Multi-ML: Programming Multi-BSP Algorithms in ML

Victor Allombert, Frédéric Gava, Julien Tesson

LACL, Université Paris Est

The BSP Model

In the BSP model [1], a computer is a set of **p** uniform processor-memory pairs and a communication network. A BSP program is executed as a sequence of *super-steps* (Fig. 1), each one divided into three successive disjoint phases:

1) Each processor only uses its local data to perform sequential computations and to request data transfers to other nodes;
2) The network delivers the requested data;
3) A global synchronisation barrier occurs, making the transferred data available for the next super-step.

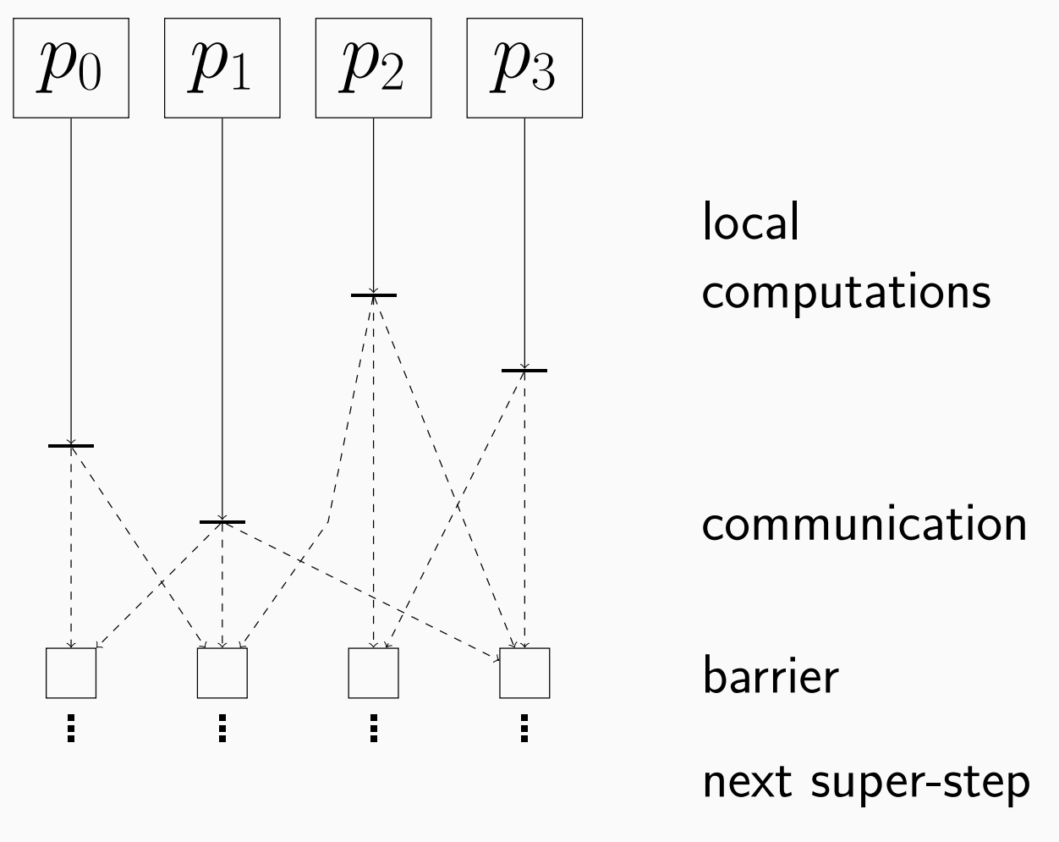

Figure 1: A BSP super-step

The Multi-BSP Model

The MULTI-BSP model [2] is another *bridging model* as the original BSP, but adapted to *clusters of multicores*. The MULTI-BSP model introduces a vision where a *hierarchical architecture* is a *tree* structure of *nested components* (*sub-machines*) where the lowest stage (*leaf*) are processors and every other stage (*node*) contains memory. A node executes some codes on its nested components (*aka* "*children*"), then waits for results, do the communication and synchronised the sub-machine.

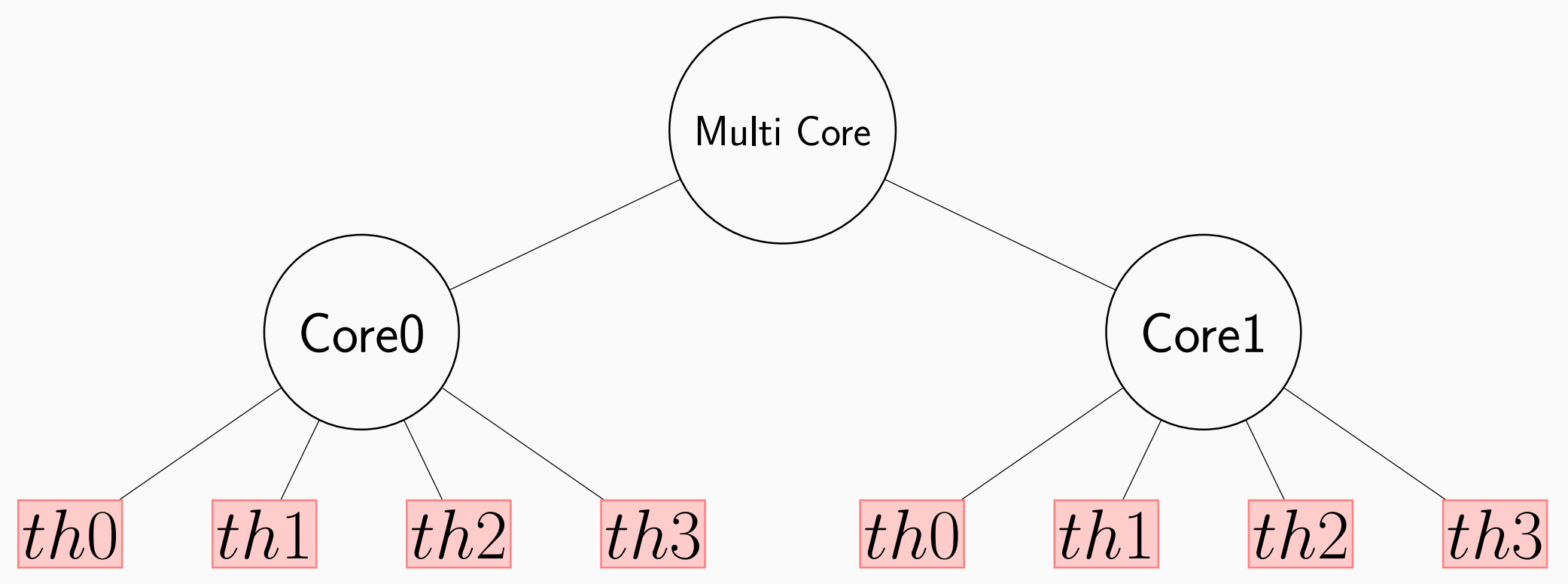

Figure 2: A MULTI-BSP view of a multi-core architecture

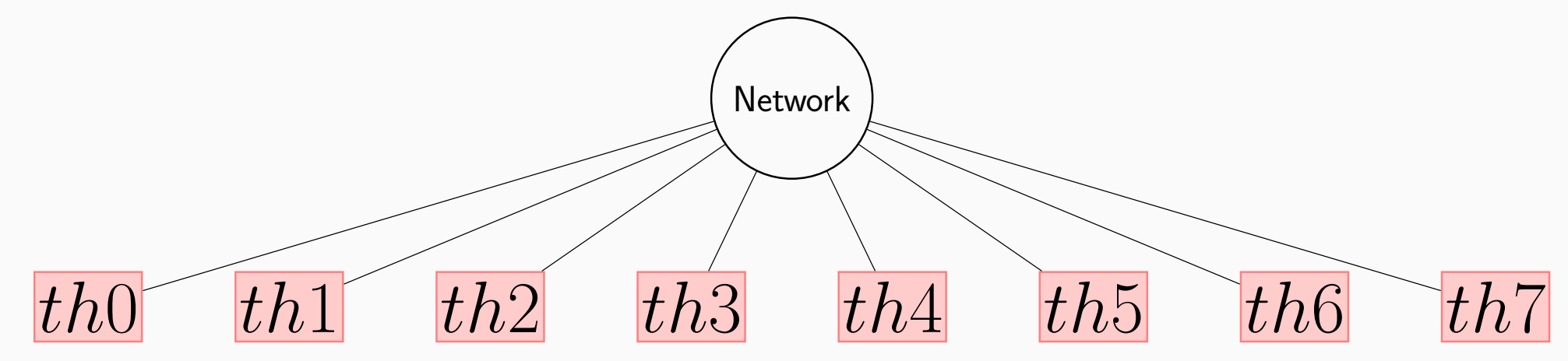

Figure 3: A BSP view of a multi-core architecture

For a multicore architecture it is possible to distinguish all the level thanks to MULTI-BSP (Fig. 2). On the contrary, the BSP model (Fig. 3) flattens the architecture.

Benchmarks

Fig. 4 shows the results of our experimentations. We can see that the efficiency on small list is poor but as the list grows, MULTI-ML exceeds BSML. This difference is due to the fact that BSML communicates through the network at every super steps; while MULTI-ML focusing on communications through local memories and finally communicates through the distributed level.

	100_000		1_000_000		3_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	0.7	1.8	125.3	430.7	...	...
16	0.5	0.8	68.1	331.5	1200.0	...
32	0.3	0.5	11.3	122.2	173.2	...
48	0.5	0.4	5.5	88.4	69.3	...
64	0.3	0.3	4.1	56.1	51.1	749.9
96	0.3	0.38	3.9	30.8	38.1	576.1
128	0.5	0.45	4.7	24.3	30.6	443.7

Figure 4: Execution time of Eratosthenes (naive) using MULTI-ML and BSML.

Fig. 5 gives the computation time of the simple scan using a summing operator. We can see that MULTI-ML introduce a small overhead due to the level management; however it is as efficient as BSML and concord to the estimated execution times.

	5_000_000			
	MULTI-ML	BSML	*Pred*_MULTI-ML	*Pred*_BSML
8	2.91	2.8	3.44	1.83
16	1.42	1.4	1.72	0.92
32	0.92	0.73	0.43	0.46
48	0.84	0.75	0.28	0.31
64	0.83	0.74	0.21	0.23

Figure 5: Execution time and predictions of scan (sum of integers)

BSP Programming in ML : BSML

BSML [3] uses a *small set of primitives* and is currently implemented as a library for the ML programming language OCAML. A BSML program is built as a ML one but using a specific data structure called *parallel vector*. Its ML type is 'a **par**. A vector expresses that each of the **p** processors *embeds* a value of any type 'a. The BSML primitives are summarized in Fig. 6 :

Primitive	Level	Type	Informal semantics
≪e≫	g	'a **par** (if e:'a)	⟨e, . . . , e⟩
pid	g	int **par**	A predefined vector: i on processor i
`$v$`	l	'a (if v: 'a **par**)	v_i on processor i, assumes $\mathsf{v} \equiv \langle \mathsf{v}_0, \dots, \mathsf{v}_{\mathbf{p}-1} \rangle$
proj	g	'a **par**→ (int→ 'a)	$\langle x_0, \dots, x_{p-1} \rangle \mapsto (\textbf{fun}\ i \rightarrow x_i)$
put	g	(int→ 'a)**par**→ (int→ 'a)**par**	$\langle f_0, \dots, f_{\mathbf{p}-1} \rangle \mapsto \langle (\textbf{fun}\ i \rightarrow f_i\ 0), \dots, (\textbf{fun}\ i \rightarrow f_i\ (\mathbf{p}-1)) \rangle$

Figure 6: The BSML primitives

An example of a parallel vector construction using the BSML toplevel :

```
#let vec = ≪ "GDR"≫ in ≪ $vec$^",␣proc␣"^(string_of_int $pid$) ≫ ;;
  val vec : string par = <"GDR,␣proc␣0", "GDR,␣proc␣1", "GDR,␣proc␣2">
```

The Multi-ML language

MULTI-ML is based on the idea of executing a BSML-like code on every stage of the MULTI-BSP architecture, that is on every sub-machine. For this, we add a *specific syntax* to ML in order to code special functions, called *multi-functions*, that recursively go through the MULTI-BSP tree. At each stage, a multi-function allows the execution of any BSML code. The main idea of MULTI-ML is to structure parallel codes to control all the stage of a tree: we generate the parallelism by allowing a node to call recursively a code on each of its sub-machines (children). When leaves are reached, they will execute their own codes and produce values, accessible by the top node using a vector. The data are distributed on the stages (toward leaves) and results are gathered on nodes toward the root node as shown in Fig. 7. Let us consider a code where, on a node, ≪ e ≫ is executed. As shown in Fig. 8, the node creates a vector containing, for each sub-machine i, the expression e. As the code is run asynchronously, the execution of the node code will continue until reaching a barrier.

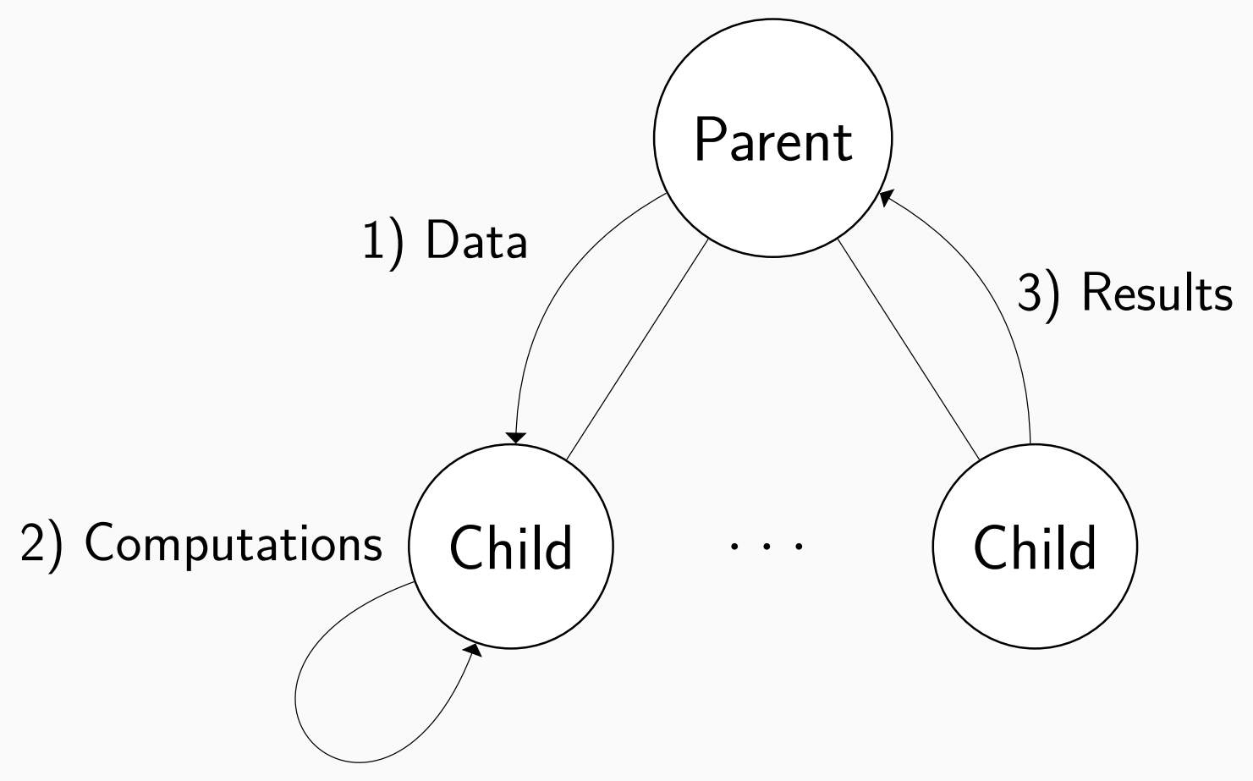

Figure 7: Code propagation

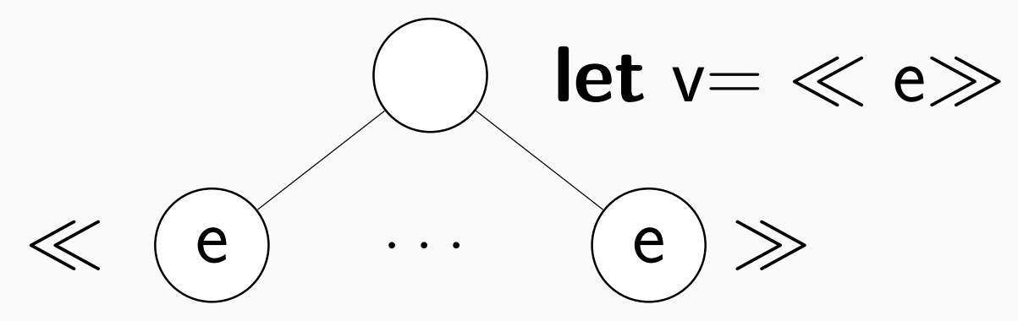

Figure 8: Data distribution

Fig. 9 shows the MULTI-ML primitives (without recall the BSML ones); their authorised level of execution and their informal semantics.

Primitive	Level	Type	Informal semantics
§e§	m	'a tree	Build ≀e≀, a tree of e
`$t$`	s	'a	In a §e§ code, t_n on node/leaf n of the tree ≀t≀
v (if v: 'a tree)	b	'a	v_n on node n of tree ≀v≀,
`$v$`	l	'a	In the ith component of a vector, $\mathsf{v}_{n.i}$ on node/leaf n of the tree ≀v≀
gid	m	id	The predefined tree of nodes and leaves ids
≪...f...≫	l	'a	In a component of a vector, recursive call of the multi-function
#x#	l	'a	In a component of a vector, reading the value x at upper stage (id)
mkpar f	b	'a **par**	$\langle v_0, \dots, v_{\mathsf{p}_n} \rangle$, where $\forall i, \mathsf{f}\ \mathsf{i} = v_i$, at id n of the tree
finally $\mathsf{v}_1\ \mathsf{v}_2$	b,s	'a	Return value v_1 to upper stage (id) and keep v_2 in the tree
this	b,l,s	'a option	Current value of the tree if exists, None otherwise

Figure 9: The MULTI-ML primitives

An example of a tree construction using the MULTI-ML toplevel :

```
#let multi f n =
  where node =
    let _=≪f ($pid$+ #n#+ 1)≫ in
    finally ~up:() ~keep:(gid^"=>"^n)
  where leaf=finally ~up:() ~keep:(gid^"=>"^n);;
 val f : int→string tree = <multi-fun>
```

```
#f 0
o "0→ 0"
|
o "0.0→ 1"
|  → "0.0.0→ 2"
|  → "0.0.1→ 3"
o "0.1→ 2"
|  → "0.1.0→ 3"
|  → "0.1.1→ 4"
```

References

[1] L. G. Valiant. "A Bridging Model for Parallel Computation". In: *Comm. of the ACM* 33.8 (1990), pp. 103–111.

[2] L. G. Valiant. "A bridging model for multi-core computing". In: *J. Comput. Syst. Sci.* 77.1 (2011), pp. 154–166.

[3] Louis Gesbert et al. "Bulk Synchronous Parallel ML with Exceptions". In: *Future Generation Computer Systems* 26 (2010), pp. 486–490.

LaTeX TikZposter