Multi-ML: Programming Multi-BSP Algorithms in ML

Victor Allombert, Frédéric Gava and Julien Tesson

Laboratory of Algorithmic Complexity and Logic
Université Paris-Est

HLPP July 2015

Table of Contents

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

Table of Contents

1 Introduction
- BSP
- BSML
- MULTI-BSP

2 Multi-ML

3 Results

4 Conclusion

Bulk Synchronous Parallelism

The BSP computer

Defined by:

Bulk Synchronous Parallelism

The BSP computer

Defined by:

- **p** pairs CPU/memory

Bulk Synchronous Parallelism

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network

Bulk Synchronous Parallelism

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronization unit

Bulk Synchronous Parallelism

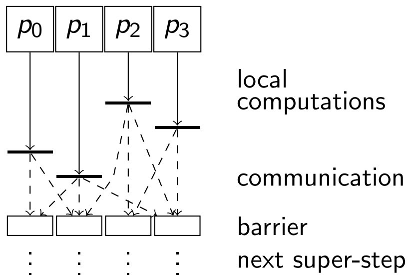
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronization unit

Properties:

- Super-steps execution



Bulk Synchronous Parallelism

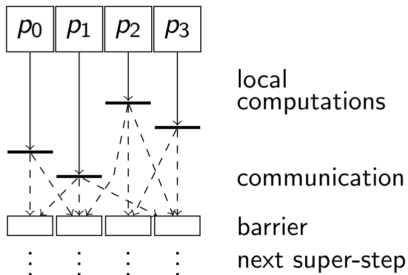
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronization unit

Properties:

- Super-steps execution
- Confluent



Bulk Synchronous Parallelism

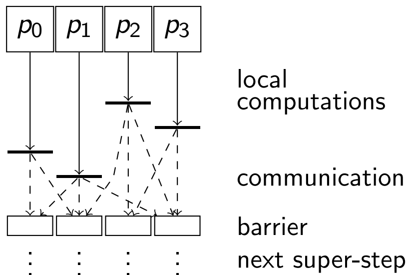
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronization unit

Properties:

- Super-steps execution
- Confluent
- Deadlock-free



Bulk Synchronous Parallelism

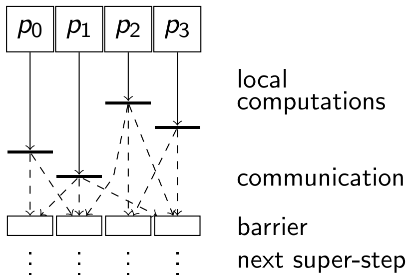
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronization unit

Properties:

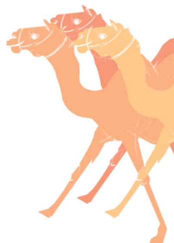
- Super-steps execution
- Confluent
- Deadlock-free
- Predictable performances



What is BSML?

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

What is BSML?

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

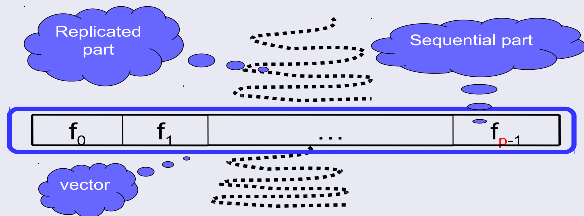
Bulk Synchronous ML

What is BSML?

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

Main idea

Parallel data structure ⇒ vectors:



Asynchronous primitives

BSML primitives

Asynchronous primitives

- $\ll e \gg \ : \langle e, \ldots, e \rangle$

Asynchronous primitives

- $\ll$ e $\gg$: $\langle e, \ldots, e \rangle$
- v : v_i on processor i, assumes $v \equiv \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$

BSML primitives

Asynchronous primitives

- $\ll$ e $\gg$: $\langle e, \ldots, e \rangle$
- \$v\$: v_i on processor *i*, assumes $v \equiv \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$
- \$**pid**\$: A predefined vector: *i* on processor *i*

BSML primitives

Asynchronous primitives

- $\ll$ e $\gg$: $\langle e, \ldots, e \rangle$
- \$v\$: v_i on processor *i*, assumes $v \equiv \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$
- \$**pid**\$: A predefined vector: *i* on processor *i*

Synchronous primitives

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

BSML primitives

Asynchronous primitives

- $\ll e \gg$: $\langle e, \ldots, e \rangle$
- v : v_i on processor i, assumes $v \equiv \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$
- $**pid**$: A predefined vector: 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 = ≪ "HLPP␣" ≫ ;;
val vec : string par = <"HLPP␣", "HLPP␣", "HLPP␣">

# let vec2 = ≪ $vec$^(string_of_int $pid$) ≫ ;;
val vec2 : string par = <"HLPP␣0", "HLPP␣1", "HLPP␣2">

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

# totex vec2;;
— : string list = ["HLPP0"; "HLPP1"; "HLPP2"]
```

What is MULTI-BSP?

The MULTI-BSP model

What is MULTI-BSP?

1. A tree structure with nested components

What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity

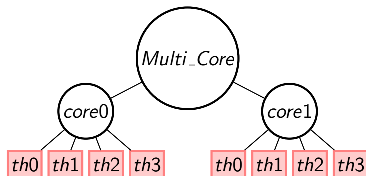
The MULTI-BSP model

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

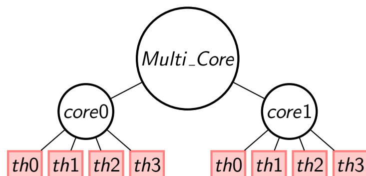


The MULTI-BSP model

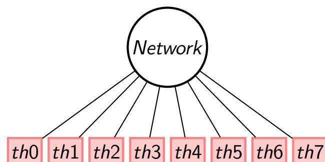
What is MULTI-BSP?

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

MULTI-BSP



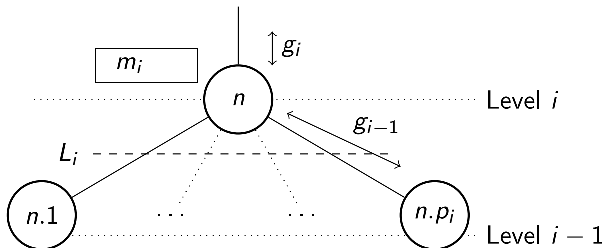
BSP



The MULTI-BSP model

Execution model

A level i superstep is:

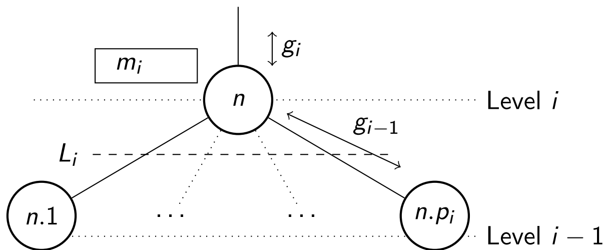


The MULTI-BSP model

Execution model

A level i superstep is:

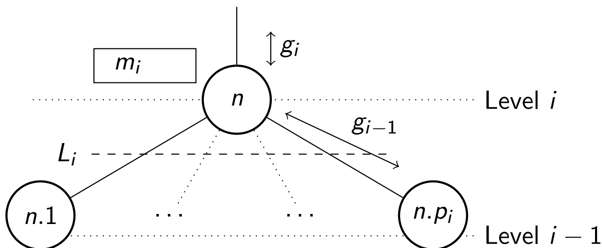
- Level $i-1$ executes code independantly



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

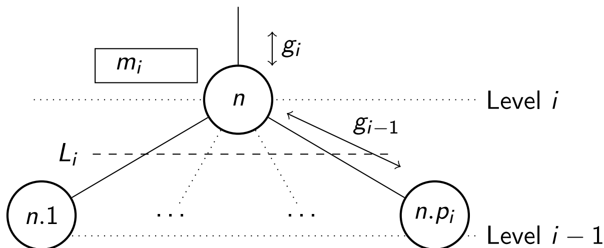


Table of Contents

1 Introduction

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

3 Results

4 Conclusion

Basic ideas:

Basic ideas:

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

Basic ideas:

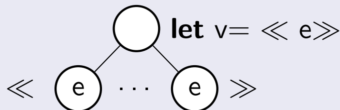
- BSML-like code on every stage of the MULTI-BSP architecture
- Specific syntax over ML: eases programming

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.

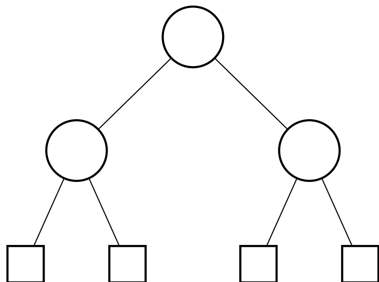


Recursion structure

let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ...
  where leaf =
   (* OCAML code *)
   ...

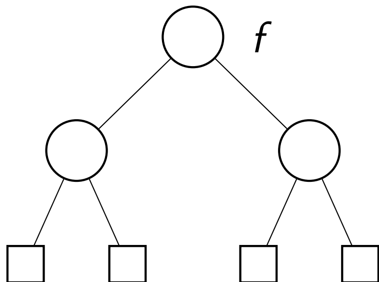
Recursion structure

```
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ...
  where leaf =
   (* OCAML code *)
   ...
```



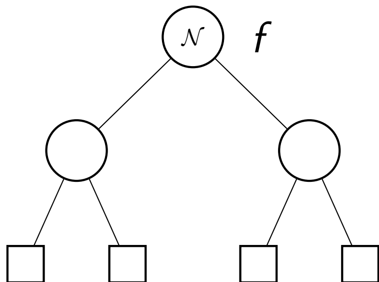
Recursion structure

let multi f [args] =
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ...
  where leaf =
    (* OCAML code *)
    ...



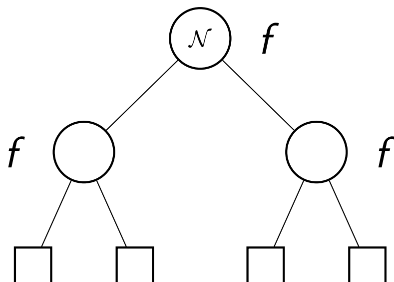
Recursion structure

let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ...
  where leaf =
   (* OCAML code *)
   ...



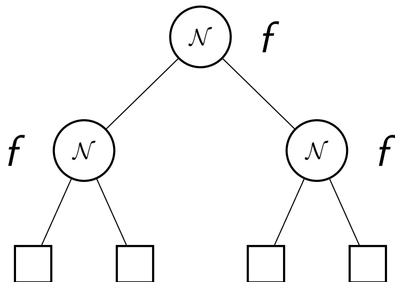
Recursion structure

let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ...
  where leaf =
   (* OCAML code *)
   ...



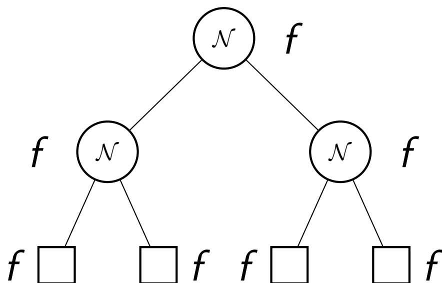
Recursion structure

let multi f [args] =
 where node =
  (* BSML code *)
  ...
  ≪ f [args] ≫
  ...
 where leaf =
  (* OCAML code *)
  ...



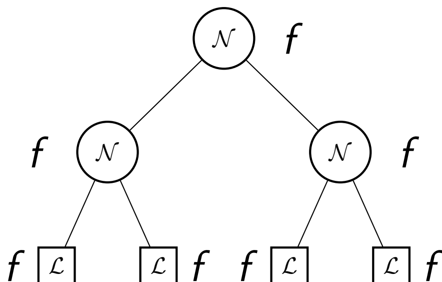
Recursion structure

let multi f [args] =
  where node =
    (* BSML code *)
    ...
    ≪ f [args] ≫
    ...
  where leaf =
    (* OCAML code *)
    ...



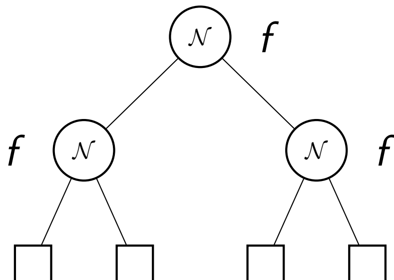
Recursion structure

let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ...
  where leaf =
   (* OCAML code *)
   ...



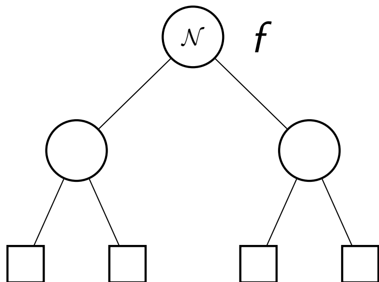
Recursion structure

```
let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ...
  where leaf =
   (* OCAML code *)
   ...
```



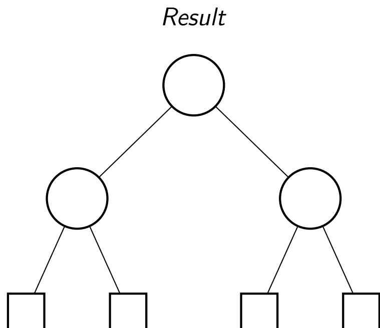
Recursion structure

let multi f [args] =
  where node =
   (* BSML code *)
   ...
   ≪ f [args] ≫
   ...
  where leaf =
   (* OCAML code *)
   ...



Recursion structure

let multi f [args] =
where node =
(* BSML code *)
...
≪ f [args] ≫
...
where leaf =
(* OCAML code *)
...



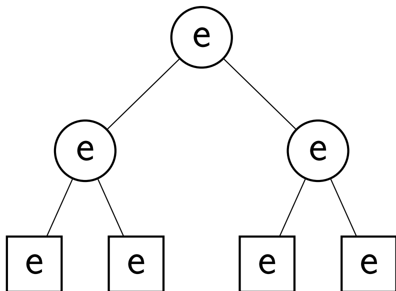
Primitives

Summary:

Primitives

Summary:

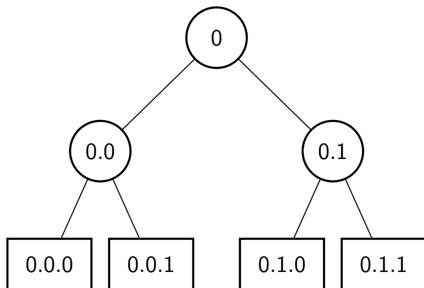
- §e§



Primitives

Summary:

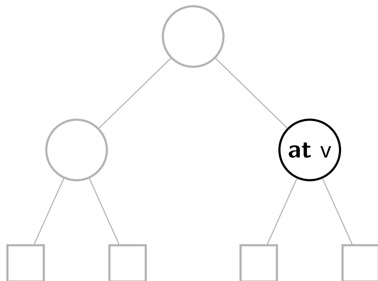
- §e§
- **gid**



Primitives

Summary:

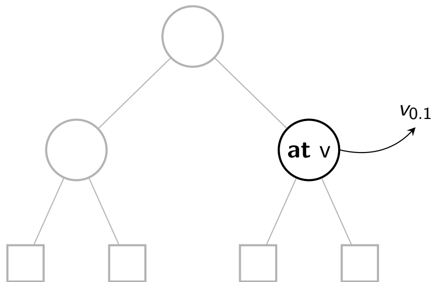
- §e§
- **gid**
- **at**



Primitives

Summary:

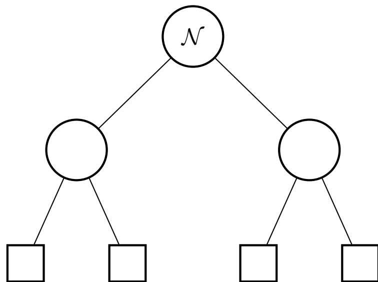
- §e§
- **gid**
- **at**



Primitives

Summary:

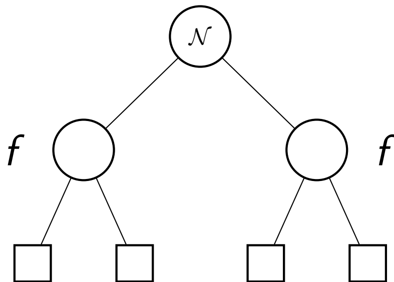
- §e§
- **gid**
- **at**
- ≪...f...≫



Primitives

Summary:

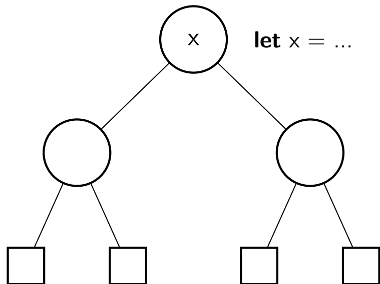
- §e§
- **gid**
- **at**
- ≪...f...≫



Primitives

Summary:

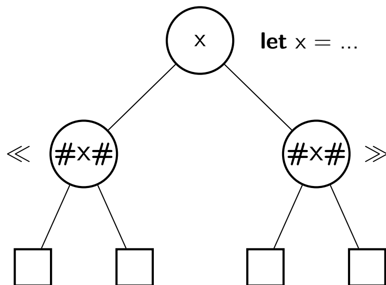
- §e§
- **gid**
- **at**
- ≪...f...≫
- **#**x**#**



Primitives

Summary:

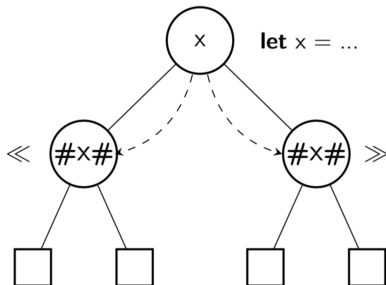
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#



Primitives

Summary:

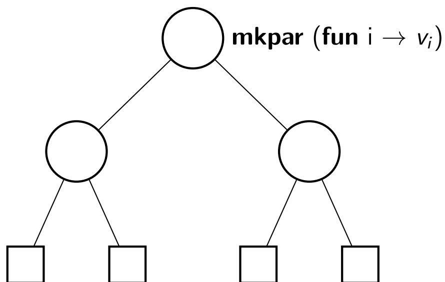
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#



Primitives

Summary:

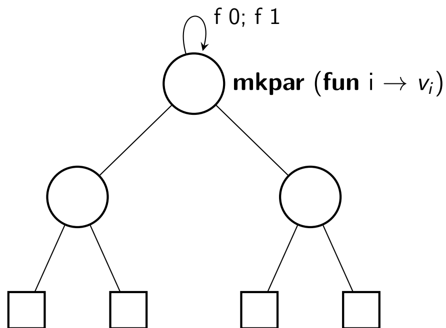
- §e§
- **gid**
- **at**
- ≪...f...≫
- **#**x**#**
- **mkpar** f



Primitives

Summary:

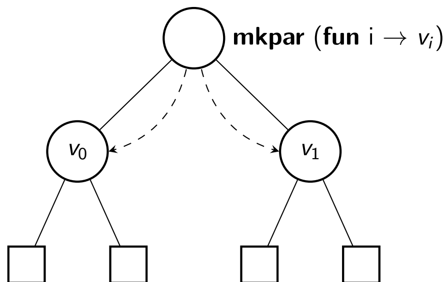
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#
- **mkpar** f



Primitives

Summary:

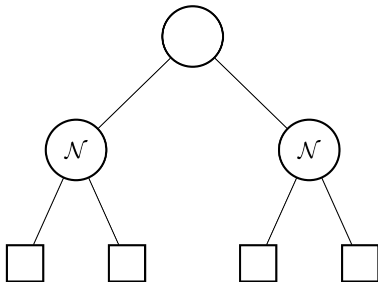
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#
- **mkpar** f



Primitives

Summary:

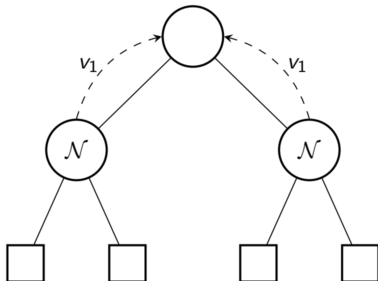
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#
- **mkpar** f
- **finally** v_1 v_2



Primitives

Summary:

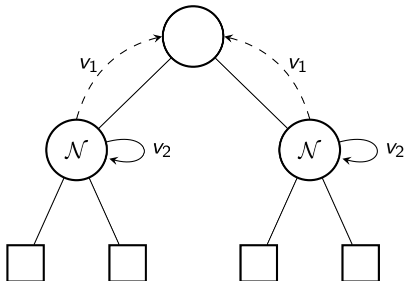
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#
- **mkpar** f
- **finally** v_1 v_2



Primitives

Summary:

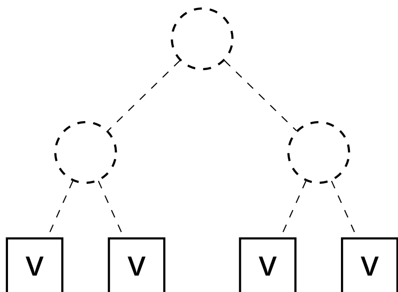
- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#
- **mkpar** f
- **finally** v_1 v_2



Primitives

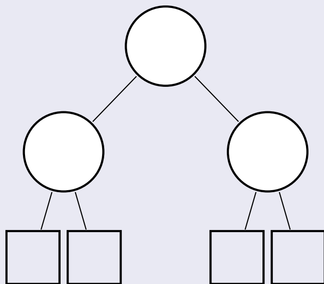
Summary:

- §e§
- **gid**
- **at**
- ≪...f...≫
- #x#
- **mkpar** f
- **finally** v_1 v_2
- **this**



Code example

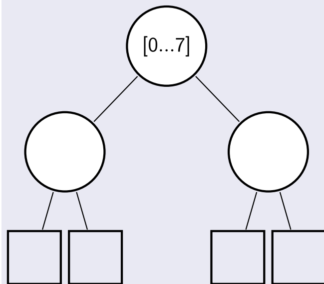
Keep the intermediate results of the sum:



```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
    finally ˜up:s ˜keep:s
  where leaf =
    let s = sumSeq l in

    finally ˜up:s ˜keep:s
```

Keep the intermediate results of the sum:

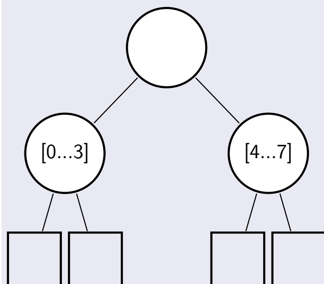



```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
    finally ˜up:s ˜keep:s
  where leaf =
    let s = sumSeq l in

    finally ˜up:s ˜keep:s
```

Code example

Keep the intermediate results of the sum:

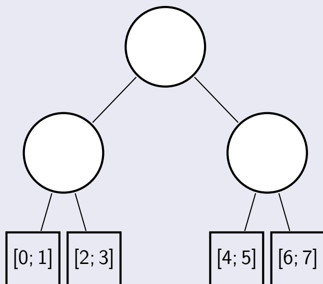



```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in

    finally ~up:s ~keep:s
```

Code example

Keep the intermediate results of the sum:

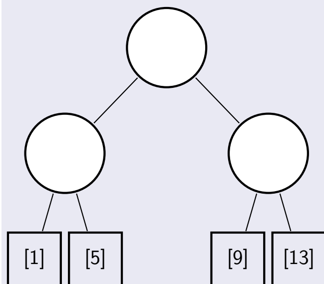



```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in

    finally ~up:s ~keep:s
```

Code example

Keep the intermediate results of the sum:

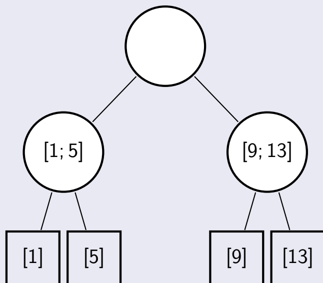



```
let multi tree sum_list l =
   where node =
      let v = mkpar (fun i → split i l) in
      let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
      finally ˜up:s ˜keep:s
   where leaf =
      let s = sumSeq l in

      finally ˜up:s ˜keep:s
```

Code example

Keep the intermediate results of the sum:

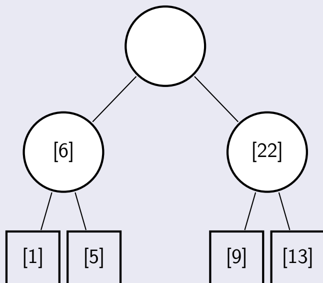



```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in

    finally ~up:s ~keep:s
```

Code example

Keep the intermediate results of the sum:

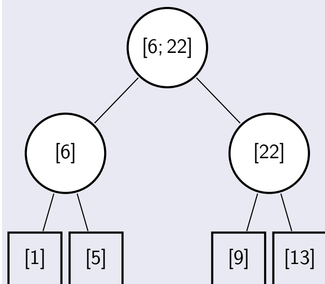



```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in

    finally ~up:s ~keep:s
```

Code example

Keep the the intermediate results of the sum:

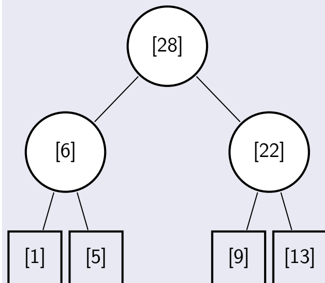



```
let multi tree sum_list l =
    where node =
        let v = mkpar (fun i → split i l) in
        let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
        finally ~up:s ~keep:s
    where leaf =
        let s = sumSeq l in

        finally ~up:s ~keep:s
```

Code example

Keep the intermediate results of the sum:

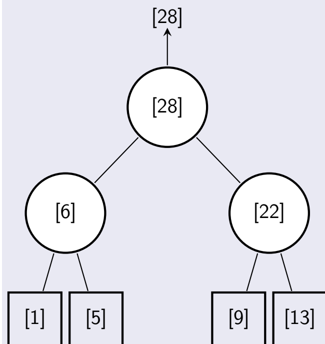



```
let multi tree sum_list l =
  where node =
    let v = mkpar (fun i → split i l) in
    let s = sumSeq (flatten ≪ sum_list $v$≫ ) in
    finally ~up:s ~keep:s
  where leaf =
    let s = sumSeq l in

    finally ~up:s ~keep:s
```

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 s = sumSeq (flatten ≪ sum_list $v$≫ ) in
    finally ˜up:s ˜keep:s
  where leaf =
    let s = sumSeq l in

    finally ˜up:s ˜keep:s
```

Formal definition of a core-languge

Useful for:

Formal definition of a core-languge

Useful for:

- Study of properties

Semantics

Formal definition of a core-languge

Useful for:

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

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

Implementation

Sequential simulator

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

```
#let multi f n =
 | where node =
 | | let _=≪f ($pid$ + #n# + 1) ≫ in
 | | | finally ~up:() ~keep:(gid^"=>"^n)
 | where leaf=finally ~up:() ~keep:(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

Distributed implementation

Our approach

- Modular

Distributed implementation

Our approach

- Modular
- Generic functors

Distributed implementation

Our approach

- Modular
- Generic functors
- Communication routines

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

Distributed implementation

Our approach

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

Current version

- Based on MPI
- SPMD

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 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

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

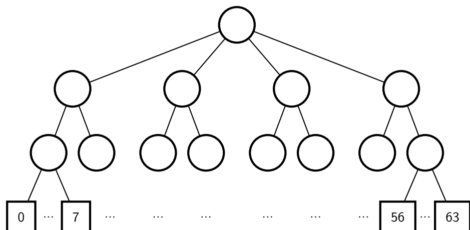
Naive Eratosthenes sieve

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

Benchmarks

Naive Eratosthenes sieve

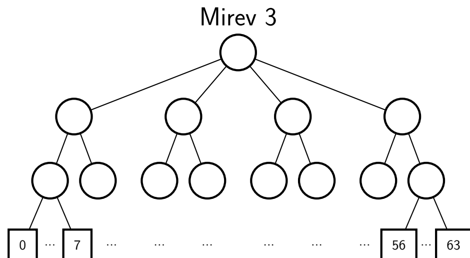
- $\sqrt(n)$th first prime numbers
- Based on scan
- Unbalanced



Benchmarks

Naive Eratosthenes sieve

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



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)
- Small number of primitives and little syntax extension

Conclusion

MULTI-ML

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

Future work

- Optimise MPI implementation

Conclusion

MULTI-ML

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

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)
- Small number of primitives and little syntax extension

Future work

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

Thank you for your attention !

Any questions ?