HAL
open science

Multi-ML: Programming Multi-BSP Algorithms in ML

Victor Allombert, Frédéric Gava, Julien Tesson

To cite this version:

Victor Allombert, Frédéric Gava, Julien Tesson. Multi-ML: Programming Multi-BSP Algorithms in ML. International Journal of Parallel Programming, 2015, pp.20. hal-01160164v1

HAL Id: hal-01160164

https://hal.science/hal-01160164v1

Submitted on 4 Jun 2015 (v1), last revised 3 Feb 2016 (v3)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

8th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2015)
Pisa, Italia
July 2-3, 2015

Multi-ML: Programming Multi-BSP Algorithms in ML

V. Allombert · F. Gava · J. Tesson

Abstract The MULTI-BSP model is an extension of the well known BSP bridging model. It brings a tree based view of nested components to represent hierarchical architectures. In the past, we designed BSML for programming BSP algorithms in ML. We now propose the MULTI-ML language as an extension of BSML for programming MULTI-BSP algorithms in ML.

Keywords BSP · MULTI-BSP · ML.

1 Introduction

Context of the work. Nowadays, parallel programming is the norm in many areas but it remains a *hard task*. And the more complex the parallel architectures become, the harder the task of programming them *efficiently* is. As we moved from *unstructured* sequential code cluttered with goto statement toward *structured* code, there has been a move in parallel programming community to leave unstructured parallelism, with pairwise communications, in favour of *global communication scheme* [2,6,26] and of structured *abstract* models like BSP [2,33] or of higher-level concepts like *algorithmic skeletons* [15].

Programming in the context of a *bridging model*, such as BSP, allows to simplify the task of the programmer, to ease the reasoning on *cost* and to ensure a better *portability* from one system to another [2,22,33]. However, designing a programming language [17,21] for such a model requires to chose a *trade-off* between the possibility to control parallel aspects necessary for *predictable* efficiency (but which make programs more difficult to write, to prove and to port) and the *abstraction* of such features which are necessary to make programming easier —but which may hampers efficiency and *performance* prediction.

With *flat homogeneous architectures*, like clusters of mono-processors, BSP has been proved to be an effective target model for the design of efficient algorithms and languages [35]: while its structured nature allows to avoid *deadlocks* and *non-determinism* with little care and to reason on program *correctness* [13,36,37] and cost, it is general enough to express many algorithms [2]. But *modern* parallel architecture have now *multiple layers* of parallelism. For example, supercomputers are made of thousands of *interconnected nodes*, each one carrying *several multicores* processors. Communications between distant nodes cannot be as fast as communications among the cores of a given proces-

LACL, University of Paris-East, Créteil, France
E-mail: victor.allombert@lacl.fr · E-mail: {frederic.gava,julien.tesson}@univ-paris-est.fr

sor; Communications between cores, by accessing shared processor cache are faster than communications between processors through RAM.

Contribution of this paper. Those architectures specifics led to a *new bridging model*, MULTI-BSP [34], where the hierarchical nature of parallel architectures is reflected by a *tree-shaped* model describing the dependencies between memories. The MULTI-BSP model gives a more precise picture of the cost of computations on modern architectures. While the model is more complex to grasp than the BSP one, it keeps structured characteristics that prevents deadlock and non-determinism. We propose a language, MULTI-ML, which aim at providing a mean to program MULTI-BSP algorithms so as our past BSML [14] is a mean to program BSP ones. MULTI-ML combines the high degree of *abstraction* of ML (without poor performances because often, ML programs are as efficient as C ones) with the *scalable* and *predictable* performances of MULTI-BSP.

Outline. The remainder of this paper is structured as follows. We first give in section 2 an overview of previous works: the BSP model at Section 2.1 and then the BSML language at Section 2.2 following with the MULTI-BSP model at Section 2.3. Our language MULTI-ML is presented at Section 3. Its formal semantics and implementation, together with examples and benchmarks are given at Section 4. Section 5 discusses some related work and finally, Section 6 concludes the paper and gives a brief outlook on future work.

2 Previous Work

In this section, we present shortly the BSP and MULTI-BSP models and how to program BSP algorithms using the BSML language. We also give an informal semantics of BSML primitives and simple examples of BSML programs. We assume the reader is familiar with ML programming.

2.1 The BSP Model of Computation

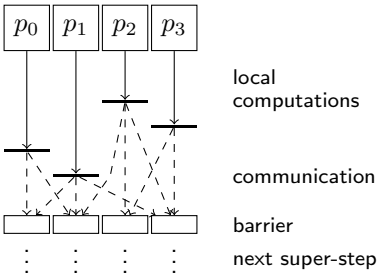

Fig. 1 A BSP super-step.

In the BSP model [2,33], 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. This *structured* model enforces a strict *separation* of communication and computation: during a super-step, no communication between the processors is allowed but only transfer requests; only at the *barrier* information is actually exchanged. Note that a BSP library can send data during the computation phase of a super-step, but this is hidden to programmers.

The *performance* of the BSP computer is characterised by 4 *parameters*: (1) the local processing speed **r**; (2) the number of processor **p**; (3) the time **L** required for a barrier; (4) and the time **g** for collectively delivering a 1-relation, a communication phase where every processor receives/sends at most one word. The network can deliver an h-relation (every processor receives/sends at most h words) in time $\mathbf{g} \times h$. To accurately *estimate* the execution time of a BSP program, these 4 parameters could be easily benchmarked [2]. The execution time (cost) of a super-step s is the sum of the maximal of the local processing time, the data delivery and the global synchronisation times. The total cost (execution time) of a BSP program is the sum of its super-steps's costs.

2.2 BSP Programming in ML

BSML [14] uses a *small set of primitives* and is currently implemented as a library (`http://bsmllib.free.fr`) for the ML programming language OCAML (`http://caml.org`). An important feature of BSML is its *confluent* semantics: whatever the order of execution of the processors, the final value will be the same. Confluence is convenient for *debugging* since it allows to get an *interactive loop* (toplevel). That also eases programming since the parallelisation can be done *incrementally* from a ML program. Last but not least, it is possible to reason about BSML programs using the COQ (`http://https://coq.inria.fr/`) interactive theorem prover [13,36] and to extract actual parallel programs from proofs.

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 processors are *labelled* with *ids* from 0 to $\mathbf{p}-1$. The nesting of vectors is not allowed. We use the following notation to describe a vector: $\langle v_0, v_1, \ldots, v_{\mathbf{p}-1} \rangle$. We distinguish a vector from an usual array because the different values, that will be called *local*, are blind from each other; it is only possible to access the local value v_i in two cases: locally, on processor i (using a specific syntax), or after some communications.

Since a BSML program deals with a whole parallel machine and individual processors at the same time, a *distinction* between the 3 *levels of execution* that take place will be needed: (1) **Replicated** execution r is the default; Code that does not involve BSML primitive is run by the parallel machine as it would be by a single processor; Replicated code is executed at the same time by every processor, and leads to the same result everywhere; (2) **Local** execution l is what happens inside parallel vectors, on each of their components; The processor uses its local data to do computation that may be different from the other's; (3) **Global** execution g concerns the set of all processors together as for BSML communication primitives. The distinction between local and replicated is strict: the replicated code cannot depend on local information. If it were to happen, it would lead to replicated inconsistency.

Parallel vectors are handled through the use of different *communication primitives*. Their implementation relies either on MPI or on TCP/IP. Fig. 2 shows their use. Informally, they work as follows: let $\ll$`e`$\gg$ be the vector holding `e` everywhere (on each processor), the $\ll \gg$ indicates that we enter a

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 v ≡ ⟨v_0, . . . , v_{p-1}⟩
proj	g	'a **par**→(int→'a)	$\langle x_0, \ldots, x_{p-1}\rangle \mapsto$ (**fun** $i \rightarrow x_i$)
put	g	(int→'a)**par**→(int→'a)**par**	$\langle f_0, \ldots, f_{p-1}\rangle \mapsto \langle$(**fun** $i \rightarrow f_i$ 0), . . . , (**fun** $i \rightarrow f_i$ (p−1))⟩

Fig. 2 Summary of the BSML primitives.

vector and switch to the local level. Replicated values are available inside the vectors. Now, only within a vector, to access to local information, we add the syntax x to read the vector x and get the local value it contains. The *ids* can be accessed with the predefined vector **pid**. For example, using the toplevel for a *simulated* BSP machine with 3 processors:

```
# let vec1 = ≪ "HLPP" ≫ in
        ≪ $vec1$^",␣proc␣"^(string_of_int $pid$) ≫ ;;
— : string par = <"HLPP,␣proc␣0", "HLPP,␣proc␣1", "HLPP,␣proc␣2">
```

The # symbol is the prompt that invites the user to enter an expression to be evaluated. Then, the toplevel gives the evaluated value with its type. Thanks to BSML confluence, it is ensured that the results of the toplevel or of the parallel implementation are identical.

The **proj** primitive is the only way to *extract* local values from a vector. Given a vector, it returns a function such that applied to the *pid* of a processor, the function returns the value of the vector at this processor. **proj** performs communication to make local results available globally and ends the current super-step. For example, if we want to convert a vector into a list, we write:

```
# let list_of_par vec = List.map (proj vec) procs;;
— : val list_of_par : 'a par → 'a list = <fun>
# list_of_par ≪ $pid$ ≫ ;;
— : int list = [0; 1; 2]
```

where procs is the list of *ids* [0; 1; · · · ;p-1].

The **put** primitive is another communication primitive. It allows any local value to be *transferred* to any other processor. It is also synchronous, and ends the current super-step. The parameter of **put** is a vector that, at each processor, holds a function of type (int→'a) returning the data to be sent to processor j when applied to j. The result of **put** is another vector of functions: At a processor j the function, when applied to i, yields the value *received from* processor i by processor j. For example, a total_exchange could be written:

```
# let total_exchange vec =
    | let msg = put ≪ fun dst → $vec$≫ in
    |   | ≪ List.map $msg$ procs ≫ ;;
— : val total_exchange : 'a par → 'a list par = <fun>
# total_exchange ≪ $pid$ ≫ ;;
— : int list par = <[0;1;2], [0;1;2], [0;1;2]>
```

where the BSP cost is $(\mathbf{p}-1)\times s\times \mathbf{g}+\mathbf{L}$ where s is the size of the biggest sent value.

2.3 The MULTI-BSP Model for Hierarchical Architectures

The MULTI-BSP model [34] 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

Fig. 4 The difference between the MULTI-BSP and BSP models for a multicore architecture.

(*node*) contains memory. Fig. 4 illustrates the difference between both models for multicores. There exist other hierarchical models [38], such as D-BSP [1] or H-BSP [7], but MULTI-BSP describes them in a simpler way. An instance of MULTI-BSP is defined by $\mathbf{d}$ the depth of a tree and 4 parameters for each *stage* i:

- $\mathbf{p_i}$ is the number of components inside the i stage. We consider $\mathbf{p}_1$ as a basic computing unit where a step on a word is considered as the unit of time.
- $\mathbf{g_i}$ is the *bandwidth* between stages i and $i+1$: the ratio of the number of operations to the number of words that can be transmitted in a second (illustrated in Fig. 3).
- $\mathbf{L_i}$ is the *synchronisation cost* of all components of $i-1$ stage, but *no* synchronisation across above branches in the tree. Every components can execute codes but they have to synchronise in favour of data exchange. Thus, MULTI-BSP does not allow subgroup synchronisation as the D-BSP does: at a stage i there is only a synchronisation of the sub-components, a synchronisation of each of the computational units that manage the stage $i-1$.
- $\mathbf{m_i}$ is the amount of memory available at stage i.

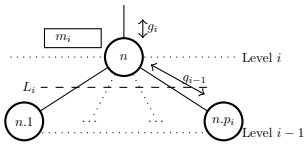

Fig. 3 MULTI-BSP parameters.

A node executes some codes on its nested components (*aka* "*children*"), then waits for results, do the communication and synchronised the sub-machine.

Considering C_j^i as the communication cost of a super-step i communicating with level j, $C_j^i = h_i \times \mathbf{g}_j + \mathbf{L}_j$. With h_i the message at step i, $\mathbf{g}_j$ communication bandwidth with level j and $\mathbf{L}_j$ the synchronisation cost. We can *recursively* express the cost of a MULTI-BSP algorithm as follow: $\sum_{i=0}^{N-1} w_i + \sum_{j=0}^{d-1} \sum_{i=0}^{M_j-1} C_j^i$ where N is the number of computational super-steps, w_i is the cost of a single computation step and M_j is the number of communication phases at level j.

3 Design of 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. We first present the execution model that follows this idea; then, we present the specific syntax and finally, we give the limitations when using some advanced OCAML features.

3.1 Execution Model

A MULTI-ML tree is formed by *nodes* and *leaves* as proposed in MULTI-BSP with the difference that a node is not only a memory but has the ability to *manage*

(*coordinate*) the values exchanged by its sub-machines. However, as common architectures do not have dedicated processors for each memory, one (or more, implementation dependent) selected computation unit has the responsibility to perform this task which is little in practice. Because leaves are their own computing units, our approach coincides with MULTI-BSP if we consider that all the computations and memory accesses, at every nodes, are performed by one (or more) leaf: *replicated codes* (outside vectors) that takes place in nodes will be costlier than in leaves. This is the reason why computations on nodes are reserved to the simple task of *coordination*. The MULTI-ML approach is also a bit more relaxed than MULTI-BSP regarding synchronisation. We allow asynchronous codes in the sub-machines when only lower memories accesses are used, unlike MULTI-BSP who forces a synchronisation. Of course, we do synchronise if a communication primitive is used. As suggested in [34], we also allow flat communications between nodes and leaves without using an upper level.

The type $'a\ tree$, symbolised by $\wr t \wr$, stands for a MULTI-ML tree. Every node and leaf contains a value of type $'a$ in its own memory. It is important to notice that the values contained in a tree are accessed by the corresponding node (or leaf) only. It is impossible to access the values of another component without using explicit communications. In MULTI-ML codes, we differentiate 4 *strictly separated execution levels*: (1) the level m (MULTI-BSP) is the upper one (*outside* trees) and is appropriate to call multi-functions and managing the trees; codes at this level are executed by all the computation units in a SPMD fashion; (2) the level b (BSP) is use inside multi-functions and is dedicated to execute BSML codes on nodes; (3) level l (local) corresponds to the codes that are run inside vectors; (4) level s stands for standard OCAML codes finally executed by the leaves. It is to notice that it is impossible to exchange vectors or trees and, like in BSML, the *nesting of parallelism* (of vectors/trees) is forbidden.

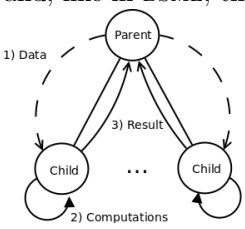

Fig. 5 Code propagation.

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. 5.

Let us consider a code where, on a node, the following code is executed: $\ll$e$\gg$. As shown in Fig. 6, 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.

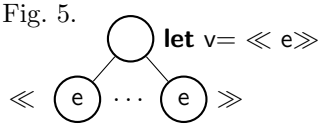

Fig. 6 Vector distribution.

3.2 The MULTI-ML Language

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, $v_{n.i}$ 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**	$\langle v_0, \ldots, v_{p_n} \rangle$, where $\forall i$, f i $= v_i$, at id n of the tree
finally v_1 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

Fig. 9 Summary of the MULTI-ML primitives.

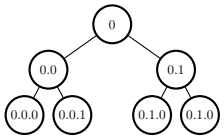

Fig. 7 Node identifiers.

n denotes the id of a node/leaf, *i.e.* its position in the tree encoded by the top-down path of positions in node's vectors. For example, 0 stands for the root, 0.0 for its first child, *etc.* For the ith component of a vector at node n, the id is $n.i$. Fig. 7 illustrates this naming. We now describe in details those primitives and let-multi functions.

The let-multi Construction. The goal is to define a multi-function *i.e.* a recursive function over the MULTI-BSP tree. Except for the tree's root, when the code ends on a stage i, the values are made available to the upper stage $i-1$ in a vector. The let-multi construction offers a syntax to declare codes for two levels, one for the nodes (level b) and one for the leaves (level s):

```
let multi f [args] =
  where node = ... (* BSML code *)
  where leaf = ... (* OCAML code *)
```

[args] are the arguments of the newly define multi-function f. In the *leaf block* (*i.e.* level s), we find usual ML computations and most of the calculation need to take place here. In the *node block* (*i.e.* level b), we find the code that will be executed at every stage of the tree, but on the leaves. Typically, a node is charged to *propagate* the computations and the data using vectors (level l); to *manage* the sub-machine computations; and finally, *gather* the results using the **proj** — extraction of values from a vector. To go one step deeper on the tree, the node code must *recursively call* the multi-function inside a vector. This call *must* be done inside a vector in order to spread the computation all over the tree in the deeper stages. It is also to notice that a multi-function can only be called at m level of the code and values at this level are available throughout the multi-function execution. Fig. 8 shows, as an example of how data moves through the MULTI-BSP tree, a simple program summing of the elements of a list.

```
1 let multi par_fold l =
2   where node =
3     let v=mkpar (fun i→split i l) in
4     let res=≪par_fold $v$≫ in
5       sum (flatten res)
6   where leaf =
7     List.fold_left (fun x y→x+y) 0 l
8 (* flatten:'a par→'a list *)
9 (* sum: int list→int *)
```

Fig. 8 Sum MULTI-ML example.

It works as follows: We define the multi-function (line 1); Lines $2-5$ give the code for the nodes and lines $6-7$ give the code for the leaves; The list is scattered across each component i of the vector (line 3); We recursively call the multi-function on the sub-lists (line 4); We finally gather the results in line 5 (sum, a BSML code, performs a **proj** to sum the $\mathbf{p}_n$ projected integers).

Tree Construction. It is not fundamentally different of the above multi-functions: Unlike generating a single usual ML value only, the "let multi" builds a *tree* ⦇t⦈ of type *'a tree*. For this, a new constructor determines the values that

are stored (*keep*) at each id and those that are ascended (*up*) to the upper stage —until now, the value returned by nodes and leaves was implicitly considered as the value given to the upper stage. This is the role of **finally** which works as follows: **finally ~up**:v1 **~keep**:v2 to up the value v1 and store to the tree the value v2 (having first replaced the previous stored value in the tree). The primitive **this** returns the last value stored using **finally** or None if it has never been called at this point. It is useful to update the tree.

Fig. 10 shows the modifications that has to be added to sum_list in order to obtain a tree containing the list of partial sums on every leaves and the maximal sub-sums on each node —thus the final sum on the root node. The code works as follows: We use a generic operator and a list to be distributed (line 1); then we traverse twice (distinguished with a boolean flag) the MULTI-BSP machine, first to split the list and then to compute the partial sums; for the nodes, we split the list (line 5) and then we do a *recursive call* over the scattered lists to continue the splitting at lower levels and then recover the partial sums of children nodes (line 6), BSP scan is used on this partial results to transfer and compose partial results from sibling nodes left to right (bsp_scan could be any BSML scan code, direct or logarithmic); finally, we do a recursive call again in a vector (lines $7-9$) to complete the partial sums with the communicated values and for this, we transmit down a list containing only the last value for each branch, keep it in the tree and give it to the upper level (line 10); when reaching the leaves the first time, we compute the partial sums (line 15) and, each time a value (communicated by other branches) comes down to a leaf, we add it to its own partial sums (line 16). Note that using two multi-functions, one to first split the list and another one to compute the partial sums, is surely easier, but using a one-shot multi-function, we exhibit more features of MULTI-ML.

```
1  let par_scan_list op li =
2    let multi m_scan flag l =
3      where node =
4        if flag then
5          let spl=mkpar (fun i→split i l) in
6          let deep=bsp_scan op ≪m_scan true $spl$≫ in
7          let v=last ≪if $pid$≠0
8                        then (m_scan false [$deep$])
9                        else noSome $this$≫  in
10         finally ~up:v ~keep:[v]
11       else
12         let v=last ≪m_scan false #l#≫  in
13           finally ~up:v ~keep:[v]
14     where leaf =
15       let final,l'= if flag then (seq_scan op l)
16         else (map_and_last (op (List.hd l)) this) in
17       finally ~up:final ~keep:l'
18 in m_scan true li
19 (* bsp_scan:'a par→('a→'a→'a)→'a par ⇒ BSML scan *)
20 (* last:'a par→'a ⇒ gives the last element of a vector *)
21 (* seq_scan:'a list→('a→'a→'a)→'a*a list
22    ⇒ computes the scan and also returns the last element *)
23 (* map_and_last:'a list→('a→'a)→'a*a list
24     ⇒ do a map and also returns the last element *)
```

Fig. 10 Scan MULTI-ML example.

Variables Accesses. There are three different ways to access to variables in addition to the usual ML access. First, v stands for reading the value of a vector "v" inside a vector (l level, as in BSML) but also for reading a previously defined tree "v": if the vector is constructed at id n, then at component i, v stands for reading the value of "v" at id $n.i$ —stage $i-1$. Within the tree construction §e§, v stands for reading the value of "v" at id n. For short, v stands for reading one of the component of "v".

The second way is to read a value *inside* a vector which had been declared *outside*. As explained above, the values available on a node are not implicitly

available on the child nodes. It is thus necessary to copy them from a node to its sub-machines. For example, the code **let** x=1 **in let** vect=≪x+1≫ is incorrect because "x" is not available on children. It is imperative to use ≪#x#+1≫ to copy the value of x from the b level into the vector (l level).

Outside a vector (g level), when the code is computing at id n, accessing to a tree t is *implicitly* reading its value at id n. Finally, **gid** is the predefined tree of nodes/leaves ids. Within the component i of a vector constructed on node n (l level), **gid** stands for the id $n.i$ whereas inside a §e§ code, it stands for the identifier of the node evaluating e; at evaluation level g, **gid** stands for n (if executed at id n) and at evaluation level m, it is the tree of node identifiers.

A Convenient Tree Construction. For *building* a tree *without using* a multi-function (which induce communications), we add the §e§ syntax. It allows to execute e on every nodes and leaves. One can read the values of a previously defined tree ≀t≀ using the t access in the code of e. In this way, using the predefined tree **gid**, we can execute *different* codes on each components of a tree without any need (and possibility) of communication between the stages.

A New Primitive. For *performances* reason, we chose to add the new primitive mkpar. Indeed, in BSML, a replicated code is duplicated on every processors so it is *not* necessary to take care of data transfer in code like: **let** lst=[...] **in** ≪split **pid** lst≫ where lst is a large list and split a splitting function. With the MULTI-ML model, data are not distributed everywhere and we have to transfer data explicitly. One can write ≪split **pid** #lst#≫ but it is not useful to copy the whole list on every children in order to extract a sub list and throw the rest. This is the reason why the **mkpar** compute first p_n values and then *distribute* them to the sub-machines thus building a vector. This method is more expensive for the node n in computation time, but it reduces drastically the amount of data transfers.

3.3 Current Limitations

Exceptions and Partially Evaluated Trees. Exceptional situations are handled in OCAML with a system of *exceptions*. In parallel, this is at least as relevant: if one processor triggers an exception during a computation, BSML [14] as well as MULTI-ML have to deal with it, like OCAML would, and prevent a crash.

The problem is when an exception is raised *locally* (l level) on (at least) one processor but other processors continue to follow the execution stream, until they are stopped by the need of synchronisation. This happens when an exception *escapes* the scope of a parallel vector. Then, a crash can occur: a processor misses the global barrier. To prevent this situation, in [14], we introduce a *specific handler* of local exceptions that have not been caught locally. The structure **trypar**...**withpar** catches any exception and handles it as usual in OCAML. To do this, a barrier occurs and exceptions are communicated to all processors in order to allow a global decision to be taken. Furthermore, any access to a vector that is in an incoherent state will raise again the exception.

For MULTI-ML, if an exception is no correctly handled in a stage, it must be propagated to the upper stage at the next barrier —as in BSML. If an exception

Core-ML inductive rules $\Downarrow_\delta$

$$\frac{}{\mathcal{E} \vdash \mathbf{cst} \Downarrow_\delta \mathbf{cst}} \quad \frac{}{\mathcal{E} \vdash \mathbf{op} \Downarrow_\delta \mathbf{op}} \quad \frac{\{x \mapsto v\} \in \|\mathcal{E}\|_\delta}{\mathcal{E} \vdash x \Downarrow_\delta v} \quad \frac{}{\mathcal{E} \vdash (\mathbf{fun}\, x \to e) \Downarrow_\delta \overline{(\mathbf{fun}\, x \to e)[\mathcal{E}]}}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \overline{(\mathbf{fun}\ x\ \to\ e)[\mathcal{E}']} \quad \mathcal{E} \vdash e_2 \Downarrow_\delta v \quad \mathcal{E}' \oplus_\delta \{x \mapsto v\} \vdash e \Downarrow_\delta v'}{\mathcal{E} \vdash (e_1\ e_2) \Downarrow_\delta v'} \quad \frac{\mathcal{E} \vdash e_1 \Downarrow_\delta v_1 \quad \mathcal{E} \vdash e_2 \Downarrow_\delta v_2}{\mathcal{E} \vdash (e_1, e_2) \Downarrow_\delta (v_1, v_2)}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \mathbf{op} \quad \mathcal{E} \vdash e_2 \Downarrow_\delta v \quad \overline{(\mathbf{op}\ v)} \equiv v'}{\mathcal{E} \vdash (e_1\ e_2) \Downarrow_\delta v'} \quad \frac{\mathcal{E} \vdash e_1 \Downarrow_\delta v \quad \mathcal{E} \oplus_\delta \{x \mapsto v\} \vdash e_2 \Downarrow_\delta v'}{\mathcal{E} \vdash \mathbf{let}\ x = e_1\ \mathbf{in}\ e_2 \Downarrow_\delta v'}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \mathbf{true} \quad \mathcal{E} \vdash e_2 \Downarrow_\delta v_2}{\mathcal{E} \vdash \mathbf{if}\, e_1 \mathbf{then}\, e_2 \mathbf{else}\, e_3 \Downarrow_\delta v_2} \quad \frac{\mathcal{E} \vdash e_1 \Downarrow_\delta \mathbf{false} \quad \mathcal{E} \vdash e_3 \Downarrow_\delta v_3}{\mathcal{E} \vdash \mathbf{if}\, e_1 \mathbf{then}\, e_2 \mathbf{else}\, e_3 \Downarrow_\delta v_3} \quad \frac{\mathcal{E} \oplus_\delta \{f \mapsto (\mathbf{rec}\ f \to e)\} \vdash e \Downarrow_\delta v}{\mathcal{E} \vdash (\mathbf{rec}\ f \to e) \Downarrow_\delta v}$$

BSML-like primitives inductive rules $\Downarrow_n^b$

$$\frac{\forall i \in \{1, \ldots, \mathbf{p}_n\} \quad \mathcal{E} \vdash e \Downarrow_{n.i}^l v_i}{\mathcal{E} \vdash < e > \Downarrow_n^b \langle v_1, \ldots, v_{\mathbf{p}_n} \rangle} \quad \frac{\mathcal{E} \vdash e \Downarrow_n^b \langle v_1, \ldots, v_{\mathbf{p}_n} \rangle}{\mathcal{E} \vdash \mathbf{proj}\ e \Downarrow_n^b \overline{(\mathbf{fun}\ i \to v_i)}} \quad \frac{\mathcal{E} \vdash e \Downarrow_n^b \langle \overline{f_1}, \ldots, \overline{f_{\mathbf{p}_n}} \rangle}{\mathcal{E} \vdash \mathbf{put}\ e \Downarrow_n^b \langle \overline{f'_1}, \ldots, \overline{f'_{\mathbf{p}_n}} \rangle}$$

$$\frac{\{x \mapsto \langle v_1, \ldots, v_i, \ldots, v_{\mathbf{p}_n} \rangle\} \in \|\mathcal{E}\|_n}{\mathcal{E} \vdash \$x\$ \Downarrow_{n.i}^l v_i} \quad \frac{\{x \mapsto v\} \in \|\mathcal{E}\|_n}{\mathcal{E} \vdash \#x\# \Downarrow_{n.i}^l v} \quad \frac{}{\mathcal{E} \vdash \$\mathbf{pid}\$ \Downarrow_{n.i}^l i}$$

$$\frac{\|\mathcal{E}\|_n \vdash e \Downarrow_\delta v}{\mathcal{E} \vdash e \Downarrow_n^{b,l} v} \quad \frac{\mathcal{E} \vdash e \Downarrow_n^b f \quad \forall i \in \{1, \ldots, \mathbf{p}_n\} \quad \mathcal{E} \vdash (f\ i) \Downarrow_n^b v_i \text{ with } f \equiv \overline{(\mathbf{fun}\ x \to e')[\mathcal{E}']}}{\mathcal{E} \vdash \mathbf{mkpar}\ e \Downarrow_n^b \langle v_1, \ldots, v_{\mathbf{p}_n} \rangle}$$

Multi functions inductive rules $\Downarrow^m$

$$\frac{}{\mathcal{E} \vdash (\mathbf{multi}\, f\, x \to e_1 \to e_2) \Downarrow^m \overline{(\mathbf{multi}\, f\, x \to e_1 \to e_2)[\mathcal{E}]}} \quad \frac{}{\mathcal{E} \vdash \mathbf{gid} \Downarrow_n^{b,l} n}$$

In what follows $g \equiv \overline{(\mathbf{multi}\ f\ x \to e'_1 \to e'_2)[\mathcal{E}']}$

$$\frac{\mathcal{E} \vdash e \Downarrow_\delta v}{\mathcal{E} \vdash e \Downarrow^m v} \quad \frac{\mathcal{E} \vdash e_1 \Downarrow^m g \quad \mathcal{E} \vdash e_2 \Downarrow^m v \quad \mathcal{E}' \oplus_0 \{x \mapsto v\} \oplus_0 \{f \to g\} \vdash e'_1 \Downarrow_0^b v'}{\mathcal{E} \vdash e_1\ e_2 \Downarrow^m v'}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_{n.i}^l g \quad \mathcal{E} \vdash e_2 \Downarrow_{n.i}^l v \quad \mathcal{E}' \oplus_{n.i} \{x \mapsto v\} \oplus_{n.i} \{f \to g\} \vdash e'_1 \Downarrow_{n.i}^b v'}{\mathcal{E} \vdash e_1\ e_2 \Downarrow_{n.i}^l v'}$$

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_{n.i}^l g \quad \mathcal{E} \vdash e_2 \Downarrow_{n.i}^l v \quad \mathcal{E}' \oplus_{n.i} \{x \mapsto v\} \oplus_{n.i} \{f \to g\} \vdash e'_2 \Downarrow_{n.i}^s v'}{\mathcal{E} \vdash e_1\ e_2 \Downarrow_{n.i}^l v'}$$

Fig. 11 Operational big-step semantics rules of core-MULTI-ML.

is no handled in a multi-function, it must be thrown at the global level m as a standard OCAML exception. An exception thrown in a node of a tree leads this node in an incoherent state until the exception has been caught in a upper level. Any access to this tree must raise again the exception. This handling has not been yet implemented for MULTI-ML but the first author works on it.

An application case is *partially evaluated trees.* Take for example the following code: ≪**if** random() **then** f **else** 0≫ where f is a multi-function. A part of the tree will never be instantiated. If a partially evaluated tree is accessed during a code execution an exception could be immediately throw.

Type System. The main limitation of our prototype is the lack of a type system. Currently, *nesting* of BSML vectors/trees are not checked. A type system for BSML exists [14] but has not been implemented yet. We are convinced that adding multi-functions will not fundamentally change the type system: it's mainly a matter of adding just a new level of execution.

Other ML *Features.* We have not yet studied all the interactions of all the OCAML features with the multi-function (as well in BSML). Mainly: objects, first-order modules and GADT. We let them for future works.

4 Semantics, Implementation and Examples

We present a formal semantics of MULTI-ML as well as two implementations. To get the *assurance* that both implementations are *coherent*, using the semantics, we first prove that MULTI-ML is confluent. A semantics is useful as a *specification* of the language so as to simplify the design of the implementations. We give some examples and benchmarks to illustrates the usefulness of MULTI-ML. Our prototype is freely available at `http://lacl.fr/allombert/multi-ml.html`.

4.1 Operational Semantics

We give a *big-step* semantics of a core-language —without tree creation to simplify the presentation. The syntax (Fig. 12) extends that of popular core-ML.

e ::= /* core-ML */
| x | **cst** | **op** | (e, e) | **let** $x = e$ **in** e | $(e\ e)$
| **if** e **then** e **else** e | (**fun** $x \to e$) | (**rec** $f \to e$)
/* BSML-like primitives */
| $\$x\$$ | $\#x\#$ | $<e>$ | **pid**
| **mkpar** e | **gid** | **proj** e | **put** e
/* multi-fun, without tree construction */
| (**multi** $f\ x \to e \to e$)

Fig. 12 Syntax of core-MULTI-ML.

Programs contain variables, constants (integers, *etc.*), operators ($\leq$, $+$, *etc.*), pairing, **let**, **if**, **fun** statements as usual in ML, **rec** for recursive calls, the BSML primitives ($<e>$, **put**, **proj**), **mkpar**, access $\$x\$$ to the local value of a vector x, local copy $\#x\#$ of a parent's variable x, the vector of **pid** component and **gid** the tree of ids. Finally, we define let-multi as particular functions with codes for nodes and leaves.

The semantics is a big-step one with *environments* that represent the *memories*. We have a tree of memories (one per node and leaf) and we note them $\mathcal{E}$. $\|\mathcal{E}\|_n$ denotes the memory of $\mathcal{E}$ at n where n is the id of the node/leaf. $\{x \mapsto v\}$ denotes a binding of a variable x to a value v in the memory; $\in$ denotes a membership test and $\oplus$ denotes an update. Those operators have the subscript n that denotes the application in a specific memory.

$\mathcal{E} \vdash e \Downarrow v$ denotes the *evaluation* of e in the environment $\mathcal{E}$ to the value v. A value is a constant, an operator or a functional *closure* (noted $\overline{(\mathbf{fun}\, x \to e)[\mathcal{E}]}$) that is a function with its own environment of execution. The rules of evaluation are define by *induction* and fully given in Fig. 11. Note that to simplify the reading of the rules, we count vector pids from 1 to $\mathbf{p}_n$ and not from 0 to $\mathbf{p}_n$–1.

Even if the semantics contains many rules, there is no surprise and it has to be read naturally. As explain before, there are 4 different levels of execution: (1) level m for the execution on all computation units; (2) BSP level b for BSML codes. These rules are subscripted with the id n of the sub-machine like for memories; (3) local level l for the codes inside a vector; (4) and finally level s on the leaves. In this way, the evaluation $\Downarrow$ is upscripted by the level. Note that a code at level l becomes at level b if a recursive call of a multi-function occurs.

The first rules are for the core-ML part and are as intended: a value returns a value; an application needs a closure (or an operator), a value and then the core of the closure is used with the properly modified environment; pairing builds two values; a "let" adds a new binding. They are used to evaluate any expression that do only contains ML code (no MULTI-ML primitives)

The rules for the BSML primitives work as follows: creating a new vector for the machine of id n is triggering $\mathbf{p}_n$ local evaluations, each with $n.i$ as

subscript since we are going one step deeper, in the ith component; **proj** and **put** rules build the functions of exchanges; $\$x\$$ rule read at $n.i$ the the ith value of the vector x created by node n; $\#x\#$ rule read the value x at the node n from its child; $\$\mathbf{pid}\$$ rule returns i on child $n.i$; **mkpar** rule creates the vector but first the node creates the values to be sent down.

For the multi-functions, we have a rule to create the function and a rule to initialise the deeper computations. In this way, at level m, we start to recurse down in the machine from id 0 with the appropriate environment and level g. Then, inside the component i of a vector of sub-machine n, the recursive call of the multi-function generates an evaluation on $n.i$. Except if we reach a leaf, then the last rule says that the code is evaluated on leaf $n.i$ with level s. We can then prove by induction the following lemma:

Lemma 1 (Confluence) $\forall \mathcal{E}$ *if* $\mathcal{E} \vdash e \Downarrow^m v_1$ *and* $\mathcal{E} \vdash e \Downarrow^m v_2$ *then* $v_1 \equiv v_2$

Co-inductive rules [25] $\Downarrow_\infty$ (for diverging programs) can be easily infered from the above rules. For sake of conciseness, we only present some typical examples:

$$\frac{\mathcal{E} \vdash e_1 \Downarrow_s \mathbf{true} \quad \mathcal{E} \vdash e_2 \Downarrow_\infty}{\mathcal{E} \vdash \mathbf{if}\, e_1 \,\mathbf{then}\, e_2 \mathbf{else}\, e_3 \Downarrow_\infty} \qquad \frac{\exists i \in \{1, \ldots, \mathbf{p}_t\} \quad \mathcal{E} \vdash e \Downarrow_\infty}{\mathcal{E} \vdash < e > \Downarrow_\infty} \qquad \frac{\mathcal{E} \vdash e \Downarrow_\infty}{\mathcal{E} \vdash \mathbf{proj}\; e \Downarrow_\infty}$$

We can then prove by co-induction the following lemma:

Lemma 2 (Mutually exclusive) $\forall \mathcal{E}$ *if* $\mathcal{E} \vdash e \Downarrow^m v$ *then* $\neg(\mathcal{E} \vdash e \Downarrow^m_\infty)$

Results do not depend of the order of evaluation of the processors nor of the BSP sub-machines. All strategy work and return the same values, especially a sequential simulation and a distributed implementation. The former is fine for debugging whereas the latter is for benchmarking. We now present both.

4.2 Sequential Simulation and Distributed Implementation

Sequential Simulation. We propose a sequential simulator that works as the OCAML toplevel. Given an architecture as a configuration file, the toplevel allows simulating MULTI-ML codes on a single core machine and printing the results. Currently, all the main features of OCAML are available, without typing. To be executed, the MULTI-ML code is converted into a sequential code using a modified OCAML parser. The simulator creates a tree structure to represent the whole MULTI-BSP architecture. Vectors are represented as arrays of data. A global hashtable is used to simulate the memory available at each levels as suggested by the semantic. Fig. 13 shows the result when using the toplevel for a simulated MULTI-BSP machine composed of 2 cores with respectively 2 threads.

Distributed Implementation. To be portable, our implementation is written to use various communication libraries. We have thus a modular approach and our implementation depends of a generic functors that requires the architecture configuration and some particular communication routines, mainly, asynchronous broadcasting and gathering values for

```
#let multi f n =
    where node =
        let _=≪f $pid$≫ 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→ 3"
|  |  → "0.0.1→ 4"
o "0.1→ 2"
|  |  → "0.1.0→ 5"
|  |  → "0.1.1→ 6"
```

Fig. 13 Example of the Toplevel.

a group of processors, total echange, and builds groups of processes. Our first implementation of this module is currently based on MPI. We create one MPI process for every nodes and leaves of the architecture. Those processes are distributed over the physical cores and threads in order to balance the charge. As the code executed by nodes is, most of the time, a simple task of parallel management, this extra job is thus distributed over the leaves to reduce its impact.

Our implementation is based on a daemon running on each MPI processes. Each daemon have 3 MPI communicators to communicate with their father (upper node), their sons (leafs) and their brothers (processes at the same sublevel).These daemon are waiting for a task given by their father. When a task is received, it is executed by the daemon, and then, they returns to the waiting state until they receive a "kill" signal, corresponding to the end of the program.

As the code is a SPMD one, every processes know the entire code and they just need to receive a signal to execute a task. To do so, and to avoid serialising codes that are inside functional values (the closures) and know by all the nodes due to a SPMD execution, when transmitting down values and thus creating parallel vectors, we identify the vectors by two kinds of identifiers: (1) a static identifier is generated globally for every vectors and reference their computations through the execution; (2) when a node need to create a parallel vector, it generates a dynamic identifier that represent the data on its leaves. Then, when a node execute some code using parallel values inside a vector, it just sends the static identifier (that references the code to execute) with the dynamic identifier (to substitute the distributed value) to its sons which can then execute the correct tasks. The main advantage of this method it to avoid serialising unnecessary codes when creating vectors, and thus reduces the size of the data exchanged by the processes. But, associating a value by a dynamic identifier can lead to a memory over-consumption, for example in loops. When the life cycle of a vector is terminated, we manually clean the memory by removing all the obsolete identifier and calling the garbage collector.

Shared memory. We propose an implementation to avoid some unnecessary copies of the transmitted data. Indeed, the OCAML memory is garbage collected and, to be safe, only a copy of the data can me transmitted. Using the standard POSIX "mmap" routine and some IPC functionalities (to synchronise the processes), the child (as daemons) can read asynchronously the transmitted serialised value in the mapping of the virtual address space of the father and synchronise with the father only, as the MULTI-BSP model suggest. As architectures can have different types of memory (distributed or shared), it is possible to mix executions schemes. Since the OCAML memory is garbage collected (currently with a global thread lock), we sadly cannot use pthreads as done in [35] to share values without performing first a copy. We are currently working on using some tricks to overcome this limitation but we leave it as future work.

4.3 Benchmarks

In this section we present the benchmark of a simple scan with integer addition and a naive implementation of the sieve of Eratosthenes. A scan can be used

to perform the sieve of Eratosthenes using a particular operator which implies more computations and communications than a simple list summing.

The MULTI-BSP cost of the scan algorithm is as follows:

$$\sum_{i\in[0\ldots\mathbf{d}[} V_{sp}(i) + \mathcal{O}(|l_{\mathbf{d}}|) + \sum_{i\in[0\ldots\mathbf{d}[} C_i.$$

Where $\sum_{i\in[0\ldots\mathbf{d}[} V_{sp}(i)$ is the total cost of splitting the list until the leaves (at depth $\mathbf{d}-1$); $\mathcal{O}(|l_{\mathbf{d}}|)$ is the time needed to compute the local sums in the leaves; and $\sum_{i\in[0\ldots\mathbf{d}[} C_i$ corresponds to the cost of diffusing partial sum back to leaves and to add these values to values held by leaves. This diffusion is done once per node. $V_{sp}(i)$ is the work done at level i to split the locally held chunk l_i and scatter it among children nodes. Splitting l_i in $\mathbf{p}_i$ chunks costs $\mathcal{O}(|l_i|)$ where $|l_i|$, the size of l_i, is $n * \prod_{j\in[0\ldots i[} \frac{1}{\mathbf{p_j}}$ where n is the size of the initial list holds by the root node. Scattering it among children nodes costs $\mathbf{p_i} * \mathbf{g_{i-1}} + \frac{n_i}{\mathbf{p_i}} + \mathbf{L_i}$. The sequential list scan cost at leaves is $\mathcal{O}(|l_d|) = \mathcal{O}(n * \prod_{i\in[0\ldots\mathbf{d}[} \frac{1}{\mathbf{p_i}})$.

The cost C_i at level i is the cost of a BSP scan, of a diffusion of the computed values until leaves and the sequential cost of a map on list held by leaves. Let s be the size of the partial sum, the cost of BSP scan at level i is $s * \mathbf{p_i} * \mathbf{g_{i-1}} + \mathbf{L_i}$, the diffusion cost is $\sum_{j\in]i\ldots\mathbf{d}]} \mathbf{g_j} * s + l_j$ and the final map is cost $\mathcal{O}(s_{\mathbf{d}})$. The size s may be difficult to evaluate: for a sum of integers it will simply be the size of an integer, but for Eratosthenes sieve, the size of exchanged lists varies depending on which data are held by the node.

```
1  let scan_direct op vv =
2  let mkmsg pid v dst =
3    if dst<pid then None else Some v in
4  let procs_lists =
5    ≪ fun pid → from_to 0 pid≫  in
6  let receivedmsgs =
7    put(apply(mkpar mkmsg) vv) in
8  let values_lists =
9    ≪ List.map ((compose noSome)
10         $receivedmsgs$) $procs_lists$ ≫
   in
11 ≪ (fun (h::t)→ List.fold_left op h t )
12      $values_lists$ ≫
```

Fig. 14 BSML direct scan code.

The sieve of Eratosthenes generates a list of primary numbers below a given integer n. From the list of all elements lesser than n, it iteratively removes elements that are a multiple of the smaller element of the list that as not been yet considered. We generate only the integers that are not multiple of the 4 first prime numbers, then we iterate $\sqrt{n}$ time (as it is known to be the maximum number of needed iteration). On our architectures, the direct and logarithmic scans are as efficient. Fig. 14 gives the BSML code of the direct scan. This code build the list of elements to communicate to its neighbours and exchange the values using the **put** primitive. Then every processes maps the received values on their own data. We used the following functions: elim:int list→ int→ int list which deletes from a list all the integers multiple of the given parameter; final_elim:int list→ int list→ int list iterates elim using elements from the first list to delete elements in the second; seq_generate:int → int→ int list which returns the list of integers between 2 bounds; and select:int → int list→ int list which gives the $\sqrt{n}$th first prime numbers of a list.

For this naive example, we use a generic scan computation with final_elim as the $\oplus$ operator. In our computation, we also did extend the scan so that the sent values are first modified by a given function (select) to just sent the $\sqrt{n}$th first prime numbers. The BSP methods is thus simple: each processor i

$p_0 = 4$	$g_0 = \infty$	$L_0 = 149000$	$m_0 = 0$	$p_0 = 8$	$g_0 = \infty$	$L_0 = 195400$	$m_0 = 0$
$p_1 = 2$	$g_1 = 89$	$L_1 = 1100$	$m_1 = 64Gb$	$p_1 = 2$	$g_1 = 14$	$L_1 = 472$	$m_1 = 16Gb$
$p_2 = 16$	$g_2 = 6$	$L_2 = 1800$	$m_2 = 20Mb$	$p_2 = 4$	$g_2 = 6$	$L_2 = 800$	$m_2 = 6Mb$
$p_3 = 0$	$g_3 = 3$	$L_3 = 0$	$m_3 = 0$	$p_2 = 0$	$g_2 = 5$	$L_2 = 0$	$m_2 = 0$

Fig. 15 Multi-BSP parameters of Mirev3 (left) and Mirev2 (right).

holds the integers between $i \times \frac{n}{\mathbf{p}} + 1$ and $(i+1) \times \frac{n}{\mathbf{p}}$. Each processor computes a local sieve (the processor 0 contains thus the first prime numbers) and then our scan is applied. We then eliminate on processor i the integers that are multiple of integers received from processors of lower identifier.

Benchmark were done on two parallel architectures named Mirev2 and Mirev3. Here are the main specifications of these machines:

- Mirev2: 8 nodes with 2 quad-core (AMD 2376) at $2.3Ghz$ with $16Gb$ of memory per node and a $1Gb/s$ network.
- Mirev3: 4 nodes with 2 hyper-threaded octo-core (16 threads) (Intel XEON $E5-2650$) at $2.6Ghz$ with $64Gb$ of memory per node and a $10Gb/s$ network.

The MULTI-BSP and BSP model can be used to estimate the cost of an algorithm. Thus, we estimated the cost of transferring values and the sequential cost of summing lists of integer. Then, we used the BSP parameters given in Fig. 15 in order to predict and compare the execution time of scan. The **g** and **L** BSP parameters are given in Fig.19, one can notice that until 64 cores **g** evolves linearly, but after that, too many cores access the network, producing a bottleneck and severely hindering performances.

We measured the time to compute the sieve without the time to generate the initial list of values. The experiments have been done on Mirev2 and Mirev3 using BSML (MPI version) and MULTI-ML implementations over lists of increasing size on an increasing number of processors. The processes have been assigned to machines in order to scatter as much as possible the computation among the machines, *i.e.* a 16 process run will uses one core on each processor of the 8 machines of Mirev3. Fig. 16 and 17 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.

Fig. 18 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.

As the experiment shows, MULTI-ML out-performs BSML when the MULTI-BSP model allows to confine heavy communications in a node before communicating with other nodes. For the sum of integers, where the MULTI-BSP model cannot bring improvements, we can see that the additional managing cost of MULTI-ML is negligible.

	100_000		500_000		1_000_000		3_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	0.7	1.8	22.4	105.0	125.3	430.7	...	...
16	0.5	0.8	13.3	50.3	68.1	331.5	1200.0	...
32	0.3	0.5	2.6	18.9	11.3	122.2	173.2	...
48	0.5	0.4	1.75	14.5	5.5	88.4	69.3	...
64	0.3	0.3	1.3	8.7	4.1	56.1	51.1	749.9
96	0.3	0.38	1.6	6.3	3.9	30.8	38.1	576.1
128	0.5	0.45	2.1	5.2	4.7	24.3	30.6	443.7

Fig. 16 Execution time of Eratosthenes using MULTI-ML and BSML on Mirev3.

	100_000		500_000		1_000_000	
	MULTI-ML	BSML	MULTI-ML	BSML	MULTI-ML	BSML
8	1.5	1.7	64.5	106.1	402.9	1538.1
16	0.45	0.93	16.0	49.3	91.4	631.7
32	0.14	0.45	4.1	18.7	21.1	219.7
48	0.13	0.40	2.6	11.0	10.8	123.5
64	0.11	0.34	1.89	7.5	8.2	80.5

Fig. 17 Execution time of Eratosthenes using MULTI-ML and BSML on Mirev2.

	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

Fig. 18 Execution time and predictions of scan (sum of integers) on Mirev3.

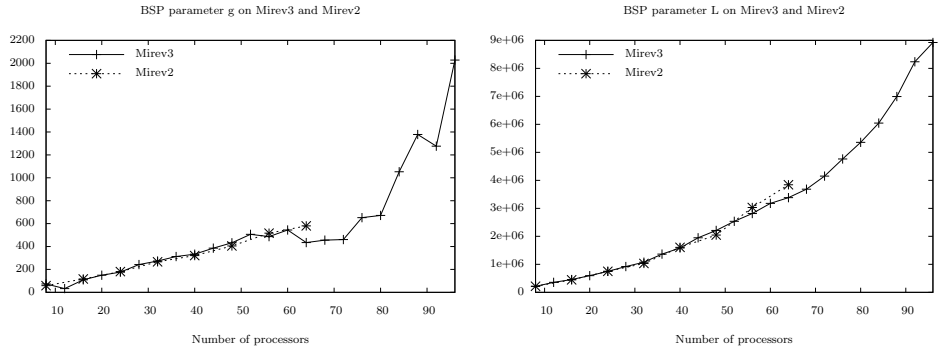

Fig. 19 **g** and **L** BSP parameters of Mirev2 and Mirev3 in flops.

5 Related Work

There are a lot parallel languages or parallel extensions of a sequential language (functional, iterative, object-oriented, *etc.*). It would be too long to list all of them. We chose to point out those that were the most important to our mind. Notice that, except in [26], there is a lack of comparisons between parallel

languages. It is difficult to compare them since many parameters have to be taken into account: efficiency, scalability, expressiveness, *etc.*

5.1 Programming Languages and Libraries for BSP like Computing

Historically, the first library for BSP computing was the BSPLIB [21] for the C language; it has been extended in the PUB library [4] by adding subgroup synchronisations, high performance operations and migration of threads. For the GPU architectures, a BSP library was provided in [23] with mainly DRMA primitives close to the BSPLIB's ones.

For JAVA, different libraries exist. The first one is [17]. There is also BSP-CORE [35] (also for C++). A library with scheduling and migration of BSP threads has been designed in [29] —the scheduling is implicit but the migration can be explicit. The library HAMA [32] is implemented using a "MAPREDUCE-like" framework. We can also highlight the work of NESTSTEP [24] which is C/JAVA library for BSP computing, which authorises nested computations in case of a cluster of multi-core but without any safety.

The BSML primitives were adapted for C++ [18]: the BSP++ library provides nested computation in the case of a cluster of multi-cores (MPI+OPEN-MP). But it is the responsibility of the programmer to avoid harmful nested parallelism. BSML also inspired BSP-PYTHON [22] and BSP-HASKELL [28].

5.2 Functional Parallel Programming

A survey to parallel functional programming can be found in [20]. It has been used as a basis for the following classification with some updates.

Data-parallel Languages. The first functional one was NESL [3]. This language allows to create particular arrays and nested computations within these arrays. The abstract machine is responsible for the distribution of the data over the available processors. For ML, there is MANTICORE [12], an extension of NESL with the dynamic creation of asynchronous threads and send/received primitives. For GPU architectures, an extension of OCAML, using a special syntax for the kernels has been developed in [5].

SAC (Single Assignment C) [16] is a lazy language (with a syntax close to C) for array processing. Some higher-order operations on multi-dimensional arrays are provided and the compiler is responsible for generating an efficient parallel code. A data-parallel extension of HASKELL has been done in [10] where the language allows to create data arrays that are distributed across the processors. And some specific operations permit to manipulate them.

The main drawback of all these languages is that cost analysis is hard to do since the system is responsible for the data distribution.

Explicit process creation. We found two extensions of HASKELL in this category: EDEN and GPH [31]. Both use a small set of syntactic constructs for explicit process creation. Their fine-grain parallelism, while providing enough

control to implement parallel algorithms efficiently, frees the programmer from the tedious task of managing low-level details of communications —which uses lazy shared data. Processes are automatically managed by sophisticated runtime systems for shared memory machines or distributed ones.

As above, cost analysis is hard to do and, sometimes, the runtime fails to distribute correctly the data [31]; it introduces too much communication and thus a lack of scalability. Another distributed language is HUME [19]. The main advantage of this language is that it is provided with a cost analysis of the programs for real-time purpose but with limitations of the expressiveness.

Algorithmic skeletons. Skeletons are patterns of parallel computations [15]. They can be seen as high-order functions that provide parallelism. They thus fall into the category of functional extensions. They are many skeleton libraries [15]. For OCAML, the most known work is the one of [11].

Distributed functional languages. In front of parallel functional languages, there are many concurrent extensions of functional languages such as ERLANG, CLEAN or JOCAML [27]. The latter is a concurrent extension of OCAML, which added explicit synchronisations of processes using specific patterns.

ALICE-ML [30] adds what is called a "future" for communicating values. A future is a placeholder for an undetermined result of a concurrent computation. When the computation delivers a result, the associated future is eliminated by globally replacing it by the result value. The language also contains "promises" that are explicit handles of futures. SCALA is a functional extension of JAVA which provides concurrency, using the actor model: mostly, creation of agents that can migrate across resources. Two others extensions of OCAML are [8] and [9]. The former uses SPMD primitives with a kind of futures. The latter allows migration of threads that communicate using particular shared data.

All these languages have the same drawbacks: they are not deadlock and race condition free; furthermore, they do not provide any cost model.

6 Conclusion and Future Work

6.1 Summary of the Contribution

The paper presents a language call MULTI-ML to program MULTI-BSP algorithms. It extends our previous BSML that has been designed for programming BSP algorithms. They both have the following advantages: *confluent* operational semantics; equivalence of the results for both toplevel and distributed implementation; cost model and efficiency.

The MULTI-BSP model extends the BSP one as a *hierarchical tree* of *nested* BSP machines. MULTI-ML extends BSML with a special syntax for define *special recursive functions* over this tree of nested machines, each of them programmed using BSML. In a tree, nodes contain codes to *manage* the submachines whereas leaves perform the *largest parts* of the computation. In this work, we focus on the informal presentation of MULTI-ML, an operational semantics of a core-language and benchmarks of simple examples with a compar-

ison with *predicted performances* associated with the MULTI-BSP cost model. We also compare MULTI-ML codes with BSML ones as well as the performances of both languages on a typical cluster of hyper-threaded multi-cores. As predicted, MULTI-ML codes run *faster* when the cores share the network: there is *no bottleneck*; And the multi-core synchronisations are cheaper.

Compared to BSML, MULTI-ML have several drawbacks. First, the codes, the semantics and the implementation are a bit more complex. Second, the cost model associated to the program is more difficult to grasp: designing MULTI-BSP algorithms and programming them in MULTI-ML is more difficult than using BSML only. But, from our experience, we can say that it is not so hard.

6.2 Future Work

In a close future, we plan to axiomatise the MULTI-ML primitives inside COQ, as we did for BSML in [13,36], in order to prove the *correctness* of MULTI-BSP algorithms. We also consider to formally prove that the implementations follow the formal semantics. We plan also to benchmark bigger examples. We think of model-checking problems and algebraic computations that better follow high-level languages than intensive float operations can do.

But the most important work to do is the *implementation of a type system* for MULTI-ML to ensure a true safety of the codes: forbid nesting of vectors, forbid data-races if imperatives features, such as handling exceptions [14], are used. In the long term, the type system could be used to optimise the compiler. Indeed, currently, even in the case of a share-memory architecture, only serialised values are exchanged between nodes. We consider implementing a dedicated *concurrent garbage collector*.

References

1. Beran, M.: Decomposable BSP Computers. In: Theory and Practice of Informatics (SOFSEM), *LNCS*, vol. 1725, pp. 349–359. Springer (1999)
2. Bisseling, R.H.: Parallel Scientific Computation. A Structured Approach Using BSP and MPI. Oxford University Press (2004)
3. Blelloch, G.: NESL. Encyclopedia of Parallel Computing, pp. 1278–1283. Springer (2011)
4. Bonorden, O., Judoiink, B., von Otte, I., Rieping, O.: The Paderborn University BSP Library. Parallel Computing **29**(2), 187–207 (2003)
5. Bourgoin, M., Chailloux, E., Lamotte, J.L.: SPOC: GPGPU Programming through Stream Processing with OCAML. Parallel Processing Letters **22**(2), 1–12 (2012)
6. Cappello, F., Guermouche, A., Snir, M.: On Communication Determinism in HPC Applications. In: Computer Communications and Networks (ICCCN), pp. 1–8. IEEE (2010)
7. Cha, H., Lee, D.: H-BSP: A Hierarchical BSP Computation Model. Journal of Supercomputing **18**(2), 179–200 (2001)
8. Chailloux, E., Foisy, C.: A Portable Implementation for Objective Caml Flight. Parallel Processing Letters **13**(3), 425–436 (2003)
9. Chailloux, E., Ravet, V., Verlaguet, J.: HIRONDML: Fair threads migrations for Objective Caml. Parallel Processing Letters **18**(1), 55–69 (2008)
10. Chakravarty, M., Leshchinskiy, R., Jones, S., Keller, G., Marlow, S.: Data Parallel HASKELL. In: Declarative Aspects of Multicore Prog. (DAMP), pp. 10–18. ACM (2007)

11. Cosmo, R.D., Li, Z., Pelagatti, S., Weis, P.: Skeletal Parallel Programming with OCAMLP3L 2.0. Parallel Processing Letters **18**(1), 149–164 (2008)
12. Fluet, M., Rainey, M., Reppy, J., Shaw, A.: Implicitly-threaded Parallelism in MANTICORE. SIGPLAN Not. **43**(9), 119–130 (2008)
13. Gava, F.: Formal Proofs of Functional BSP Programs. PPL **13**(3), 365–376 (2003)
14. Gesbert, L., Gava, F., Loulergue, F., Dabrowski, F.: Bulk Synchronous Parallel ML with Exceptions. Future Generation Computer Systems **26**, 486–490 (2010)
15. González-Vélez, H., Leyton, M.: A Survey of Algorithmic Skeleton Frameworks. Software, Practrice & Experience **40**(12), 1135–1160 (2010)
16. Grelck, C., Scholz, S.B.: Classes and Objects as Basis for I/O in SAC. In: Implementation of Functional Languages (IFL), pp. 30–44 (1995)
17. Gu, Y., Le, B.S., Wentong, C.: JBSP: A BSP Programming Library in JAVA. Journal of Parallel and Distributed Computing **61**(8), 1126–1142 (2001)
18. Hamidouche, K., Falcou, J., Etiemble, D.: A Framework for an Automatic Hybrid MPI +OPEN-MP Code Generation. In: SpringSim (HPC), pp. 48–55. ACM (2011)
19. Hammond, K.: The Dynamic Properties of HUME: A Functionally-based Concurrent Language with Bounded Time and Space Behaviour. In: Implementation of Functional Languages (IFL), *LNCS*, vol. 2011, pp. 122–139. Springer (2000)
20. Hammond, K., Michaelson, G. (eds.): Research Directions in Parallel Functional Programming. Springer (2000)
21. Hill, J.M.D., McColl, B., et *al*: BSPLIB: The BSP Programming Library. Parallel Computing **24**, 1947–1980 (1998)
22. Hinsen, K., Langtangen, H.P., Skavhaug, O., Ødegård, Å.: Using BSP and PYTHON to Simplify Parallel Programming. Future Generation Comp. Syst. **22**(1-2), 123–157 (2006)
23. Hou, Q., Zhou, K., Guo, B.: BSPGP: Bulk-Synchronous GPU Programming. ACM Trans. Graph. **27**(3), pp. 1–30 (2008)
24. Keßler, C.W.: NESTSTEP: Nested Parallelism and Virtual Shared Memory for the BSP Model. The Journal of Supercomputing **17**(3), 245–262 (2000)
25. Leroy, X., Grall, H.: Coinductive Big-step Operational Semantics. Inf. Comput. **207**(2), 284–304 (2009)
26. Loidl, H.W., et *al*: Comparing Parallel Functional Languages: Programming and Performance. Higher Order and Symb. Comp. **16**(3), 203–251 (2003)
27. Mandel, L., Maranget, L.: Programming in JOCAML. In: European Symposium on Programming (ESOP), no. 4960 in LNCS, pp. 108–111. Springer (2008)
28. Miller, Q.: BSP in a Lazy Functional Context. In: Trends in Functional Programming, vol. 3. Intellect Books (2002)
29. da Rosa Righi, R., et *al*: MIGBSP: A Novel Migration Model for BSP Processes Re-scheduling. In: HPC and Communications (HPCC), pp. 585–590. IEEE (2009)
30. Rossberg, A.: Typed Open Programming – a Higher-order, Typed Approach to Dynamic Modularity and Distribution. Ph.D. thesis, Universität des Saarlandes (2007)
31. Scaife, N., Michaelson, G., Horiguchi, S.: Empirical Parallel Performance Prediction From Semantics-Based Profiling. Scalable Computing: Prac. and Exp. **7**(3) (2006)
32. Seo, S., Yoon, et *al*: HAMA: An Efficient Matrix Computation with the MapReduce Framework. In: Cloud Computing (CloudCom), pp. 721–726. IEEE (2010)
33. Valiant, L.G.: A Bridging Model for Parallel Computation. Comm. of the ACM **33**(8), 103–111 (1990)
34. Valiant, L.G.: A Bridging Model for Multi-core Computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)
35. Yzelman, A.N., Bisseling, R.H.: An Object-oriented BSP Library for Multicore Programming. Concurrency and Computation: Practice and Experience **24**(5), 533–553 (2012)
36. Gesbert, L., Hu, Z., Loulergue, F., Matsuzaki, K., Tesson, J.: Systematic Development of Correct BSP Programs. In: PDCAT, pp. 334–340. IEEE (2010)
37. Fortin, J., Gava, F.: BSP-WHY: a Tool for Deductive Verification of BSP Algorithms with Subgroup ynchronization. Journal of Parallel Programming. To appear. 2015.
38. Li, C., Hains, G.: SGL: Towards a Bridging Model for Heterogeneous Hierarchical Platforms IJHPCN. **7**(2), pp. 139–151 (2012)