Multi-ML: Programming
Multi-BSP Algorithms in ML

Victor Allombert, Frédéric Gava and Julien Tesson

Laboratory of Algorithmic Complexity and Logic
Université Paris-Est

Siam PP16 - Paris
Table of Contents

1 Introduction
2 Multi-ML
3 Results
4 Conclusion
Table of Contents

1 Introduction
   OCAML
   BSML
   MULTI-BSP

2 Multi-ML

3 Results

4 Conclusion
Ocaml : a ML language

Strengths of Ocaml

- A functionnal programming language
- A powerful type system
- User-definable algebraic data types and pattern matching
- Automatic memory management
- Efficient native code compilers
# let f = fun x -> "Hello "^(string_of_int x) in
  let lst = [0;1;2] in
  List.map f lst;;
- : string list = ["Hello 0"; "Hello 1"; "Hello 2"]

# let pair = ([0;1;2],true);;
val pair : int list * bool = ([0; 1; 2], true)

# type 'a list =
  Nil
  | Node of 'a*'a list ;;
type 'a list = Nil | Node of 'a * 'a list
Bulk Synchronous ML

What is BSML?

• Explicit BSP programming with a functional approach
• Based upon ml implemented over ocaml
• Formal semantics → computer-assisted proofs (coq)

Main idea
Parallel data structure ⇒ Vector:
What is BSML?

- Explicit BSP programming with a functional approach
Bulk Synchronous ML

What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML
- Formal semantics $\rightarrow$ computer-assisted proofs (COQ)
**What is BSML?**

- Explicit **BSP** programming with a functional approach
- Based upon **ML** an implemented over **OCAML**
- Formal semantics $\rightarrow$ computer-assisted proofs (**COQ**)

**Main idea**

Parallel data structure $\Rightarrow$ Vector:

![Diagram of parallel data structure](image-url)
BSML primitives

Asynchronous primitives
Asynchronous primitives

- $\langle e, \ldots, e \rangle$
Asynchronous primitives

- \( << e >> \) : \( \langle e, \ldots, e \rangle \)
- \( v \) : \( v_i \) on processor \( i \), assumes \( v \equiv \langle v_0, \ldots, v_{p-1} \rangle \)

Synchronous primitives

- \( \text{proj} \) : \( \langle x_0, \ldots, x_{p-1} \rangle \mapsto \langle \text{fun} \, i \rightarrow x_i \rangle \)
- \( \text{put} \) : \( \langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle \text{fun} \, i \rightarrow f_i \rangle, \ldots, \langle \text{fun} \, i \rightarrow f_{p-1} \rangle \)
BSML primitives

Asynchronous primitives

- $<< e >>$: $\langle e, \ldots, e \rangle$
- $\$v\$: $v_i$ on processor $i$, assumes $v \equiv \langle v_0, \ldots, v_{p-1} \rangle$
- $\$pid\$: $i$ on processor $i$
BSML primitives

Asynchronous primitives

- \( \langle e, \ldots, e \rangle \)
- \( v_i \) on processor \( i \), assumes \( v \equiv \langle v_0, \ldots, v_{p-1} \rangle \)
- \( pid \) on processor \( i \)

Synchronous primitives

- \( \text{proj} : \langle x_0, \ldots, x_{p-1} \rangle \mapsto (\text{fun } i \rightarrow x_i) \)
BSML primitives

Asynchronous primitives

- \( << e >> : \langle e, \ldots, e \rangle \)
- \( \$v\$: \( v \) on processor \( i \), assumes \( v \equiv \langle v_0, \ldots, v_{p-1} \rangle \)
- \( \$pid\$: \( i \) on processor \( i \)

Synchronous primitives

- \( \text{proj} : \langle x_0, \ldots, x_{p-1} \rangle \mapsto (\text{fun } i \rightarrow x_i) \)
- \( \text{put} : \langle f_0, \ldots, f_{p-1} \rangle \mapsto \langle (\text{fun } i \rightarrow f_i \, 0), \ldots, (\text{fun } i \rightarrow f_i \, (p - 1)) \rangle \)
For a BSP machine with 3 processors:

```ocaml
# let vec = << "Hello" >>;;
val vec : string par = <"Hello", "Hello", "Hello">

# let vec2 = << $vec$^(string_of_int $pid$) >>;;
val vec2 : string par = <"Hello0", "Hello1", "Hello2">

# let totex v = List.map (proj v) procs;;
val totex : 'a par -> 'a list = <fun>

# totex vec2;;
- : string list = ["Hello0"; "Hello1"; "Hello2"]
```
The multi-BSP model

What is multi-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
The MULTI-BSP model

What is MULTI-BSP?

1. A tree structure with nested components
The **MULTI-BSP** model

**What is MULTI-BSP?**

1. A tree structure with nested components
2. Where nodes have a storage capacity
The MULTI-BSP model

What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
The **MULTI-BSP** model

**What is MULTI-BSP?**

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors

---

**MULTI-BSP**

```
Multi_Core
```

```
<table>
<thead>
<tr>
<th>Core0</th>
<th>Core1</th>
</tr>
</thead>
<tbody>
<tr>
<td>th0</td>
<td>th0</td>
</tr>
<tr>
<td>th1</td>
<td>th1</td>
</tr>
<tr>
<td>th2</td>
<td>th2</td>
</tr>
<tr>
<td>th3</td>
<td>th3</td>
</tr>
</tbody>
</table>
```

**BSP**

```
Network
```

```
<table>
<thead>
<tr>
<th>Core0</th>
<th>Core1</th>
</tr>
</thead>
<tbody>
<tr>
<td>th0</td>
<td>th0</td>
</tr>
<tr>
<td>th1</td>
<td>th1</td>
</tr>
<tr>
<td>th2</td>
<td>th2</td>
</tr>
<tr>
<td>th3</td>
<td>th3</td>
</tr>
<tr>
<td>th4</td>
<td>th4</td>
</tr>
<tr>
<td>th5</td>
<td>th5</td>
</tr>
<tr>
<td>th6</td>
<td>th6</td>
</tr>
<tr>
<td>th7</td>
<td>th7</td>
</tr>
</tbody>
</table>
```
The MULTI-BSP model

Execution model

A level $i$ superstep is:

- Level $i - 1$ executes code independently
- Exchanges information with the $m_i$ memory
- Synchronises Level $i$ and $i - 1$
The **MULTI-BSP** model

**Execution model**

A level $i$ superstep is:

- Level $i - 1$ executes code independently
The **MULTI-BSP** model

**Execution model**

A level $i$ superstep is:

- Level $i - 1$ executes code independently
- Exchanges informations with the $m_i$ memory
The **MULTI-BSP** model

### Execution model

A level $i$ superstep is:

- Level $i - 1$ executes code independently
- Exchanges informations with the $m_i$ memory
- Synchronises
Table of Contents

1 Introduction

2 Multi-ML
   Overview
   Primitives
   Semantics
   Typing
   Implementation

3 Results

4 Conclusion
Basic ideas
Basic ideas

- BSML-like code on every stage of the MULTI-BSP architecture
Basic ideas

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
Basic ideas

- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming
- *Multi-functions* that recursively go through the tree.
Basic ideas

• BSML-like code on every stage of the MULTI-bsp architecture
• Specific syntax over ML: eases programming
• *Multi-functions* that recursively go through the tree.

```ml
let v = <<e>>

<< e ... e >>
```
Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ...
    in v
  where leaf =
    (* OCaml code *)
    ...
    in v
```
**MULTI-ML: Tree recursion**

Recursion structure

```ocaml
let multi f [args]=
  where node =
  (* BSML code *)
  ...  
  << f [args] >>
  ... in v
  where leaf =
  (* OCaml code *)
  ... in v
```

Result

![Tree diagram](image)
Multi-ML: Tree recursion

Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ...
    in v
  where leaf =
    (* OCaml code *)
    ...
    in v
```

Result: $v_0$
Recursion structure

\[
\text{let multi } f \ [\text{args}] = \\
\text{  where node } = \\
\text{    (* BSML code *)} \\
\text{    ...} \\
\text{    }<< f \ [\text{args}] >> \\
\text{    ... in v} \\
\text{  where leaf } = \\
\text{    (* OCaml code *)} \\
\text{    ... in v}
\]

MULTI-ML: Tree recursion
MULTI-ML: Tree recursion

Recursion structure

```ml
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    ... in v
  where leaf =
    (* OCaml code *)
    ... in v
```

Result $v$.
MULTI-ML: Tree recursion

Recursion structure

```ocaml
let multi f [args] =
    where node =
        (* BSML code *)
        ...
        << f [args] >>
        ...
        in v
    where leaf =
        (* OCaml code *)
        ...
        in v
```

![Diagram of tree recursion]

Result

v_{0}0.
0.
0.
1.
1.
1.
0.
1.
1.
0.
V. Allombert et al.
Siam PP16 - 14 April 2016
10 / 22
**MULTI-ML: Tree recursion**

### Recursion structure

```
let multi f [args]=
  where node =
    (* BSML code *)
    ...
  ... in v
  where leaf =
    (* OCaml code *)
    ...
  ... in v
```
MULTI-ML: Tree recursion

Recursion structure

let multi f [args] =
  where node =
    (* BSML code *)
    ...
  in v
  where leaf =
    (* OCaml code *)
    ...
  in v
**MULTI-ML: Tree recursion**

Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
    ...
  <<< f [args] >>>
  ... in v
where leaf =
  (* OCaml code *)
  ...
  in v
```
MULTI-ML: Tree recursion

Recursion structure

```ocaml
let multi f [args] =
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ...
    in v
where leaf =
  (* OCaml code *)
  ...
  in v
```

![Tree diagram]
Recursion structure

```ocaml
define multi f [args]=
  where node =
    (* BSML code *)
    ...
    << f [args] >>
    ...
    in v
  where leaf =
    (* OCaml code *)
    ...
    in v
```

Result

![Tree diagram]
MULTI-ML: Tree construction

Tree construction

let multi tree f [args]=
  where node =
    (* BSML code *)
    ...
    in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ...
    in v
**MULTI-ML: Tree construction**

**Tree construction**

```
let multi tree f [args]=
  where node =
    (* BSML code *)
    ...
    in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ...
    in v
```
MULTI-ML: Tree construction

```ocaml
define multi-tree (
  f : [args]
  .
  node = (* BSML code *)
  .
  in (* OCaml code *)
  leaf =
  .
  out (f [args] >>, v)
  .
  in v
)
MULTI-ML: Tree construction

let multi tree f [args]=
  where node =
    (* BSML code *)
    ...
    in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ...
    in v
MULTI-ML: Tree construction

let multi tree f [args]=
    where node =
        (* BSML code *)
        ... in
        (<< f [args] >>, v)
    where leaf =
        (* OCaml code *)
        ... in v
MULTI-ML: Tree construction

Tree construction

```
let multi tree f [args]=
  where node =
    (* BSML code *)
    ... in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ... in v
```
**MULTI-ML: Tree construction**

**Tree construction**

```
let multi tree f [args]=
   where node =
      (* BSML code *)
      ... in
      (<< f [args] >>, v)
   where leaf =
      (* OCaml code *)
      ... in v
```

```
\[
\text{Tree:}
\begin{array}{c}
\phantom{f} \node[N] \node[N]
\end{array}
\]
```

V. Allombert et al.

Siam PP16 - 14 April 2016

11 / 22
**MULTI-ML: Tree construction**

```
let multi tree f [args]=
  where node =
    (* BSML code *)
    ... in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ... in v
```
MULTI-ML: Tree construction

let multi tree f [args] =
  where node =
    (* BSML code *)
    ...
    in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ...
    in v

Tree construction
MULTI-ML: Tree construction

Tree construction

let multi tree f [args]=
  where node =
    (* BSML code *)
    ... in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ... in v
let multi tree f [args]=
  where node =
    (* BSML code *)
    ...
    in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ...
    in v
Primitives

Summary
Primitives

Summary

- `mktree e`

```
  e
 / 
 e  e
 / 
 e  e
```
Primitives

Summary

- mktree
- gid
Primitives

Summary

- mktree
- gid
- at
Primitives

Summary

- mktree $e$
- gid
- $t$
Primitives

Summary

- mktree e
- gid
- at
- <<...f...>>
Primitives

Summary

- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
Primitives

Summary

- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`

```
x
let x = ...
```
Primitives

Summary

- \texttt{mktree e}
- \texttt{gid}
- \texttt{at}
- \texttt{<<...f...>>}
- \texttt{#x#}

\begin{center}
\begin{tikzpicture}
    \node (x) {\texttt{x}};
    \node (x1) at (x -| 0, -1) {#x#};
    \node (x2) at (x -| 0, 1) {#x#};
    \node (f1) at (x1 -| 0, 0) {\texttt{\ldots}};
    \node (f2) at (x2 -| 0, 0) {\texttt{\ldots}};
    \node (empty1) at (f1 -| -1, 0) {\texttt{\ldots}};
    \node (empty2) at (f1 -| 1, 0) {\texttt{\ldots}};
    \node (empty3) at (f2 -| -1, 0) {\texttt{\ldots}};
    \node (empty4) at (f2 -| 1, 0) {\texttt{\ldots}};
    \draw (x) -- (x1);
    \draw (x) -- (x2);
    \draw (x1) -- (f1);
    \draw (x1) -- (empty1);
    \draw (x1) -- (empty2);
    \draw (x2) -- (f2);
    \draw (x2) -- (empty3);
    \draw (x2) -- (empty4);
\end{tikzpicture}
\end{center}

\texttt{let x = \ldots}
Summary

- mktree e
- gid
- at
- ＜＜...f...＞＞
- #x#
Primitives

Summary

- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`

```
mkpar (fun i -> vi)
```

```
f 0; f 1
v0
v1
```
Primitives

Summary

- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`

```ml
mkpar (fun i -> vi)
```

```
f 0; f 1
```
Primitives

Summary

- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`
Keep the intermediate results of the sum

```ocaml
let multi tree sum_list l =  
  where node =  
    let v = mkpar (fun i -> split i l) in  
    let rc = << sum_list $v$ >> in  
    let s = sumSeq (flatten << at $rc$ >>)  
    in (rc,s)  
  where leaf =  
    sumSeq l
```

V. Allombert et al.

Siam PP16 - 14 April 2016
Code example

Keep the intermediate results of the sum

let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
  in (rc,s)
  where leaf =
    sumSeq l
Keep the intermediate results of the sum

let multi tree sum_list l =  
  where node =  
    let v = mkpar (fun i -> split i l) in  
    let rc = << sum_list $v$ >> in  
    let s = sumSeq (flatten << at $rc$ >>)  
    in (rc,s)  
  where leaf =  
    sumSeq l
**Code example**

Keep the intermediate results of the sum

```
let multi tree sum_list l =
    where node =
        let v = mkpar (fun i -> split i l) in
        let rc = << sum_list $v$ >> in
        let s = sumSeq (flatten << at $rc$ >>)
    in (rc, s)
    where leaf =
        sumSeq l
```
Let's keep the intermediate results of the sum.

```ocaml
let multi_tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>) in
    (rc, s)
  where leaf =
    sumSeq l
```
Keep the intermediate results of the sum

```ocaml
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>) in
    (rc,s)
  where leaf =
    sumSeq l
```
Keep the intermediate results of the sum

```
let multi tree sum_list l =  
  where node =  
    let v = mkpar (fun i -> split i l) in  
    let rc = << sum_list $v$ >> in  
    let s = sumSeq (flatten << at $rc$ >>)  
    in (rc,s)  
  where leaf =  
    sumSeq l
```
Keep the intermediate results of the sum

```
let multi_tree sum_list l =
    where node =
        let v = mkpar (fun i -> split i l) in
        let rc = sum_list $v$ in
        let s = sumSeq (flatten << at $rc$ >>) in
        (rc, s)
    where leaf =
        sumSeq l
```
Keep the intermediate results of the sum

let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in (rc,s)

where leaf =
  sumSeq l
Semantics

Formal definition of a core-language

Useful for:
- Study of properties
- Proof of programs/compiler/typing rules

Currently
- Inductive big-step: confluent
- Co-inductive: mutually exclusive
Purely Constraint-Based system: PCB(X)

- Constraint based
- Extension of DM’s type system
- Easy to extend
- Related to HM(X)

MULTI-ML type extension

- Add parallel constructions
- Introduce localities using effects (s, ℓ, b and m)
- Control parallel structure imbrications
Type localities
Type localities

$m$
Type localities

$m$

$b$

V. Allombert et al.
Type localities
Type localities

$m$

$b$

$s$
Type localities

\[ m \]

\[ b \]

\[ s \]

\[ c \]
Type localities

\( m \)

\( b \)

\( s \)

\( c \)

\( \ell \)
Type localities

Accessibility: ▽

Definability: ▲
Tagged type

\( \tau ::= \alpha_\pi \)

- type variable
- Base_\pi base type
- \((\tau_\pi \rightarrow \tau_\pi)_\pi\) arrow type
- \((\tau_\pi, \tau_\pi)_\pi\) pairs
- \(\tau_\pi \text{Par}_b\) parallel vector
- \(\tau_\pi \text{Tree}_\pi\) tree

Latent effect

\( f : (\text{int}_a \rightarrow \text{int}_c \text{par}_b)_m \)
Implementation

Sequential simulator

- OCAML-like toplevel
- Test and debug
- Tree structure
- Hash tables to represent memories

```ocaml
#let multi tree f n =
   where node =
     let r =<<f ($pid$ + #n# + 1) >> in
     (r,(gid^"=>"^n))
   where leaf=
     (gid^"=>"^n);;

- : val f : int -> string tree = <multi-fun>
# (f 0)
  o "0->0"
  | --o "0.0->1"
  | |--o "0.0.0-> 2"
  | |--o "0.0.1-> 3"
  --o "0.1->2"
  | |-- "0.1.0-> 3"
  | |-- "0.1.1-> 4"
```
Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories
## Distributed implementation

### Our approach

- Modular
- Generic functors
- Communication routines
- Portable on shared and distributed memories

### Current version

- Based on **MPI**
- **SPMD**
- One process for each nodes/leaves
- Distributed over physical cores
- Shared/Distributed memory optimisations
# Table of Contents

1. Introduction
2. Multi-ML
3. Results
4. Conclusion
Benchmarks

Naive Eratosthenes algorithm

- \( \sqrt{n} \)th first prime numbers
- Based on scan
- Unbalanced
Naive Eratosthenes algorithm

- $\sqrt{n}$th first prime numbers
- Based on scan
- Unbalanced

Benchmarks

Mirev 3
Naive Eratosthenes algorithm

- $\sqrt{(n)}$th first prime numbers
- Based on scan
- Unbalanced

Benchmarks

Results

<table>
<thead>
<tr>
<th></th>
<th>100_000</th>
<th>500_000</th>
<th>1_000_000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MULTI-ML</td>
<td>BSML</td>
<td>MULTI-ML</td>
</tr>
<tr>
<td>8</td>
<td>0.7</td>
<td>1.8</td>
<td>22.4</td>
</tr>
<tr>
<td>64</td>
<td>0.3</td>
<td>0.3</td>
<td>1.3</td>
</tr>
<tr>
<td>128</td>
<td>0.5</td>
<td>0.45</td>
<td>2.1</td>
</tr>
</tbody>
</table>
# Table of Contents

1. Introduction
2. Multi-ML
3. Results
4. Conclusion
Conclusion

MULTI-ML
## Conclusion

### MULTI-ML

- Recursive multi-functions
Conclusion

**MULTI-ML**

- Recursive multi-functions
- Structured nesting of BSML codes
Conclusion

**MULTI-ML**

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
## Conclusion

### MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Type system
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confluent)
- Type system
- Small number of primitives and little syntax extension

Current/Future work
## Conclusion

### MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Type system
- Small number of primitives and little syntax extension

### Current/Future work

- Optimise MPI implementation
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (confuent)
- Type system
- Small number of primitives and little syntax extension

Current/Future work

- Optimise MPI implementation
- Type system for MULTI-ML
Conclusion

MULTI-ML

- Recursive multi-functions
- Structured nesting of BSML codes
- Big-steps formal semantics (consequent)
- Type system
- Small number of primitives and little syntax extension

Current/Future work

- Optimise MPI implementation
- Type system for MULTI-ML
- Real life benchmarks
Merci !

Any questions?