Programmation Parallèle BSP fonctionnelle

Frédéric Loulergue
Laboratoire d'Informatique Fondamentale d'Orléans (LIFO)
Université d'Orléans

Septembre 2005

Sommaire

Sommaire

- Contexte : laboratoires et projets

Sommaire

- Contexte : laboratoires et projets
- Programmation en Caml

Sommaire

- Contexte : laboratoires et projets
- Programmation en Caml
- Parallélisme

Sommaire

- Contexte : laboratoires et projets
- Programmation en Caml
- Parallélisme
- *Bulk Synchronous Parallel ML* (BSML)

Contexte

Laboratoire d'Algorithmique, Complexité et Logique

Laboratoire d'Algorithmique, Complexité et Logique

- Laboratoire de l'Université Paris XII & du CNRS

Laboratoire d'Algorithmique, Complexité et Logique

- Laboratoire de l'Université Paris XII & du CNRS
- Les membres du LACL :
  - 5 Professeurs
  - 10 Maîtres de conférences
  - 8 Doctorants

Laboratoire d'Algorithmique, Complexité et Logique

- Laboratoire de l'Université Paris XII & du CNRS
- Les membres du LACL :
  - 5 Professeurs
  - 10 Maîtres de conférences
  - 8 Doctorants
- Trois équipes :
  - Algorithmique et vérification
  - Logique et complexité
  - Systèmes communicants

Laboratoire d'Algorithmique, Complexité et Logique

- Laboratoire de l'Université Paris XII & du CNRS
- Les membres du LACL :
  - 5 Professeurs
  - 10 Maîtres de conférences
  - 8 Doctorants
- Trois équipes :
  - Algorithmique et vérification
  - Logique et complexité
  - Systèmes communicants
- http ://www.univ-paris12.fr/lacl

Laboratoire d'Informatique Fondamentale d'Orléans

Laboratoire d'Informatique Fondamentale d'Orléans

- Laboratoire de l'Université d'Orléans & du CNRS

Laboratoire d'Informatique Fondamentale d'Orléans

- Laboratoire de l'Université d'Orléans & du CNRS
- Les membres du LIFO :
  - 10 Professeurs
  - 19 Maîtres de conférences
  - 15 Doctorants

Laboratoire d'Informatique Fondamentale d'Orléans

- Laboratoire de l'Université d'Orléans & du CNRS
- Les membres du LIFO :
  - 10 Professeurs
  - 19 Maîtres de conférences
  - 15 Doctorants
- Trois équipes :
  - Contraintes et Apprentissage
  - Graphes et Algorithmes
  - Vérification Parallélisme et Sécurité

Laboratoire d'Informatique Fondamentale d'Orléans

- Laboratoire de l'Université d'Orléans & du CNRS
- Les membres du LIFO :
  - 10 Professeurs
  - 19 Maîtres de conférences
  - 15 Doctorants
- Trois équipes :
  - Contraintes et Apprentissage
  - Graphes et Algorithmes
  - Vérification Parallélisme et Sécurité
- http ://www.univ-orleans.fr/lifo

Le projet Caraml 2002-2004

- CoordinAtion et Répartition d'Applications multiprocesseurs en objective camML

- CoordinAtion et Répartition d'Applications multiprocesseurs en objective camML
- Projet de l'ACI GRID (Globalisation des ressources informatiques et des données)

- CoordinAtion et Répartition d'Applications multiprocesseurs en objective camML
- Projet de l'ACI GRID (Globalisation des ressources informatiques et des données)
- Partenaires :

  Université d'Orléans (LIFO) Université Paris XII (LACL)
  Université Paris VII (PPS) INRIA

Le projet Caraml 2002-2004

- CoordinAtion et Répartition d'Applications multiprocesseurs en objective camML
- Projet de l'ACI GRID (Globalisation des ressources informatiques et des données)
- Partenaires :

  Université d'Orléans (LIFO) Université Paris XII (LACL)
  Université Paris VII (PPS) INRIA
- Développement de bibliothèques pour le calcul haute-performance et globalisé autour du langage Objective Caml de l'INRIA :bibliothèques de primitives parallèles et globalisées, bibliothèques applicatives.
- http ://www.caraml.org

Le projet Propac 2004-2007

- PROgrammation PArallèle Certifée

- PROgrammation PArallèle Certifée
- Projet de l'ACI Jeunes Chercheuses et Chercheurs

- PROgrammation PArallèle Certifée
- Projet de l'ACI Jeunes Chercheuses et Chercheurs
- Exécution sûre de programmes parallèles certifiés :
  - Programmes BSML certifiés en Coq
  - Machine abstraite & compilateur BSML certifiés
  - Prévision de performance

- PROgrammation PArallèle Certifée
- Projet de l'ACI Jeunes Chercheuses et Chercheurs
- Exécution sûre de programmes parallèles certifiés :
  - Programmes BSML certifiés en Coq
  - Machine abstraite & compilateur BSML certifiés
  - Prévision de performance
- http ://wwwpropac.free.fr

Rappels de programmation en Objective Caml

Logiciels de qualité

Pour écrire un logiciel de qualité il faudrait :

- Une spécification
- Un programme
- Une preuve de correction du programme par rapport à la spécification

Logiciels de qualité

Pour écrire un logiciel de qualité il faudrait :

- Une spécification
- Un programme
- Une preuve de correction du programme par rapport à la spécification

Sans aller jusqu'à la preuve formelle :

écrire rapidement des programmes corrects

- Langage fonctionnel
- Typé statiquement
- Polymorphe paramétrique
- Inférence de types

- Langage fonctionnel
- Typé statiquement
- Polymorphe paramétrique
- Inférence de types
- Exceptions
- Traits impératifs
- Système de modules
- Interaction avec C
- Processus légers

- Langage fonctionnel
- Typé statiquement
- Polymorphe paramétrique
- Inférence de types
- Exceptions
- Traits impératifs
- Système de modules
- Interaction avec C
- Processus légers
- Objets

Expressions

```
# 2+2 ; ;
```

Expressions

```
# 2+2 ; ;
- : int = 4
```

```
# 2+2;;
- : int = 4
```

“L’expression que vous venez de saisir est de type `int` *et sa valeur est 4”*

Définitions

Définitions

- Associent dans l'environnement de l'utilisateur des identificateurs (variables) à des valeurs
- Identificateurs : suites de lettres (première en minuscule), chiffres, ' et _

Définitions

- Associent dans l'environnement de l'utilisateur des identificateurs (variables) à des valeurs
- Identificateurs : suites de lettres (première en minuscule), chiffres, ' et _
- Exemples :

```
# let x = 1-(2+3) ;;
val x : int = -4

# x*x ;;
- : int = 16
```

```
# let euro = 6.55957 ;;
val euro : float = 6.55957

# 100./.euro ;;
- : float = 15.2449017237
```

Définitions locales (1)

Définitions locales (1)

- Portée limitée à une expression

Définitions locales (1)

- Portée limitée à une expression
- Exemple :

```
# let a=4 and b=2 in a*b ; ;
- : int = 8
```

Définitions locales (1)

- Portée limitée à une expression
- Exemple :

```
# let a=4 and b=2 in a*b;;
- : int = 8
```

- Il n'y a pas de portée globale :

```
# a;;
Characters 0-1 :
a;;
^
Unbound value a
```

Définitions locales (2)

```
# let x = 4;;
val x : int = 4

# let x=5 in x*x+2*x+1;;
- : int = 36

# x;;
- : int = 4
```

Entiers

- Type : `int`
- Compris entre `min_int` et `max_int` :

```
# max_int ;;
- : int = 1073741823

# min_int ;;
- : int = -1073741824
```

- Opérateurs : `*`, /, +, −, `mod`

Entiers

Entiers

- Type : `int`
- Compris entre `min_int` et `max_int` :

```
# max_int ;;
- : int = 1073741823

# min_int ;;
- : int = -1073741824
```

- Opérateurs : `*`, `/`, `+`, `-`, `mod`

Type unit

- Type : `unit`
- Constante : ()

Booléens

Booléens

- Type : `bool`
- Constantes : `true` et `false`
- Opérateurs : `||` (ou), `&` (et), `not`

Booléens

- Type : `bool`
- Constantes : `true` et `false`
- Opérateurs : `||` (ou), `&` (et), `not`
- Exemples :

```
# 1=2 ; ;
- : bool = false

# 1<2 ; ;
- : bool = true

# (1<2)&(2<3) ; ;
- : bool = true
```

Expression conditionnelle

- Syntaxe : `if b then e1 else e2`

Expression conditionnelle

- Syntaxe : `if b then e1 else e2`
- Typage :
  - `b` doit être une expression booléenne
  - `e1` et `e2` sont deux expressions de même type

Expression conditionnelle

- Syntaxe : `if b then e1 else e2`
- Typage :
  - `b` doit être une expression booléenne
  - `e1` et `e2` sont deux expressions de même type
- Évaluation :
  - Si `b` vaut `true` alors c'est `e1` qui est évaluée
  - Si `b` vaut `false` alors c'est `e2` qui est évaluée

Expression conditionnelle

- Syntaxe : `if b then e1 else e2`
- Typage :
  - `b` doit être une expression booléenne
  - `e1` et `e2` sont deux expressions de même type
- Évaluation :
  - Si `b` vaut `true` alors c'est `e1` qui est évaluée
  - Si `b` vaut `false` alors c'est `e2` qui est évaluée
- Remarque : les deux branches sont obligatoires

Expression conditionnelle

- Syntaxe : `if b then e1 else e2`
- Typage :
  - `b` doit être une expression booléenne
  - `e1` et `e2` sont deux expressions de même type
- Évaluation :
  - Si `b` vaut `true` alors c'est `e1` qui est évaluée
  - Si `b` vaut `false` alors c'est `e2` qui est évaluée
- Remarque : les deux branches sont obligatoires
- Exemple :

```
# let x=100 in if x>1000 then "grand" else "petit";;
- : string = "petit"
```

Produit cartésien

Produit cartésien

- Pour construire des n-uplets

Produit cartésien

- Pour construire des n-uplets
- Syntaxe : `(e1,e2,...,en)`

Produit cartésien

- Pour construire des n-uplets
- Syntaxe : `(e1,e2,...,en)`
- Pour les couples uniquement :
  - `fst` renvoie la première composante
  - `snd` renvoie la seconde composante

Produit cartésien

- Pour construire des n-uplets
- Syntaxe : `(e1,e2,...,en)`
- Pour les couples uniquement :
  - `fst` renvoie la première composante
  - `snd` renvoie la seconde composante
- Exemples :

```
# let x=(1,2);;
val x : int * int = (1, 2)

# fst x;;
- : int = 1

# snd x;;
- : int = 2

# let truc = (true,"toto");;
val truc : bool * string = (true, "toto")

# fst truc;;
- : bool = true
```

- Définitions :

```
# let f1 (x,y) = x*x + y*y ;;
val f1 : int * int -> int = <fun>

# let f2 = fun (x,y) -> x*x + y*y ;;
val f2 : int * int -> int = <fun>
```

- Définitions :

```
# let f1 (x,y) = x*x + y*y;;
val f1 : int * int -> int = <fun>

# let f2 = fun (x,y) -> x*x + y*y;;
val f2 : int * int -> int = <fun>
```

- Application :

```
# f1 (2,3);;
- : int = 13

# f2 (2,3);;
- : int = 13
```

- Définitions :

```
# let f1 (x,y) = x*x + y*y ;;
val f1 : int * int -> int = <fun>

# let f2 = fun (x,y) -> x*x + y*y ;;
val f2 : int * int -> int = <fun>
```

- Application :

```
# f1 (2,3) ;;
- : int = 13

# f2 (2,3) ;;
- : int = 13
```

- Les fonctions n'ont pas besoin d'être nommées :

```
# fun x -> x*x + 2*x + 1 ;;
- : int -> int = <fun>
```

Fonctions d'ordre supérieur (1)

Fonctions d'ordre supérieur (1)

- Les fonctions sont des valeurs comme les autres
- En particulier on peut les passer en argument

Fonctions d'ordre supérieur (1)

- Les fonctions sont des valeurs comme les autres
- En particulier on peut les passer en argument
- Exemple :

```
# let g f = fun x -> (f(x)+f(-x))/2 ;;
val g : (int -> int) -> int -> int = <fun>

# let f x = 2*x + 1 ;;
val f : int -> int = <fun>

# g(f) ;;
- : int -> int = <fun>

# (g(f))(1) ;;
- : int = 1
```

Fonctions d'ordre supérieur (1)

- Les fonctions sont des valeurs comme les autres
- En particulier on peut les passer en argument
- Exemple :

```
# let g f = fun x -> (f(x)+f(-x))/2 ;;
val g : (int -> int) -> int -> int = <fun>

# let f x = 2*x + 1 ;;
val f : int -> int = <fun>

# g(f) ;;
- : int -> int = <fun>

# (g(f))(1) ;;
- : int = 1
```

- On dit que g est une fonction d'ordre supérieur

- Autre exemple :

```
# (g (fun x->x+1))(2) ; ;
- : int = 1
```

Fonctions d'ordre supérieur (2)

- Autre exemple :

```
# (g (fun x->x+1))(2) ;;
- : int = 1
```

- Plus simplement pour l'application on peut écrire :

```
# g f ;;
- : int -> int = <fun>

# g f 1 ;;
- : int = 1
```

Listes (1)

Listes (1)

- Une liste contient une suite de valeurs de même type.
- Structure de données polymorphe

- Une liste contient une suite de valeurs de même type.
- Structure de données polymorphe
- Liste vide []

```
# [];;
- : 'a list = []
```

- Une liste contient une suite de valeurs de même type.
- Structure de données polymorphe
- Liste vide []

```
# [];;
- : 'a list = []
```

- Ajout d'un élément en début de liste ::
  C'est un opérateur infixe.

```
# 1::[];;
- : int list = [1]
# 2::1::[];;
- : int list = [2; 1]
```

- Autre notation :

```
# [2;1];;
- : int list = [2; 1]

# ["Bonjour";"tout";"le";"monde"];;
- : string list = ["Bonjour"; "tout"; "le"; "monde"]
```

- Autre notation :

```
# [2;1];;
- : int list = [2; 1]

# ["Bonjour";"tout";"le";"monde"];;
- : string list = ["Bonjour"; "tout"; "le"; "monde"]
```

- Attention : La virgule est la notation pour séparer les composantes d'un tuple, pas d'une liste !

```
# [1,2,3,4];;
- : (int * int * int * int) list = [(1, 2, 3, 4)]
# [1;2;3;4];;
- : int list = [1; 2; 3; 4]
```

Dans le module `List` :

Dans le module `List` :

- hd: α list $\rightarrow$ α

  (List.hd l) renvoie le premier élément de la liste l. Ne doit pas être appliqué à une liste vide. On appelle cet élément la tête de la liste.

Dans le module `List` :

- hd: α list $\rightarrow$ α

  (List.hd l) renvoie le premier élément de la liste l. Ne doit pas être appliqué à une liste vide. On appelle cet élément la tête de la liste.

- tl: α list $\rightarrow$ α list

  (List.tl l) renvoie la liste sans son premier élément. Ne doit pas être appliqué à une liste vide. On appelle cette liste la la queue de la liste l.

(∗ val length : α list → int ∗)

```
let rec length l =
  if l=[]
  then 0
  else 1+length (List.tl l)
```

(∗ val length : α list → int ∗)

let rec length l =
  if l=[]
  then 0
  else 1+length (List.tl l)

```
# length [1;2;3];;
- : int = 3
```

(∗ val length : α list → int ∗)

let rec length l =
  if l=[]
  then 0
  else 1+length (List.tl l)

```
# length [1;2;3];;
- : int = 3

# length [ [1;2]; [2;3;4] ];;
- : int = 2
```

(∗ val length : α list $\rightarrow$ int ∗)

let rec length l =
  if l=[]
  then 0
  else 1+length (List.tl l)

```
# length [1;2;3];;
- : int = 3

# length [ [1;2]; [2;3;4] ];;
- : int = 2

# length [ (+); (-); (/); (fun x y->2*x*y) ];;
- : int = 4
```

Programmation parallèle

Un ordinateur parallèle

Un ordinateur parallèle

- Il en existe de différentes architectures

Un ordinateur parallèle

- Il en existe de différentes architectures
- Un type très répandu : les grappes de PC

Un ordinateur parallèle

- Il en existe de différentes architectures
- Un type très répandu : les grappes de PC

- On les utilise pour calculer plus vite le résultat des fonctions et/ou pour manipuler des données plus grandes.

Le plus répandu : Single Program Multiple Data

Le plus répandu : Single Program Multiple Data

- on munit chaque processeur d'un numéro appelé "pid" (compris entre 0 et $p-1$)

Programmation parallèle

Le plus répandu : Single Program Multiple Data

- on munit chaque processeur d'un numéro appelé “pid” (compris entre 0 et $p-1$)
- on écrit un programme avec des expressions du genre :

  ```
  if pid=0 then ... else
  if pid=1 then ... else ...
  ```

  C'est ce même programme qui sera exécuté sur tous les processeurs

Le plus répandu : Single Program Multiple Data

- on munit chaque processeur d'un numéro appelé “pid” (compris entre 0 et $p-1$)
- on écrit un programme avec des expressions du genre :

  ```
  if pid=0 then ... else
  if pid=1 then ... else ...
  ```

  C'est ce même programme qui sera exécuté sur tous les processeurs
- chaque processeur peut contenir des données différentes.
  Une variable x de type int peut avoir une valeur différente sur chaque processeur, par exemple elle peut contenir la valeur du “pid” $+ 1$.

Le plus répandu : Single Program Multiple Data

- on munit chaque processeur d'un numéro appelé "pid" (compris entre 0 et $p-1$)
- on écrit un programme avec des expressions du genre :

  ```
  if pid=0 then ... else
  if pid=1 then ... else ...
  ```

  C'est ce même programme qui sera exécuté sur tous les processeurs
- chaque processeur peut contenir des données différentes. Une variable x de type int peut avoir une valeur différente sur chaque processeur, par exemple elle peut contenir la valeur du "pid" $+ 1$.
- les processeurs s'échangent des données par passage de messages avec deux opérations : envoi et réception.

Indéterminisme (1)

- On suppose qu'on a pid:unit$\longrightarrow$ int qui renvoie le “pid” du processeur

- On suppose qu'on a pid:unit$\longrightarrow$ int qui renvoie le "pid" du processeur
- Programme SPMD Caml : print_int (pid())

- On suppose qu'on a pid:unit$\rightarrow$ int qui renvoie le "pid" du processeur
- Programme SPMD Caml : print_int (pid())
- On peut par exemple avoir les résultats suivants :
  - 012345
  - 021345
  - 534120
  - ...

Indéterminisme (1)

- On suppose qu'on a pid:unit→ int qui renvoie le "pid" du processeur
- Programme SPMD Caml : print_int (pid())
- On peut par exemple avoir les résultats suivants :
  - 012345
  - 021345
  - 534120
  - ...
- Ce programme est indéterministe

- On suppose qu'on a deux opérations blocantes :
  - **receive**: unit$\rightarrow$ α
    receive() attend de recevoir une valeur puis envoie une confirmation de réception et retourne la valeur reçue
  - **send**: α $\rightarrow$ int$\rightarrow$ unit
    (**send** x i) envoie une valeur *x* à un processeur *i* et qui attend la confirmaton de réception

- On suppose qu'on a deux opérations blocantes :
  - **receive**: unit→ α
    receive() attend de recevoir une valeur puis envoie une confirmation de réception et retourne la valeur reçue
  - **send**: α → int→ unit
    (**send** x i) envoie une valeur *x* à un processeur *i* et qui attend la confirmaton de réception
- Programme SPMD pseudo-Caml :

```
let f x = if pid=0
  then let y = receive() and z = receive() in x::y::z::[]
  else
    if pid=1 then let _ = send x 0 in []
    else if pid=2 then let _ = send x 0 in []
    else [];;
f (pid+1)
```

- On suppose qu'on a deux opérations blocantes :
  - **receive**: unit→ α
    receive() attend de recevoir une valeur puis envoie une confirmation de réception et retourne la valeur reçue
  - **send**: α → int→ unit
    (**send** x i) envoie une valeur x à un processeur i et qui attend la confirmaton de réception
- Programme SPMD pseudo-Caml :

```
let f x = if pid=0
  then let y = receive() and z = receive() in x::y::z::[]
  else
    if pid=1 then let _ = send x 0 in []
    else if pid=2 then let _ = send x 0 in []
    else [];;
f (pid+1)
```

- Au processeur 0, f n'est pas déterministe.
  On peut avoir [1;2;3] ou [1;3;2].

- Programme SPMD pseudo-Caml :

```
let decalage_droit x =
  let _ = send x ((pid+1) mod p) in
  receive()
```

- Tous les processeurs attendent : c'est un interblocage

Le modèle *Bulk Synchronous Parallelism* (BSP)

Le modèle *Bulk Synchronous Parallelism* (BSP)

- Début des années 90 par Valiant (Cambridge) et McColl (Oxford)

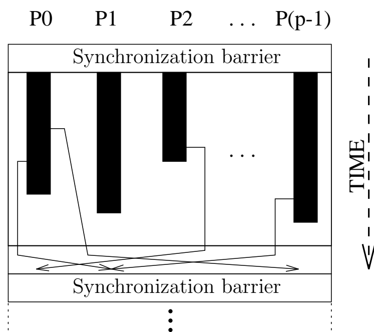
Le modèle *Bulk Synchronous Parallelism* (BSP)

- Début des années 90 par Valiant (Cambridge) et McColl (Oxford)
- Propose :
  - un modèle abstrait d'architecture
  - un modèle d'exécution
  - un modèle de performances (ou de coûts)

Le modèle *Bulk Synchronous Parallelism* (BSP)

- Début des années 90 par Valiant (Cambridge) et McColl (Oxford)
- Propose :
  - un modèle abstrait d'architecture
  - un modèle d'exécution
  - un modèle de performances (ou de coûts)
- La machine parallèle BSP est composée de :
  - p paires identiques processeur-mémoire
  - un réseau permettant les communications 2 à 2
  - une unité de synchronisation globale

BSP : modèle d'exécution et de performances



$$T(s) = \max_{0 \leq i < p} w_i + h \times g + L$$

Bulk Synchronous Parallel ML

Introduction

- The design of a parallel language is a trade-off between :
  - expressivity
  - simple semantics & performance prediction

Introduction

- The design of a parallel language is a trade-off between :
  - expressivity
  - simple semantics & performance prediction
- Bulk Synchronous Parallel ML :
  - describes Bulk Synchronous Parallel algorithms (explicit processes & communications)
  - deadlock free & deterministic
  - pure functional semantics (based on a confluent extension of the λ-calculus)

Overview

Overview

- Functional language (ML family) + BSP operations
- BSML prog. = usual ML prog. + operations on parallel vectors

Overview

- Functional language (ML family) + BSP operations
- BSML prog. = usual ML prog. + operations on parallel vectors
- Parallel vector : α **par** (size *p*)

Overview

- Functional language (ML family) + BSP operations
- BSML prog. = usual ML prog. + operations on parallel vectors
- Parallel vector : α **par** (size p)
- Access to BSP parameters :

  bsp_p: unit $\rightarrow$ int
  bsp_g: unit $\rightarrow$ float
  bsp_l: unit $\rightarrow$ float

Creation of parallel vectors

Creation of parallel vectors

$$\textbf{mkpar} : (\text{int} \longrightarrow \alpha) \longrightarrow \alpha\ \textbf{par}$$

Creation of parallel vectors

$$\textbf{mkpar} : (\text{int} \rightarrow \alpha) \rightarrow \alpha \ \textbf{par}$$

$$(\textbf{mkpar}\ f)$$

$(f\ 0)$	$(f\ 1)$	$\cdots$	$(f\ (p-1))$

Creation of parallel vectors

$$\textbf{mkpar} : (\text{int} \rightarrow \alpha) \rightarrow \alpha \ \textbf{par}$$

$$(\textbf{mkpar}\ f)$$

$(f\ 0)$	$(f\ 1)$	$\cdots$	$(f\ (p-1))$

$$\text{BSP cost} : \max_{0 \leq i < p} w_i$$

Point-wise parallel application

Point-wise parallel application

$$\textbf{apply} : (\alpha \rightarrow \beta)\ \textbf{par} \rightarrow \alpha\ \textbf{par} \rightarrow \beta\ \textbf{par}$$

Point-wise parallel application

$$\textbf{apply} : (\ \alpha \rightarrow \ \beta\ )\ \textbf{par} \rightarrow \ \alpha\ \textbf{par} \rightarrow \ \beta\ \textbf{par}$$

apply

f_0	f_1	$\cdots$	f_{p-1}

v_0	v_1	$\cdots$	v_{p-1}

$=$

$(f_0\ \ v_0)$	$(f_1\ \ v_1)$	$\cdots$	$(f_{p-1}\ \ v_{p-1})$

Point-wise parallel application

$$\textbf{apply} : (\ \alpha \rightarrow\ \beta\ )\ \textbf{par} \rightarrow\ \alpha\ \textbf{par} \rightarrow\ \beta\ \textbf{par}$$

$$\left(\textbf{apply}\ \begin{array}{|c|c|c|c|}\hline f_0 & f_1 & \cdots & f_{p-1} \\ \hline\end{array}\ \begin{array}{|c|c|c|c|}\hline v_0 & v_1 & \cdots & v_{p-1} \\ \hline\end{array}\right)$$

$$= \begin{array}{|c|c|c|c|}\hline (f_0\ \ v_0) & (f_1\ \ v_1) & \cdots & (f_{p-1}\ \ v_{p-1}) \\ \hline\end{array}$$

$$\text{BSP cost} : \max_{0 \leq i < p} w_i$$

Communication

Communication

$$\textbf{type}\ \ \alpha\ \text{option} = \text{None} \mid \text{Some}\ \textbf{of}\ \ \alpha$$
$$\textbf{put}\text{: } (\text{int} \to\ \alpha\ \text{option})\ \textbf{par} \to (\text{int} \to\ \alpha\ \text{option})\ \textbf{par}$$

Communication

$$\textbf{type} \;\; \alpha \text{ option} = \text{None} \mid \text{Some } \textbf{of} \;\; \alpha$$

$$\textbf{put}\text{: } (\text{int} \to \;\alpha \text{ option})\ \textbf{par} \to (\text{int} \to \;\alpha \text{ option})\ \textbf{par}$$

$$\left(\textbf{put} \quad \boxed{f_0 \mid f_1 \mid \cdots \mid f_{p-1}}\right) \quad = \quad \boxed{g_0 \mid g_1 \mid \cdots \mid g_{p-1}}$$

$$\textbf{type}\ \ \alpha\ \text{option} = \text{None} \mid \text{Some}\ \textbf{of}\ \ \alpha$$
$$\textbf{put}\text{: } (\text{int} \rightarrow\ \alpha\ \text{option})\ \textbf{par} \rightarrow (\text{int} \rightarrow\ \alpha\ \text{option})\ \textbf{par}$$

$$\left(\textbf{put}\ \boxed{f_0 \mid f_1 \mid \cdots \mid f_{p-1}}\right) = \boxed{g_0 \mid g_1 \mid \cdots \mid g_{p-1}}$$

0	1	2	3
None	$(f_1\ 0)$	$(f_2\ 0)$	$(f_3\ 0)$
None	$(f_1\ 1)$	$(f_2\ 1)$	$(f_3\ 1)$
None	$(f_1\ 2)$	$(f_2\ 2)$	$(f_3\ 2)$
Some v	$(f_1\ 3)$	$(f_2\ 3)$	$(f_3\ 3)$

$\Rightarrow$

0	1	2	3
None	None	None	Some v
$(g_0\ 1)$	$(g_1\ 1)$	$(g_2\ 1)$	$(g_3\ 1)$
$(g_0\ 2)$	$(g_1\ 2)$	$(g_2\ 2)$	$(g_3\ 2)$
$(g_0\ 3)$	$(g_1\ 3)$	$(g_2\ 3)$	$(g_3\ 3)$

type α option = None | Some **of** α
put: (int $\rightarrow$ α option) **par** $\rightarrow$ (int $\rightarrow$ α option) **par**

$$\left(\textbf{put}\ \ \boxed{f_0 \mid f_1 \mid \cdots \mid f_{p-1}}\right) = \boxed{g_0 \mid g_1 \mid \cdots \mid g_{p-1}}$$

0	1	2	3
None	(f_1 0)	(f_2 0)	(f_3 0)
None	(f_1 1)	(f_2 1)	(f_3 1)
None	(f_1 2)	(f_2 2)	(f_3 2)
Some v	(f_1 3)	(f_2 3)	(f_3 3)

$\Rightarrow$

0	1	2	3
None	None	None	Some v
(g_0 1)	(g_1 1)	(g_2 1)	(g_3 1)
(g_0 2)	(g_1 2)	(g_2 2)	(g_3 2)
(g_0 3)	(g_1 3)	(g_2 3)	(g_3 3)

$$\text{BSP cost} : \max_{0 \leq i < p} w_i + h \times g + L$$

Projection

Projection

proj: α option **par** $\longrightarrow$ (int $\longrightarrow$ α option)

Projection

$$\text{proj}: \ \alpha \text{ option } \textbf{par} \rightarrow (\text{int} \rightarrow \ \alpha \text{ option})$$

$$\left(\text{proj} \ \boxed{v_0 \mid v_1 \mid \cdots \mid v_{p-1}}\right) = \begin{array}{llll} \textbf{function} & 0 & \rightarrow & v_0 \\ \mid & 1 & \rightarrow & v_1 \\ & \vdots & & \\ \mid & p-1 & \rightarrow & v_{p-1} \end{array}$$

Projection

proj: α option **par** $\rightarrow$ (int $\rightarrow$ α option)

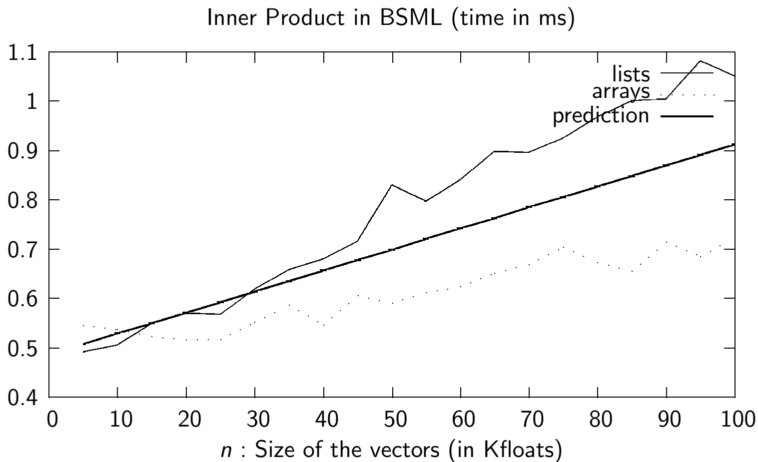
$$\left(\text{proj} \quad \boxed{v_0 \mid v_1 \mid \cdots \mid v_{p-1}}\right) = \begin{array}{lll} \textbf{function} & 0 \rightarrow & v_0 \\ \mid & 1 \rightarrow & v_1 \\ & \vdots & \\ \mid & p-1 \rightarrow & v_{p-1} \end{array}$$

$$\text{BSP cost} : \max_{0 \leq i < p} w_i + h \times g + L$$

```
let noSome (Some x) = x

let procs () = [0;1;2;...;bsp_p()−1] (* Pseudo code *)

let replicate x = mkpar(fun pid→ x)

let parfun f vec = apply (replicate f) vec

let parfun2 f vec1 vec2 = apply (parfun f vec1) vec2

let _ =
  let print = fun i n → Printf.printf "Processor %d of %d\n" i n in
  parfun2 print (mkpar(fun pid→ pid)) (replicate (bsp_p()))
```

```
(* totex: α par → (int → α ) par *)
let totex vv = parfun (compose noSome)
  (put(parfun (fun v dst→ Some v) vv))

(* total_exchange: α par → α list par *)
let total_exchange vec =
  parfun2 List.map (totex vec) (replicate (procs()))

(* fold_direct: ( α → α → α ) → α par → α par *)
let fold_direct op vec =
  let local_reduce = function h::t→ List.fold_left op h t in
  parfun local_reduce (total_exchange vec)
```

```
let inprod_array v1 v2 = let s = ref 0. in
    for i = 0 to (Array.length v1)−1 do
       s:=!s+.(v1.(i)*.v2.(i));
    done; !s

let inprod_list v1 v2 =
    List.fold_left2 (fun s x y→ s+.x*.y) 0. v1 v2

let inprod seqinprod v1 v2 =
  let local_inprod = parfun2 seqinprod v1 v2 in
  fold_direct (+.) local_inprod
```


Inner Product in BSML (time in ms)
lists
arrays
prediction
1.1
1
0.9
0.8
0.7
0.6
0.5
0.4
0
10
20
30
40
50
60
70
80
90
100
n : Size of the vectors (in Kfloats)

Overview

Overview

- Two parts :
  - Primitives
  - Standard library
- Modular implementation :
  - The module of primitives is a functor
  - Low-level communication module (MPI, PUB, TCP/IP, SEQ)
  - Module for input/output (PAR, SEQ)
  - Module for parallel composition (Generic)

Conclusions and Future Work

Conclusions and Future Work

- Bulk Synchronous Parallel ML is :
  - Simple
  - Efficient
  - Predictable

Conclusions and Future Work

- Bulk Synchronous Parallel ML is :
  - Simple
  - Efficient
  - Predictable
  - Partially certified

Conclusions and Future Work

- Bulk Synchronous Parallel ML is :
  - Simple
  - Efficient
  - Predictable
  - Partially certified
- Current implementation : 0.25 (MPI)
- Next release : 0.5 (soon)

Conclusions and Future Work

- Bulk Synchronous Parallel ML is :
  - Simple
  - Efficient
  - Predictable
  - Partially certified
- Current implementation : 0.25 (MPI)
- Next release : 0.5 (soon)

`http://bsmllib.free.fr`