Programming MULTI-BSP Algorithms in ML

Victor ALLOMBERT

LIFO - Université d’Orléans

15 January 2018
Table of Contents

1 Introduction

2 The MULTI-ML language

3 Type system

4 Implementation

5 Conclusion
Table of Contents

1 Introduction
   The world of parallel computing

2 The MULTI-ML language

3 Type system

4 Implementation

5 Conclusion
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

Hardware

x86
x64
ARM
PowerPC

Software

Quick Sort
Compiler X
ML
C

Von Neumann
A parallel bridging model

Hardware

Multi-Core

Cluster

Super-computer

GPU

FPGA

Software

Sorting algorithms

Compilers

Language

Skeletons
A parallel bridging model

- Hardware
  - Multi-core
  - Cluster
  - Super-computer

- Software
  - Parallel Sorting by Regular Sampling
  - Heat equation
  - BSPLIB
  - BSML
**Bulk Synchronous Parallelism**

The **BSP** computer

Defined by:
- \( p \) pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

**Properties:**
- Deadlock-free
- Predictable performances
A parallel bridging model

Hardware
- Multi-core
- Cluster
- Super-computer

Software
- Parallel Sorting by Regular Sampling
- Heat equation
- BSPLIB
- BSML
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)
**Bulk Synchronous ML**

**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*:

Replicated part (BSP) →

Sequential part

\[ f_0 \ 
\[ f_1 \ 
\[ ... \ 
\[ f_{p-1} \ 
\]

parallel vector
A parallel bridging model

Hardware

Multi-core

Cluster

Super-computer

Hierarchical architecture

Software

Parallel Sorting by Regular Sampling

Heat equation

BSPLIB

BSML
A parallel bridging model

- Flat memories
- No sub-synchronisation

Why?

Parallel Sorting by Regular Sampling

Heat equation

Hardware

- Multi-core
- Cluster
- Super-computer
- Hierarchical architecture

Software

- BSPLIB
- BSML
A parallel bridging model

Hardware

- Multi-core
- Cluster
- Super-computer
- Hierarchical architecture

Software

- MULTI-BSP sorting
- State space
- MPI (sub-group)
- ???-ML

Multi-core & Cluster & Super-computer & Hierarchical architecture & MULTI-BSP
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

![Diagram of MULTI-BSP structure]

Stage 3

Stage 2

Network

RAM

$L_2/L_1$

Core

L3

RAM

$P_3 = 4$

$P_1 = 8$

$P_2 = 4$
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
The MULTI-BSP model

Execution model

A level $i$ superstep is:

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

1 Introduction

2 The **MULTI-ML** language
   - **MULTI-ML** overview
   - The **MULTI-ML** primitives
   - A code example

3 Type system

4 Implementation

5 Conclusion
## The MULTI-ML language

**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

Replicated part (multi-bsp)
The MULTI-ML language

Basic ideas

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

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

```
<< e ... e >>
```

Replicated part (BSP) →

```
\[ f_0 \quad f_1 \quad \ldots \quad f_{p-1} \]
```

vector parallel

Sequential part
The **MULTI-ML** language

**Basic ideas**

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

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

```
<< e ... e >>
```

```
parallel vector
Sequential part
Replicated part (BSP)
```

V. Allombert

Seminar - LIP
The **MULTI-ML** language

**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

```plaintext
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
```
**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
```

```
<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
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 recursion diagram]
**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:

```
\[
\begin{array}{c}
  f \\
  f \\
  f \\
  f \\
  f \\
  f
\end{array}
\]
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

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

where leaf =
  (* OCaml code *)
  ...
  in v
```

Result
MULTI-ML: Tree recursion

Recursion structure

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

Result

\[ \uparrow v_0 \]
Tree construction

```ocaml
let multi_tree f [args] =
  where node =
    (* BSML code *)
    ...
  in
  finally « f [args] »> v
  where leaf =
    (* OCaml code *)
    ...
  in v
```
MULTI-ML: Tree construction

Tree construction

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

Diagram:
```
          v
        /|
        / \
      v   v
     /     \
   v       v
  /       /\n v         v
```

V. Allombert
MULTI-ML: Tree construction

Tree construction

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

```ocaml

```

V. Allombert
MULTI-ML: Tree construction

Tree construction

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

Let `multi tree f [args] =`

where `node`

(* BSML code *)

... in

finally `<< f [args] >> v`

where `leaf`

(* OCaml code *)

... in `v`
MULTI-ML: Tree construction

Tree construction

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

```
V. Allombert
```

```
Seminar - LIP
```

16 / 37
multi-ml: Tree construction

Tree construction

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

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

Summary

- `mktree e`

Diagram:
```
  e
 / \
 e   e
```

V. Allombert
Seminar - LIP
Primitives

Summary

- mktree e
- gid

![Binary tree diagram with nodes labeled 0, 0.0, 0.1, 0.0.0, 0.0.1, 0.1.0, 0.1.1]
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
- \langle\langle\ldots f \ldots\rangle\rangle
Primitives

Summary

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

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

```
    x
   / 
  /   
/     
```

V. Allombert
Primitives

**Summary**

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

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

```
x
<<
#x#
'''

```
#x#
'''
```
Primitives

Summary

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

Summary

- mktree e
- gid
- at
- \(<<...f...>>\)
- #x#
- mkpar f

\(\text{mkpar (fun } i \rightarrow vi)\)
Primitives

Summary

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

Summary

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

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

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 finally rc s  
  where leaf =  
    sumSeq l
```

Keep the intermediate results of the sum

```ml
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 finally rc s
  where leaf =
    sumSeq l
```

[Diagram of a tree with nodes labeled [0...7]]
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 finally 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
    in finally rc s
  where leaf =
    sumSeq l
```

Diagram:
- Root node
- Subtree [0...3]
- Subtree [4...7]
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 finally rc s
  where leaf =
    sumSeq l
```

V. Allombert
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 finally rc s
  where leaf =
    sumSeq l
```

V. Allombert
Keep the intermediate results of the sum

def multi_tree(sum_list, l):
    node =
    v = mkpar(lambda i: split(i, l))
    rc = sum_list(v)
    s = sumSeq(flatten(at(rc)))
    finally rc s
    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 = unsafe_sum_list $v$ in
    let s = sumSeq (flatten unsafe_at $rc$) in
    in finally rc s
  where leaf =
    sumSeq l
```

V. Allombert

Seminar - LIP
Keep the intermediate results of the sum

```ocaml
define 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 $>$> $rc$ $>$>)
            in
                finally rc s
    where leaf =
        sumSeq l
```

V. Allombert
```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 finally rc s
where leaf =
  sumSeq l
```

Keep the intermediate results of the sum
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 finally rc s  
    where leaf =  
        sumSeq l
```

Diagram:
- Top node: [6; 22]
- Left branch: [6]
  - Left leaf: [1]
  - Right leaf: [5]
- Right branch: [22]
  - Left leaf: [9]
  - Right leaf: [13]
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 finally 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 finally rc s
  where leaf =
    sumSeq l
```
Table of Contents

1 Introduction

2 The MULTI-ML language

3 Type system
   Parallel program safety
   The MULTI-ML typing system

4 Implementation

5 Conclusion
Parallel program safety

- Replicated coherency

Replicated coherency

```java
if random_bool () then
    multi_fct ()
else
    (fun _ -> ...) ()
```
Parallel program safety

- Replicated coherency

Why?

Replicated coherency

```python
if random_bool()
    multi_fct()
else
    (fun _-> ...)
```

V. Allombert
## Type system

### Parallel program safety

- Replicated coherency
- Level (memory) compatibility

### Level (memory) compatibility

```plaintext
<< let multi f x = ... >>
let x = #y#
let z = $v$
```
Type system

### Parallel program safety

- Replicated coherency
- Level (memory) compatibility
- Control parallel structure nesting
  - vector
  - tree

### Parallel structure nesting

```
<< let v = << 1 >> in v >>

let v = << 1 >> in << v >>
```
Type localities
Type localities

$m$
Type localities

$m$

$b$

V. Allombert
Type localities

$m$

$b$

$s$
Type localities

$m$

$b$

$s$
Type localities
Type localities

\[ \text{Type localities} \]

\[ m \]

\[ b \]

\[ c \]

\[ \ell \]

\[ s \]
Type annotations

Type grammar

\[
\begin{align*}
\tau & ::= \\
& \alpha_\pi \quad \text{type variable} \\
& \text{Base}_\pi \quad \text{base type} \\
& (\tau, \tau)_\pi \quad \text{pair} \\
& \tau \text{ Par}_b \quad \text{vector} \\
& \tau \text{ Tree}_\pi \quad \text{tree} \\
& (\tau \xrightarrow{\pi} \tau)_\pi \quad \text{arrow type}
\end{align*}
\]

\[
\pi ::= m \mid b \mid c \mid l \mid s
\]
Type annotations

Latent effect

\[(\tau \xrightarrow{\pi} \tau)_{\pi'}\]

Where \(\pi\) is the effect *emitted* by the evaluation and \(\pi'\) the locality of definition.

A BSP function

```ocaml
#let f = fun x ->
  let v = "..." in 1
  in:
  val f : ('a_'z -> ('b-> int_b))_m
```

\[f: ('a'_z \rightarrow b \text{int}_b)_m\]
Accessibility: \( \triangleleft \)

\[
\begin{align*}
m, c & \triangleleft m \\
m, b & \triangleleft b \\
m, l, c & \triangleleft l \\
m, l, c & \triangleleft c \\
m, s & \triangleleft s
\end{align*}
\]

\( \lambda_2 \triangleleft \lambda_1 : \text{"} \lambda_1 \text{ can read in } \lambda_2 \text{ memory. } \)
Accessibility

Accessibility: $\triangleleft$

$m, c \triangleleft m$
$m, b \triangleleft b$
$m, l, c \triangleleft l$
$m, l, c \triangleleft c$
$m, s \triangleleft s$

$\lambda_2 \triangleleft \lambda_1 : \text{« \lambda}_1 \text{ can read in } \lambda_2 \text{ memory. »}$

Example:

$f : (\acute{a}_z \xrightarrow{b} \text{int}_b)_m$

$f \ 1 \leadsto b \triangleleft m$
Accessibility

Accessibility: △

\[ m, c \triangleright m \]
\[ m, b \triangleright b \]
\[ m, l, c \triangleright l \]
\[ m, l, c \triangleright c \]
\[ m, s \triangleright s \]

\[ \lambda_2 \triangleright \lambda_1 : \text{« } \lambda_1 \text{ can read in } \lambda_2 \text{ memory. »} \]

Example:

\[ f : (\text{'a} \rightarrow b \rightarrow \text{int}_b)_m \]
\[ f \ 1 \sim b \triangleright m \]

Error
Definability:

\[ s, b, m \rightarrow m \]
\[ b \rightarrow b \]
\[ l, c \rightarrow c \]
\[ l, c \rightarrow l \]
\[ s \rightarrow s \]

\( \lambda_1 \rightarrow \lambda_2 \): « \( \lambda_1 \) can be defined in \( \lambda_2 \) memory. »
Definability:

$s, b, m \triangleleft m$

$b \triangleleft b$

$l, c \triangleleft c$

$l, c \triangleleft l$

$s \triangleleft s$

\[ \lambda_1 \triangleleft \lambda_2 : \text{«} \lambda_1 \text{ can be defined in } \lambda_2 \text{ memory. } \text{»} \]

Example:

\[
\langle\langle \text{let multi } f \ x = \ldots \rangle\rangle \rightsquigarrow m \triangleleft c
\]
Definability:

\[ s, b, m \triangleleft m \]
\[ b \triangleleft b \]
\[ l, c \triangleleft c \]
\[ l, c \triangleleft l \]
\[ s \triangleleft s \]

\( \lambda_1 \triangleleft \lambda_2 : \text{«} \lambda_1 \text{ can be defined in } \lambda_2 \text{ memory. } \text{»} \)

Example:

\[
\langle\langle \text{let multi f x = ... } \rangle\rangle \rightsquigarrow m \triangleleft c
\]

Error
Other relations

Propagation

This relation returns the prevailing effect among $\varepsilon$ and $\varepsilon'$.

![Diagram showing relationships between variables]

Serialisation

Is it safe to communicate $\tau_\pi$ to locality $\Lambda$?

\[
\begin{align*}
\text{Seria}_\Lambda(\text{Base}_\pi) &= \text{Base}_\Lambda \text{ if Base = int, string, float,Bool,...} \\
\text{Seria}_\Lambda(\text{Base}_\pi) &= \text{Fail if Base = i/o,...} \\
\text{Seria}_\Lambda(\tau_\pi) &= \begin{cases} 
\tau_\Lambda, & \text{if } \pi \prec \Lambda \\
\text{Fail,} & \text{otherwise}
\end{cases} \\
\text{Seria}_\Lambda(\tau_\pi \text{ par}_b) &= \text{Fail}
\end{align*}
\]
Formal properties

Operational semantics

- Big Step semantics (deterministic)
- Big Step semantics for diverging terms (mutually exclusive)
- Programs that “do not go wrong”:
  - \( v : \mathbb{L}p v \) or \( \mathbb{L}p1 \)

Type safety of a multi-ML program

- Let \( e \) be an expression,
- \( a \) typing environment,
- and \( c \) a set of constraints.
- Then: \( \vdash e : \mathbb{L}[c] \) implies that \( e \) “does not go wrong” (safe)
Formal properties

Operational semantics

- Big Step semantics (deterministic)
Formal properties

Operational semantics

- Big Step semantics (deterministic)
- Big Step semantics for diverging terms (mutually exclusive)
Formal properties

Operational semantics

- Big Step semantics (deterministic)
- Big Step semantics for diverging terms (mutually exclusive)
- Programs that “do not go wrong” : \( \exists v. \downarrow_p^C v \) or \( \downarrow_p^C \infty \)
Formal properties

Operational semantics

- Big Step semantics (deterministic)
- Big Step semantics for diverging terms (mutually exclusive)
- Programs that “do not go wrong”: \((\exists v. \downarrow^L_P v)\) or \((\downarrow^L_P \infty)\)

Type safety of a MULTI-ML program

- Let \(e\) be an expression,
- \(\Gamma\) a typing environment,
- and \(c\) a set of constraint.

Then: \(\Gamma \vdash e : \tau_\pi / \varepsilon [c]\) implies that \(e\) “does not go wrong” \((e \Rightarrow_{safe})\)
Table of Contents

1 Introduction

2 The MULTI-ML language

3 Type system

4 Implementation
   Execution scheme
   Parallel and sequential implementations
   Benchmarks

5 Conclusion
Execution scheme

- One process per leaf/node
- Distributed over physical cores
Execution scheme

Orders/Signals

\[
\begin{align*}
\text{Master} & : \quad f = \text{recv}() \\
\text{Slaves} & : \quad f() \\
p0 & : \quad \text{Signal}^1 \text{ job} \quad \text{Signal}^2 \text{ job} \\
p1 & : \quad f() \\
p2 & : \quad f() 
\end{align*}
\]
Correctness of a **MULTI-ML** program

If $e \Rightarrow_{safe}$ and $\mathcal{WF}(e)$ we have: $\left\langle \left\langle [e]_M, \ldots, [e]_M \right\rangle \right\rangle \Rightarrow_{safe}$
Cost model

Big step semantics with costs

\[ P \frac{\mathcal{M} \vdash e \downarrow^C_p \ e' / C_p}{\text{and}} \frac{\mathcal{M} \vdash e \downarrow^b_p \ e' / C_p / < C_{p.i} >}{\text{and}} \]
Cost model

Big step semantics with costs

\[ \mathcal{P} \]
\[ \mathcal{M} \vdash e \downarrow_f^e e'/C_p \]
and

\[ \mathcal{P} \]
\[ \mathcal{M} \vdash e \downarrow_b^e e'/C_p/ < C_{p,i} > \]

Cost algebra

\[ C ::= \]
\[ 1 \quad \text{Arbitrary unit cost} \]
\[ g \quad \text{g parameter} \]
\[ l \quad \text{l parameter} \]
\[ C \oplus C \quad \text{Addition} \]
\[ C \otimes C \quad \text{Multiplication} \]
\[ \max_{i=0}^n (C_i) \quad \text{Maximum} \]
\[ \sum_{i=0}^n (C_i) \quad \text{Sum} \]
\[ S_{\varepsilon}(v) \quad \text{Data size} \]

C_{1} \oplus C_{2} \equiv C_{2} \oplus C_{1} \]
\[ C_{1} \otimes C_{2} \equiv C_{2} \otimes C_{1} \]
\[ \max_{i=0}^n (C_i) \equiv \text{maximum}(C_0, \ldots, C_n) \]
\[ \sum_{i=0}^n (C_i) \equiv C_0 \oplus \ldots \oplus C_n \]
Distributed implementation

Module
- Communication library
- Based on operational semantics

Current implementation
- MPI processes
- Distributed over physical cores
- Shared/Distributed memory

Future implementations
- TCP/IP
- PTHREAD
- ...

V. Allombert
Sequential 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
    finally 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"
  | |__o "0.1.0-> 3"
  | |__o "0.1.1-> 4"
```
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

![Diagram showing network, machine, multi-core, and thread connections with numbers 0 to 7 and 56 to 63.]
Naive Eratosthenes sieve

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

Results

<table>
<thead>
<tr>
<th></th>
<th>100,000</th>
<th></th>
<th>500,000</th>
<th></th>
<th>1,000,000</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>MULTI-ML</td>
<td>BSML</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>
<td>105.0</td>
<td>125.3</td>
</tr>
<tr>
<td>64</td>
<td>0.3</td>
<td>0.3</td>
<td>1.3</td>
<td>8.7</td>
<td>4.1</td>
</tr>
</tbody>
</table>
Table of Contents

1 Introduction

2 The MULTI-ML language

3 Type system

4 Implementation

5 Conclusion
- BSP \neq \text{Hierarchical architecture}
- BSML \rightarrow \text{BSP \ à la ML}
- No language dedicated to \text{MULTI-BSP}
• MULTI-BSP extension of ML
  - Recursive multi-functions  
    \[
    \text{let } \text{multi } f \ x = \ldots \text{where } \text{node} = \langle \langle f \ldots \rangle \rangle
    \]
  - BSML like code
  - Small syntax extension  
    
    #,$,at,mkpar,finally,mktree,...

• Type system
  - Constraints
  - Effects

• Operational semantics (even for diverging terms)

• Compilation scheme

⇒ Type safety from programs to abstract machines
Before ...

Hardware

- Multi-core
- Cluster
- Super-computer
- Hierarchical architecture

Software

- MULTI-BSP sorting
- State space
- MPI (sub-group)
- ???-ML
... Now

Hardware

- Multi-core
- Cluster
- Super-computer
- Hierarchical architecture

Software

- MULTI-BSP sorting
- State space
- MPI (sub-group)
- MULTI-ML
Future Work

Ongoing work

- Code examples
  - FFT, TDS, PPP, Sort, Nbody, State-Space, MM, ...
- Extensions
  - Language
  - Type system
- MULTI-ML + GPU ⇒ Hybrid architectures

Future work

- Automatic cost analysis
- Certified parallel programming
Thank you for your attention 😊

Questions?
Fusion Sort on Mirev3 with 32 and 64 threads

V. Allombert
Fast Fourier Transform

FFT on Mirev2 (8 machines) with 64 and 128 threads
Fast Fourier Transform

FFT on Mirev2 (6 machines) and Mirev3 (2 machines) with 64 and 128 threads

![Graph showing execution time vs. input size for FFT on Mirev2 and Mirev3 with 64 and 128 threads.](image-url)
## Typing rules

\[
\begin{align*}
\text{LET IN} & \quad \Lambda, \Gamma \vdash e_1 : \tau_{\pi_1}^1 / \varepsilon_1 \quad [c_1] \\
\Lambda, \Gamma; x : \text{Weak}(\tau_{\pi_1}^1, \varepsilon_1) \vdash e_2 : \tau_{\pi_2}^2 / \varepsilon_2 \quad [c_2] \\
& \quad c_3 \equiv [\Psi = \text{Propgt}(\varepsilon_1, \varepsilon_2, c_1, c_2)] \\
\hline
\Lambda, \Gamma \vdash \text{let } x = e_1 \text{ in } e_2 : \tau_{\pi_2}^2 / \Psi \quad [c_3]
\end{align*}
\]

\[
\begin{align*}
& \quad << \text{fun } _\rightarrow \text{ let } x = \text{at } t \text{ in } x >> \\
& \quad << \text{let } x = \text{at } t \text{ in } \text{fun } _\rightarrow \text{ x >>}
\end{align*}
\]