Multi-BSML, une approche à la ML pour la programmation Multi-BSP

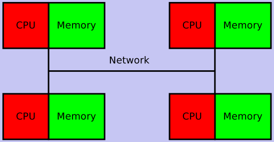
Victor Allombert & Frédéric Gava & Julien Tesson

Laboratory of **A**lgorithms, **C**omplexity and **L**ogic (LACL)
University of Paris-East

Outline

Parallel Architectures

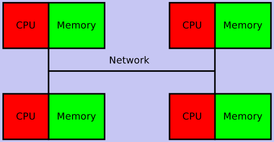
Distributed computing



Clusters

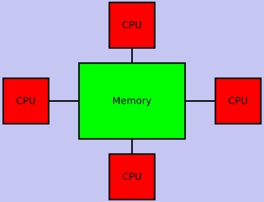
Parallel Architectures

Distributed computing



Clusters

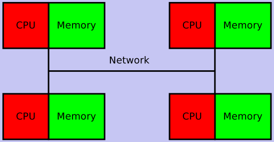
Shared memory



Multi-core

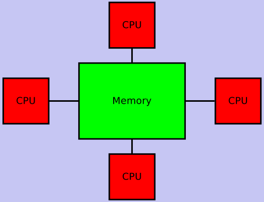
Parallel Architectures

Distributed computing



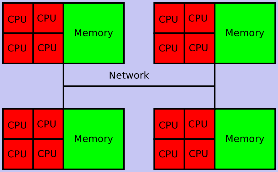
Clusters

Shared memory



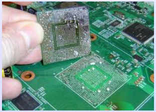
Multi-core

Hybrid model



"Think Parallel or Perish"

GPU



Phone

Tablet

Laptop

PC

Cluster

Supercomputer:

(Parallel) Software Errors

Risks

- Over-consumption
- Erroneous results

Typical bugs

	Distributed	Shared memory
Deadlocks:		

(Parallel) Software Errors

Risks

- Over-consumption
- Erroneous results

Typical bugs

	Distributed	Shared memory
Deadlocks:		

(Parallel) Software Errors

Risks

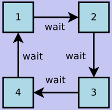
- Over-consumption
- Erroneous results

Typical bugs

Distributed

Shared memory

Deadlocks:



(Parallel) Software Errors

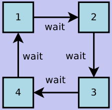
Risks

- Over-consumption
- Erroneous results

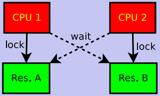
Typical bugs

Deadlocks:

Distributed



Shared memory



(Parallel) Software Errors

Risks

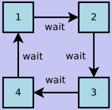
- Over-consumption
- Erroneous results

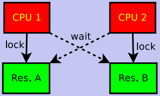
Typical bugs

Distributed

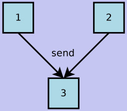
Shared memory

Deadlocks:





Race condition:



(Parallel) Software Errors

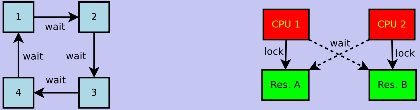
Risks

- Over-consumption
- Erroneous results

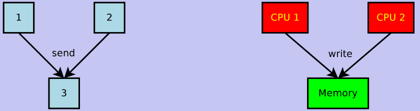
Typical bugs

Distributed | Shared memory

Deadlocks:



Race condition:



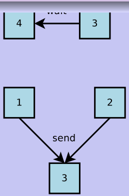
(Parallel) Software Errors

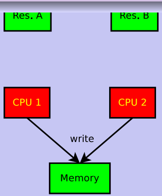
Risks

- Over-consumption
- Erroneous results

Considered solutions

1. Well structured parallelism;
2. Design a high-level language for “hybrid architectures”
3. Software-hardware bridging model ⇒ Portability, scalability





Race condition:

Why Structured Parallelism?

Safety, debugging and verification

Reasoning about cost

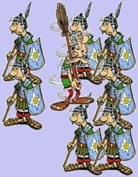
Why Structured Parallelism?

Safety, debugging and verification

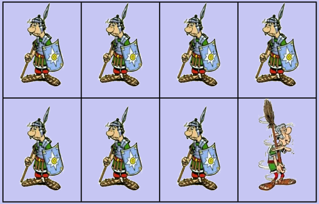
Reasoning about cost

Why Structured Parallelism?

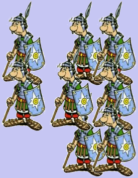
Safety, debugging and verification



⇒



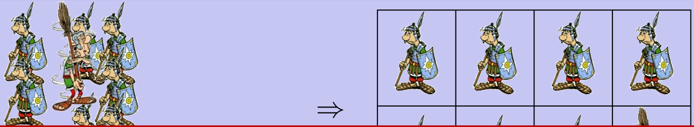
Reasoning about cost



⇒

Why Structured Parallelism?

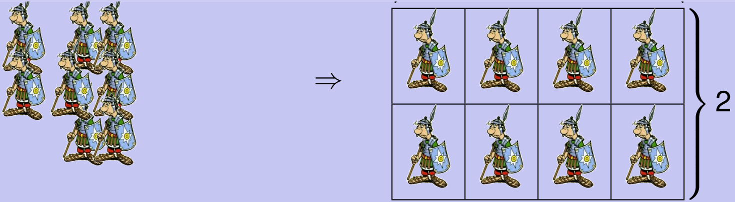
Safety, debugging and verification



Considered solutions

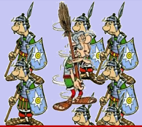
"*Send-receive considered harmful*" (Sergei GORLATCH)

1. Distributed extension of a functional language;
2. Tools for correctness; (mechanized) Semantics ⇒ Coq

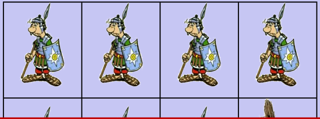


Why Structured Parallelism?

Safety, debugging and verification



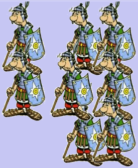
⇒



Considered solutions

"*Send-receive considered harmful*" (Sergei Gorlatch)

1. Distributed extension of a functional language;
2. Tools for correctness; (mechanized) Semantics ⇒ Coq



⇒

Outline

Bridging Model: Bulk Synchronous Parallelism (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

Bridging Model: Bulk Synchronous Parallelism (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

Bridging Model: Bulk Synchronous Parallelism (BSP)

The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

- "Confluent"
- "Deadlock-free"
- Predictable performances

Bridging Model: Bulk Synchronous Parallelism (BSP)

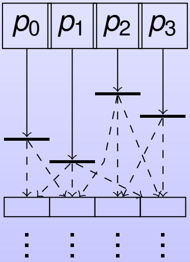
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

- “Confluent”
- “Deadlock-free”
- Predictable performances



local computations

communication

barrier

next super-step

Bridging Model: Bulk Synchronous Parallelism (BSP)

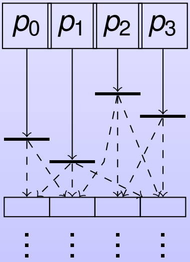
The BSP computer

Defined by:

- p pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

- "Confluent"
- "Deadlock-free"
- Predictable performances



local computations

communication

barrier

next super-step

Bridging Model: Bulk Synchronous Parallelism (BSP)

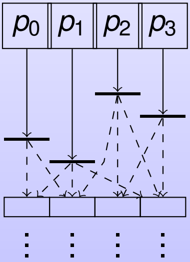
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

- "Confluent"
- "Deadlock-free"
- Predictable performances



local computations

communication

barrier

next super-step

Bridging Model: Bulk Synchronous Parallelism (BSP)

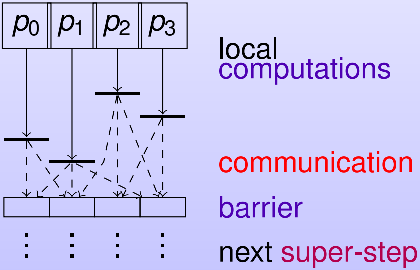
The BSP computer

Defined by:

- **p** pairs CPU/memory
- Communication network
- Synchronisation unit
- Super-steps execution

Properties:

- “Confluent”
- “Deadlock-free”
- Predictable performances



The BSML Language

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaml
- Formal semantics (confluent) → Coq

Main idea

The BSML Language

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaml
- Formal semantics (confluent) → Coq

Main idea

The BSML Language

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaml
- Formal semantics (confluent) $\rightarrow$ Coq

Main idea

The BSML Language

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaml
- Formal semantics (confluent) → Coq

Main idea

- $\langle v_0, \ldots, v_{p-1} \rangle : \alpha$ par $\equiv v_i$ on node i
- Four primitives ⇒ simple semantics

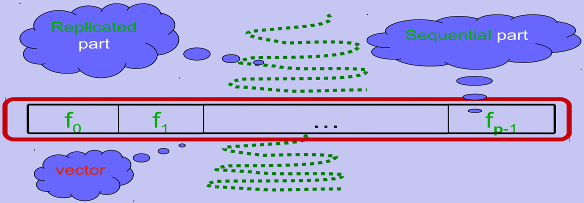
The BSML Language

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaml
- Formal semantics (confluent) → Coq

Main idea

Parallel data structure ⇒ vectors:



1. $\langle\, v_0, \ldots, v_{p-1} \,\rangle : \alpha\ par \equiv v_i$ on node i
2. Four primitives ⇒ simple semantics

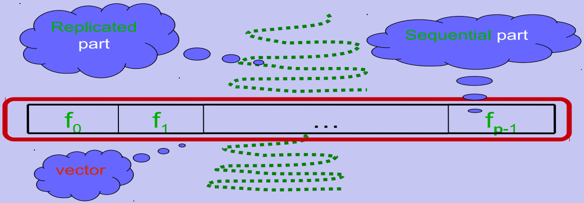
The BSML Language

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaml
- Formal semantics (confluent) → Coq

Main idea

Parallel data structure ⇒ vectors:



1. $\langle\, v_0, \ldots, v_{p-1} \,\rangle : \alpha\, par \equiv v_i$ on node i
2. Four primitives ⇒ simple semantics

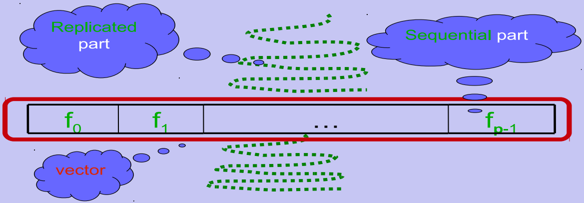
The BSML Language

BSML

- Explicit BSP programming with a functional approach
- Based upon ML; Implemented over OCaml
- Formal semantics (confluent) → Coq

Main idea

Parallel data structure ⇒ vectors:



1. $\langle\ v_0, \ldots, v_{p-1}\ \rangle : \alpha\,\text{par} \equiv v_i$ on node i
2. Four primitives ⇒ simple semantics

The BSML Primitives

Asynchronous operations

- $\langle\langle \ldots \rangle\rangle$: local execution (vector)
- \$v\$: element of a parallel vector v
- \$pid\$: id of the processor

The BSML Primitives

Asynchronous operations

- $\langle\langle \ \dots \ \rangle\rangle$: local execution (vector)
- v: element of a parallel vector v
- pid: id of the processor

Communication

The BSML Primitives

Asynchronous operations

- ⟪ . . . ⟫ : local execution (vector)
- \$v\$: element of a parallel vector v
- \$pid\$: id of the processor

Communication

The BSML Primitives

Asynchronous operations

- $\langle\langle \ldots \rangle\rangle$: local execution (vector)
- \$v\$: element of a parallel vector v
- \$**pid**\$: id of the processor

Communication

- **proj:** $\langle x_0, \ldots, x_{p-1} \rangle \mapsto \begin{pmatrix} x_0 \\ \vdots \\ x_{p-1} \end{pmatrix}$
- **put:** $\langle f_0, \ldots, f_{p-1} \rangle \mapsto \left\langle \begin{matrix} f_0\ 0 \\ \vdots \\ f_{p-1}\ 0 \end{matrix}, \ldots, \begin{matrix} f_0\ (p-1) \\ \vdots \\ f_{p-1}\ (p-1) \end{matrix} \right\rangle$
- super: evaluation of two expressions into super-threads

The BSML Primitives

Asynchronous operations

- $\langle\langle \ldots \rangle\rangle$: local execution (vector)
- \$v\$: element of a parallel vector v
- \$**pid**\$: id of the processor

Communication

- **proj**: $\langle\ x_0, \ldots, x_{\mathbf{p}-1}\ \rangle \mapsto \begin{array}{c} x_0 \\ \vdots \\ x_{\mathbf{p}-1} \end{array}$
- **put**: $\langle\ f_0, \ldots, f_{\mathbf{p}-1}\ \rangle \mapsto \left\langle \begin{array}{c} f_0\ 0 \\ \vdots \\ f_{\mathbf{p}-1}\ 0 \end{array}, \ldots, \begin{array}{c} f_0\ (\mathbf{p}-1) \\ \vdots \\ f_{\mathbf{p}-1}\ (\mathbf{p}-1) \end{array} \right\rangle$
- **super**: evaluation of two expressions into super-threads

The BSML Primitives

Asynchronous operations

- $\langle\langle \ldots \rangle\rangle$: local execution (vector)
- \$v\$: element of a parallel vector v
- \$**pid**\$: id of the processor

Communication

- **proj**: $\langle\ x_0, \ldots, x_{\mathbf{p}-1}\ \rangle \mapsto \begin{array}{c} x_0 \\ \vdots \\ x_{\mathbf{p}-1} \end{array}$
- **put**: $\langle\ f_0, \ldots, f_{\mathbf{p}-1}\ \rangle \mapsto \left\langle \begin{array}{c} f_0\ 0 \\ \vdots \\ f_{\mathbf{p}-1}\ 0 \end{array}, \ldots, \begin{array}{c} f_0\ (\mathbf{p}-1) \\ \vdots \\ f_{\mathbf{p}-1}\ (\mathbf{p}-1) \end{array} \right\rangle$
- **super**: evaluation of two expressions into super-threads

The BSML Primitives

Asynchronous operations

- $\langle\!\langle \ldots \rangle\!\rangle$: local execution (vector)
- v: element of a parallel vector v
- pid: id of the processor

Communication

- **proj**: $\langle\, x_0, \ldots, x_{\mathbf{p}-1} \,\rangle \mapsto \begin{array}{c} x_0 \\ \vdots \\ x_{\mathbf{p}-1} \end{array}$
- **put**: $\langle\, f_0, \ldots, f_{\mathbf{p}-1} \,\rangle \mapsto \left\langle \begin{array}{c} f_0\ 0 \\ \vdots \\ f_{\mathbf{p}-1}\ 0 \end{array}, \ldots, \begin{array}{c} f_0\ (\mathbf{p}-1) \\ \vdots \\ f_{\mathbf{p}-1}\ (\mathbf{p}-1) \end{array} \right\rangle$
- **super**: evaluation of two expressions into super-threads

The BSML Primitives

Asynchronous operations

- $\langle\langle \ldots \rangle\rangle$: local execution (vector)
- \$v\$: element of a parallel vector v
- \$**pid**\$: id of the processor

Communication

- **proj**: $\langle\, x_0, \ldots, x_{\mathbf{p}-1} \,\rangle \mapsto \begin{array}{c} x_0 \\ \vdots \\ x_{\mathbf{p}-1} \end{array}$
- **put**: $\langle\, f_0, \ldots, f_{\mathbf{p}-1} \,\rangle \mapsto \left\langle \begin{array}{c} f_0\ 0 \\ \vdots \\ f_{\mathbf{p}-1}\ 0 \end{array}, \ldots, \begin{array}{c} f_0\ (\mathbf{p}-1) \\ \vdots \\ f_{\mathbf{p}-1}\ (\mathbf{p}-1) \end{array} \right\rangle$
- **super**: evaluation of two expressions into super-threads

Implementation

Modular

- Based on a semantic study:
  BSML $\equiv$ **p**$\times$ ML + 2 BSP instructions (SPMD style)
- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

Inspire:

Implementation

Modular

- Based on a semantic study:

  BSML $\equiv$ **p**$\times$ ML + 2 BSP instructions (SPMD style)
- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching implemented using a modification of the source code
- Superposition using

Inspire:

Implementation

Modular

- Based on a semantic study:

  BSML $\equiv$ **p** $\times$ ML + 2 BSP instructions (SPMD style)
- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:

Inspire:

Implementation

Modular

- Based on a semantic study:

  BSML $\equiv$ **p**$\times$ ML + 2 BSP instructions (SPMD style)

- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:

Implementation

Modular

- Based on a semantic study:
  BSML $\equiv \mathbf{p} \times$ ML + 2 BSP instructions (SPMD style)
- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:
  1. System threads $\Rightarrow$ slowdown if there is too many threads
  2. A CPS (Continuation Passing Style) transformation

Inspire:

Implementation

Modular

- Based on a semantic study:
  BSML $\equiv$ **p**$\times$ ML + 2 BSP instructions (SPMD style)
- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:
  1. System threads $\Rightarrow$ slowdown if there is too many threads
  2. A CPS (Continuation Passing Style) transformation

Inspire:

Implementation

Modular

- Based on a semantic study:

  BSML $\equiv$ **p**$\times$ ML + 2 BSP instructions (SPMD style)

- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:
  1. System threads $\Rightarrow$ slowdown if there is too many threads
  2. A CPS (Continuation Passing Style) transformation

Inspire:

- Imperative: BSP++ (J. Falcou), BSP-Python (K. Hinsen)
- Functional: BSP-Haskell (Q. Miller)

Implementation

Modular

- Based on a semantic study:

  BSML $\equiv$ **p**$\times$ ML + 2 BSP instructions (SPMD style)

- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:
  1. System threads $\Rightarrow$ slowdown if there is too many threads
  2. A CPS (Continuation Passing Style) transformation

Inspire:

- Imperative: BSP++ (J. Falcou) , BSP-Python (K. Kinsen)
- Functional: BSP-Haskell (Q. Miller), Snow/BSP-R (N. Li)

Implementation

Modular

- Based on a semantic study:

  BSML $\equiv$ **p** $\times$ ML + 2 BSP instructions (SPMD style)

- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:
  1. System threads $\Rightarrow$ slowdown if there is too many threads
  2. A CPS (Continuation Passing Style) transformation

Inspire:

- Imperative: BSP++ (J. Falcou) , BSP-Python (K. Kinsen)
- Functional: BSP-Haskell (Q. Miller), Snow/BSP-R (N. Li)

Implementation

Modular

- Based on a semantic study:
  BSML $\equiv$ **p** $\times$ ML + 2 BSP instructions (SPMD style)
- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:
  1. System threads $\Rightarrow$ slowdown if there is too many threads
  2. A CPS (Continuation Passing Style) transformation

Inspire:

- Imperative: BSP++ (J. Falcou) , BSP-Python (K. Kinsen)
- Functional: BSP-Haskell (Q. Miller), Snow/BSP-R (N. Li)

Implementation

Modular

- Based on a semantic study:

  BSML $\equiv$ **p** $\times$ ML + 2 BSP instructions (SPMD style)
- Different implementations: TCP/IP, MPI, PUB, $\cdots$

Extensions

- Exception mechanism and pattern matching: implemented using a modification of the source code
- Superposition using:
  1. System threads $\Rightarrow$ slowdown if there is too many threads
  2. A CPS (Continuation Passing Style) transformation

Inspire:

- Imperative: BSP++ (J. Falcou) , BSP-Python (K. Kinsen)
- Functional: BSP-Haskell (Q. Miller), Snow/BSP-R (N. Li)

Example : BSP Sampling Sort

```
(* psrs: int par → 'a list → 'a list *)
let psrs lvlengths lv =
    (* super−step 1(a): local sorting *)
  let locsort = ≪ List.sort compare $lv$ ≫ in

    (* super−step 1(b): selection of the primary samples *)
  let regsampl = ≪ extract_n P $lvlengths$ $locsort$ ≫ in

    (* super−step 2(a): total exchange of the primary samples;*)
  let glosampl = List.sort compare (proj regsampl) in

    (* super−step 2(b): selection of the secondary samples *)
  let pivots = extract_n P (P*(P−1)) glosampl in

    (* super−step 2(c) : building the communicated lists of values *)
  let comm = ≪ slice_p $locsort$ pivots ≫ in

   (* super−step 3: sended them and merging of the received values *)
  let recv = put ≪ List.nth $comm$ ≫ in
    ≪ p_merge P (List.map $recv$ procs_list) ≫
```

Advantages and Drawbacks

Advantages

- Easy to learn
- "All" OCaml codes can be used
- Easy to get a BSML code from a BSP algorithm

Drawbacks

Advantages and Drawbacks

Advantages

- Easy to learn
- "All" OCaml codes can be used
- Easy to get a BSML code from a BSP algorithm

Drawbacks

Advantages and Drawbacks

Advantages

- Easy to learn
- “All” OCaml codes can be used
- Easy to get a BSML code from a BSP algorithm

Drawbacks

- Hierarchical architecture as a flat one
- Congestion using network

Advantages and Drawbacks

Advantages

- Easy to learn
- "All" OCaml codes can be used
- Easy to get a BSML code from a BSP algorithm

Drawbacks

- Hierarchical architecture as a flat one
- Congestion using network

Advantages and Drawbacks

Advantages

- Easy to learn
- "All" OCaml codes can be used
- Easy to get a BSML code from a BSP algorithm

Drawbacks

- Hierarchical architecture as a flat one
- Congestion using network

Advantages and Drawbacks

Advantages

- Easy to learn
- “All” OCaml codes can be used
- Easy to get a BSML code from a BSP algorithm

Drawbacks

- Hierarchical architecture as a flat one
- Congestion using network

Advantages and Drawbacks

Advantages

- Easy to learn
- “All” OCaml codes can be used
- Easy to get a BSML code from a BSP algorithm

Drawbacks

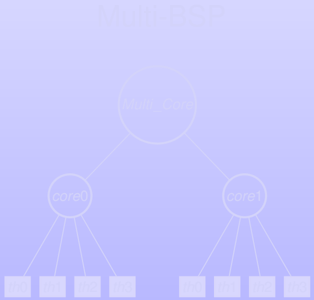
- Hierarchical architecture as a flat one
- Congestion using network

Outline

Multi-BSP Model (1)

What is Multi-BSP? (Valiant)

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

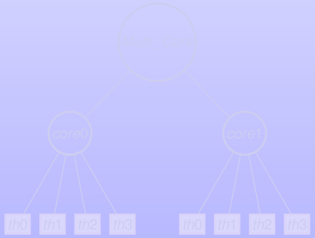


Multi-BSP Model (1)

What is Multi-BSP? (Valiant)

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

Multi-BSP



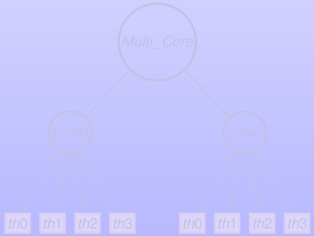
BSP

Multi-BSP Model (1)

What is Multi-BSP? (Valiant)

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

Multi-BSP

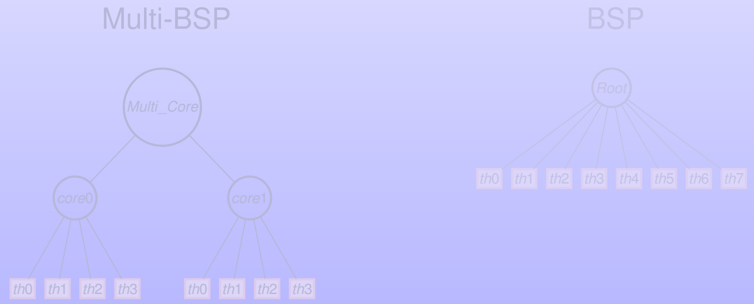


BSP

Multi-BSP Model (1)

What is Multi-BSP? (Valiant)

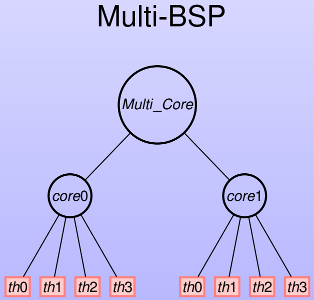
1. A tree structure with nested components
2. Where nodes have a storage capacity
3. And leaf are homogenous processors

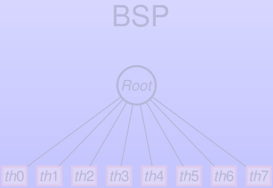


Multi-BSP Model (1)

What is Multi-BSP? (Valiant)

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



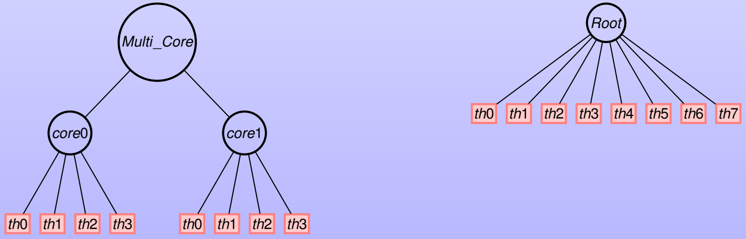


Multi-BSP Model (1)

What is Multi-BSP? (Valiant)

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

Multi-BSP BSP



Multi-BSP Model (2)

Cost model

A d depth tree is specified by $4 \times d$ parameters:

- **p** : Number of sub-components
- **m** : Available memory at a level
- **g** : Bandwidth with the upper level
- **L** : Synchronisation

Multi-BSP Model (2)

Cost model

A d depth tree is specified by $4 \times d$ parameters:

- **p** : Number of sub-components
- **m** : Available memory at a level
- **g** : Bandwidth with the upper level
- **L** : Synchronisation

Example: 16 quad-chips with octo-cores

- Level 4 ($\mathbf{p}=16, \mathbf{g}=\infty, \mathbf{L}=1000, \mathbf{m}=16Tb$) (RAM/IO)
- Level 3 ($\mathbf{p}=4, \mathbf{g}=150, \mathbf{L}=100, \mathbf{m}=64Gb$) (RAM)
- Level 2 ($\mathbf{p}=8, \mathbf{g}=5, \mathbf{L}=10, \mathbf{m}=2Mb$) (L2 cache)
- Level 1 ($\mathbf{p}=1, \mathbf{g}=1, \mathbf{L}=1, \mathbf{m}=8Kb$) (L1 cache)

Multi-BSP Model (3)

Execution model

At a level i, a super-step is:

- Each component at level $i-1$ does its own super-steps
- Then each copies some data to the memory at level i
- Then synchronisation
- Finaly copy of some data from level i to $i-1$

Multi-BSP Model (3)

Execution model

At a level i, a super-step is:

- Each component at level $i - 1$ does its own super-steps
- Then each copies some data to the memory at level i
- Then synchronisation
- Finaly copy of some data from level i to $i - 1$

Advantages and drawbacks

- Implicit subgroup synchronisation
- Recursive decomposition of problems
- Harder to design/cost some algorithms

Multi-BSML Language (1)

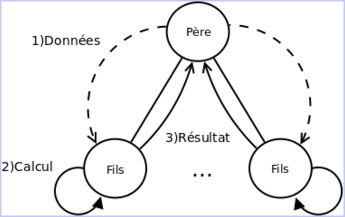
Syntaxic construction

```
let multi f [args] =
 let cst = CodeOCaml
 where node [args] = CodeBSML ...
                     ≪ f args ≫
                   ... CodeBSML
 where leaf [args] = CodeOCaml
 in f ...
```

Multi-BSML Language (1)

Syntaxic construction

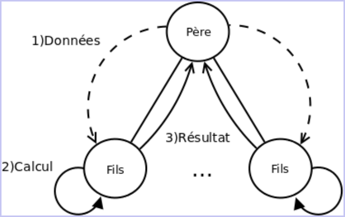
let multi f [args] =
  let cst = CodeOCaml
  where node [args] = CodeBSML ...
      ≪ f args ≫
      ... CodeBSML
  where leaf [args] = CodeOCaml
  in f ...



Multi-BSML Language (1)

Syntaxic construction

let multi f [args] =
let cst = CodeOCaml
where node [args] = CodeBSML ...
$\ll$ f args $\gg$
... CodeBSML
where leaf [args] = CodeOCaml
in f ...



Limitations and differences

- Nodes are implicit computation units
- Horizontal communications between level components
- Garbage collector $\Rightarrow$ no L1, L2 caches.

Multi-BSML Language (2)

Semantics of multi

- BSML code to distribute values
- $\langle\langle \ldots \rangle\rangle$ and **proj**; level changing
- Mutual recursive functions of standard OCaml values
- (Formal) Big-steps $\Longleftrightarrow$ small-steps for a mini-Multi-BSML

Multi-BSML Language (2)

Semantics of multi

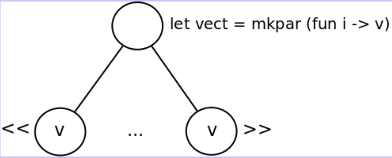
- BSML code to distribute values
- $\langle\langle \ldots \rangle\rangle$ and **proj**; level changing
- Mutual recursive functions of standard OCaml values
- (Formal) Big-steps $\Longleftrightarrow$ small-steps for a mini-Multi-BSML

Copy memory values and distribution of values

- **let** x = 1 **in** $\ll$ #x# + 1 $\gg$
- **mkpar** (**fun** i $\rightarrow$ e)

  $\mapsto \langle\, v_0, \ldots, v_{p-1} \,\rangle$

  where $(e\ i) \mapsto v_i$



Example : Sum Of The Elements Of A List

```
let multi sum_list l =

   where node l =
     let v = mkpar (fun i → split i l) in
      sumSeq (flatten ≪ sum_list $v$ ≫ ) (* flatten uses a proj *)
                                          (* sumSeq is List.fold_left *)
   where leaf l = sumSeq l

 in ... (sum_list lst) ...
```

Multi With Tree Construction

Goals

- Keep values on each node and leaf
- To program multiple phases of multi

Multi With Tree Construction

Goals

- Keep values on each node and leaf
- To program multiple phases of multi

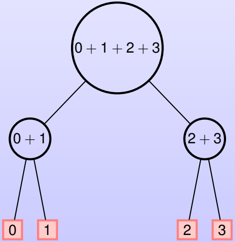
Extension

Two new keywords:

- finally; pushes up a value and keeps a value
- where default; keeps a value even if the recursive calls generates partial trees; Optional if the language allows to raise exceptions

Example

Keep the intermediate results of the sum




```
let multi sum_list l =

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

  where leaf l =
    let s = sumSeq l in
     finally ~up:s ~keep:s

  where default = 0 (* not used *)
```

Implementation

Sequential (currently test phase)

For debugging and toplevel

- tree structure of data
- A global tree of Hashtables to represent the memories

Implementation

Sequential (currently test phase)

For debugging and toplevel

- tree structure of data
- A global tree of Hashtables to represent the memories

Distributed (ongoing work)

Modular (MPI, TCP/IP, etc.) and based on formal semantics.

Implementation

Sequential (currently test phase)

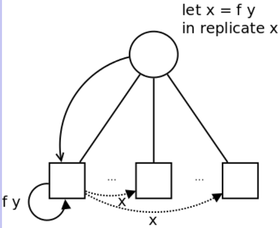
For debugging and toplevel

- tree structure of data
- A global tree of Hashtables to represent the memories

Distributed (ongoing work)

Modular (MPI, TCP/IP, etc.) and based on formal semantics.

Shared memory



Implementation

Sequential (currently test phase)

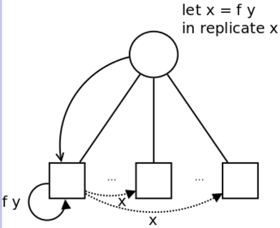
For debugging and toplevel

- tree structure of data
- A global tree of Hashtables to represent the memories

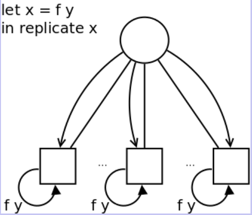
Distributed (ongoing work)

Modular (MPI, TCP/IP, etc.) and based on formal semantics.

Shared memory



Distributed



Skeleton (For Coq) Example

Why ?

Embed Multi in Coq:

- Syntax extensions not friendly in Coq
- **finally** too close to a monad (side effect)

Skeleton (For Coq) Example

Why ?

Embed Multi in Coq:

- Syntax extensions not friendly in Coq
- **finally** too close to a monad (side effect)

The multi skeleton

mktree : 'a tree → 'b → 'e tree → down → leaff → up → control → ('c∗'e tree)

where

down: 'a → 'b → 'b par

leaf: 'e → 'a → 'b → ('c ∗ 'e)

up: 'd par → 'e → 'a → 'b → ('c ∗ 'e)

control: 'c par→ 'e→ 'a→ 'b→ UP **of** ('d par ∗ 'e) | DOWN **of** ('b ∗ 'e)

Outline

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP ⇒ BSML
- Multi-BSP ⇒ Multi-BSML

Multi-BSML

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP ⇒ BSML
- Multi-BSP ⇒ Multi-BSML

Multi-BSML

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP $\Rightarrow$ BSML
- Multi-BSP $\Rightarrow$ Multi-BSML

Multi-BSML

- Recursive functions on different memories of chips
- Structured nesting of codes
- Big-steps and small-steps formal semantics (confluent)
- A skeleton for
- Small number of primitives and little syntax extension

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP $\Rightarrow$ BSML
- Multi-BSP $\Rightarrow$ Multi-BSML

Multi-BSML

- Recursive functions on different memories of chips
- Structured nesting of BSML codes
- Big-steps and small-steps formal semantics (confluent
- A skeleton for
- Small number of primitives and little syntax extension

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP $\Rightarrow$ BSML
- Multi-BSP $\Rightarrow$ Multi-BSML

Multi-BSML

- Recursive functions on different memories of chips
- Structured nesting of BSML codes
- Big-steps and small-steps formal semantics (confuent)
- A skeleton for Coq
- Small number of primitives and little syntax extension

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP $\Rightarrow$ BSML
- Multi-BSP $\Rightarrow$ Multi-BSML

Multi-BSML

- Recursive functions on different memories of chips
- Structured nesting of BSML codes
- Big-steps and small-steps formal semantics (confuent)
- A skeleton for Coq
- Small number of primitives and little syntax extension

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP ⇒ BSML
- Multi-BSP ⇒ Multi-BSML

Multi-BSML

- Recursive functions on different memories of chips
- Structured nesting of BSML codes
- Big-steps and small-steps formal semantics (confuent)
- A skeleton for Coq
- Small number of primitives and little syntax extension

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP ⇒ BSML
- Multi-BSP ⇒ Multi-BSML

Multi-BSML

- Recursive functions on different memories of chips
- Structured nesting of BSML codes
- Big-steps and small-steps formal semantics (confuent)
- A skeleton for Coq
- Small number of primitives and little syntax extension

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP ⇒ BSML
- Multi-BSP ⇒ Multi-BSML

Multi-BSML

- Recursive functions on different memories of chips
- Structured nesting of BSML codes
- Big-steps and small-steps formal semantics (confuent)
- A skeleton for Coq
- Small number of primitives and little syntax extension

Conclusion

Multi

- Multi-BSP extension of BSP for hierarchical architectures
- BSP $\Rightarrow$ BSML
- Multi-BSP $\Rightarrow$ Multi-BSML

Multi-BSML

- Recursive functions on different memories of chips
- Structured nesting of BSML codes
- Big-steps and small-steps formal semantics (confuent)
- A skeleton for Coq
- Small number of primitives and little syntax extension

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again

Long term (team)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad [illegible] of data
  - Bad [illegible] of the parallel operators

Long term (team)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

- Mechanized semantics in Coq
- Embed in for proofs and extraction of programs as for
- Examples and libraries
- Same work for other languages (C++, Java)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

- Mechanized semantics in Coq
- Embed in Coq for proofs and extraction of multi-BSML programs as for BSML
- Examples and libraries
- Same work for other languages (C++, Java)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

- Mechanized semantics in Coq
- Embed in Coq for proofs and extraction of multi-BSML programs as for BSML
- Examples and libraries
- Same work for other languages (C++, Java)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

- Mechanized semantics in Coq
- Embed in Coq for proofs and extraction of multi-BSML programs as for BSML
- Examples and libraries
- Same work for other languages (C++, Java)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

- Mechanized semantics in Coq
- Embed in Coq for proofs and extraction of multi-BSML programs as for BSML
- Examples and libraries
- Same work for other languages (C++, Java)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

- Mechanized semantics in Coq
- Embed in Coq for proofs and extraction of multi-BSML programs as for BSML
- Examples and libraries
- Same work for other languages (C++, Java)

Perspectives (Ongoing/Future Work)

Short term (for Victor's Phd)

- Implementation using MPI
- Examples and benchmarks
- Type system for a subpart of OCaml again
  - Bad nesting of parallelism
  - Bad copy of data
  - Bad use of the parallel operators

Long term (team)

- Mechanized semantics in Coq
- Embed in Coq for proofs and extraction of multi-BSML programs as for BSML
- Examples and libraries
- Same work for other languages (C++, Java)

Merci !