre à un processus de ne pas envoyer de message à un autre lors d'un échange, la fonction put travaille avec le type Option qui encapsule les données à échanger. Le type $Option < A >$ posséde deux constructeurs. Le premier sans paramètre permet de spécifier qu'aucun message n'est envoyé. Le second prend en paramètre la valeur à envoyer. Ainsi la fonction f, définissant les échanges de données en paramètre à la fonction put, doit être définie de telle façon que

$$\begin{cases} f(i) = Option < A > (a) & \text{si la valeur si la valeur a doit être envoyée au processus i,} \\ f(i) = Option < A > () & \text{si aucune valeur ne doit l'être.} \end{cases}$$

La valeur de retour de la fonction put est un vecteur parallèle de fonctions des indices de processus $< g_0, g_1, \ldots, g_{p-1} >$ telles que $g_i(j)$ donne la valeur reçue par le processus i du processus j.

$$put : FunType < Par < Fun1 < int, Option < A >>>, \ Par < Fun1 < int, Option < A >>>>$$

$$\frac{f \ \Rightarrow \ <f_0,f_1,\ldots,f_{p-1}> \quad \forall i,j \in \{1,2,\ldots,p-1\} f_i(j) \Rightarrow v_{i,j}}{put(f) \ \Rightarrow \ <g_0,g_1,\ldots,g_{p-1}>}$$

avec $g_i(j) = f_j(i)$

Exemple 9 *Chaque processus i envoie à chaque processus j le point de coordonnée (i,j).*

- *ligne 10 : Fonction décrivant les données à envoyer.*
- *ligne 24 : envoie des données et récupération des valeurs reçues du processus 0 par chaque processus.*

```
1   #include "bsfcpp.h"
2
3   typedef struct Point_{
4           int x;
5           int y;
6           Point_():x(0),y(0){};
7           Point_(int a, int b):x(a),y(b){};
8   }Point;
9
10  Option<Point> fun(int src, int dst){
11          if(src!=dst)
12                  return Option<Point>(Point(src, dst));
13          else
14                  return Option<Point>();
15  }
16
17  int fun2(int pid){return 0;}
18
19  int main(int argc, char *argv){
20
21          bsml_begin(argc, argv);
22
23          Par<Option<Point> > i =
24                  applypar(       put(mkpar(makeFun2(ptr_to_fun(&fun)))),
25                                  mkpar(makeFun1(ptr_to_fun))));
26          bsml_end();
27          return 0;
28  }
```

3.2.5 at_par

bfatpar évalue la valeur booléene correspondant à un processus dans un vecteur parallèle de booléens. La sémentique BSFC++ impose que atpar ne soit utilisé que dans une structure `if (atpar(v,n)) then` c_1 `;` `else` c_2 `;` de telle manière que, selon le résultat de l'évaluation booléene de la projection du vecteur parallèle v sur le processus i, c_1 ou c_2 soit exécuté.

$$\boxed{atpar : FunType < Par < bool >,\ int,\ bool >}$$

$$\frac{v \Rightarrow < v_0, v_1, ..., v_{p-1}}{atpar(v,i) \Rightarrow v_i}$$

Exemple 10

```
1   #include "bsfcpp.h"
2
3   bool fun1(int pid){
4           if(pid==0) return true;
5           else return false;
6   }
7   int fun2(int pid, int i){
8           return i*i;
9   }
10  int fun3(int pid, int i){
11          return i*i*i;
12  }
13  int id(int pid){
14          return pid;
15  }
16  int main(int argc, char *argv[]){
17
18          bsml_begin(argc, argv);
19
20          Par<int> x = mkpar(makeFun1(ptr_to_fun(&id)));
21          Par<bool> b = mkpar(makeFun1(ptr_to_fun(&fun1)));
22          Par<Fun1<int,int> > f = mkpar(makeFun2(ptr_to_fun(&fun2)));
23          Par<Fun1<int,int> > g = mkpar(makeFun2(ptr_to_fun(&fun3)));
24          Par<int> i;
25
26          if(atpar(b,0)) i = applypar(f,x);
27          else i = applypar(g,x);
28
29          bsml_end();
30          return 0;
31  }
```

3.3 Implémentation

3.3.1 Fonctions MPI

Les calculs BSFC++ se trouvent entre les instructions **bsfcc_begin** et **bsfcpp_end** qui reposent respectivement sur les fonctions `MPI_Initialize` qui débute l'exécution parallèle d'un programme et `MPI_Finalize` qui y met fin. Les fonctions permettant la mesure du temps d'exécution des programmes utilisent `MPI_WTime` qui renvoie le temps écoulé depuis le début de l'exécution parallèle du programme. La fonction **bsp_p** utilisent la fonction `MPI_Comm_size` qui permet de connaître le nombre de processus participant à l'exécution parallèle du programme. **mkpar** utilise la fonction `MPI_Comm_rank` pour connaître le pid de chaque processus

et évaluer la fonction qui lui est passée en argument. Les autres fonctions sont décrites plus en détails ci-dessous.

MPI_Alltoall

```
MPI_Alltoall(  void *sendbuf, int sendcount,
               MPI_Datatype sendtype,
               void* recvbuf, int recvcount,
               MPI_Dataype recvtype, MPI_Comm comm)
```

sendbuf	addresse de départ du buffer d'envoi
sendcount	nombre d'éléments envoyés à chaque processus
sendtype	type de données du buffer d'envoi
recvbuf	addresse de départ du buffer de réception
recvcount	nombre d'éléments reçus de chaque processus
recvtype	type de données du buffer de réception
comm	communicator

Le j^{eme} bloc du buffer d'envoi, envoyé du processus i, est reçu par le processus j et est placé dans le i^{eme} bloc du buffer de réception. Les quantités de données envoyées à chaque processus doivent être égales.

MPI_Alltoallv

```
MPI_Alltoall(  void* sendbuf, int* sendcount,
               int* sdispls,
               MPI_Datatype sendtype,
               void* recvbuf, int* recvcount,
               int* rdispls,
               MPI_Dataype recvtype, MPI_Comm comm)
```

`sendbuf`	addresse de départ du buffer d'envoi
`sendcount`	tableau d'entiers (nombre d'éléments envoyés à chaque processus)
`sdispls`	tableau d'entiers (déplacements des éléments dans le buffer d'envoi)
`sendtype`	type de données du buffer d'envoi
`recvbuf`	addresse de départ du buffer de réception
`recvcount`	tableau d'entiers (nombre d'éléments reçus de chaque processus)
`sdispls`	tableau d'entiers (déplacements des éléments dans le buffer de réception)
`recvtype`	type de données du buffer de réception
`comm`	communicator

Le j^{eme} bloc du buffer d'envoi, envoyé du processus i, est reçu par le processus j et est placé dans le i^{eme} bloc du buffer de réception. Cette opération ajoute au `MPI_Alltoall` la possibilité d'avoir des messages de taille variable.

3.3.2 put

La fonction **put** prend en paramètre un vecteur parallèle de fonctions $< f_0, f_1, \ldots, f_{p-1} >$ tel que $f_i(j)$ est la valeur de type $Option < A >$ encapsulant la valeur de type A envoyée par le processus i au processus j. Pour chaque processus, les $f_i(j)$ sont évaluées pour chaque j processus de destination. Si $f_i(j)$ correspond effectivement à une valeur à envoyer, celle-ci est placée dans un buffer. Deux tableaux *sendcounts* et *sdispls* stockent respectivement les tailles et les déplacements par rapport au début du buffer de chacune des valeurs.

1. Un `Alltoall` permet d'échanger les tailles des messages à envoyer, les déplacements par rapport au buffer de réception étant calculés par rapport à ces dernières.
2. Les messages sont effectivement echangés par un `Alltoallv` et placés dans un buffer de réception.

Chaque processus posséde ainsi un buffer contenant les messages reçus de chacun ainsi que leurs tailles et leurs déplacements par rapport au début de ce buffer. Ces trois élements sont passés en paramètre du constructeur d'une fonction qui à chaque indice de processus associe la valeur de type $Option < A >$ reçue de ce processus. Le vecteur parallèle formé des p fonctions de ce type (correspondant à chaque processus) est finalement renvoyé.

3.3.3 atpar

La fonction **atpar** effectue simplement une diffusion BSP de la projection du vecteur parallèle sur le processus demandé et renvoie cette valeur.

Chapitre 4

BSFC++ : librairie standard

La librairie BSFC++ repose sur une architecture à trois niveaux. Le noyau qui contient les opérations de base de communication et de manipulation de vecteurs parallèles. La librairie standard, présentée ici qui étend le noyau par un certain nombre d'opérations plus élaborées et qui servira de base au troisième niveau constitués des patrons parallèles. Nous présentons en premier lieu un mécanisme de sérialisation permettant d'étendre les opérations de communication à des types complexes.

4.1 Sérialisation

Comme précisé dans la documentation du noyau de bsfcpp, la commande `put` ne permet d'échanger que des messages constitués de types simples n'étant pas des pointeurs ou alors des structures ne contenant pas elles même de pointeurs. Nous présentons ici un mécanisme rudimentaire de serialisation d'objets plus complexes, à savoir des classes pouvant contenir des membres de type pointeurs ou des instances d'autres classes. Une telle classe doit hériter de l'interface `Serializable` et implémenter les méthodes `serialize()` et un constructeur ayant un pointeur sur bytes comme argument. Un objet membre d'une classe sérialisable doit lui même hériter de l'interface sérialisable. De telles objets pourront être utilisé dans des communications par un appel à la fonction putSer selon le même mode que la fonction `put`.

4.1.1 serialize()

La méthode `sérialize` est appelée par la fonction `put` pour récuperer la valeur sérialisée de l'objet. Dans cette méthode, pour chaque membre de l'objet, la méthode `pack` doit être appelée. Si le membre est de type pointeur, la taille en octets de la zone mémoire pointée doit être spécifiée. Dans le cas de

pointeurs de profondeur supérieure à 1 ou dans le cas de pointeurs sur des objets sérialisables, chaque valeurs pointée doit être traitée individuellement (ex : ligne 60). La méthode `serialize` doit toujours se terminer par `return serial();` pour permettre à la fonction **put** de récupérer la copie de l'objet.

Type	valeur	appel
Simple (int, char, struct,...)	int n	pack(n)
Pointeur vers un type simple	char *c ="hello"	pack(c,6)
Objet sérialisable	Matrice : :Serializable m	pack(m)

4.1.2 Serializable(byte *c)

Ce constructeur permet de créer une copie d'un objet sérialisable à partir de sa valeur sérialisée. En premier lieu, ce constructeur doit appeler la méthode `init_ser(c)`, si c est l'argument, qui effectue une copie du buffer passé en argument. Ensuite à chaque appel de `pack` dans la méthode `serialize` doit correspondre, dans le même ordre, un appel à la méthode `unpack`. Contrairement à `pack`, `unpack` qui prend en paramètre l'adresse de la variable à désérialiser, n'a pas besoin de connaître la taille de la zone mémoire à restituer dans le cas d'une variable de type pointeur(ex : ligne21). En revanche dans le cas d'une profondeur de pointeurs supérieure à 1 ou dans le cas de pointeurs sur des objets sérialisables les espace mémoires de chaque pointeur doivent être alloués avant l'appel à `unpack`(ex : ligne 53).

4.1.3 Exemple

```
1   #include "serial.h"
2   #include <iostream>
3
4
5   class Vect:Serializable{
6
7           public :
8
9           int *valeurs;
10          int size;
11
12          Vect(int *v, int s){
13
14                  valeurs = (int*)malloc(sizeof(int)*s);
15                  std::memcpy(valeurs, &v[0], s*sizeof(int));
```

```
16                  size = s;
17          }
18          Vect(byte *c){
19
20                  init_ser(c);
21                  unpack(&valeurs);
22                  unpack(&size);
23          }
24          byte* serialize(){
25                  pack(valeurs, size*sizeof(int));
26                  pack(size);
27                  return serial();
28          }
29  };
30
31  class Matrice:Serializable{
32
33          public :
34
35          Vect * vecteurs;
36          int lignes;
37          int colonnes;
38
39          Matrice(Vect *v, int l, int c){
40
41                  vecteurs = (Vect*)malloc(sizeof(Vect)*c);
42                  for(int i=0; i< c;i++) vecteurs[i] =  Vect(v[i]);
43                  lignes = l;
44                  colonnes = c;
45          }
46
47          Matrice(byte *c){
48
49                  init_ser(c);
50                  unpack(&lignes);
51                  unpack(&colonnes);
52                  vecteurs = (Vect*)malloc(sizeof(Vect)*colonnes);
53                  for (int i=0; i<colonnes; i++) unpack(&vecteurs[i]);
54          }
55
56          byte* serialize(){
57
58                  pack(lignes);
59                  pack(colonnes);
60                  for (int i=0; i<colonnes; i++) pack(vecteurs[i]);
61                  return serial();
```

```
62          }
63
64  };
65
66  int main(){
67          int tab1[]={1,0,0,0};
68          int tab2[]={0,1,0,0};
69          int tab3[]={0,0,1,0};
70          int tab4[]={0,0,0,1};
71
72          Vect v1(tab1,4);
73          Vect v2(tab2,4);
74          Vect v3(tab3,4);
75          Vect v4(tab4,4);
76
77          Vect tab[] = {v1,v2,v3,v4};
78
79          Matrice m1(tab,4,4);
80
81          Matrice m2(m1.serialize());
82
83  }
```

4.2 replicatepar

replicatepar génère un vecteur parallèle dont toutes les projections sont égales à la valeur passées en argument.

$$replicatepar : FunType < A, Par < A >>$$

$$\frac{}{replicate(v) \Rightarrow <v_1,v_2,...,v_{p-1}>} \; \forall\, i \in \{1,2,\ldots,v_{p-1}\}.v_i = v$$

4.3 parfun

`parfun(f, p)` est le vecteur parallèle résultant de l'application de f à toutes les projections du vecteur p.

$$parfun : FunType < A,\ Par < B >,\ Par < RT < A, B >:: ResultType >>$$

$$\frac{v \Rightarrow <v_0,\ v_1,...,\ v_{p-1}> \quad \forall\ i \in \{1,2,\ ...,v_{p-1}\}.f(v_i) \Rightarrow w_i}{parfun(f,\ v) \Rightarrow <w_1,w_2,...,w_{p-1}>}$$

4.4 filterFunSome

Les valeurs de retour des fonctions de communication sont des fonctions à valeurs de type `template <class A> Option<A>` , la fonction `filterFunSome` permet de récupérer les valeurs de type `A` correspondantes. Si f est de type $FunType < int,\ Option < A > >$, `filterFunSome(f,i)` renvoit la valeur de type A correspondant à la valeur $f(i)$ de type `Option<A>` si celle-ci est définit sinon l'exception `OptionException` est levée.

$$\begin{aligned} atpar : FunType < A,\ & \\ & B,\ \\ & Par < typenameA :: ElementType,\ \\ & \quad typename\ B :: ElementType \\ & \quad >:: ResultType :: OptionType > \end{aligned}$$

$$\frac{v \Rightarrow <v_0,\ v_1,...,\ v_{p-1}> \quad \forall\ i \in \{1,2,\ ...,v_{p-1}\}.f(i) \Rightarrow w_i}{parfun(f,\ v) \Rightarrow <w_1,w_2,...,w_{p-1}>}$$

4.5 get

La fonction **get** permet à chaque processus de récupérer les projections d'un vecteurs parallèles sur un sous ensemble des processus. Elle prend en argument le vecteur parallèle dont les valeurs doivent être échangées et un vecteur parallèle de fonction des indices de processus à valeurs booléenes, true ou false selon que la valeur correspondant à l'indice doit être récupérée ou non. La valeur de retour est une fonction des indices de processus à valeur de type `template <class A>`

`Option<A>` donnant les valeur reçue en fonction de l'indice du processus source.

$$get : FunType < Par < A >,\ Par < Fun1 < int, bool > >,\ Par < Fun1 < int,\ Option < bool > > > >$$

$$\frac{v \Rightarrow <v_0,\ v_1,...,\ v_{p-1}>\quad f \Rightarrow <f_1,\ f_2,...,\ f_{p-1}>\quad \forall\ i,j\ in\ \{1,\ 2,...,\ p-1\}.f_i(j)=b_{ij}}{get(p,f) \Rightarrow <g_1,\ g_2,...,\ g_{p-1}>}\ _{g}$$

4.6 get_one

La fonction `get_one` est une version simplifiée de la fonction **get** où chaque processus récupère exactement une valeur.

$$get_one : FunType < Par < A >,\ Par < int >,\ Par < A > >$$

$$\frac{v \Rightarrow <v_1,\ v_2\ ,...,\ v_{p-1}>\quad s \Rightarrow <s_1,\ s_2\ ,...,\ s_{p-1}>}{get\_one(v,\ s) \Rightarrow <v_{s_1},\ v_{s_2}\ ,...,\ v_{s_{p-1}}>}$$

4.7 put_one

La fonction **put_one** est une version simplifiée de la fonction `put` qui permet à chaque processus d'envoyer exactement une valeur d'un vecteur parallèle.

$$put_one : template < classA > FunType < Par < A >,\ Par < int > ,\ Par < Fun1 < int, Option < A > > >$$

$$\frac{v \Rightarrow <v_1,\ v_2\ ,...,\ v_{p-1}> \quad d \Rightarrow <d_1,\ d_2\ ,...,\ d_{p-1}>}{get\_one(v,\ s) \Rightarrow <f_1,\ f_2\ ,...,\ f_{p-1}>}$$

$$\text{avec } f_i(j) = \begin{cases} Option < A > (v_j) & \text{si } d_j\text{=i} \\ Option < A > () & \text{sinon.} \end{cases}$$

4.8 totex

Réalise un échange total entre les valeurs d'un vecteur parallèle.

$$totex : template < classA > FunType < Par < A >,\ Par < Fun1 < int, A >> >$$

$$\frac{v \Rightarrow <v_1,\ v_2\ ,...,\ v_{p-1}>}{totex(v) \Rightarrow <f_1,\ f_2\ ,...,\ f_{p-1}>}$$

$$\text{avec } \forall i, j \in \{1,\ 2, \ldots,\ p-1\}.f_i(j) = v_j$$

4.9 shiftl

Réalise un décalage à gauche des projections d'un vecteur parallèle.

$$shitfl : template < classA > FunType < Par < A >,\ Par < A > >$$

$$\frac{v \Rightarrow <v_1,\ v_2\ ,...,\ v_{p-1}>}{shiftl(v) \Rightarrow <v_2,\ v3_2\ ,...,\ v_1>}$$

4.10 shiftr

Réalise un décalage à droite des projections d'un vecteur parallèle.

$$shitfr : template < classA > FunType < Par < A >,\ Par < A > >$$

$$\frac{v \ \Rightarrow \ <v_1,\ v_2\ ,...,\ v_{p-1}>}{shiftr(v) \ \Rightarrow \ <v_2,\ v_3\ ,...,\ v_1>}$$

4.11 bcast_direct

Effectue une diffusion de la projection d'un vecteur parallèle sur un processus particulier et renvoie le vecteur parallèle dont toutes les projections sont égales à cette valeur.

$$bcast_direct : template < classA > FunType < Par < A >,\ Par < Fun1 < int, A >> >$$

$$\frac{v \ \Rightarrow \ <v_1,\ v_2\ ,...,\ v_{p-1}>}{bcast\_direct(i,v) \ \Rightarrow \ <w_1,\ w_2\ ,...,\ w_{p-1}>}$$

avec $\forall j \in \{1,\ 2, \ldots,\ p-1\}.w_j \ = \ v_i$

4.12 cutList

Cette fonction prend une Liste L et un entier i en argument et renvoie la sous liste correspondant au i^{eme} morceau de L découpée en p sous listes si p est le nombre de processus.

$$cutList : template < classA > FunType < List < A >,\ int,\ List < A > >$$

4.13 gatherList

gatherList prend en argument un vecteur parallèle de listes et un entier i et renvoie le vecteur parallèle dont la projection sur le processus i est la liste obtenue par composition des listes du vecteur argument (par ordre croissant des indices de processus).

$$gatherList : template < classA > FunType < Par < List < A > > , \ int,\ Par < List < A > > >$$

$$\frac{v \Rightarrow <l_1, l_2, ..., l_{p-1}>}{gatherList(v,i) \Rightarrow <m_1, m_2, ..., m_{p-1}>}$$

avec $\forall j \neq i . m_j = l_j$ et $m_i = l_1 l_2 \ldots l_n$

4.14 scatterList

scatterList prend en argument un vecteur parallèle de listes et un indice de processus i et renvoie le vecteur parallèle composée des sous listes obtenues par cutList.

$$scatterList : template < classA > FunType < Par < List < A > > , \ int,\ Par < List < A > > >$$

$$\frac{v \Rightarrow <l_1, l_2, ..., l_{p-1}>}{scatterList(v,i) \Rightarrow <m_1, m_2, ..., m_{p-1}>}$$

avec $\forall j \neq i . m_j = cutList(l, j)$

4.15 parFoldList

foldList prend en argument un vecteur parallèle de liste et une fonction et renvoie un vecteur parallèle dont chaque projection est le résultat de l'exécution d'un fold sur l'ensemble des listes.

$$parFoldList : template < classA > FunType < Par < List < A > >, \ Par < A > \ >$$

$$\frac{v \Rightarrow <l_1, l_2, ..., l_{p-1}>}{parFoldList(f,i) \Rightarrow <a_1, a_2, ..., a_{p-1}>}$$

avec $\forall j . a_j = f(\ldots(f(fold(f, l1), fold(f, l_2)) \ldots, l_{p-1}))$

Chapitre 5

Tests

Nous avons effectué sur un *cluster* de PC une série de tests basés sur une fonction **parallel_fold** qui prend en argument une fonction et un vecteur parallèle de listes, et réalise un *fold* sur l'ensemble des listes du vecteurs. Le résultat global du *fold* se trouve sur toues les projections du vecteur parallèle retourné. Les test ont été effectués avec 2, 4, 6 et 8 machines en considérant des listes d'entiers de 1000,10000,100000 et 1000000 d'éléments, l'opération effectuée est l'addition. Le tableau suivant présente les résultats comparativement à ceux obtenus avec la BSMLLib. Les temps de calculs sont données en secondes et la fonction **parallel_fold** est executé à chaque fois 100 fois pour avoir un ordre de grandeur suffisant.

nombre de processeurs	2	4	6	8
1000 entiers				
BSFC++	0.109	0.139	0.165	0.194
BSMLLib	0.31	0.53	0.63	0.7
10000 entiers				
BSFC++	0.140	0.141	0.176	0.293
BSMLLib	1.18	1.52	1.67	1.89
100000 entiers				
BSFC++	10.79	11.29	11.48	11.53
BSMLLib	23.1	23.4	23.6	24
1000000 entiers				

Les résultats de la librairie semblent, dans ce cas, acceptables,. Des tests plus poussés devront être réalisés.

Chapitre 6

Conclusion

Nous avons présenté ici la première version de la librairie BSFC++, d'autres fonctions devront encore être ajoutés à la librairie standard afin de permettre l'implémentation d'un ensemble de patrons parallèles. Ils restent également à préciser les coûts de chacune des opérations afin de permettre à ce troisième niveau de l'architecture BSFC++ de bénéficier d'un modèle de coût simple. Pour l'instant les test présentés dans le chapître précédent ont été les seuls à être réalisé sur de véritables architectures parallèles, cependant de nombreux tests ont été effectués sur une seule machine grâce à l'utilisation de plusieurs processus et la librairie semble stable.

Bibliographie

[1] Olivier Ballereau, Frédéric Loulergue, and Gaétan Hains. *The BSMLlib version 0.1 Reference Manual*, January 2000. available from http ://www.univ-paris12.fr/lacl/loulergu.

[2] Murray I. Cole. *Algorithmic Skeletons : Structured Management of Parallel Computation*. Research Monographs in Parallel and Distributed Computing. Pitman, London, 1989.

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

[4] Brian McNamaran and Smaragdakis Yannis. Functionnal programming in c++ using the fc++ library. available from www.cc.gatech.edu/ yannis/fc++.

[5] Jones S. Peyton and J.Hughes. *Report on the programming language Haskell 98*, February 1999. available from www.haskell.org.

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

[7] L.G. Valiant. A bridging model for computation. *Communications of the ACM*, 33(8) :103–111, 1990.

Index