Multi-ML: Programming Multi-BSP Algorithms in ML

Victor Allombert, Frédéric Gava and Julien Tesson

Laboratory of Algorithmic Complexity and Logic
Université Paris-Est

Journées du GDR GPL - Besançon

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

Syntaxe overview

```
# 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
```

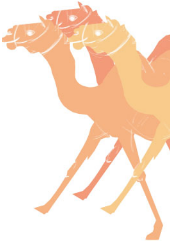
Bulk Synchronous ML

What is BSML?

Bulk Synchronous ML

What is BSML?

- Explicit BSP programming with a functional approach



What is BSML?

- Explicit BSP programming with a functional approach
- Based upon ML an implemented over OCAML

Bulk Synchronous ML

What is BSML?

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

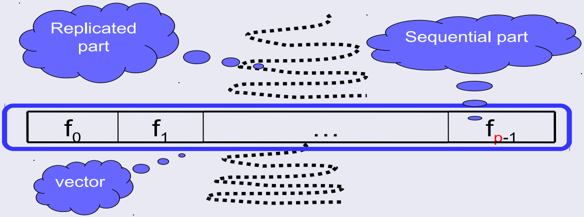
Bulk Synchronous ML

What is BSML?

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

Main idea

Parallel data structure ⇒ Vector:



BSML primitives

Asynchronous primitives

BSML primitives

Asynchronous primitives

- `<< e >>` : $\langle e, \ldots, e \rangle$

BSML primitives

Asynchronous primitives

- `<< e >>`: $\langle \mathsf{e}, \ldots, \mathsf{e} \rangle$
- `$v$`: v_i on processor *i*, assumes $\mathrm{v} \equiv \langle \mathrm{v}_0, \ldots, \mathrm{v}_{\mathbf{p}-1} \rangle$

BSML primitives

Asynchronous primitives

- `<< e >>` : $\langle \mathsf{e}, \ldots, \mathsf{e} \rangle$
- `$v$` : v_i on processor *i*, assumes $\mathrm{v} \equiv \langle \mathrm{v}_0, \ldots, \mathrm{v}_{\mathbf{p}-1} \rangle$
- `$pid$` : *i* on processor *i*

BSML primitives

Asynchronous primitives

- `<< e >>` : $\langle \mathrm{e}, \ldots, \mathrm{e} \rangle$
- `$v$` : v_i on processor i, assumes $\mathrm{v} \equiv \langle \mathrm{v}_0, \ldots, \mathrm{v}_{\mathbf{p}-1} \rangle$
- `$pid$` : i on processor i

Synchronous primitives

- `proj` : $\langle x_0, \ldots, x_{\mathbf{p}-1} \rangle \mapsto (\textbf{fun } i \rightarrow x_i)$

BSML primitives

Asynchronous primitives

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

Synchronous primitives

- `proj` : $\langle x_0, \ldots, x_{\mathbf{p}-1} \rangle \mapsto (\textbf{fun } i \to x_i)$
- `put` : $\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle (\textbf{fun } i \to f_i\ 0), \ldots, (\textbf{fun } i \to f_i\ (\mathbf{p}-1)) \rangle$

Code example

For a BSP machine with 3 processors:

```
# 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?

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?

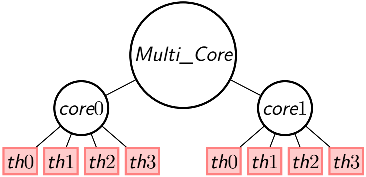
1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors

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

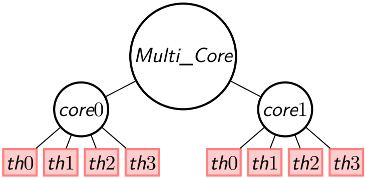


The MULTI-BSP model

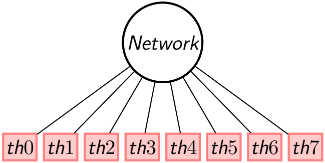
What is MULTI-BSP?

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

MULTI-BSP



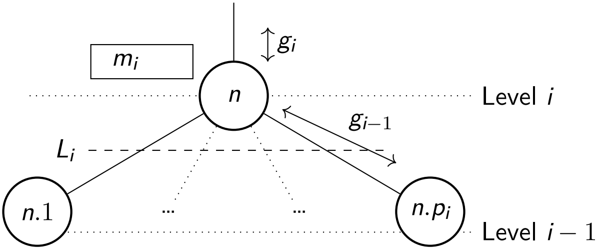
BSP



The MULTI-BSP model

Execution model

A level i superstep is:

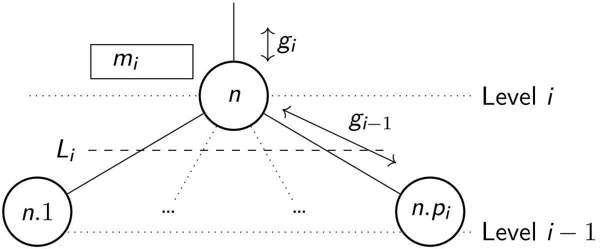


The MULTI-BSP model

Execution model

A level i superstep is:

- Level $i-1$ executes code independantly

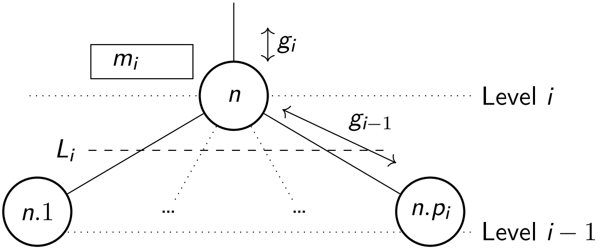


The MULTI-BSP model

Execution model

A level i superstep is:

- Level $i-1$ executes code independantly
- Exchanges informations with the m_i memory

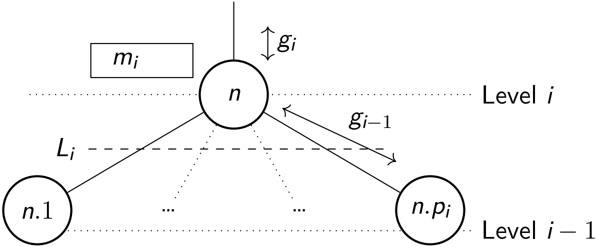


The MULTI-BSP model

Execution model

A level i superstep is:

- Level $i-1$ executes code independantly
- Exchanges informations with the m_i memory
- Synchronises



The MULTI-BSP model

Cost model

- L : Tree levels
- N : Supersteps
- $h_{k,i}$: Max of h-relations within the i^{th} superstep at level k
- $w_{k,i}$: Max of work within the i^{th} superstep at level k

MULTI-BSP cost

$$\sum_{k=0}^{L-1} (\sum_{i=0}^{N_k-1} w_{k,i} + h_{k,i} g_k + l_k)$$

Table of Contents

1 Introduction

2 Multi-ML
   - Overview
   - Primitives
   - Semantics
   - Typing
   - Implementation

3 Results

4 Conclusion

MULTI-ML

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

MULTI-ML

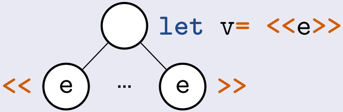
Basic ideas

- 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.

MULTI-ML

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.



MULTI-ML: Tree recursion

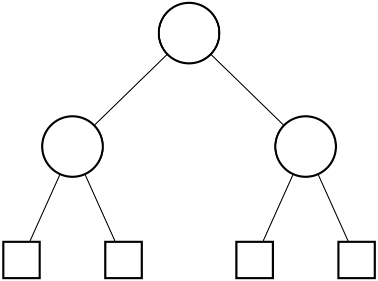
Recursion structure

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

MULTI-ML: Tree recursion

Recursion structure

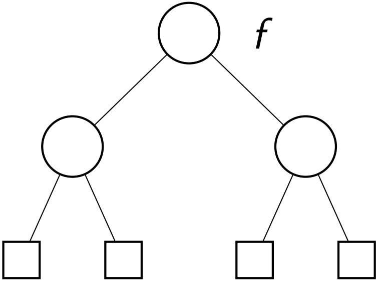
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

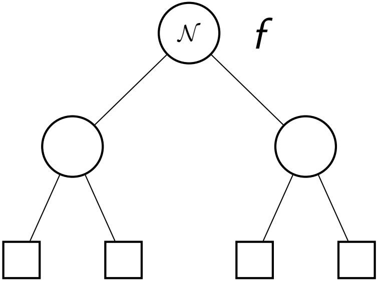
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

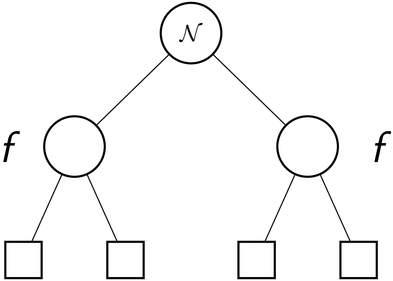
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

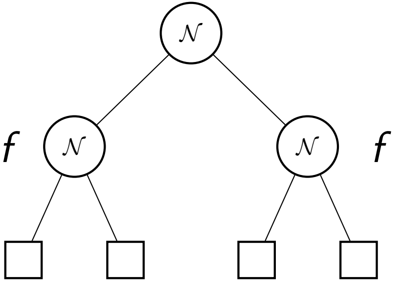
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

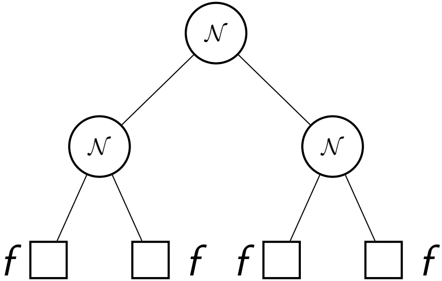
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

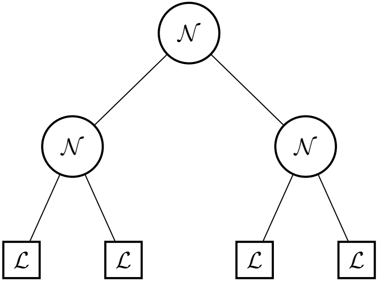
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

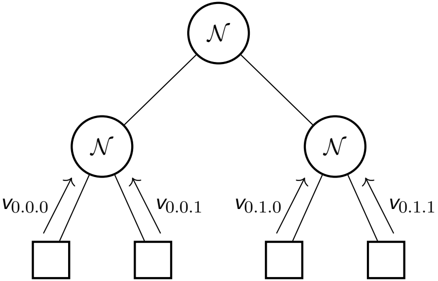
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

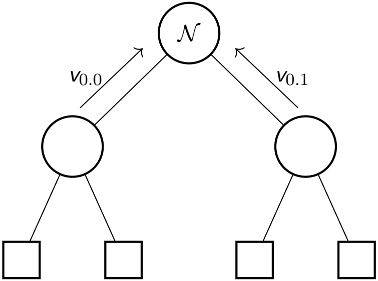
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

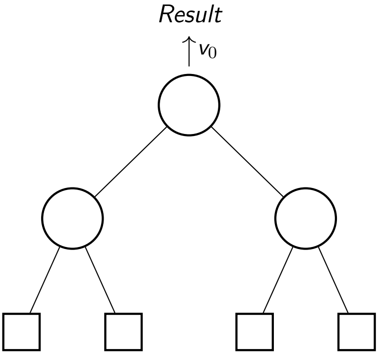
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

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



MULTI-ML: Tree construction

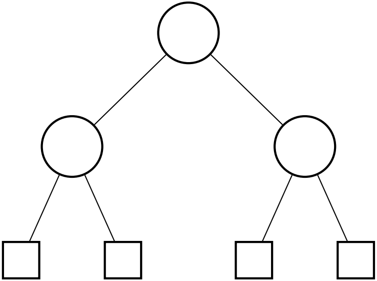
Tree construction

```
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

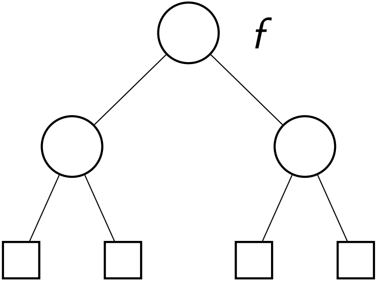
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   (<< f [args] >>, v)
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

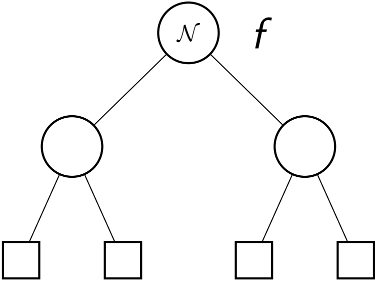
```
let multi tree f [args]=
  where node =
    (* BSML code *)
    ... in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree construction

Tree construction

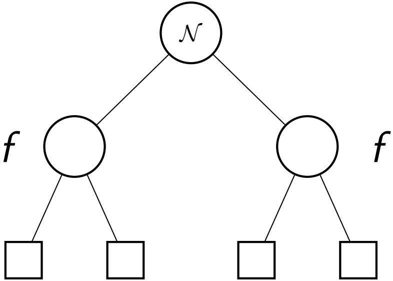
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   (<< f [args] >>, v)
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

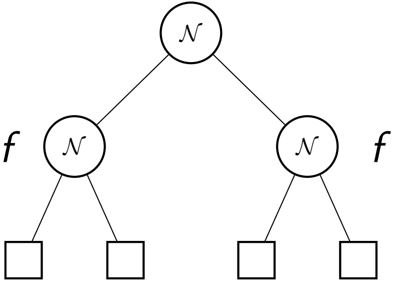
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   (<< f [args] >>, v)
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

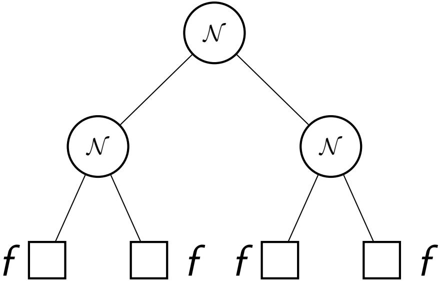
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   (<< f [args] >>, v)
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

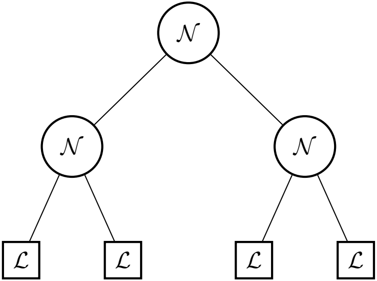
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   (<< f [args] >>, v)
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

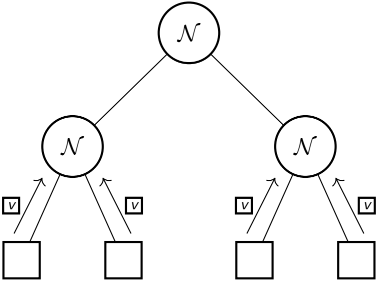
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   (<< f [args] >>, v)
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree construction

Tree construction

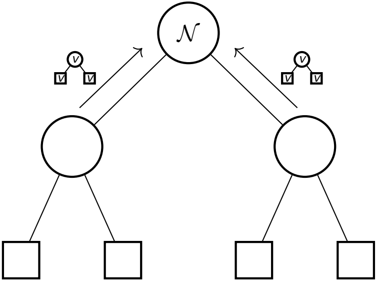
```
let multi tree f [args]=
  where node =
    (* BSML code *)
    ... in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ... in v
```



MULTI-ML: Tree construction

Tree construction

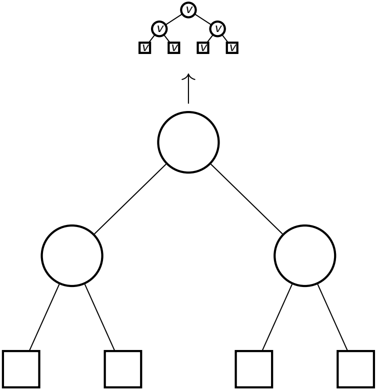
```
let multi tree f [args]=
  where node =
    (* BSML code *)
    ... in
    (<< f [args] >>, v)
  where leaf =
    (* OCaml code *)
    ... in v
```



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
```



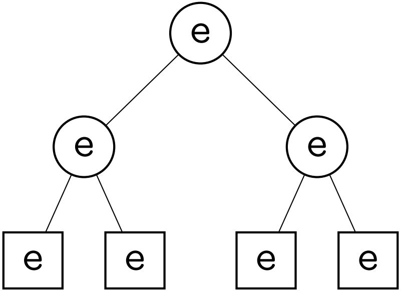
Primitives

Summary

Primitives

Summary

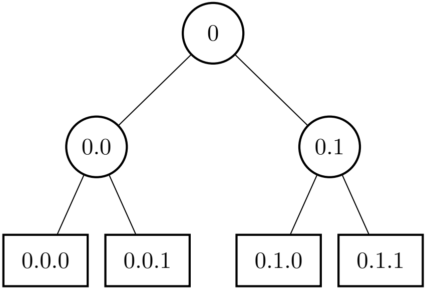
- `mktree` e



Primitives

Summary

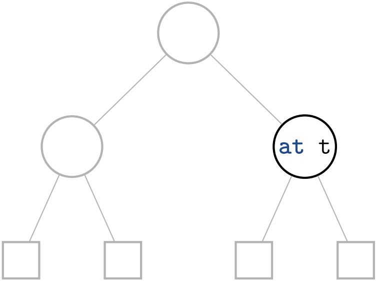
- `mktree e`
- `gid`



Primitives

Summary

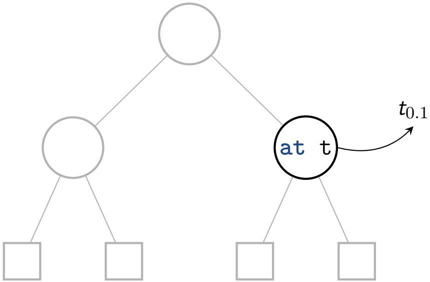
- `mktree e`
- `gid`
- `at`



Primitives

Summary

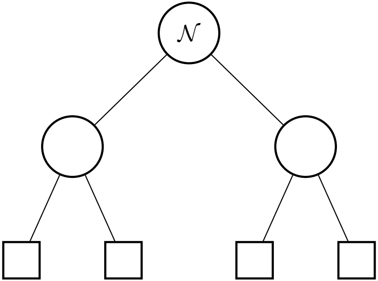
- `mktree e`
- `gid`
- `at`



Primitives

Summary

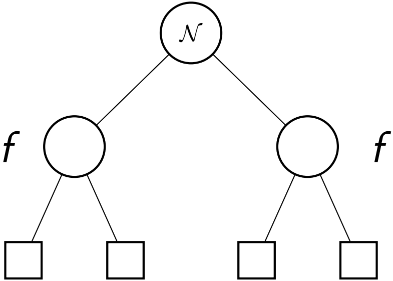
- **mktree** e
- **gid**
- **at**
- <<...f...>>



Primitives

Summary

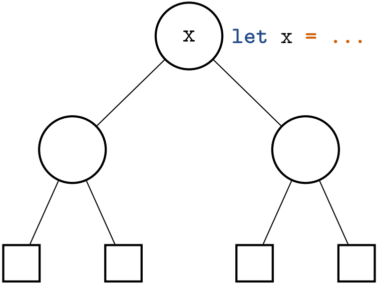
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`



Primitives

Summary

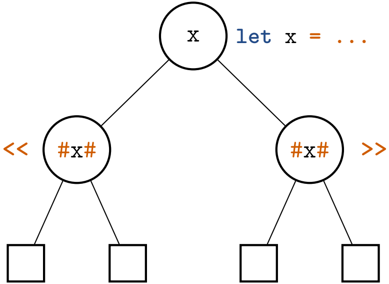
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`



Primitives

Summary

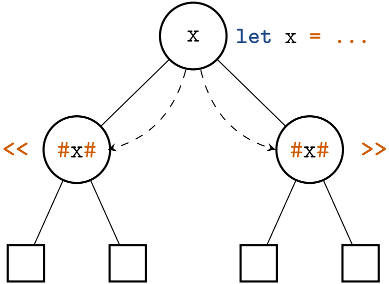
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`



Primitives

Summary

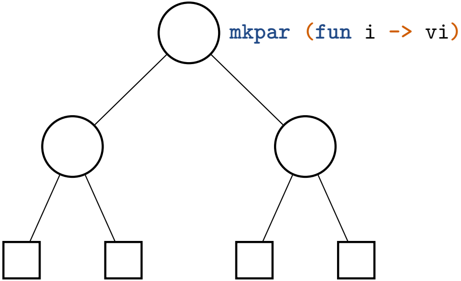
- mktree e
- gid
- at
- <<...f...>>
- #x#



Primitives

Summary

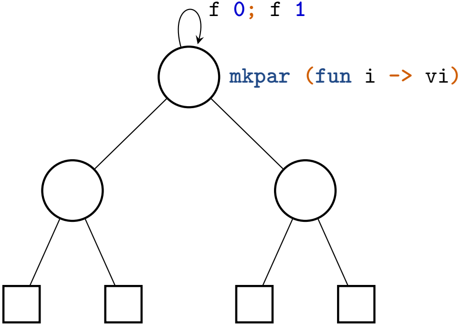
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`



Primitives

Summary

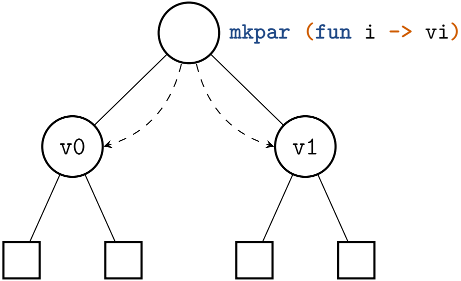
- `mktree e`
- `gid`
- `at`
- `<<...f...>>`
- `#x#`
- `mkpar f`



Primitives

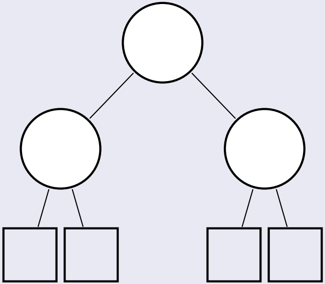
Summary

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



Code example

Keep the intermediate results of the sum

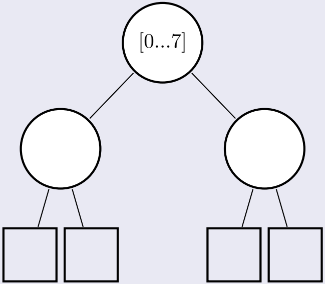


```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in (rc,s)
  where leaf =

    sumSeq l
```

Code example

Keep the intermediate results of the sum

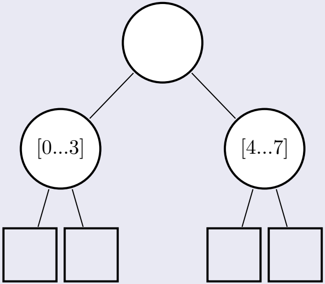


```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i -> split i l) in
      let rc = << sum_list $v$ >> in
      let s = sumSeq (flatten << at $rc$ >>)
      in (rc,s)
   where leaf =

      sumSeq l
```

Code example

Keep the intermediate results of the sum

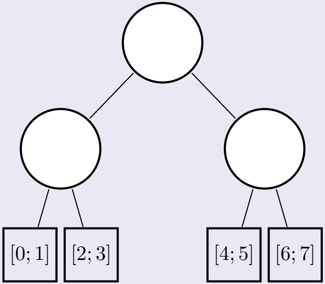


```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in (rc,s)
  where leaf =

    sumSeq l
```

Code example

Keep the intermediate results of the sum

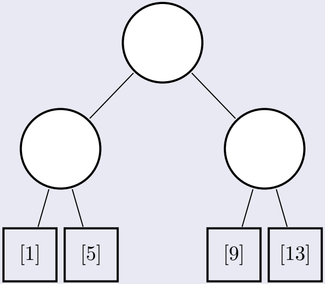


```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in (rc,s)
  where leaf =

    sumSeq l
```

Code example

Keep the intermediate results of the sum

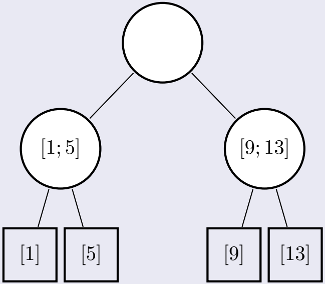


```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in (rc,s)
  where leaf =

    sumSeq l
```

Code example

Keep the intermediate results of the sum

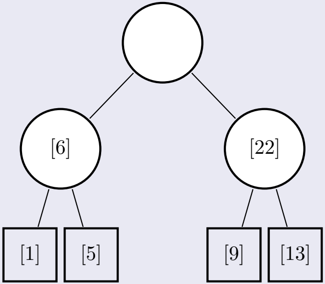


```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in (rc,s)
  where leaf =

    sumSeq l
```

Code example

Keep the intermediate results of the sum

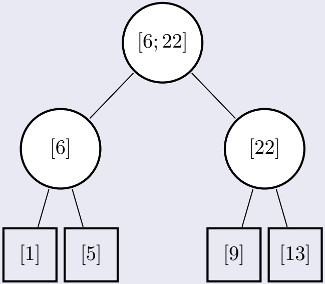


```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in (rc,s)
  where leaf =

    sumSeq l
```

Code example

Keep the intermediate results of the sum

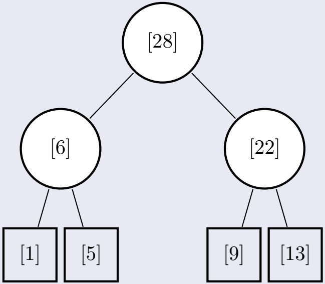


```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i -> split i l) in
    let rc = << sum_list $v$ >> in
    let s = sumSeq (flatten << at $rc$ >>)
    in (rc,s)
  where leaf =

    sumSeq l
```

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
```

Semantics

Formal definition of a core-languge

Useful for:

- Study of properties
- Proof of programs/compiler/typing rules

Currently

- Inductive big-step: confluent
- Co-inductive: mutually exclusive

Typing

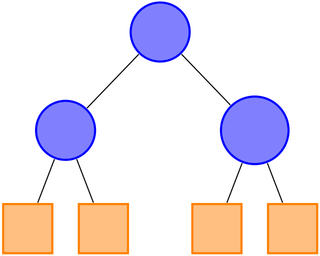
Purely **C**onstraint-**B**ased system : **PCB(X)**

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

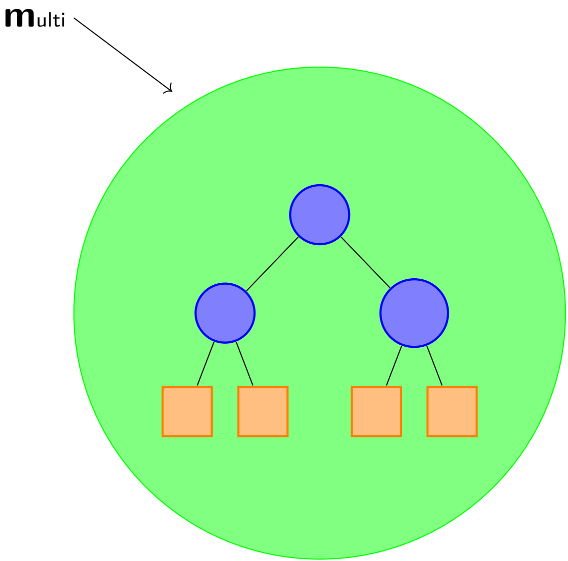
MULTI-ML type extension

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

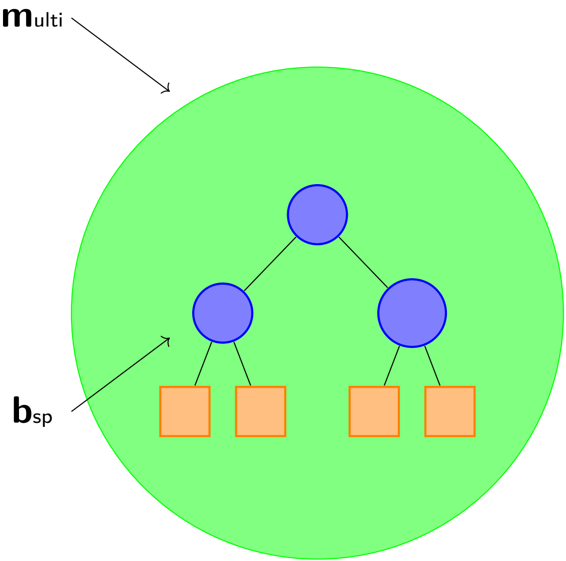
Type localities



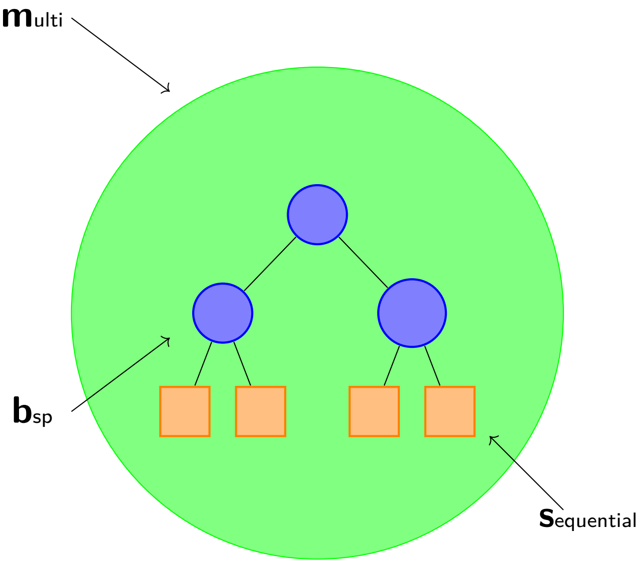
Type localities



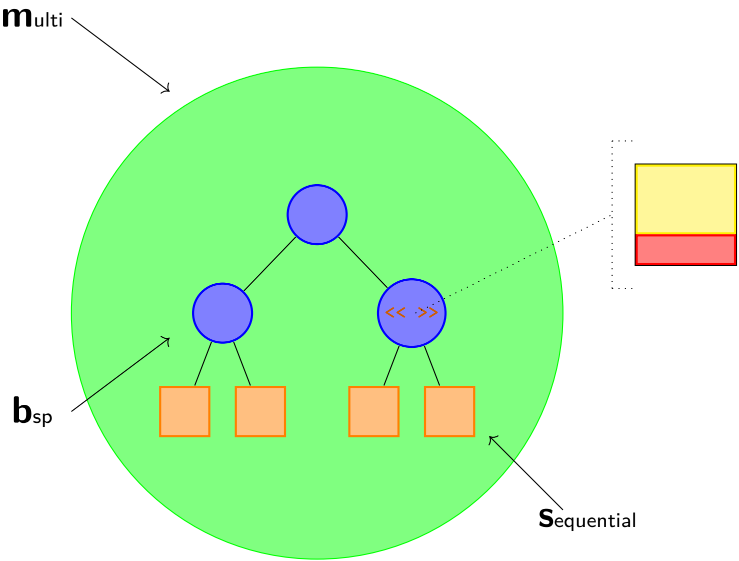
Type localities



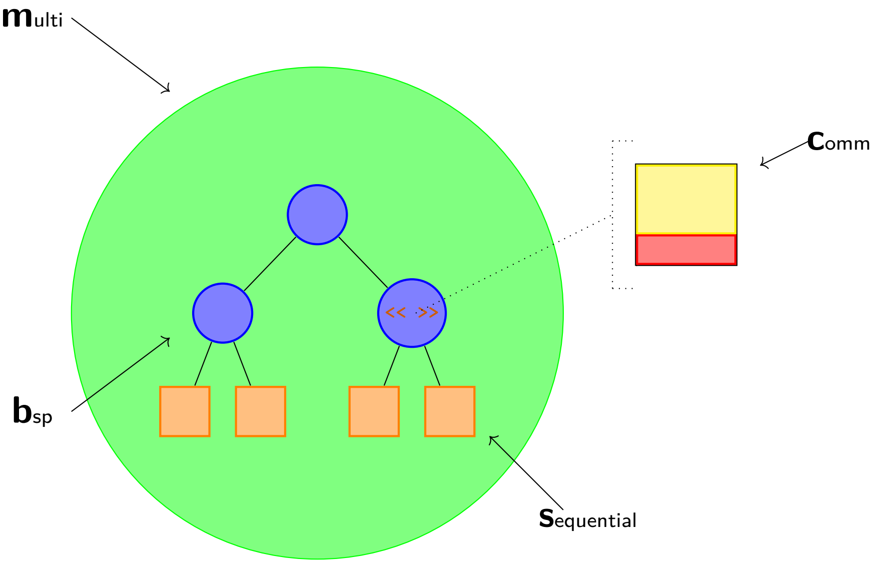
Type localities



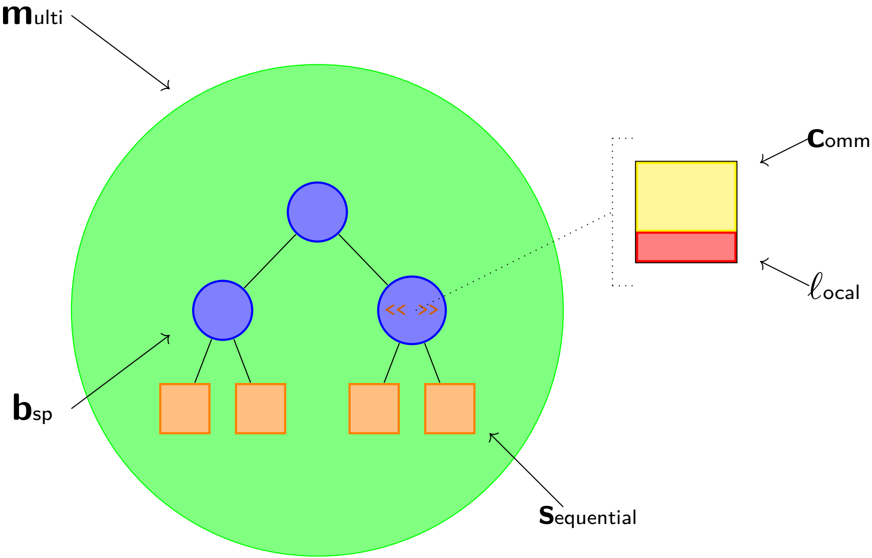
Type localities



Type localities



Type localities



multi

comm

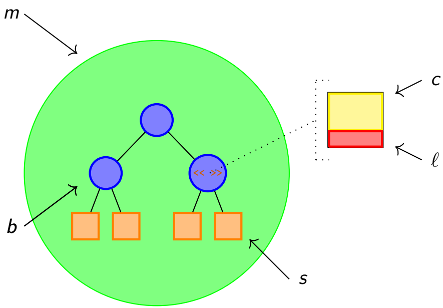
ℓocal

<< >>

bsp

sequential

Type localities



Accessibility: ◁

$$\begin{tikzcd}
 & & m & & \\
b \arrow[urr] & c \arrow[ur] \arrow[rr, leftrightarrow] & & \ell \arrow[ul] & s \arrow[ull]
\end{tikzcd}$$

Definability: ◀

$$\begin{tikzcd}
 & & m & & \\
b \arrow[urr] & c \arrow[rr, leftrightarrow] & & \ell \arrow[ul] & s \arrow[ull]
\end{tikzcd}$$

Type syntax

Tagged type

τ	$::=$	α_π	*type variable*
		$\texttt{Base}_\pi$	*base type*
		$(\tau \xrightarrow{\pi} \tau)_\pi$	*arrow type*
		$(\tau, \tau)_\pi$	*pairs*
		$\tau\ \texttt{Par}_b$	*parallel vector*
		$\tau\ \texttt{Tree}_\pi$	*tree*

Latent effect

$\mathsf{f} : (\texttt{int}`_a \xrightarrow{b} \texttt{int}_c\, \texttt{par}_b)_m$

Implementation

Sequential simulator

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

```
#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"
| |--> "0.0.0-> 2"
| |--> "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

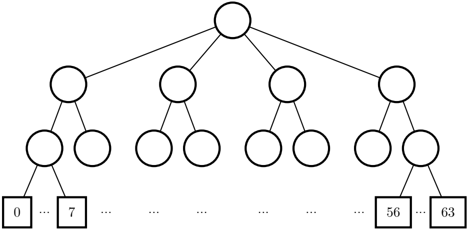
- $\sqrt(n)$th first prime numbers
- Based on scan
- Unbalanced

Benchmarks

Naive Eratosthenes algorithm

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

Mirev 3

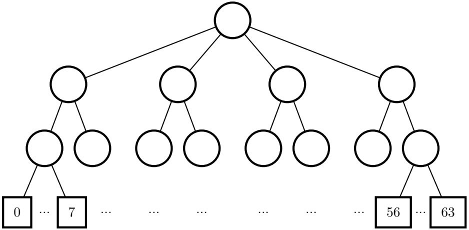


Benchmarks

Naive Eratosthenes algorithm

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

Mirev 3



Results

	100_000		500_000		1_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	0.7	1.8	22.4	105.0	125.3	430.7
64	0.3	0.3	1.3	8.7	4.1	56.1
128	0.5	0.45	2.1	5.2	4.7	24.3

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 (confuent)
- 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 (confuent)
- 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 ?