A formal semantics of the MULTI-ML language

Victor ALLOMBERT[1], Frédéric GAVA[2], Julien TESSON[2]

[1]LIFO - Université d'Orléans, France
[2]LACL - Université Paris Est, France

27 June 2018

Table of Contents

Table of Contents

The world of parallel computing

Simulations:

Fluid simulation
3D Visualisation

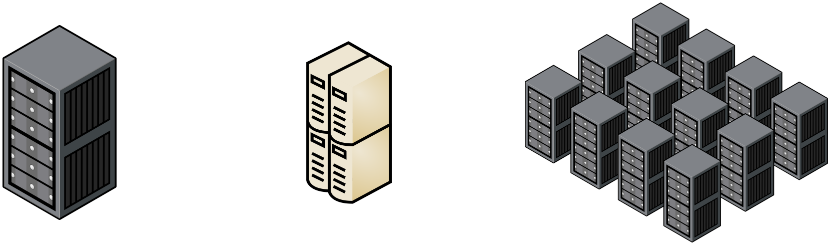
Big-Data:

IoT
Social Networking
Data science

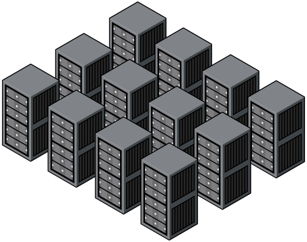
Symbolic computation:

Model-Checking
Formal computing

Super-computer

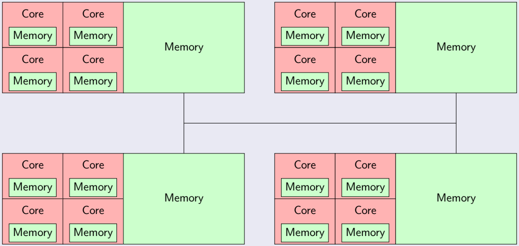


Hierarchical architectures

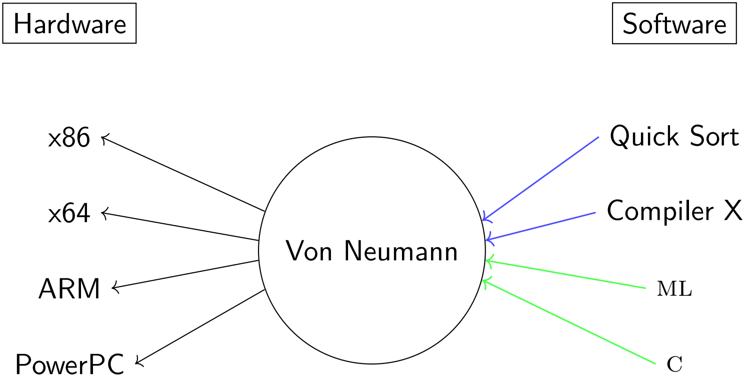


Characterised by:

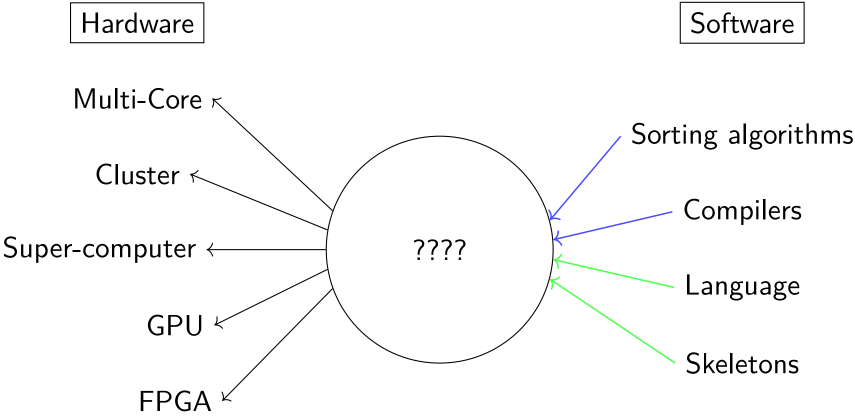
- Interconnected units
- Both shared and distributed memories
- Hierarchical memories



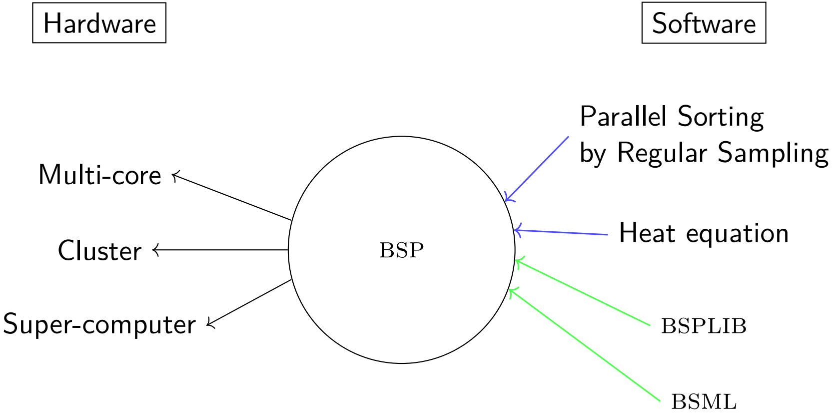
A sequential bridging model



A parallel bridging model



A parallel bridging model



Bulk Synchronous Parallelism

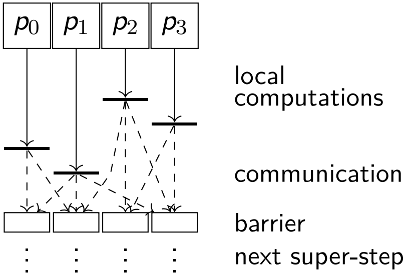
The BSP computer

Defined by:

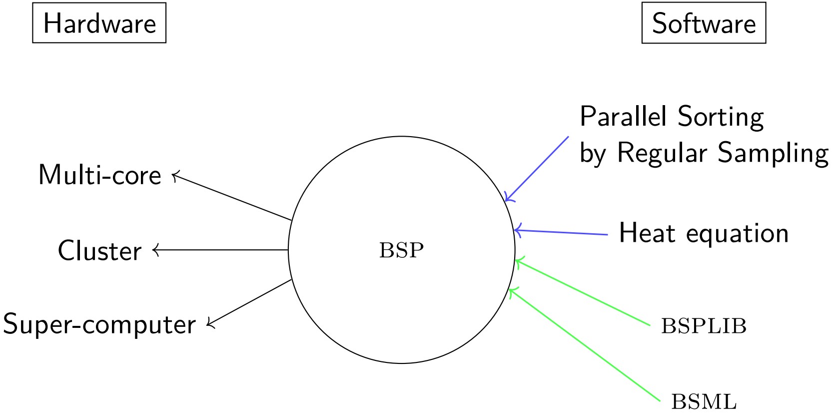
- $\mathbf{p}$ pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

- Deadlock-free
- Predictable performances



A parallel bridging model



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 and implemented over OCAML

What is BSML?

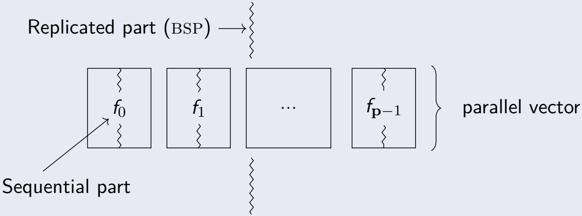
- Explicit BSP programming with a functional approach
- Based upon ML and implemented over OCAML
- Formal semantics → computer-assisted proofs (COQ)

What is BSML?

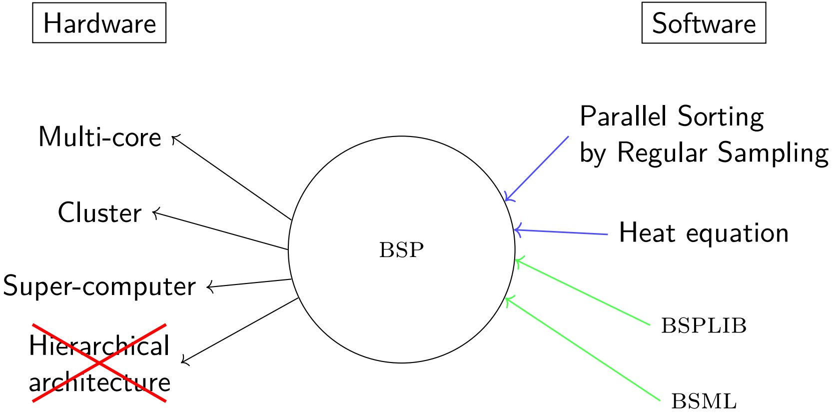
- Explicit BSP programming with a functional approach
- Based upon ML and implemented over OCAML
- Formal semantics → computer-assisted proofs (COQ)

Main idea

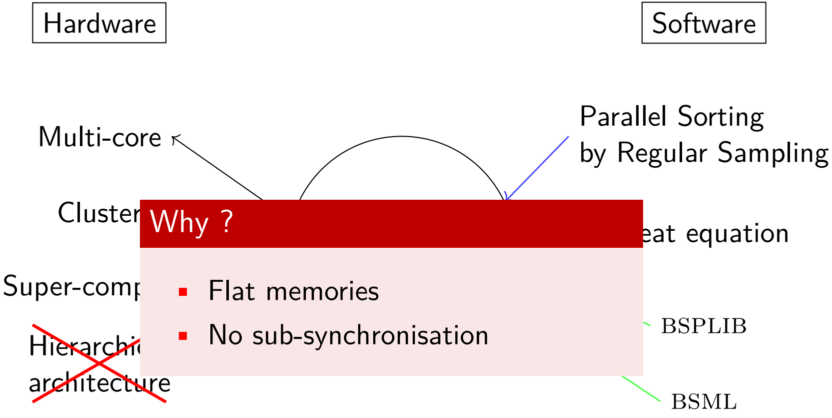
Parallel data structure ⇒ *parallel vector*:



A parallel bridging model



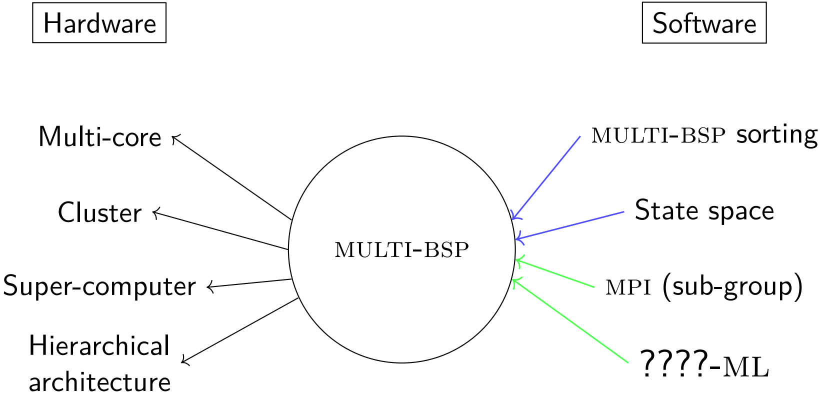
A parallel bridging model



Why ?

- Flat memories
- No sub-synchronisation

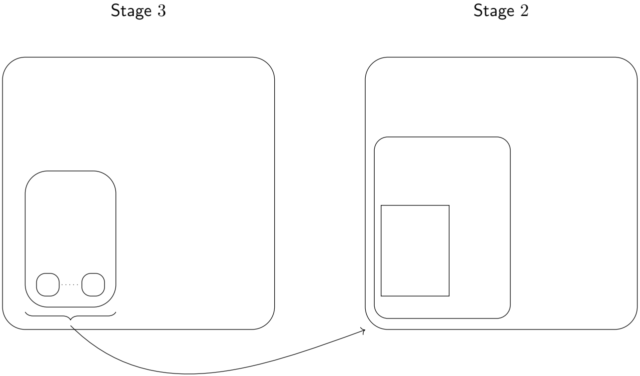
A parallel bridging model



What is MULTI-BSP?

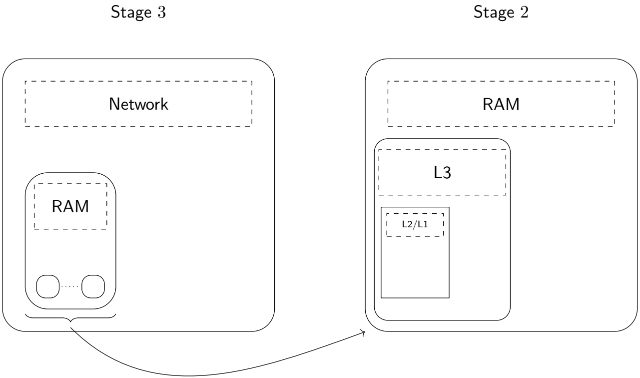
What is MULTI-BSP?

1. A tree structure with nested components



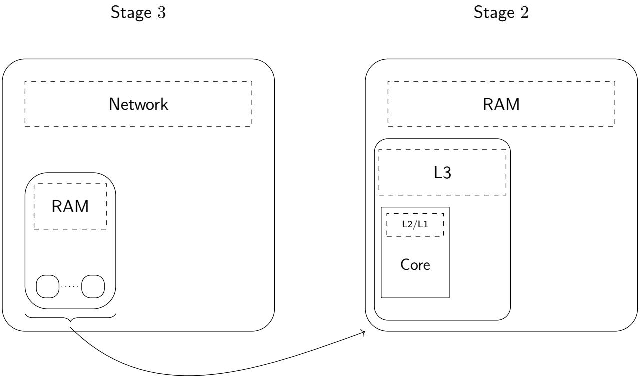
What is MULTI-BSP?

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



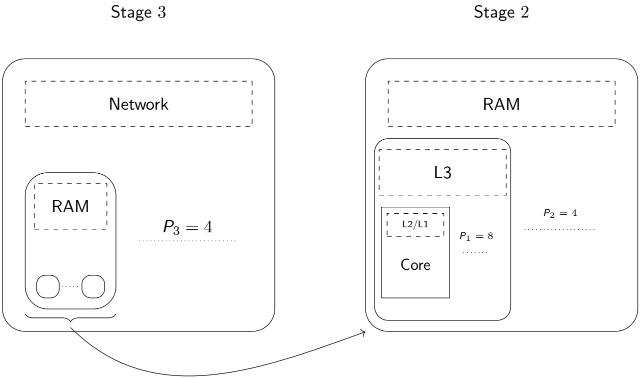
What is MULTI-BSP?

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



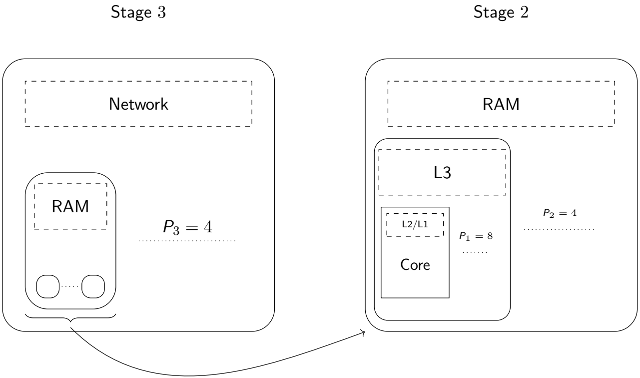
What is MULTI-BSP?

1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaves are processors
4. With sub-synchronisation capabilities



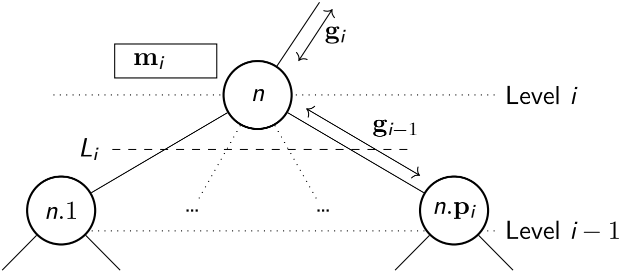
What is MULTI-BSP?

- Stage 3: 4 nodes with a network access
- Stage 2: one node has 4 chips plus RAM
- Stage 1: one chip has 8 cores plus L3 cache
- Stage 0: one core with L1/L2 caches



Execution model

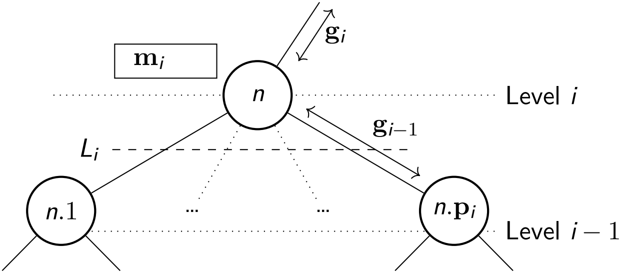
A level i superstep is:



Execution model

A level i superstep is:

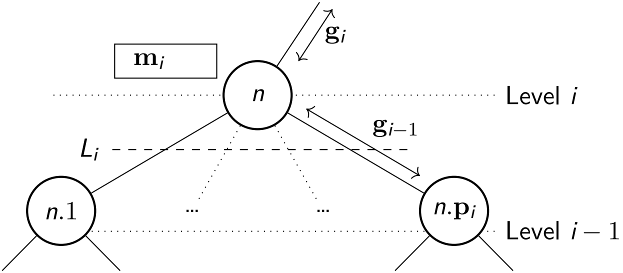
- Level $i-1$ executes code independently



Execution model

A level i superstep is:

- Level $i-1$ executes code independently
- Exchanges information with the m_i memory

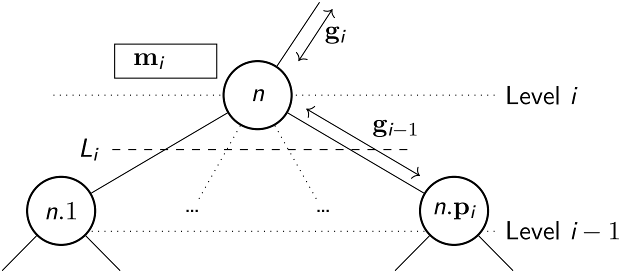


The MULTI-BSP model

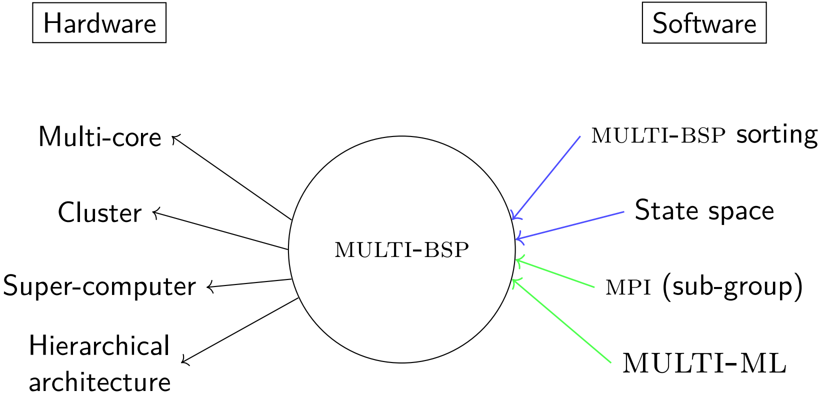
Execution model

A level i superstep is:

- Level $i-1$ executes code independently
- Exchanges information with the m_i memory
- Synchronises



A parallel bridging model

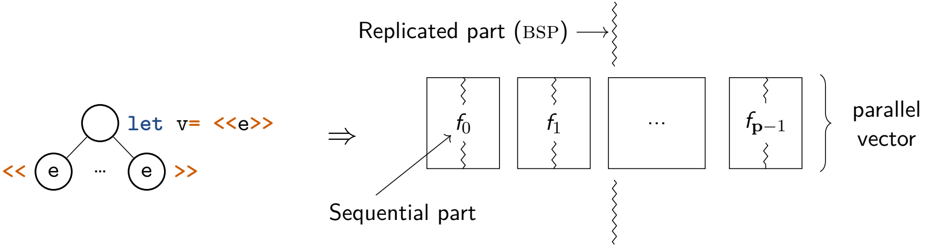


Basic ideas

The MULTI-ML language

Basic ideas

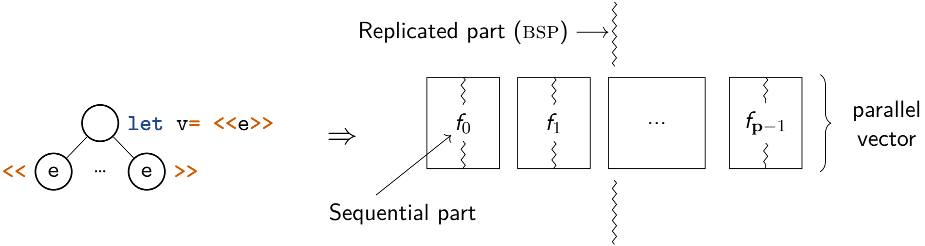
- BSML-like code on every stage of the MULTI-BSP architecture



The MULTI-ML language

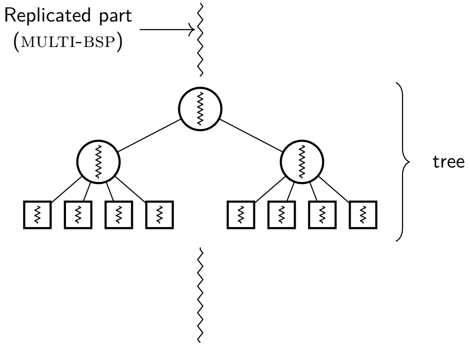
Basic ideas

- 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 MULTI-BSP tree



MULTI-ML: Tree recursion

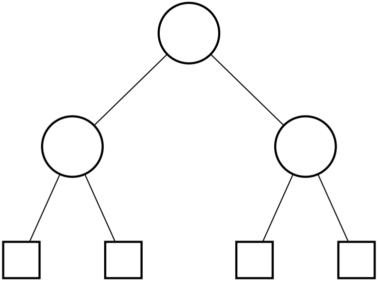
Recursion structure

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

MULTI-ML: Tree recursion

Recursion structure

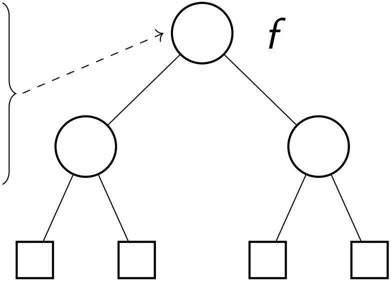
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

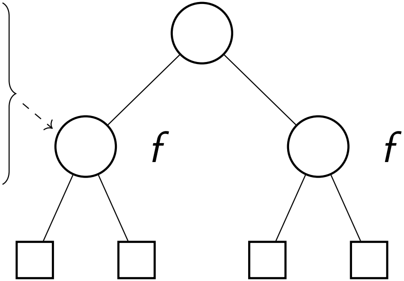
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

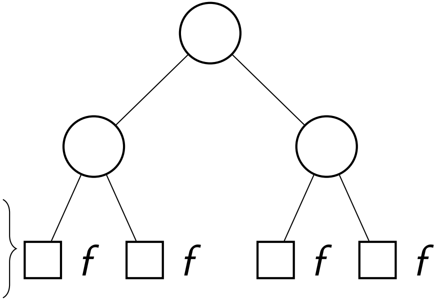
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

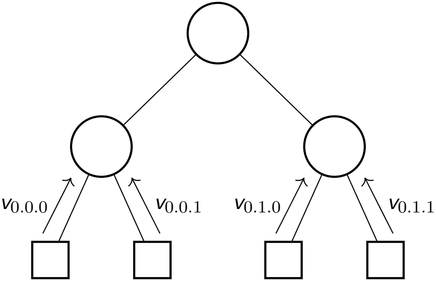
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



MULTI-ML: Tree recursion

Recursion structure

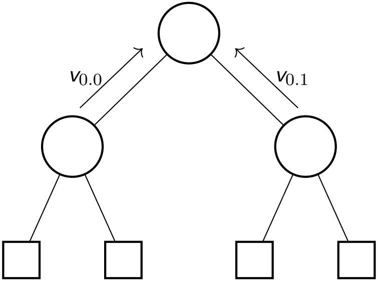
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



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
```

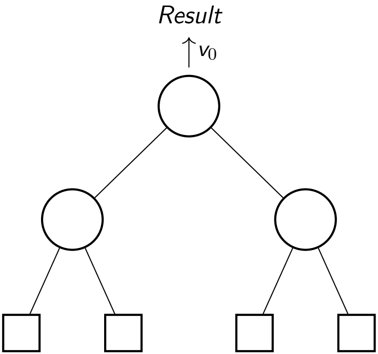
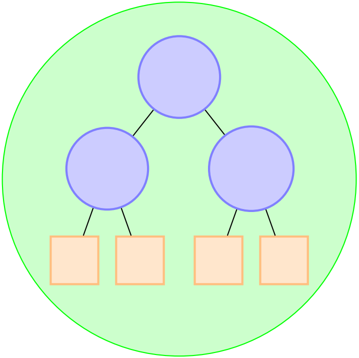
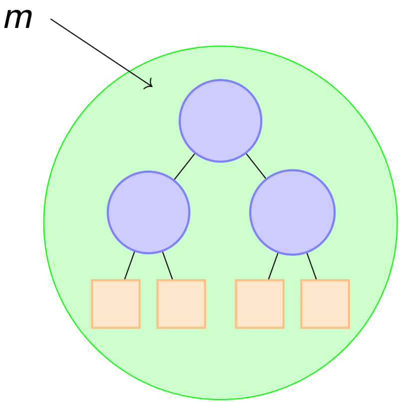


Table of Contents

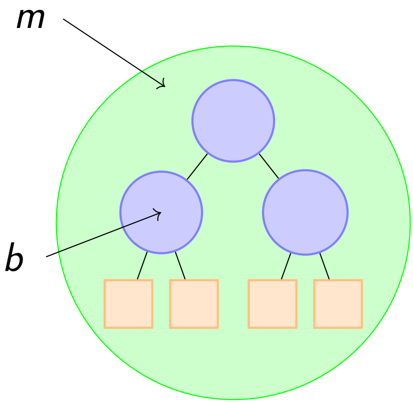
The MULTI-ML localities



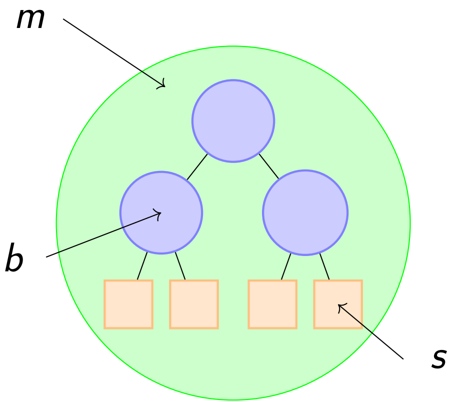
The MULTI-ML localities



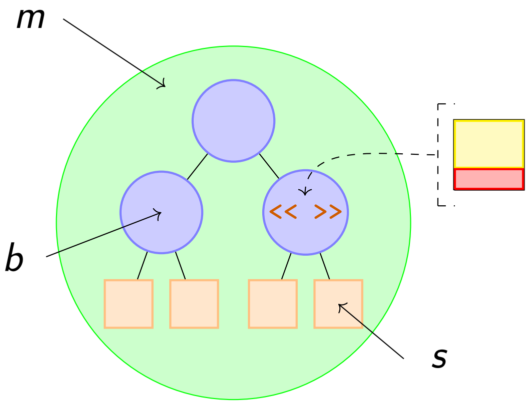
The MULTI-ML localities



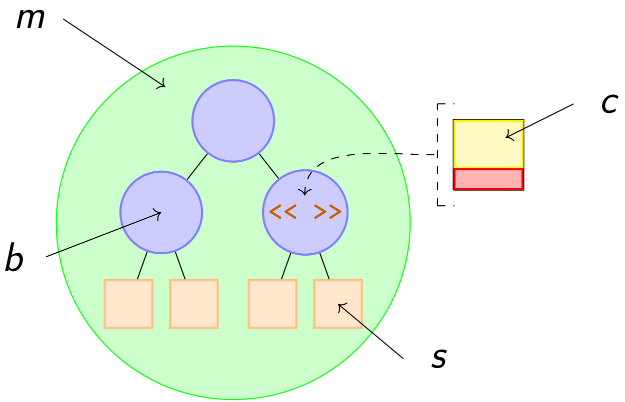
The MULTI-ML localities



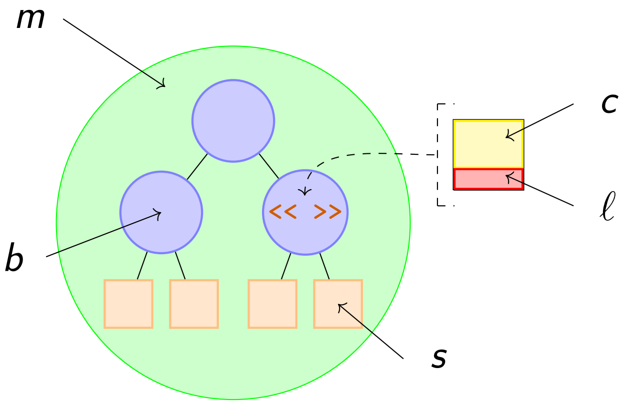
The MULTI-ML localities



The MULTI-ML localities



The MULTI-ML localities



A core language: μMULTI-ML

$$
\begin{array}{llll}
e & ::= & \mathtt{x} & \textit{Variable} \\
 & \mid & \mathbf{op}, \mathbf{cst} & \textit{Operator, Constant} \\
 & \mid & \mathbf{let}\ \ x = e\ \ \mathbf{in}\ \ e & \textit{Let binding} \\
 & \mid & \mathbf{fun}\ \ x \to e & \textit{Function} \\
 & \mid & \mathbf{multi}\ \ f\ x \to e \dagger e & \textit{Multi-function} \\
 & \mid & (e\ \ e) & \textit{Application} \\
 & \mid & \mathbf{if}\ \ e\ \ \mathbf{then}\ \ e\ \ \mathbf{else}\ \ e & \textit{Conditional} \\
 & \mid & \mathtt{mkpar}\ \ e & \textit{Parallel primitives} \\
 & \mid & \mathtt{proj}\ \ e \mid \mathtt{put}\ \ e & \textit{Synchro. parallel primitives} \\
 & \ldots & & \\
\mathtt{v} & ::= & & \textit{Values} \\
 & \mid & \mathbf{op}, \mathbf{cst} & \textit{Operator, Constant} \\
 & \mid & \overline{(\mathbf{fun}\ \ x \to e\ \ )[\mathcal{E}]} & \textit{Closure} \\
 & \mid & \overline{(\mathbf{multi}\ \ f\ x \to e \dagger e)[\mathcal{E}]} & \textit{Multi-function closure} \\
 & \mid & (v, v) & \textit{Pair} \\
 & \mid & < v, \ldots, v > & \textit{Parallel vector} \\
\mathbf{op} & ::= & +, -, *, /, \mathtt{fst}, \mathtt{snd}, \ldots & \textit{Basic operators} \\
\mathbf{cst} & ::= & 1, 2, \ldots, \mathit{true}, \mathit{false}, (), \ldots & \textit{Constants} \\
\mathcal{E} & ::= & \{x_1 \mapsto v_1, \ldots x_n \mapsto x_n\} & \textit{Environment}
\end{array}
$$

Inference rules

Inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v}$$

Co-inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty}$$

Inference rules

Inductive inference rule:

(Multi-)environment

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v}$$

Co-inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty}$$

Inference rules

Inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v}$$

(Multi-)environment

Expression

Co-inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty}$$

Inference rules

Inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v}$$

(Multi-)environment

Expression

Position

Co-inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty}$$

Inference rules

Inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v}$$

(Multi-)environment

Expression

Position

Locality

Co-inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty}$$

Inference rules

Inductive inference rule:

(Multi-)environment

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_{p}^{\mathcal{L}} v}$$

Value

Expression Position Locality

Co-inductive inference rule:

$$\begin{array}{c} \mathcal{P} \\ \hline\hline \mathcal{M} \vdash e \Downarrow_{p}^{\mathcal{L}} \infty \end{array}$$

Inference rules

Inductive inference rule:

Premises

(Multi-)environment

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_{p}^{\mathcal{L}} v}$$

Value

Expression Position Locality

Co-inductive inference rule:

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_{p}^{\mathcal{L}} \infty}$$

Inference rules

Inductive inference rule:

Premises

(Multi-)environment

$$\frac{\mathcal{P}}{\mathcal{M} \vdash e \Downarrow_{p}^{\mathcal{L}} v}$$

Value

Expression

Position

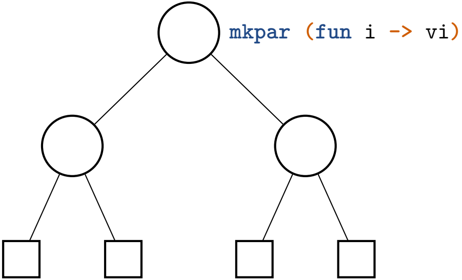
Locality

Co-inductive inference rule:

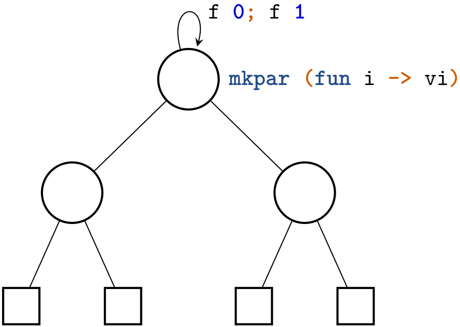
$$\begin{array}{c} \mathcal{P} \\ \hline\hline \mathcal{M} \vdash e \Downarrow_{p}^{\mathcal{L}} \infty \end{array}$$

Divergence

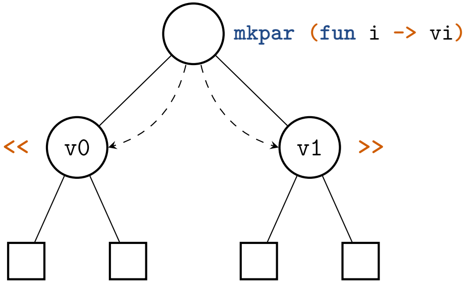
The `mkpar` case



The mkpar case



The mkpar case



The `mkpar` case

Inductive rule:

$$\text{MKPAR}\quad \frac{\begin{array}{c}\mathcal{M} \vdash e \Downarrow_p^{\mathtt{b}} \overline{(e')[\mathcal{E}']} \\ \forall i \in p \quad \mathcal{M} \oplus_p \mathcal{E}' \vdash e' \;\; i \Downarrow_p^{\mathtt{b}} v_i\end{array}}{\mathcal{M} \vdash \texttt{mkpar}\;\; e \Downarrow_p^{\mathtt{b}} < v_0, ..., v_n >}$$

The `mkpar` case

Co-inductive rule:

$$\text{MKPAR-E}\ \frac{\mathcal{M} \vdash e \Downarrow_p^{\mathtt{b}} \infty}{\mathcal{M} \vdash \mathtt{mkpar}\ \ e \Downarrow_p^{\mathtt{b}} \infty}$$

$$\text{MKPAR-V}\ \frac{\begin{array}{c}\mathcal{M} \vdash e \Downarrow_p^{\mathtt{b}} \overline{(e')[\mathcal{E}']} \\ \exists i \in p \quad \mathcal{M} \oplus_p \mathcal{E}' \vdash e'\ \ i \Downarrow_p^{\mathtt{b}} \infty\end{array}}{\mathcal{M} \vdash \mathtt{mkpar}\ \ e \Downarrow_p^{\mathtt{b}} \infty}$$

The MULTI-ML semantics ...

VALUES $\dfrac{}{\mathcal{M} \vdash \textbf{cst} \Downarrow_p^{\mathcal{L}} \textbf{cst}}$

VAR $\dfrac{\{x \mapsto v\} \in lookup(x, \mathcal{M}, p, \mathcal{L})}{\mathcal{M} \vdash x \Downarrow_p^{\mathcal{L}} v}$

CLS $\dfrac{\begin{array}{l} \mathcal{E} = select(\mathcal{M}, \mathcal{F}(\textbf{fun}\ x \to v\ ), p, \mathcal{L}) \\ v \equiv \overline{(\textbf{fun}\ x \to e\ )[\mathcal{E}]} \end{array}}{\mathcal{M} \vdash \textbf{fun}\ x \to e\ \ \Downarrow_p^{\mathcal{L}} v}$

APP $\dfrac{\begin{array}{l} \mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} \overline{(\textbf{fun}\ x \to e\ )[\mathcal{E}]} \quad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathcal{L}} v \\ \mathcal{M} \oplus_p \mathcal{E} \oplus_p \{x \mapsto v\} \vdash e \Downarrow_p^{\mathcal{L}} v' \end{array}}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_p^{\mathcal{L}} v'}$

LET_IN $\dfrac{\begin{array}{l} \mathcal{M} \vdash e_1 \Downarrow_p^{\mathcal{L}} v_1 \\ \mathcal{M} \oplus_p \{x \mapsto v_1\} \vdash e_2 \Downarrow_p^{\mathcal{L}} v_2 \end{array}}{\mathcal{M} \vdash \textbf{let}\ x = e_1\ \textbf{in}\ e_2 \Downarrow_p^{\mathcal{L}} v_2}$

REPLICATE $\dfrac{\begin{array}{l} \forall i \in p \quad \mathcal{M} \oplus_{p_i} \{f \mapsto (\overline{\textbf{fun}\ \_ \to e\ [\ ]})\} \vdash f\ ()\ \Downarrow_{p_i}^{\mathtt{l}} v_i \\ \mathcal{C}omm(\overline{(\textbf{fun}\ \_ \to e\ [\ ]})) \end{array}}{\mathcal{M} \vdash \texttt{replicate}\ (\textbf{fun}\ \_ \to e\ ) \Downarrow_p^{\mathtt{b}} < v_0, \ldots, v_n >}$

DOWN $\dfrac{\begin{array}{l} \mathcal{M} \vdash x \Downarrow_p^{\mathtt{b}} v \\ \mathcal{C}omm(v) \end{array}}{\mathcal{M} \vdash \texttt{down}\ x \Downarrow_p^{\mathtt{b}} < v >}$

MKPAR $\dfrac{\begin{array}{l} \mathcal{M} \vdash e \Downarrow_p^{\mathtt{b}} \overline{(e')[\mathcal{E}']} \\ \forall i \in p \quad \mathcal{M} \oplus_p \mathcal{E}' \vdash e'\ i \Downarrow_p^{\mathtt{b}} v_i \\ \mathcal{C}omm(v_i) \end{array}}{\mathcal{M} \vdash \texttt{mkpar}\ e \Downarrow_p^{\mathtt{b}} < v_0, \ldots, v_n >}$

APPLY $\dfrac{\begin{array}{l} \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{b}} < \overline{(e_i)[\mathcal{E}_i]} > \qquad \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{b}} < v_0, \ldots, v_n > \\ \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}, x_i \mapsto v_i\} \vdash f_i\ x_i \Downarrow_{p.i}^{\mathtt{l}} v_i' \end{array}}{\mathcal{M} \vdash \texttt{apply}\ e_1\ e_2 \Downarrow_p^{\mathtt{b}} < v_0', \ldots, v_n' >}$

PROJ $\dfrac{\begin{array}{l} \mathcal{M} \vdash e \Downarrow_p^{\mathtt{b}} < v_0, \ldots, v_n > \\ \forall i \in p \quad \mathcal{M} \oplus_{p.i} \{f \mapsto \overline{(e')[\mathcal{E}']}\} \vdash f\ i \Downarrow_{p.i}^{\mathtt{b}} v_i \\ \mathcal{C}omm(v_i) \end{array}}{\mathcal{M} \vdash \texttt{proj}\ e \Downarrow_p^{\mathtt{b}} \overline{(e')[\mathcal{E}']}}$

PUT $\dfrac{\begin{array}{l} \mathcal{M} \vdash e \Downarrow_p^{\mathtt{b}} < \overline{(e_0)[\mathcal{E}_0]}, \ldots, \overline{(e_n)[\mathcal{E}_n]} > \\ \forall i, j \in p \\ \mathcal{M} \oplus_{p.i} \{f_i \mapsto \overline{(e_i)[\mathcal{E}_i]}\} \vdash f_i\ j \Downarrow_{p.i}^{\mathtt{l}} v_{ij} \\ \mathcal{M} \oplus_{p.j} \{f_j' \mapsto \overline{(e_j')[\mathcal{E}_j']}\} \vdash f_j'\ i \Downarrow_{p.j}^{\mathtt{l}} v_{ij} \end{array}}{\mathcal{M} \vdash \texttt{put}\ e \Downarrow_p^{\mathtt{b}} < \overline{(e_0')[\mathcal{E}_0']}, \ldots, \overline{(e_n')[\mathcal{E}_n']} > f}$

MULTI_NODE $\dfrac{\begin{array}{l} isNode(p) \\ \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{l}} \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']} \\ \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{l}} v \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \oplus_p \\ \quad \{f \mapsto \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_p^{\mathtt{b}} v' \end{array}}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_p^{\mathtt{l}} v'}$

MULTI_LEAF $\dfrac{\begin{array}{l} isLeaf(p) \\ \mathcal{M} \vdash e_1 \Downarrow_p^{\mathtt{l}} \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']} \\ \mathcal{M} \vdash e_2 \Downarrow_p^{\mathtt{l}} v \\ \mathcal{M}' \oplus_p \{x \mapsto v\} \vdash e_2' \Downarrow_p^{\mathtt{s}} v' \end{array}}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_p^{\mathtt{l}} v'}$

MULTI_DEF $\dfrac{\begin{array}{l} \mathcal{M}' = select(||\mathcal{M}||_{multi}, \mathcal{F}(\textbf{multi}\ f\ x \to e_1 \dagger e_2)) \\ v \equiv \overline{(\textbf{multi}\ f\ x \to e_1 \dagger e_2)[\mathcal{M}']} \end{array}}{\mathcal{M} \vdash (\textbf{multi}\ f\ x \to e_1 \dagger e_2) \Downarrow_{multi}^{\mathtt{m}} v}$

MULTI_CALL $\dfrac{\begin{array}{l} \mathcal{M} \vdash e_1 \Downarrow_{multi}^{\mathtt{m}} \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']} \\ \mathcal{M} \vdash e_2 \Downarrow_{multi}^{\mathtt{m}} v \\ \mathcal{M} \oplus_{root} \{x \mapsto v\} \oplus_{root} \\ \quad \{f \mapsto \overline{(\textbf{multi}\ f\ x \to e_1' \dagger e_2')[\mathcal{M}']}\} \vdash e_1' \Downarrow_{root}^{\mathtt{b}} v' \\ \mathcal{C}omm(v') \end{array}}{\mathcal{M} \vdash (e_1\ e_2) \Downarrow_{multi}^{\mathtt{m}} v'}$

But why ?

- Ensure consistency with the MULTI-BSP model
- Prove:
  - Evaluation determinism

  $>$ *If* $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_1$ *and* $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v_2$ *then* $v_1 = v_2$.

  - Evaluation (or not)

  $>$ *It is impossible to obtain a value v such that* $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$,

  $>$ *or there exists a value v such that* $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$.

  - Evaluation *"does not go wrong"*

  $>$ *If* $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} v$ *and* $\mathcal{M} \vdash e \Downarrow_p^{\mathcal{L}} \infty$ *then there is a contradiction.*

Table of Contents

Conclusion

Presented work

- MULTI-BSP extension of ML: MULTI-ML
- Operational semantics
  - Evaluation determinism
  - Evaluation (or not)
  - Evaluation *"does not go wrong"* ...

Ongoing and future work

- Automatic cost analysis
- Type system
- Certified parallel programming

Thank you for your attention ☺

Questions ?