Parallel Programming with OCaml: A Tutorial

Victor Allombert[1] Mathias Bourgoin[1] Frédéric Loulergue[2]

[1]University of Orléans – The Computer Science Laboratory of Orléans
[2]Northern Arizona University – School of Informatics Computing and Cyber Systems

July 16, 2018 – HPCS, Orléans, France

Outline of the Talk

OCaml

What is OCaml ?

- Functional programming language
- From the ML (Meta Language) family

Why OCaml ?

- Powerful type system
- High level features
- Modules and object oriented approach
- Embedded Garbage Collector
- Byte and native code compilers
- Interactive loop (toplevel)

OCaml code execution

Toplevel (ocaml/utop):

```
# 3 + 4;;
- : int = 7
# 8 / 3;;
- : int = 2
# 3.5 +. 6.;;
- : float = 9.5
# 30_000_000 / 300_000;;
- : int = 100
# sqrt 9.;;
- : float = 3.
```

Compilation

- Bytecode: ocamlc −o main main.ml
- Native: ocamlopt −o main main.ml

OCaml syntax I

Variables

```
# let x = 1
val x : int = 1
# let x = 1 in x + 2
- : int = 3
```

Functions

```
# let f x = x * x
val f : int -> int = <fun>
# f 10
- : int = 100
```

OCaml syntax II

Partial application

```
# let f x y = x +. y in f 1
val f : float → float → float = <fun>
# let g = f 13.
val g : float → float = <fun>
# g 29.
- : float = 42.
```

OCaml syntax III

Polymorphism

```
# let h x = x
val h : 'a → 'a
# h 3
– : int = 3
# h true
– : bool = true
# h f
– : int → int = <fun>
```

Conditional

```
# if true then 1 else 2
```

OCaml syntax IV

Lists

```
# let l1 = [1;2;3]
val l1 : int list = [1;2;3]
# let l2 = 0::l1
val l2 : int list [0;1;2;3]
# let l3 = l2@[4]
val l3 : int list = [0; 1; 2; 3; 4]
# List.map (fun x → x +1) l3
– : int list = [1; 2; 3; 4; 5]
```

OCaml syntax V

Arrays

```
# let a1= [|1;2;3;4|]
val a1 : int array = [|1; 2; 3; 4|]
# a1.(0)
- : int = 1
# a1.(0) <- 0
- : unit = ()
```

Imperative features

- References: `let x = ref 1 in x := 2`
- Sequences: `x := 42; print_int !x`
- For loops: `for i = 0 to n do e done`
- While loops: `while bool_expr do e done`

Simple exercises I

Exercise 2.1

Write a OCaml function to compute the ratio x/y.

Exercise 2.2

Write a (recursive) OCaml function to compute factorial.

Exercise 2.3

Write a OCaml function to generate a random list of integers of size n.
(Random.int v returns a random integer between 0 (inclusive) and v (exclusive))

Simple exercises II

Exercise 2.4

Write a function taking, as argument, a function f and a list l and returns the mapping of the f on l such that: `lmap f [1;2] = [f 1; f 2]`.

Exercise 2.5

Using exercises 2.1,2.3 and 2.4, write a function taking an argument n and divide by n all elements of a given list. Apply it on random generated lists.

Parallel Programming

Automatic Parallelization

Concurrent & Distributed Programming

Parallel Programming

Automatic Parallelization

Structured Parallelism

- Declarative Parallel Programming
- Algorithmic Skeletons
- Bulk Synchronous Parallelism
- …

Concurrent & Distributed Programming

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

$\Rightarrow$ a high-level parallel language with formal semantics

…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

$\Rightarrow$ 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 **(OCaml 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)

Bulk Synchronous Parallelism (BSP)

Research on BSP

90' by Valiant (Cambridge) and McColl (Oxford)

Three models

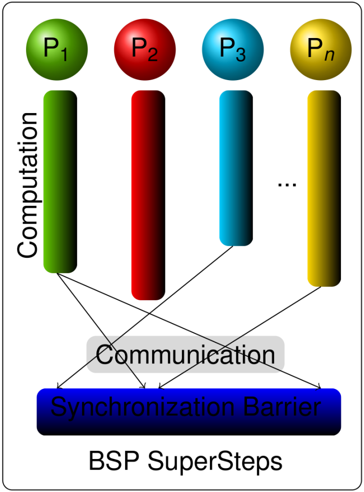
- abstract architecture
- execution model
- cost model

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



Cost model

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

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

- w_i processing time at processor i
- h_i^+ words sent by processor i
- h_i^- words received by processor i

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

$\Rightarrow$ Direct mapping eases the reasonning about performances

Notation

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

BSML Primitives

Access to the BSP parameters

bsp_p: int
bsp_r: float
bsp_g: float
bsp_l: float

$\Rightarrow$ 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

BSML Tools

Interative Loops (sequential simulators)

- On the VM: `bsml`
- Online: `http://tesson.julien.free.fr/try-bsml`

Compilation

Two modes:

- Sequential: `.seq` variants of the scripts
- Parallel (on top of MPI): `.mpi` variants of the scripts

Two targets:

- OCaml Bytecode: `bsmlc`
- Native code: `bsmlopt`

Creation of parallel vectors

Signature

mkpar : (int $\rightarrow$ 'a) $\rightarrow$ 'a par

Informal semantics

mkpar f $= \langle$ f 0, f 1, $\ldots$, f $(p-1)$ $\rangle$

Examples

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

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

BSP Cost

$\max_{0 \leq i < p} \|f\ i\|$ where $\|e\|$ is the time required to evaluate e

Point-wise parallel application

Signature

apply : $('a \to 'b)$ par $\to$ 'a par $\to$ 'b par

Informal semantics

apply $\langle\, f_0 \,, \ldots, \, f_{p-1} \,\rangle \langle\, v_0 \,, \ldots, \, v_{p-1} \,\rangle = \langle\, f_0\, v_0 \,, \ldots, \, f_{p-1}\, v_{p-1} \,\rangle$

Example

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

BSP Cost

$$\max_{0 \le i < p} \| f_i\, v_i \|$$

Exercises

Exercise 3.1

Write a BSML expression that creates a parallel vector of list of numbers, where a processor i contains the list $[10 \times i\,;\ \ldots\,;\ 10 \times (i+1) - 1]$

Exercise 3.2

Write a BSML function taking as argument a paralle vector of lists, and returning a parallel vector of the lengths of these lists.

Exercises

Exercise 3.3 (Parallel Map)

- *We consider a value of type 'a list par as a distributed list*
- *Write a BSML function*

  $$map\colon ('a \to 'b) \to 'a\ \ list\ \ par \to 'b\ \ list\ \ par$$

  that applies f to all the elements of the distributed list
- *Use map to transform the value of Exercise 3.1 into a parallel list of strings using the sequential function string_of_int*

Exercise 3.4

Write a BSML function taking as argument a positive number n and returning a parallel vector of lists of numbers, such that the contatenation of all the lists is the list from 0 to $n - 1$, and such that the lists are evenly distributed (difference between length of the smallest list and the length of the biggest list at most 1).

Projection

Signature

proj: 'a par → (int→

Informal semantics

$$\textbf{proj}\langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle = \begin{array}{lccc} \textbf{function} & 0 & \rightarrow & v_0 \\ & \vdots & & \\ \mid & p-1 & \rightarrow & v_{p-1} \end{array}$$

Remark

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

Example

```
# Bsml.proj (Bsml.mkpar string_of_int) 2;;
- : string = "2"
```

BSP Cost

$\max_{0\leq i<p}\{|v_i|, \sum_{j\neq i}|v_j|\} \times g + L$ where $|e|$ is the value of e's size

Exercises

Exercise 3.5

Write a function

$$to\_list\ :\ 'a\ par \rightarrow 'a\ \ list$$

that transforms a parallel vector into a sequential list

Exercise 3.6

- *Write a function reduce*: $('a \rightarrow 'a \rightarrow 'a) \rightarrow\ 'a \rightarrow\ 'a\ \ list\ \ par \rightarrow\ 'a$ *that for a binary associative operator* op *and its unit* e*, reduces the distributed list given in argument*
- *Example (assuming* ***bsp_p*** *= 8):*

```
# reduce (+) 0 (mkpar(fun i->[i]));;
- : int = 28
```

Exercises

Exercise 3.7 (Variance)

- *For a set of equally likely values x_i the variance is:*

$$Var = \frac{1}{n}\sum_{i=1}^{n}(x_i - \mu)^2$$
$$\mu = \frac{1}{n}\sum_{i=1}^{n} x_i$$

- *Assuming x_i is represented as a value of type* float list par*, write a BSML program to compute the variance*

Communication I

Signature

put: $(\text{int} \to \text{'a})\ \text{par} \to (\text{int} \to \text{'a})\ \text{par}$

Informal semantics

put $\langle\, f_0\ , \ldots ,\ f_{p-1}\, \rangle = \langle\, g_0\ , \ldots ,\ g_{p-1}\, \rangle$ with $g_j \equiv$ **fun** src$\to f_{\text{src}}\ j$

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

```
Bsml.apply (mkpar(fun _-> (fun f->List.map f Stdlib.Base.procs)))
           (Bsml.put(Bsml.mkpar(fun pid dst->
                                     if dst=(pid+1) mod Bsml.bsp_p
                                       then Some pid
                                       else None)));;
- : int option list Bsml.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] >
```

Communication III

BSP Cost

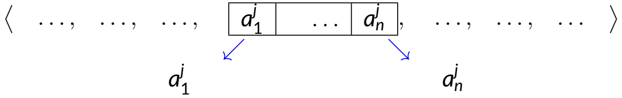
$$\max_{0\le i<p} \left(\sum_{j=0}^{p-1} \|f_i\ j\|\right) + \max_{0\le i<p} \left\{\sum_{j\neq i} |f_i\ j|, \sum_{j\neq i} |f_j\ i|\right\} \times g + L$$

Remark

- The first constant constructor of a sum type has size 0
- Examples: None, [], ...

A More Complicated Example

Communication pattern to implement



Implementation

```
let getBounds first last v = let p = Bsml.bsp_p in
  let parfun f v = Bsml.apply (Bsml.mkpar(fun _ → f)) v in
  let lasts = parfun last v and firsts = parfun first v in
  let msg = Bsml.put(Bsml.apply(Bsml.apply
    (Bsml.mkpar(fun pid first last dst→
                  if dst=(pid+1) mod p then Some last
                  else if dst=(p+pid−1) mod p then Some first else None))
     firsts ) lasts ) in
  ( Bsml.apply msg (Bsml.mkpar(fun pid→ (p+pid−1) mod p)),
    Bsml.apply msg (Bsml.mkpar(fun pid→ (pid+1) mod p)) )
```

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

Alternative Syntax

Two syntaxes for BSML

- Classic BSML: impossible to use vectors in a local section
- Alternative syntax: access to *l*ocal information of vector v noted \$v\$, possible only in a *l*ocal section, written $\ll$ e $\gg$

Examples

```
let mkpar f = << f $this$ >>
let apply fv vv = << $fv$ $vv$ >>
let parfun f v = << f $v$ >>
```

$\Rightarrow$ **mkpar** and **apply** are no longer primitives

A Revised More Complicated Example

Implementation

```
let getBounds first last v =
  let p = Bsml.bsp_p in
  let lasts  = ≪ last $v$ ≫ in
  let firsts = ≪ first $v$ ≫ in
  let msg = Bsml.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) ≫
```

Exercise

Exercise 3.8 (1D heat-equation)

- $\dfrac{\delta u}{\delta t} - \gamma \frac{\delta^2 u}{\delta^2 x} = 0$
- $u(x, t + dt) = \dfrac{\gamma dt}{dx^2} \left(u(x+dx, t) + u(x-dx, t) - 2u(x, t)\right) + u(x, t)$
- *Implement a sequential version (on a list or an array), taking two values for the bounds*
- *Use it to implement a parallel version*

The Multi-BSP bridging model

A bridging model for multi-core computing

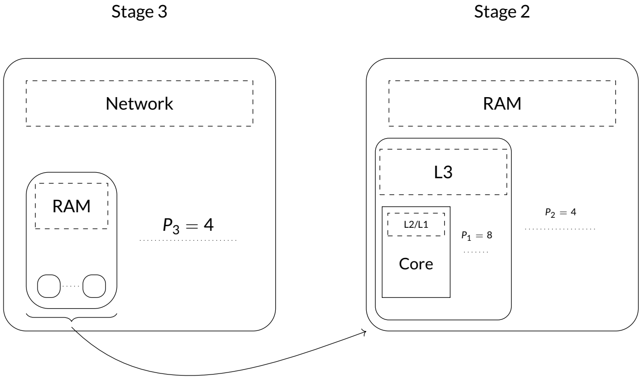
Proposed by Valiant in 2011

Approach

- Abstract multi-level model
- Execution model
- Cost model (BSP-like)

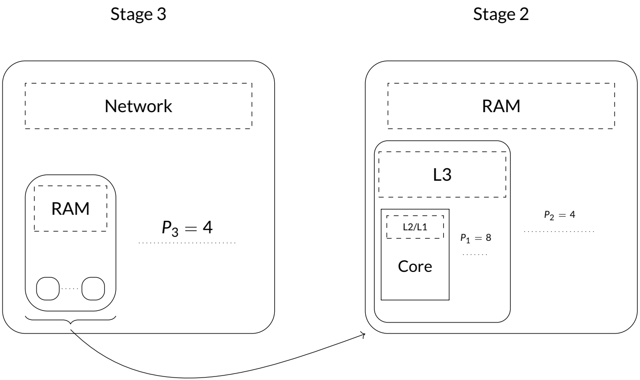
A Multi-BSP computer

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



A Multi-BSP computer

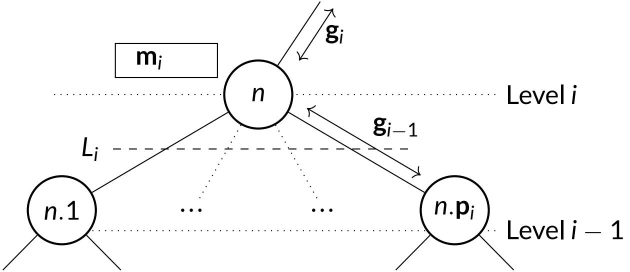
- Stage 3: 4 nodes with a network access
- Stage 2: one node has 4 chips plus RAM
- Stage 1: one chip has 8 cores plus L3 cache
- Stage 0: one core with L1/L2 caches



The Multi-BSP execution model

Execution model

A level i superstep is:

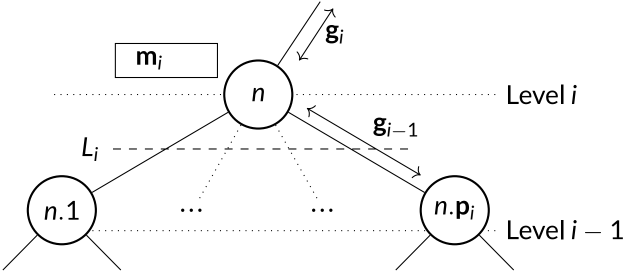


The Multi-BSP execution model

Execution model

A level i superstep is:

- Level $i-1$ executes code independently

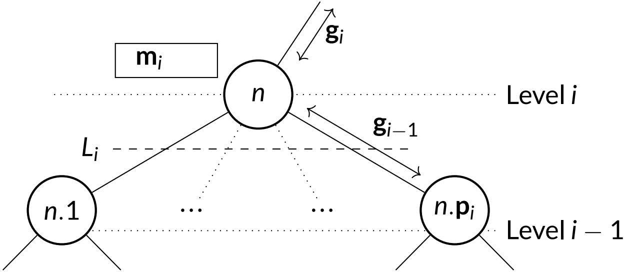


The Multi-BSP execution model

Execution model

A level i superstep is:

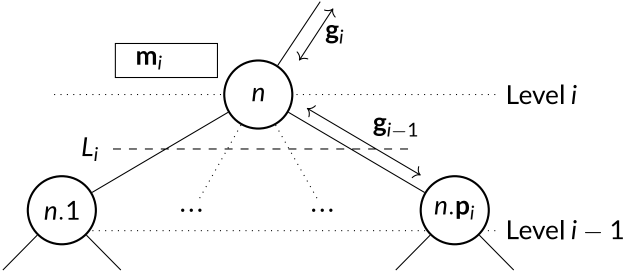
- Level $i-1$ executes code independently
- Exchanges information with the m_i memory



Execution model

A level i superstep is:

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

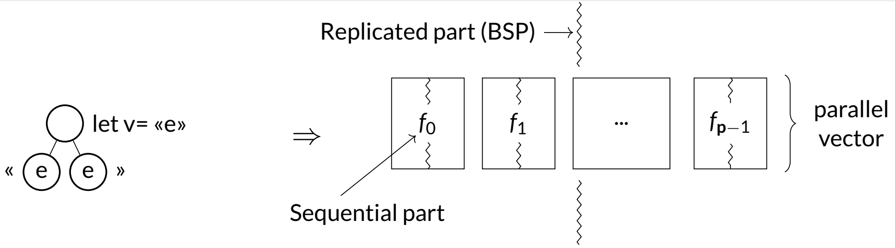


The Multi-ML language

Basic ideas

Basic ideas

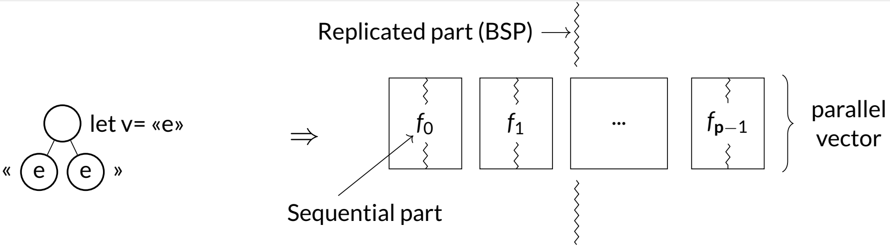
- BSML-like code on every stage of the Multi-BSP architecture



The Multi-ML language

Basic ideas

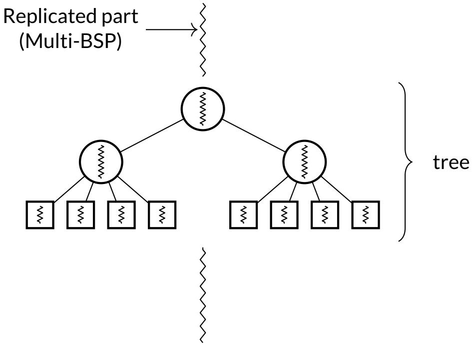
- BSML-like code on every stage of the Multi-BSP architecture
- Specific syntax over ML : eases programming



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



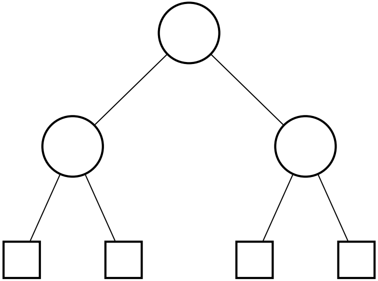
Multi-ML : Tree recursion

Recursion structure

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

Recursion structure

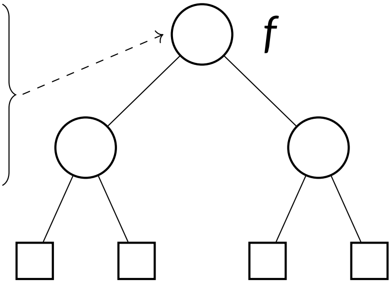
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



Multi-ML : Tree recursion

Recursion structure

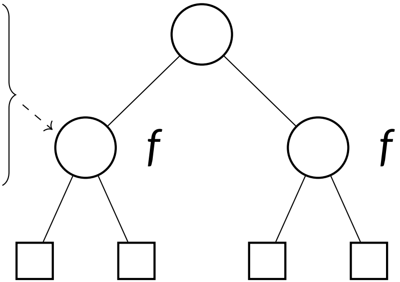
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



Multi-ML : Tree recursion

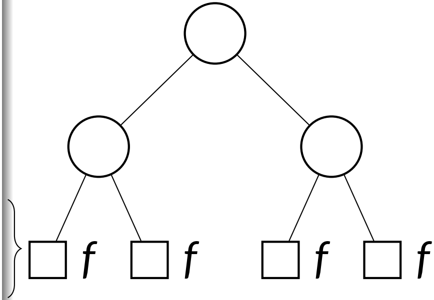
Recursion structure

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



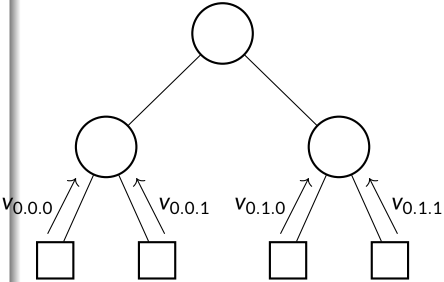
Recursion structure

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



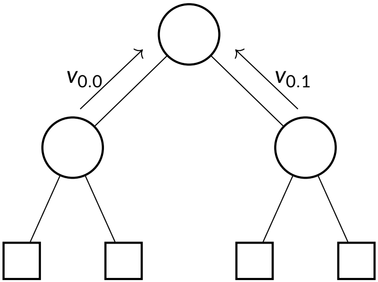
Recursion structure

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



Recursion structure

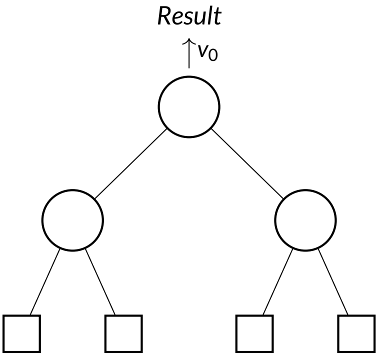
```
let multi f [args]=
  where node =
   (* BSML code *)
   ...
   << f [args] >>
   ... in v
  where leaf =
   (* OCaml code *)
   ... in v
```



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 construction

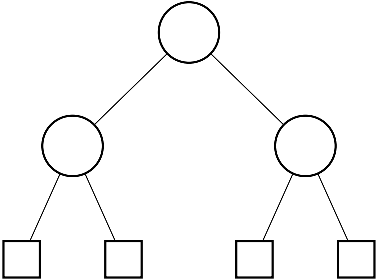
Tree construction

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

Multi-ML : Tree construction

Tree construction

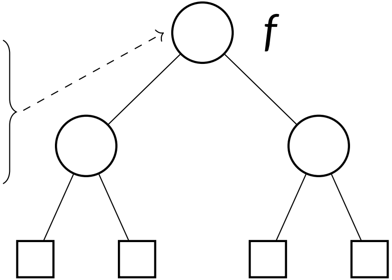
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



Multi-ML : Tree construction

Tree construction

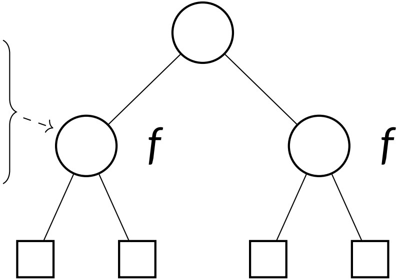
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



Multi-ML : Tree construction

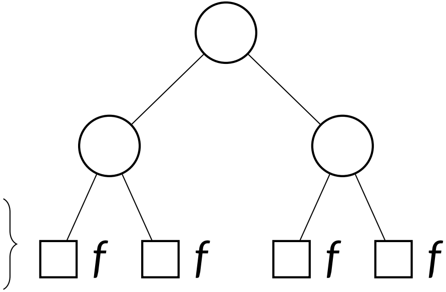
Tree construction

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



Tree construction

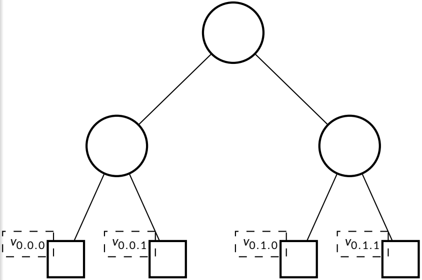
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



Multi-ML : Tree construction

Tree construction

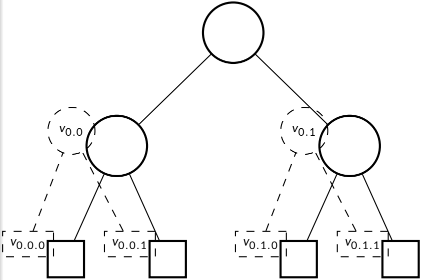
```
let multi tree f [args]=
  where node =
   (* BSML code *)
   ... in
   finally << f [args] >> v
  where leaf =
   (* OCaml code *)
   ... in v
```



Multi-ML : Tree construction

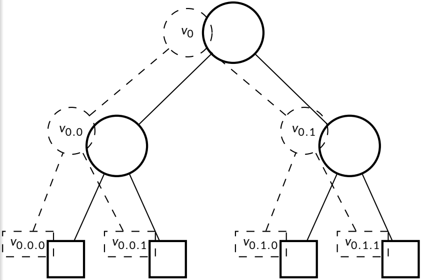
Tree construction

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



Tree construction

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



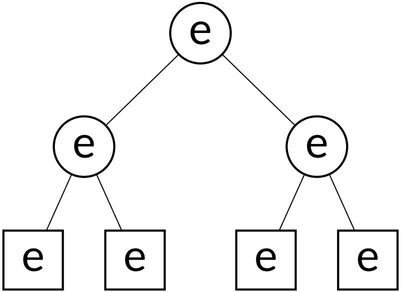
Primitives

Summary

Primitives

Summary

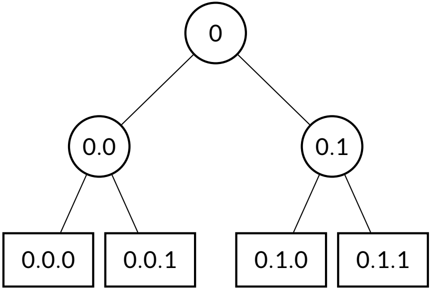
- mktree e



Primitives

Summary

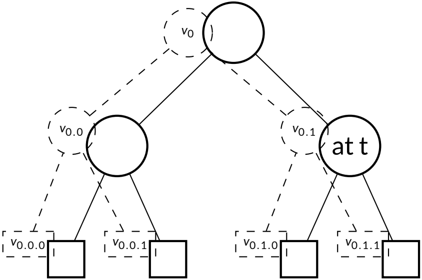
- mktree e
- gid



Primitives

Summary

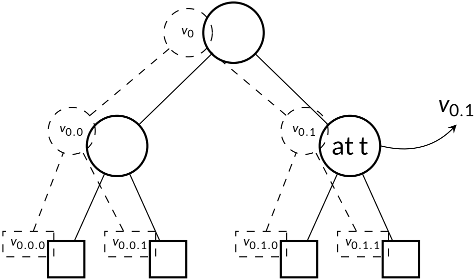
- mktree e
- gid
- at



Primitives

Summary

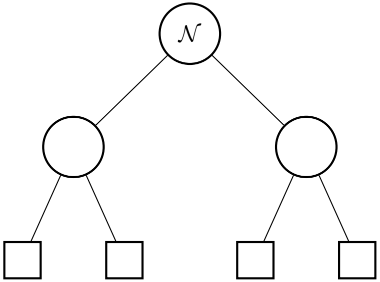
- mktree e
- gid
- at



Primitives

Summary

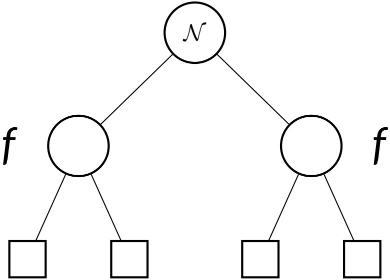
- mktree e
- gid
- at
- $<< \ldots f \ldots >>$



Primitives

Summary

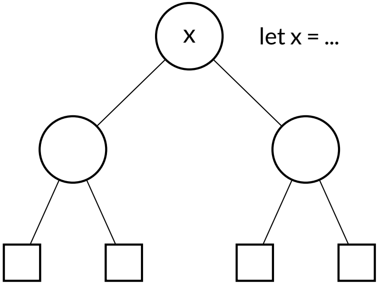
- mktree e
- gid
- at
- $<< \ldots f \ldots >>$



Primitives

Summary

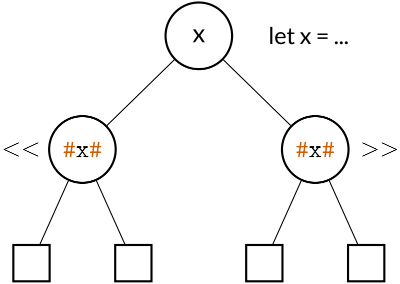
- mktree e
- gid
- at
- << ...*f*... >>
- #x#



Primitives

Summary

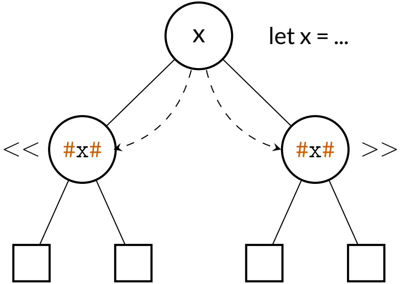
- mktree e
- gid
- at
- << ...*f*... >>
- #x#



Primitives

Summary

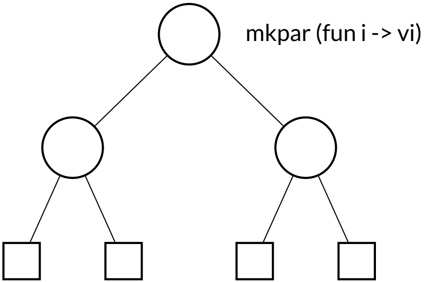
- mktree e
- gid
- at
- $<<$ ...f... $>>$
- #x#



Primitives

Summary

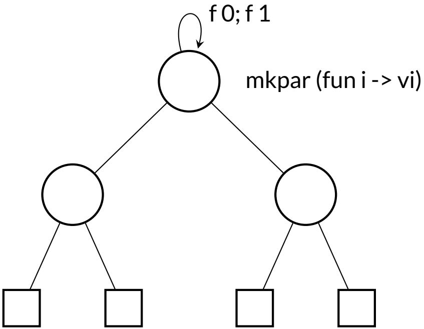
- mktree e
- gid
- at
- $<<\ ...f...\ >>$
- #x#
- mkpar f



Primitives

Summary

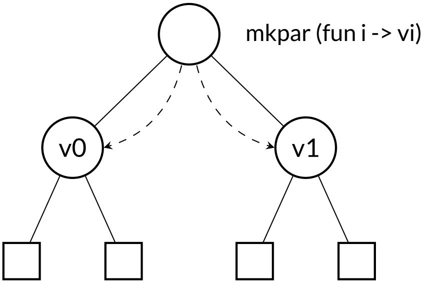
- mktree e
- gid
- at
- << ...*f*... >>
- #x#
- mkpar f



Primitives

Summary

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



Multi-ML code execution

Implementation

- Generic communication module (currently over MPI)
- Shared and distributed memory

Compilation

- mmlopt.mpi -o main main.mml

Toplevel (Beta version)

- multiml

Exercises I

Exercise 4.1

Write a (multi-)function which display the global identifier of each components of the Multi-BSP architecture.

Exercise 4.2

Write a code to build a tree of randomly generated values.

Exercise 4.3

Write a multi-functions taking as argument a tree of values and returning a list which is the concatenation of every lists of the given tree.

Exercises II

Exercise 4.4 (Data distribution)

Write a multi-function which distribute a given list of values toward the leaves and returns a tree with empty lists on nodes and lists on values on leaves.

Exercise 4.5 (Reduce)

Write a multi-function taking as argument a tree with lists of values on leaves, an associative operator and reduces the list toward the root node.

Exercise 4.6 (1-D heat equation)

Based on the heat equation implemented in 3.8, write a code using Multi-ML.

Heterogeneous computing

Multiple types of processing elements

- Multicore CPUs
- **GPUs**
- FPGAs
- Cell
- Other co-processors

Each with its own (specific) programming environment

- Programming languages (often subsets of C/C++ or assembly language)
- Compilers
- Libraries
- Debuggers and profilers

GPGPU Programming

Two main frameworks

- **Cuda** (NVidia)
- **OpenCL** (Consortium OpenCL)

Different languages

- To write **kernels**
  - **Assembly** (PTX, SPIR, IL,...)
  - Subsets of **C/C++**
- To manage kernels from the **host**
  - ***C/C++/Objective-C***
  - Bindings : Fortran, Python, Java, ...

SPOC framework

Composed of

- SPOC: An OCaml runtime library
- Sarek: A DSL dedicated to GPGPU kernels
- Multiple **experimental** libraries
  - (Maybe incomplete) Bindings to C/C++ GPGPU libraries (CUBLAS, MAGMA, CUFFT)
  - Pure OCaml (without using Sarek) parallel skeletons libraries
  - Hybrid (using Sarek) parallel skeleton libraries
  - Samples
  - A (deprecated – thanks to WebCL demise) JavaScript port of SPOC and Sarek

Open Source distribution: `http://mathiasbourgoin.github.io/SPOC/`

SPOC framework

Runtime Library: Host (CPU) code

- Not a “simple” CUDA/OpenCL OCaml binding
- Detects compatible devices at runtime
- Handles memory transfers between CPU-GPGPU accelerators automatically
- Can launch native (CUDA/OpenCL) or Sarek (DSL) kernels

DSL: Kernel (GPU) code

- Built as a syntax extension
- Static type checking
- Translated into an AST that is embedded into the OCaml host code
- Comes with a dedicated library to
  - Compile the AST to actual CUDA/OpenCL C code
  - Use the SPOC library to launch kernels on GPUs

First contact with SPOC

Launch SPOC and detect compatible devices

Devices.init : unit → device array

Device

```
type device = {
  general_info : generalInfo;
  specific_info : specificInfo ;
  gc_info : gcInfo;
  events: events;
}
```

```
type generalInfo = {
  name : string ;
  totalGlobalMem : int;
  localMemSize : int ;
  clockRate : int ;
  totalConstMem : int;
  multiProcessorCount : int;
  eccEnabled : bool;
  id : int ;
  ctx : context;
}
```

Exercise

Exercise 5.1

Write an OCaml program that prints info on the GPGPU-compatible accelerators present on your system.

Sharing data using SPOC vectors

Vector creation example

```
(* create a vector of 1024 32-bts ints *)1
 let v_ints = Vector.create Vector.int32 1024 in
 Mem.set v_ints i 0l;
 let a = Mem.get v_ints 42 in

(* create a vector of n 64-bit floats (C doubles) *)
 let v_doubles = Vector.create Vector.float64 n
 Mem.set v_floats i 32.
```

SPOC vectors are

- Automatically transferred between Host and GPGPU memory.
- Managed by the OCaml garbage collector
- Automatically freed from either (Host/GPGPU) memory
- Once created your good to go

How to launch a kernel

First, let's go back to classic GPGPU programming

- Frameworks demand to describe a 3D grid of blocks of threads
- In this grid, each thread runs an instance of the kernel code.

Example of a kernel launch

```
let n = 1000000
let block =
  {blockX = 1024; blockY = 1;  blockZ = 1} in
let grid =
  {gridX=(n+1024−1)/1024; gridY=1; gridZ=1} in
Kirc.run kernel args (block,grid) 0 device;
```

Here, 1 000 000 threads are launched, grouped into blocks of 1024 threads, using only the first dimension of the grid

Kernel launch arguments

Kernel launch

Kirc.run kernel args (block,grid) stream device;

Arguments

- `kernel` : name of a kernel described using the DSL (see next slide)
- `args` : tuple containing kernel arguments : example

```
let v1 = Vector.create .... in
let v2 = Vector.create .... in
let n = 1000000 in
let args = (v1,v2,n)
```

- `(block,grid)` : see previous slide
- `stream` : used for concurrent kernel launches (for this tutorial use 0)
- `device` : device (returned previously by `Devices.init ()`)

Generating CUDA/OpenCL code

Kirc is a DSL

Actual CUDA/OpenCL code has to be generated prior to launching the kernel:

```
Kirc.gen kernel;
```

Host programming is easy, let's write kernels

Sarek DSL (not really OCaml)

- No recursion
- No functions*
- No complex data-structures*
- No pattern matching*
- Only basic imperative code, with OCaml-like syntax, type inference and static checking

A small example

```
let multiply = kern a b c →
  let open Std in
  let idx = global_thread_id in
  b.[<idx>] ← a.[<idx>] ∗ c
```

*Available in experimental versions of SPOC/Sarek

Small kernel example

Std

The Std module contains all the necessary variables to know thread ids and locations in the grid :

```
module Std :
sig
  val thread_idx_x : Int32.t
  val thread_idx_y : Int32.t
  val thread_idx_z : Int32.t
  val block_idx_x  : Int32.t
   ...
  val block_dim_x : Int32.t
   ...
  val grid_dim_x : Int32.t
   ...
  val global_thread_id : Int32.t


   ...
end
```

Exercises

Exercise 5.2 (Vector Addition)

Write a program adding 2 vectors of 1 000 000 floats on the GPGPU accelerator using SPOC and Sarek.

Exercise 5.3 (Heat Equation)

Use the sequential version of the heat equation implemented in exercise 3.8 to implement a GPGPU version computing on SPOC vectors using a Sarek kernel.