Outline

1. Motivations

2. BSML: Bulk Synchronous Parallel ML
   - The BSP Model
   - Overview of the BSML Language
   - Classic BSML Primitives
   - Revised Bulk Synchronous Parallel ML

3. BSML in Coq
   - Shallow Embedding of BSML in Coq
   - BSML Programming and Reasoning in Coq
   - Extraction

4. Summary
1 Motivations

2 BSML: Bulk Synchronous Parallel ML

3 BSML in Coq

4 Summary

Our Goal

To ease the development of correct and verifiable parallel programs with predictable performances

We should address:

- the easy development of **correct** and verifiable programs
- the easy development of **parallel** programs
- the easy development of parallel programs with **predictable** performances
### Easy Development of Correct and Verifiable Programs

- **high-level languages**: expressive, modular, less error-prone
- **high-level languages** have simpler semantics, and could have a complete formal semantics (e.g. Standard ML, ISO Prolog)
- therefore verification of programs is possible and easier

⇒ a **high-level parallel language with formal semantics**

### The Easy Development of Parallel Programs with Predictable Performances

- assumption: the goal is to program functions
- issues: non-determinism, deadlocks, difficulty to read programs, complex semantics and verification, portability . . .
- it is also very important for the programmer to be able to reason about the performance of the programs

⇒ a **structured parallel model which allows the design of portable parallel algorithms with a simple cost model**
The Bulk Synchronous Parallel ML Approach

Choices

- an efficient functional programming language with formal semantics and easy reasoning about the performance of programs (strict evaluation):
  
  **ML** (Objective Caml flavor)

- a restricted model of parallelism with no deadlock, very limited cases of non-determinism, a simple cost model:

  **Bulk Synchronous Parallelism**

The result is:

**Bulk Synchronous Parallel ML (BSML)**

---

Outline

1. Motivations
2. BSML: Bulk Synchronous Parallel ML
3. BSML in Coq
4. Summary
Outline

1 Motivations

2 BSML: Bulk Synchronous Parallel ML
   The BSP Model
   Overview of the BSML Language
   Classic BSML Primitives
   Revised Bulk Synchronous Parallel ML

3 BSML in Coq
   Shallow Embedding of BSML in Coq
   BSML Programming and Reasoning in Coq
   Extraction

4 Summary

Bulk Synchronous Parallelism (BSP)

Research on BSP
90’ by Valiant (Cambridge) and McColl (Oxford)

Three models

- abstract architecture
- execution model
- cost model

BSP Computer

- p processor / memory pairs (of speed \( r \))
- a communication network (of speed \( g \))
- a global synchronisation unit (of speed \( L \))
Bulk Synchronous Parallelism

### Execution model

![Diagram showing the execution model of Bulk Synchronous Parallelism](image)

### Cost model

\[ T(s) = \max_{0 \leq i < p} w_i + h \times g + L \]

where \( h = \max_{0 \leq i < p} \{ h_i^+, h_i^- \} \)

---

### Outline

1. Motivations
2. BSML: Bulk Synchronous Parallel ML
   - The BSP Model
   - Overview of the BSML Language
   - Classic BSML Primitives
   - Revised Bulk Synchronous Parallel ML
3. BSML in Coq
   - Shallow Embedding of BSML in Coq
   - BSML Programming and Reasoning in Coq
   - Extraction
4. Summary
Bulk Synchronous Parallel ML

Design principles
- Small set of parallel primitives
- Universal for bulk synchronous parallelism
- Global view of programs
- Simple semantics

BSML
- a sequential functional language
  + a parallel data structure
  + parallel operations on this data structure

A Parallel Data Structure

Parallel Vectors
- An abstract polymorphic datatype: 'a par
- Fixed size p: each processor has a value of type 'a
- no nesting allowed

⇒ Direct mapping eases the reasoning about performances

Notation
\[ \langle v_0, \ldots, v_{p-1} \rangle \]
Outline

1 Motivations

2 BSML: Bulk Synchronous Parallel ML
   The BSP Model
   Overview of the BSML Language
   Classic BSML Primitives
   Revised Bulk Synchronous Parallel ML

3 BSML in Coq
   Shallow Embedding of BSML in Coq
   BSML Programming and Reasoning in Coq
   Extraction

4 Summary

Classic BSML Primitives

- Access to the BSP parameters:

  - `bsp_p`: int
  - `bsp_r`: float
  - `bsp_g`: float
  - `bsp_l`: float

  ⇒ Programs with performance portability

- Manipulation of parallel vectors:

  - `mkpar`: (int → 'a) → 'a par
  - `proj`: 'a par → (int → 'a)
  - `apply`: ('a → 'b) par → 'a par → 'b par
  - `put`: (int → 'a) par → (int → 'a) par
Creation of parallel vectors

**Signature**

\[ \text{mkpar : (int \to 'a) \to 'a par} \]

**Informal semantics**

\[ \text{mkpar } f = \langle f\ 0,\ f\ 1,\ \ldots,\ f\ (p - 1) \rangle \]

**Examples**

```ocaml
# let this = mkpar(fun pid -> pid);;
val this : int par = <0, 1, 2, 3, 4, 5, 6, 7>

# let plusMinus = mkpar(fun pid ->if pid mod 2=0
then fun x->x+1
else fun x->x-1);;
val plusMinus : (int -> int) par =
  <<fun>, <fun>, <fun>, <fun>, <fun>, <fun>, <fun>, <fun>>
```

**BSP Cost**

\[
\max_{0 \leq i < p} \| f\ i \| \quad \text{where} \quad \| e \| \text{ is the time required to evaluate } e
\]

---

Projection

**Signature**

\[ \text{proj: 'a par \to (int \to 'a)} \]

**Informal semantics**

\[
\text{proj } \langle v_0, \ldots, v_{p-1} \rangle =
\begin{array}{c}
\text{function} \quad 0 \rightarrow v_0 \\
\vdots \\
p - 1 \rightarrow v_{p-1}
\end{array}
\]

**Remark**

- Should not be evaluated in the context of a \text{mkpar}
- Returned function is partial: \text{proj (mkpar f) \neq f}

**Example**

```ocaml
# proj (mkpar string_of_int) 2;;
val it : string = "2"
```

**BSP Cost**

\[
\max_{0 \leq i < p} \{ |v_i|, \sum_{j \neq i} |v_j| \} \times g + L \quad \text{where } |e| \text{ is the value of } e\text{'s size}
\]
Point-wise parallel application

**Signature**
apply : ('a → 'b) par → 'a par → 'b par

**Informal semantics**
apply (f_0, ..., f_{p-1}) (v_0, ..., v_{p-1}) = (f_0 v_0, ..., f_{p-1} v_{p-1})

**Example**
# let v = apply plusMinus this;;
val v : int par = <1, 0, 3, 2, 5, 4, 7, 6>

**BSP Cost**
\[
\max_{0 \leq i < p} \|f_i v_i\|
\]

Communication I

**Signature**
put: (int → 'a) par → (int → 'a) par

**Informal semantics**
put (f_0, ..., f_{p-1}) = (g_0, ..., g_{p-1}) with g_j ≡ fun src → f_{src} j

**Remark**
- function f_i encodes the p messages to be sent from processor i
- (f_i j) is the message to be sent from i to j
- function g_j encodes the p messages received by processor j
- (g_j i) is the message received by j from i
Communication II

Example
apply(mkpar(fun _->fun f->List.map f Stdlib.Base.procs))
(put(mkpar(fun pid dst->
    if dst=(pid+1) mod bsp_p
    then Some pid
    else None)));;
- : int option list par =
< [None; None; None; None; None; None; None; Some 7],
[Some 0; None; None; None; None; None; None; None],
[None; Some 1; None; None; None; None; None; None],
[None; None; Some 2; None; None; None; None; None],
[None; None; None; Some 3; None; None; None; None],
[None; None; None; None; Some 4; None; None; None],
[None; None; None; None; None; Some 5; None; None],
[None; None; None; None; None; None; Some 6; None]>
A More Complicated Example

Communication pattern to implement

\[ \langle \ldots, \ldots, a_1^i, \ldots, a_n^i, \ldots, \ldots, \rangle \]

Implementation

\[
\begin{align*}
\text{let getBounds first last v =} \\
\text{let p = bsp_p in} \\
\text{let parfun f v = apply (mkpar(fun _ -> f)) v in} \\
\text{let lasts = parfun last v in} \\
\text{let firsts = parfun first v in} \\
\text{let msg = put(apply\(apply\(}
\text{ (mkpar(fun pid first last dst->}
\text{ if dst=(pid+1) mod p then Some last}
\text{ else if dst=(p+pid-1) mod p}
\text{ then Some first else None))}
\text{ firsts) lasts) in} \\
\text{ ( apply msg (mkpar(fun pid->(p+pid-1) mod p)),}
\text{ apply msg (mkpar(fun pid->(pid+1) mod p)))}
\end{align*}
\]

Outline

1. Motivations
2. BSML: Bulk Synchronous Parallel ML
   - The BSP Model
   - Overview of the BSML Language
   - Classic BSML Primitives
   - Revised Bulk Synchronous Parallel ML
3. BSML in Coq
   - Shallow Embedding of BSML in Coq
   - BSML Programming and Reasoning in Coq
   - Extraction
4. Summary
Levels of Execution in BSML

- **Replicated execution** (default)
  - “sequential” ML code
  - every processor does the same

- **Local execution**
  - what happens inside parallel vectors, on each of their components
  - uses local data
  - may be different on different processors

- **Global execution**
  - concerns the set of all processors as a whole
  - example: communications

Revised BSML

- Classic BSML: impossible to use vectors in a local section
- Revised BSML: access to local information of vector $v$ noted $\$v\$, possible only in a local section, written $\ll e \gg$
- Examples:
  
  ```
  let mkpar f = $\ll f \$this$ \gg$
  let apply fv vv = $\ll f\$v$ $\$v\$ \gg$
  let parfun f v = $\ll f \$v$ \gg$
  
  => mkpar and apply are no longer primitives in Revised BSML
  ```

A Revised More Complicated Example

Implementation

let getBounds first last v =
  let p = bsp_p in
  let lasts = « last $v$ » in
  let firsts = « first $v$ » in
  let msg = put « fun dst -> if dst=(this + 1) mod p
                      then Some lasts
                      else if dst=(p + this - 1) mod p
                      then Some firsts
                      else None » in
  « msg ((p + this - 1) mod p) »,
  « msg ((this + 1) mod p) »

Parallel Implementation

Version 0.5 (November 2010) & Version 0.51 (November 2013)

- Library for OCaml: Primitives + a Standard Library
- Modular implementation: MPI, TCP/IP, sequential
- What’s new?
  - Revised BSML
  - New configuration and build process
- http://traclifo.univ-orleans.fr/BSML
Motivations

BSML: Bulk Synchronous Parallel ML
- The BSP Model
- Overview of the BSML Language
- Classic BSML Primitives
- Revised Bulk Synchronous Parallel ML

BSML in Coq
- Shallow Embedding of BSML in Coq
- BSML Programming and Reasoning in Coq
- Extraction

Summary
Deep vs. Shallow Embedding

Deep Embedding
- Use Coq as a mathematical modelling language
- Language syntax: new induction data-type
- Language semantics: inductive predicates or functions

Shallow Embedding
- Use Coq as a functional programming language
- Library data structure and operations:
  - signature of a module: types and functions
  - in the Coq way: including properties of these functions

Shallow Embedding of BSML in Coq (1)

Module Type PRIMITIVES.

Section Processors.
  Parameter bsp_p : nat.
  Axiom bsp_pLtZero: 0 \not\in bsp_p.
  Definition processor : Type := \{ pid: nat \mid pid \in bsp_p \}.
End Processors.

Section Parallel_vectors.
  Parameter par : Type \to Type.
  Parameter get: \forall A: Type, par A \to processor \to A.
  Parameter par_eq: \forall (A:Type) (v v': par A),
  (\forall (i: processor), get v i = get v' i) \to v = v'.
End Parallel_vectors.
Section \textit{Primitives}.

Variable $A : \text{Type}$.

Parameter $\text{mkpar}$:
\[
\forall f : \text{processor} \rightarrow A, \\
\{ X : \text{par} A \mid \forall i : \text{processor}, \text{get} X i = f i \}.
\]

Parameter $\text{apply}$:
\[
\forall (B : \text{Type}) (\text{vf} : \text{par} (A \rightarrow B)) (\text{vx} : \text{par} A), \\
\{ X : \text{par} B \mid \forall i : \text{processor}, \text{get} X i = (\text{get vf} i)(\text{get vx} i) \}.
\]

Parameter $\text{put}$:
\[
\forall (\text{vf} : \text{par} (\text{processor} \rightarrow A)), \\
\{ X : \text{par} (\text{processor} \rightarrow A) \mid \\
\forall i j : \text{processor}, \text{get} X i j = \text{get vf} j i \}.
\]

Parameter $\text{proj}$:
\[
\forall (v : \text{par} A), \\
\{ X : \text{processor} \rightarrow A \mid \forall i : \text{processor}, X i = \text{get} v i \}.
\]

End \textit{Primitives}.

End \textit{PRIMITIVES}.
Coherence with Coq’s Logic

- Implementation of this module type using Coq’s vectors
- Bonus: it is a verified sequential implementation

Outline

1. Motivations
2. BSML: Bulk Synchronous Parallel ML
   The BSP Model
   Overview of the BSML Language
   Classic BSML Primitives
   Revised Bulk Synchronous Parallel ML
3. BSML in Coq
   Shallow Embedding of BSML in Coq
   BSML Programming and Reasoning in Coq
   Extraction
4. Summary
BSML Programming and Reasoning in Coq (1)

Module Type TYPE
(Import Bsml : Primitives.Specification.PRIMITIVES)
(Import PropM : Primitives.Properties.TYPE Bsml)
(Import ParSM : Support.ParSig.TYPE Bsml PropM).

Definition replicate (A:Type)(a:A):
{ vr : par A | \forall (i:processor), get vr i = a } :=
mkpar(fun _ => a).

Definition lreplicate(A:Type)(a:A) :=
proj2_sig(replicate a).

Hint Rewrite lreplicate : bsmlbase.

Program Definition parfun (A B : Type)(f:A\to B)(v:par A):
{ vr: par B | \forall (i:processor), get vr i = f (get v i) } :=
apply(replicate f) v.

Next Obligation.
autorewrite with bsml. reflexivity.
Defined.
...

End Make.

BSML Programming and Reasoning in Coq (2)

Program Definition getBoundsAux (A:Type)(l r:A)(v:par(list A))(H:\forall i, get v i\neq[]) :
{ vr: par (option A) | \forall (i:processor),
  get vr i = Some ( if ( i == firstProcessor ) then l else sLast (get v (i-1)) ) } \times
{ vr: par (option A) | \forall (i:processor),
  get vr i = Some ( if ( i == lastProcessor ) then r
    else sHead (get v (min (i+1) lastProcessor))) } :=

let msg := put(apply(mkpar(fun (pid:processor) data (dst:processor) \Rightarrow
  if ( dst == (pid+1) ) && negb(pid == (bsp_p-1)) then Some (sLast data)
  else if ( dst == (pid-1) ) && (negb(pid == 0)) then Some (sHead data)
  else None)) (parSig v \_ H) ) in
  applyat firstProcessor (constantFunPar processor (Some l)) msg
  (parSig (mkpar(fun pid\Rightarrow pid-1)) \_ \_ ),
  applyat lastProcessor (constantFunPar processor (Some r)) msg
  (parSig (mkpar(fun pid\Rightarrow min (pid+1) lastProcessor)) \_ \_ )).

Next Obligation.
...
Outline

1 Motivations

2 BSML: Bulk Synchronous Parallel ML
   The BSP Model
   Overview of the BSML Language
   Classic BSML Primitives
   Revised Bulk Synchronous Parallel ML

3 BSML in Coq
   Shallow Embedding of BSML in Coq
   BSML Programming and Reasoning in Coq
   Extraction

4 Summary

From Coq to OCaml

In Coq

- the BSML module type
- BSML programs: in functors taking a implementation as argument

⇒ extraction provides OCaml functors

In OCaml

- the BSML library provides a implementation of the module type
- almost . . . because processors:
  - type nat in Coq
  - type int in OCaml

⇒ Wrapper OCaml module BsmlNat
Summary

It is possible:

- to write BSML programs in Coq
- to extract actual OCaml BSML programs
- to compile them with OCaml compilers and BSML implementation

Current BSML in Coq:

- BSML Primitives
- Big subset of BSML pure functional standard library
- Dedicated tactics
- Some “applications”:
  - Heat Equation
  - Prototype verified implementation of OSL skeletons
  - Verified implementation of the BH skeleton