Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features

Frédéric Gava and Frédéric Loulergue
Laboratory of Algorithms, Complexity and Logic
University Paris XII, Val-de-Marne
61, avenue du général de Gaulle
94010 Créteil cedex – France
Tel: +33 (0)1 45 17 16 50
Fax: +33 (0)1 45 17 66 01
{gava,loulergue}@univ-paris12.fr

Research Report TR-2002-14

Abstract

The Bulk Synchronous Parallel ML (BSML) is a functional language for Bulk Synchronous Parallel (BSP) programming, based on the sequential functional language Objective Caml. It is based on an extension of the λ-calculus by parallel operations on a parallel data structure named parallel vector, which is given by intention. The Objective Caml language is a functional language but it also offers imperative features. This paper presents formal semantics of BSML with references, affectation and dereferencing.

Keywords: Parallel Programming, Bulk Synchronous Parallelism, Functional Programming, Imperative Features

1 Introduction

Declarative parallel languages are needed to ease the programming of massively parallel architectures. Moreover, those languages does not enforce in the syntax itself an order of evaluation, and thus appear more suitable to automatic parallelization. Functional languages are often considered. Nevertheless, even if some problems encountered in the parallelization of sequential imperative languages are avoided, some still remains (for example two different but denotationally equivalent programs may lead to very different parallel programs) and some are added, for example the fact that in those languages data-structures are always dynamic ones. It makes the amount and/or the grain of parallelism often too low. An opposite direction of research is to give the programmer the entire control over parallelism. Message passing facilities are added to functional languages. But in this case, the obtained parallel

languages are either non-deterministic [21], or non-functional (i.e. referential transparency is lost) [1, 8].

An intermediate approach is to offer only a set of algorithmic skeletons [4, 7, 23, 22, 3] (in the case of functional languages, it is a set of higher-order functions) that are implemented in parallel. Those algorithmic skeletons have sequential counterparts. For example, the `map` function, which takes a function f, a list $[x_0; \ldots; x_n]$ and returns the list $[(f\, x_0); \ldots; (f\, x_n)]$ is a classical algorithmic skeleton. Its usual parallel implementation scatters the list on different processors and evaluates the $(f\, x_i)$ in parallel, then gathers the results on one processor. The denotational semantics of the `map` skeleton is the same than the sequential `map` function. Its parallel semantics remains implicit in most algorithmic skeletons approaches.

If parallel programming is easier using algorithmic skeletons, there are some drawbacks. First, the set of algorithmic skeletons is finite and often depends on the domain of application. Most parallel languages based on algorithmic skeletons [22, 3, 28, 11, 9] rely on a specific compiler. So it is impossible for the programmer to extend himself the set of algorithmic skeletons. In other approaches [24, 6, 5, 12], the set of algorithmic skeletons is given as a library. Nevertheless those libraries are implemented using MPI [25]. Thus to program new skeletons the programmer may have to deal with indeterminism and deadlocks. Second, portable performance prediction is either not considered, or based on cost models too complex to be useful for the programmer (but they are used by compilers).

We are exploring thoroughly the intermediate position of the paradigm of algorithmic skeletons in order to obtain universal parallel languages whose execution costs can be easily determined from the source code (in this context, cost means the estimate of parallel execution time). This last requirement forces the use of explicit processes corresponding to the parallel machine's processors. *Bulk Synchronous Parallel* (BSP) computing [27, 19] is a parallel programming model which uses explicit processes, offers a high degree of abstraction and yet allows portable and predictable performance on a wide variety of architectures. Our BSML [18, 2] can be seen as an algorithmic skeletons language, because only a finite set of operations are parallel, but is different by two main points:

- our operations are universal for BSP programming and thus allow the implementation of more classical algorithmic skeletons. It is also possible for the programmer to implement additional skeletons. Moreover performance prediction is possible [2] and the associated cost model is the BSP cost model. Those operations are implemented as a library for the functional programming language Objective Caml [17].

- the parallel semantics of BSML are formal ones. We have a confluent calculus [14], a distributed semantics [15] and a parallel abstract machine [20], each semantics has been proved correct with respect to the previous one.

Our semantics are based on extension of the λ-calculus, but our `BSMLlib` library is for the Objective Caml language which contains imperative features. In the current version of the `BSMLlib` library the use of imperative features is unsafe and may lead to runtime errors. This paper explores formal semantics of BSML with imperative features.

We first describe functional bulk synchronous parallel programming and the problems that appear with the use of imperative features (section 2). We then give two formal seman-

tics of BSML with imperative features whose parallel executions are different (section 3) and conclude (section 4).

2 Functional Bulk Synchronous Parallelism

2.1 Bulk Synchronous Parallelism

The Bulk Synchronous Parallel (BSP) model[19] describes: an abstract parallel computer, a model of execution and a cost model.

A BSP computer has three components:

- a set of processor-memory pairs,
- a communication network allowing inter processor delivery of messages,
- a global synchronization unit which executes collective requests for a synchronization barrier.

The performance of the BSP computer is characterized by three parameters (often expressed as multiples the local processing speed):

- the number of processor-memory pairs p,
- the time l required for a global synchronization
- the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation (communication phase where every processor receives/sends at most h words) in time $g \times h$.

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjoint phases (Figure 1):

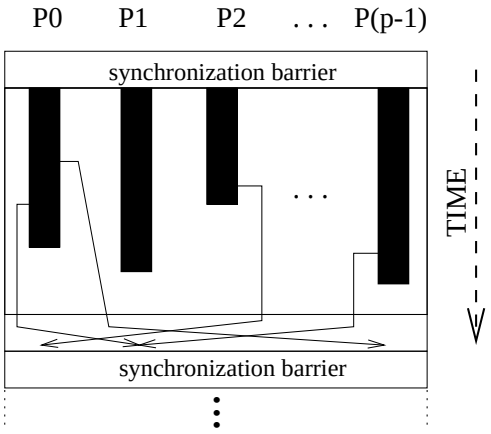

Figure 1: A BSP super-step

- each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes,
- the network delivers the requested data transfers,
- a global synchronization barrier occurs, making the transferred data available for the next super-step.

The execution time of a super-step s is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$\text{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} \times g + l$$

where $w_i^{(s)}$ = local processing time on processor i during super-step s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during super-step s.

The execution time $\sum_s \text{Time}(s)$ of a BSP program composed of S super-steps is therefore a sum of 3 terms:

$$W + H \times g + S \times l$$

where $W = \sum_s \max_i w_i^{(s)}$ and $H = \sum_s \max_i h_i^{(s)}$.

In general W, H and S are functions of p and of the size of data n, or of more complex parameters like data skew. To minimize execution time the BSP algorithm design must jointly minimize the number S of super-steps, the total volume h and imbalance of communication and the total volume W and imbalance of local computation.

2.2 Bulk Synchronous Parallel ML

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation as a library for Objective Caml. The so-called `BSMLlib` library is based on the following elements.

It gives access to the BSP parameters of the underling architecture. In particular, it offers the function `bsp_p:unit->int` such that the value of `bsp_p()` is p, the static number of processes of the parallel machine. The value of this variable does not change during execution (for "flat" programming, it is not true if a parallel composition is added to the language [16]).

There is also an abstract polymorphic type `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. Our type system enforces this restriction [**?**]. This improves on the earlier design DPML/Caml Flight [10, 8] in which the global parallel control structure `sync` had to be prevented *dynamically* from nesting.

This is very different from SPMD programming (Single Program Multiple Data) where the programmer must use a sequential language and a communication library (like MPI [25]). A parallel program is then multiple copies of a sequential program which exchange messages using the communication library. In this case messages and processes are explicit, but programs may be non deterministic or may contain deadlocks.

Another drawback of SPMD programming is the use of a variable containing the processor name (usually called "pid" for Process Identifier) which is bound outside the source program. A SPMD program is written using this variable. When it is executed, if the parallel machine contains p processors, p copies of the program are executed on each processor with the pid variable bound to the number of the processor on which it is ran. Thus parts of the program that are specific to each processor are those which depends on the pid variable. On the contrary, parts of the program which make global decision about the algorithms are those which do not depends on the pid variable. This dynamic and undecidable property is given the role of defining the most elementary aspect of a parallel program, namely it local vs global parts.

The BSML parallel constructs operates on parallel vectors. Those parallel vectors are created by

```
mkpar:  (int -> 'a) -> 'a par
```

so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$. We usually write `f` as `fun pid->e` to show that the expression `e` may be different on each processor. This expression `e` is said to be *local*. The expression `(mkpar f)` is a parallel object and it is said to be *global*.

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a superstep) and phases of global communication (second phase of a superstep) with global synchronization (third phase of a superstep). Asynchronous phases are programmed with `mkpar` and with

```
apply:  ('a -> 'b) par -> 'a par -> 'b par
```

`apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. Neither the implementation of `BSMLlib`, nor its semantics [15] prescribe a synchronization barrier between two successive uses of `apply`.

Readers familiar with BSPlib will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by

```
put:(int->'a option) par -> (int->'a option) par
```

where `'a option` is defined by: `type 'a option=None | Some of 'a`.

Consider the expression:

```
put(mkpar(fun i->fs_i))(*)
```
(1)

To send a value `v` from process `j` to process `i`, the function $\texttt{fs}_j$ at process `j` must be such that $(\texttt{fs}_j\ \texttt{i})$ evaluates to `Some v`. To send no value from process `j` to process `i`, $(\texttt{fs}_j\ \texttt{i})$ must evaluate to `None`.

Expression (1) evaluates to a parallel vector containing a function $\texttt{fd}_i$ of delivered messages on every process. At process `i`, $(\texttt{fd}_i\ \texttt{j})$ evaluates to `None` if process `j` sent no message to process `i` or evaluates to `Some v` if process `j` sent the value `v` to the process `i`.

The full language would also contain a synchronous conditional operation:

```
ifat:  (bool par) * int * 'a * 'a -> 'a
```

such that `ifat (v,i,v1,v2)` will evaluate to `v1` or `v2` depending on the value of `v` at process `i`. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core `BSMLlib` contains the function: `at:bool par -> int -> bool` to be used only in the construction: `if (at vec pid) then... else...` where `(vec:bool par)` and `(pid:int)`. `if at` expresses communication and synchronization phases.

The meaning of

```
if (at vec pid) then expr1 else expr2
```

is that of

```
ifat(vec,pid,expr1,expr2).
```

Global conditional is necessary of express algorithms like:

```
Repeat
  Parallel Iteration
Until Max of local errors < epsilon
```

Without it, the global control cannot take into account data computed locally.

2.3 Overview of Imperative Features

2.3.1 Sequential imperative features

Objective Caml offers to the programmer an important extension of functional languages: imperative features. Imperative features have been added to functional languages to offer more expressiveness. Classically, this modification is added to functional languages by the possibility of *affectation* and *allocation* of a variable or a data structure.

The idea is to add *references*. A reference is a cell of the memory which could be modified by the program. One creates a reference with the allocation's **ref**(e) construction which gives a new reference in the memory initialized to the value of e. The value kept by the reference is called the *stored value*. To use and read the stored value (*dereferencing*), we need an operation, written **!**, to extract it. Finally, we can modify the content of our reference by replacing this value by another. This operation is called *affectation* and written: $e_1 := e_2$. We use the same notations than Objective Caml. A reference binding to an identifier in a functional language is like a variable in a imperative language.

Imperative features are not a trivial extension of functional language. First,in a functional language, the value of an expression changes with the values of the free variables. If these variables have a known value, the evaluation of the sub-expressions could be done independently. For imperative language, it is not the case (and also for imperative extensions): the evaluation of a sub-expression could modify a reference by an affectation and thus affects the evaluation of the other sub-expressions which used this reference. A second difficulty came from the shared references which could not pass the well-known and classical β-reduction of the functional languages. Take for example:

```
let r=ref 2 in r:=!r*!r;(!r+1)
```

The instances of `r` are for the reference, allocated by the `ref 2` sub-expression. If we make a natural β-reduction to our expression, we would have:

```
(ref 2):=!(ref 2)*!(ref 2); (!(ref 2)+1)
```

which allocates four different references and do not have the same behavior.

To extend the dynamic semantics of functional languages and keep out the problem of the shared allocations, *locations* (written ℓ) and *store* (written s) have been added. A store, is a partial function from locations to values and a reference is evaluated to a location. In the following, we give the reduction of expressions starting from an empty store.

First, the left sub-expression of the `let` construction is evaluated and a new location ℓ is created in the store. Second, the β-reduction can be applied and finally the right sub-expression of the `let` construction is evaluated with the classical rules of a functional language:

$$
\begin{array}{lll}
 & \textbf{let } r = \textbf{ref}(2) \textbf{ in } r :=!r*!r;(!r+1) & / \quad \emptyset \\
\rightharpoonup & \textbf{let } r = \ell \textbf{ in } r :=!r*!r;(!r+1) & / \quad \{\ell \mapsto 2\} \\
\rightharpoonup & \ell :=!\ell*!\ell;(!\ell+1) & / \quad \{\ell \mapsto 2\} \\
\rightharpoonup & \ell := 2*2;(!\ell+1) & / \quad \{\ell \mapsto 2\} \\
\rightharpoonup & \ell := 4;!(\ell+1) & / \quad \{\ell \mapsto 2\} \\
\rightharpoonup & (!\ell+1) & / \quad \{\ell \mapsto 4\} \\
\rightharpoonup & (4+1) & / \quad \{\ell \mapsto 4\} \\
\rightharpoonup & 4 & / \quad \{\ell \mapsto 4\}
\end{array}
$$

2.3.2 BSML with imperative features

BSML is a parallel functional language based on BSP whose architecture model contains a set of processor-memory pairs and a network. Thus in the current implementation each processor can reach its own memory, and it causes problems.

```
let a=ref(0) in
let danger=mkpar(fun pid ->
                   a:=pid;
                   pid mod 2=0)
in if (at danger !a) then e1 else e2
```

Figure 2: Example of interaction of references with **ifat**

Take for example, the expression of figure 2. First, this expression creates a location `a` at each processor which is initialized at 0 everywhere. For the `BSMLlib` library each processor has this value in its memory. Second, a boolean parallel vector `danger` is created which is trivially `true` if the processor number is even or `false` otherwise. Thus, from the `BSMLlib` point of view, the location `a` has now a different value at each processor. After the **ifat** operator, some processor would executed `E1` and some other `E2`. But, the **ifat** is a global synchronous operator and all the processors need to executed the same branch of

the conditional. If this expression would have been evaluated with the `BSMLlib` library, we would have obtained an incoherent result and a crash of the BSP machine. The goal of our new semantics is to dynamically rejected this kind of problems (and to have an exception raised in the implementation).

3 Dynamic Semantics of BSML with Imperative Features

For the sake of conciseness, we limit our study to a subpart of the BSML language. In order to simplify the presentation and to ease the formal reasoning, this section introduces a core language. It is an attempt to trade between integrating the principal features of the BSML language with an imperative extension and being simple. This section introduces its syntax, its dynamic semantics together with some conventions, definitions and notations that are used in the paper.

3.1 Syntax

The expressions of mini-BSML, written e possibly with a prime or subscript, have the following abstract syntax:

$e ::=$	x		variable
	$\mid$	c	constant
	$\mid$	op	primitive operation
	$\mid$	**fun** $x \rightarrow e$	function abstraction
	$\mid$	$(e\ e)$	application
	$\mid$	**let** $x = e$ **in** e	local binding
	$\mid$	(e, e)	pair
	$\mid$	ℓ	locations

In this grammar, x ranges over a countable set of identifiers. The form $(e\ e')$ stands for the application of a function or an operator e, to an argument e'. The form **fun** $x \rightarrow e$ is the so-called and well-known lambda-abstraction that defines the function whose parameter is x and whose result is the value of e. Constants c are the integers 1, 2, the booleans and we assume having a unique value: () that have the type *unit*. This is the result type of affectation (like in Objective Caml). The set of primitive operations *op* contains arithmetic operations, fix-point operator **fix**, conditional, test function **isnc** of the **nc** constructor (which plays the role of the `None` constructor in Objective Caml), our parallel operations (**mkpar**, **apply**, **put**, **ifat**) and our store operation **ref**, **!** and **:=**. The construction $e_1 := e_2$ is an extended syntax for **:=**(e_1, e_2). We note $\mathcal{F}(e)$, the set of free variables of an expression e. **let** and **fun** are the binding operators and the free variables of a location is the empty set. It is defined by trivial structural induction on e.

Before presenting the dynamic semantics of the language, i.e., how the expressions of mini-BSML are computed to *values*, we present the values theirself. There is one semantics

per value of p, the number of processes of the parallel machine. In the following, $\forall i$ means $\forall i \in \{0, \ldots, p-1\}$. The values of mini-BSML are defined by the following grammar:

$v ::=$	**fun** $x \ \rightarrow e$	functional value
	$\mid c$	constant
	$\mid op$	primitive
	$\mid (v, v)$	pair value
	$\mid \langle v, \ldots, v \rangle$	p-wide parallel vector value
	$\mid \ell$	location

3.2 Rules

The dynamic semantics is defined by an evaluation mechanism that relates expressions to values. To express this relation, we used a small-step semantics. It consists of a predicate between expressions and another expression defined by a set of axioms and rules called steps. A step tells whether an expression evaluates to a given result. The small-step semantics describes all the steps of the calculus from an expression to a value.

Unlike in a sequential computer and a sequential language, an unique store is not sufficient. We need to express the store of all our processors. We assume a finite set $\mathcal{N} = \{0, \ldots, p-1\}$ which represents the set of processors names and we write i for these names and $\bowtie$ for all the network. Now, we can formalize the location and the store for each processor and for the network. We write s_i for the store of processor i with $i \in \mathcal{N}$. We assume that each processor has a store and a infinite set of addresses which are different at each processor (we could distinguish them by the name of the processor). We write $S = [s_0, \ldots, s_{p-1}]$ for the sequence of all the stores of our parallel machine. The imperative version of the small-steps semantics has the following form: $e/S \rightharpoonup e'/S'$. We will also write $e/s \rightharpoonup e'/s'$ when only one store of the parallel machine can be modified.

We note $\stackrel{*}{\rightharpoonup}$, for the transitive closure of $\rightharpoonup$ and note $e_0/S_0 \stackrel{*}{\rightharpoonup} v/S$ for $e_0/S_0 \rightharpoonup e_1/S_1 \rightharpoonup e_2/S_2 \rightharpoonup \ldots \rightharpoonup v/S$. We begin the reduction with a set of empty stores $\{\emptyset_0, \ldots, \emptyset_{p-1}\}$ noted $\emptyset_\bowtie$.

To define the relation $\rightharpoonup$, we begin with some axioms for two kinds of reductions:

1. $e/s_i \stackrel{i}{\rightharpoonup} e'/s'_i$ which could be read has "in the initial store s_i, at processor i, the expression e is reduced to e' in the store s'_i".

2. $e/S \stackrel{\bowtie}{\rightharpoonup} e'/S'$ which could be read has "in the initial network store S, the expression e is reduced to e' in the network store S'".

We write $s + \{\ell \mapsto v\}$ for the extension of s to the mapping of ℓ to v. If, before this operation, we have $\ell \in Dom(s)$, we can replace the range by the new value for the location ℓ.

To define these relations, we begin with some axioms for the relation of head reduction. We write $e_1[x \leftarrow e_1]$ the expression obtained by substituting all the free occurrences of x in

e_1 by e_2:
For a single processor:

$$(\textbf{fun } x \to e)\ v \ /\ s_i \quad \xrightarrow[i]{\varepsilon} \quad e[x \leftarrow v] \ /\ s_i \quad (\beta^i_{fun})$$
$$(\textbf{let } x = v \textbf{ in } e) \ /\ s_i \quad \xrightarrow[i]{\varepsilon} \quad e[x \leftarrow v] \ /\ s_i \quad (\beta^i_{let})$$

and for the whole parallel machine:

$$(\textbf{fun } x \to e)\ v \ /\ S \quad \xrightarrow[\bowtie]{\varepsilon} \quad e[x \leftarrow v] \ /\ S \quad (\beta^{\bowtie}_{fun})$$
$$(\textbf{let } x = v \textbf{ in } e) \ /\ S \quad \xrightarrow[\bowtie]{\varepsilon} \quad e[x \leftarrow v] \ /\ S \quad (\beta^{\bowtie}_{let})$$

For primitive operators we have some axioms, the δ-rules. For each classical δ-rule, we have two new reduction rules: $e \ / \ s_i \xrightarrow[\delta_i]{\varepsilon} e' \ / \ s'_i$ and $e \ / \ S \xrightarrow[\delta\bowtie]{\varepsilon} e' \ / \ S'$. Indeed, these reductions do not change the stores and do not depend on the stores. Those δ-rules are given in Figure 3.

Naturally, for the parallel operators, we have also some δ-rules but we do not have those δ-rules on a single processor but only for the network (Figure 4).

A problem appears with the **put** operator. The **put** operator is used for the exchange of values, in particular, locations. But a location could be seen as a pointer to the memory (a location is a memory's addresses). If we send a local allocation to a processor that do not has the location in its store, there is no reduce rule to apply and the program stop with an error (the famous *segmentation fault* of the C language) if it deferences this location (if it reads "out" of the memory). A dynamic solution is to communicate the value contained by the location and to create a new location for this value (as in the `Marshal` module of Objective Caml [13]). This solution implies the renaming of locations that are communicated to other processes. For this, we define Loc the set of location of a value. It is defined by trivial structural induction on the value. We define how to add a sequence of pair of location and value to a store with:

$$\begin{aligned} s + \emptyset &= s \\ s + [\ell_0 \mapsto v_0, \ldots, \ell_n \mapsto v_n] &= (s + \{\ell_0 \mapsto v_0\}) + \\ &\quad [\ell_1 \mapsto v_1, \ldots, \ell_n \mapsto v_n] \end{aligned}$$

We note $\varphi = \{\ell_0 \mapsto \ell'_0, \ldots \ell_n \mapsto \ell'_n\}$ for the substitution, i.e, a finite application from location ℓ_i to another location ℓ'_i with $\{\ell_0, \ldots \ell_n\}$ is the domain of φ.

Now we complete our semantics by giving the δ-rules of the operators on the stores and the references. We need two kinds of reductions. First for a single processor, δ-rules are (δ^i_{ref}), $(\delta^i_{!})$ and $(\delta^i_{:=})$ (given in figure 5). Those operations work on the store of the processor where this operation is executed. The **ref** operation creates a new allocation in the store of the processor, the **!** operation give the value contained in the location of the store and the **:=** operation change this value by another one.

For the whole network, we have to distinguish between the name of a location created outside a **mkpar** which is used in expressions and its "projections" in the stores of each process. We note $l^{\bowtie}$ in the first case and $l^{\bowtie}_i$ for its projection in the store of process i. When an expression outside a **mkpar** creates a new location, each process creates a new location (an address) on its store (rule $(\delta^{\bowtie}_{ref})$, figure 5). The affectation of location $l^{\bowtie}$ (rule $(\delta^{\bowtie}_{:=})$, figure 5) modifies the values of locations $l^{\bowtie}_i$.

For a single processor:

$$
\begin{array}{lll}
+(n_1, n_2) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & n \ / \ s_i \text{ with } n = n_1 + n_2 \qquad (\delta^i_+) \\
\mathbf{fst}(v_1, v_2) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & v_1 \ / \ s_i \qquad (\delta^i_{fst}) \\
\mathbf{snd}(v_1, v_2) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & v_2 \ / \ s_i \qquad (\delta^i_{snd}) \\
\mathbf{fix}(\mathbf{fun}\ x \rightarrow\ e) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & e[x \leftarrow \mathbf{fix}(\mathbf{fun}\ x \rightarrow\ e)] \ / \ s_i \qquad (\delta^i_{fix}) \\
\mathbf{fix}(\mathbf{op}) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & \mathbf{op} \ / \ s_i \\
\mathbf{ifthenelse}(\mathbf{true}, (e_1, e_2)) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & e_1 \ / \ s_i \qquad (\delta^i_{ifthenelseT}) \\
\mathbf{ifthenelse}(\mathbf{false}, (e_1, e_2)) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & e_2 \ / \ s_i \qquad (\delta^i_{ifthenelseF}) \\
\mathbf{isnc}(v) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & \mathbf{false} \ / \ s_i \text{ if } v \neq \mathbf{nc()} \qquad (\delta^i_{isnc}) \\
\mathbf{isnc}(\mathbf{nc()}) \ / \ s_i & \xrightarrow[\delta_i]{\varepsilon} & \mathbf{true} \ / \ s_i \qquad (\delta^i_{isnc})
\end{array}
$$

and for the whole parallel machine:

$$
\begin{array}{lll}
+(n_1, n_2) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & n \ / \ S \text{ with } n = n_1 + n_2 \qquad (\delta^\bowtie_+) \\
\mathbf{fst}(v_1, v_2) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & v_1 \ / \ S \qquad (\delta^\bowtie_{fst}) \\
\mathbf{snd}(v_1, v_2) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & v_2 \ / \ S \qquad (\delta^\bowtie_{snd}) \\
\mathbf{fix}(\mathbf{fun}\ x \rightarrow\ e) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & e[x \leftarrow \mathbf{fix}(\mathbf{fun}\ x \rightarrow\ e)] \ / \ S \qquad (\delta^\bowtie_{fix}) \\
\mathbf{fix}(\mathbf{op}) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & \mathbf{op} \ / \ S \\
\mathbf{ifthenelse}(\mathbf{true}, (e_1, e_2)) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & e_1 \ / \ S \qquad (\delta^\bowtie_{ifthenelseT}) \\
\mathbf{ifthenelse}(\mathbf{false}, (e_1, e_2)) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & e_2 \ / \ S \qquad (\delta^\bowtie_{ifthenelseF}) \\
\mathbf{isnc}(v) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & \mathbf{false} \ / \ S \text{ if } v \neq \mathbf{nc()} \qquad (\delta^\bowtie_{isnc}) \\
\mathbf{isnc}(\mathbf{nc()}) \ / \ S & \xrightarrow[\delta_\bowtie]{\varepsilon} & \mathbf{true} \ / \ S \qquad (\delta^\bowtie_{isnc})
\end{array}
$$

Figure 3: "Functional" δ-rules

$$\mathbf{mkpar}(\mathbf{fun}\ x \to\ e)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \langle e[x \leftarrow 0], \ldots, e[x \leftarrow (p-1)]\rangle\ /\ S \qquad (\delta^{\bowtie}_{mkpar})$$

$$\mathbf{apply}(\langle \mathbf{fun}\ x \to\ e_0, \ldots, \mathbf{fun}\ x \to\ e_{p-1}\rangle, \langle v_0, \ldots, v_{p-1}\rangle)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \langle e_0[x \leftarrow v_0], \ldots, e_{p-1}[x \leftarrow v_{p-1}]\rangle\ /\ S \qquad (\delta^{\bowtie}_{apply})$$

$$\mathbf{ifat}(v, (\langle \ldots, \overbrace{\mathbf{true}}^{n}, \ldots\rangle, (e_1, e_2)))\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} e_1\ /\ S \text{ if } v = n \qquad (\delta^{\bowtie}_{ifatT})$$

$$\mathbf{ifat}(v, (\langle \ldots, \overbrace{\mathbf{false}}^{n}, \ldots\rangle, (e_1, e_2)))\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} e_2\ /\ S \text{ if } v = n \qquad (\delta^{\bowtie}_{ifatF})$$

$$\mathbf{put}(\langle \mathbf{fun}\ dst \to e_0, \ldots, \mathbf{fun}\ dst \to e_{p-1}\rangle)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \langle r_0, \ldots, r_{p-1}\rangle\ /\ S' \qquad (\delta^{\bowtie}_{put})$$

where $S = [s_0, \ldots, s_{p-1}]$ and $S' = [s'_0, \ldots, s'_{p-1}]$
where $\forall j\ .\ s'_j = s_j + h'_0 + \ldots + h'_{p-1}$ where $h'j = [\ell'_0 \mapsto v_0, \ldots, \ell'_n \mapsto v_n]$ and
$h_j = \{(\ell_0, v_0), \ldots, (\ell_n, v_n)\}$ where $\ell_k \in \mathrm{Loc}(e_j)$ and $\{l_k \mapsto v_k\} \in s_j$ and
$\varphi_j = \{\ell_0 \mapsto \ell'_0, \ldots, \ell_n \mapsto \ell'_n\}$ and $e'_j = \varphi_j(e_j)$ and
$\forall i\ r_i = (\mathbf{let}\ v^i_0 = e'_0[dst \leftarrow i]\ \mathbf{in} \ldots v^i_{p-1} = e'_{p-1}[dst \leftarrow i]\ \mathbf{in} f_i)$ where
$f_i = \mathbf{fun}\ x \to \mathbf{if}\ x = 0\ \mathbf{then}\ v^i_0\ \mathbf{else} \ldots \mathbf{if}\ x = (p-1)\ \mathbf{then}\ v^i_{p-1}\ \mathbf{else}\ \mathbf{nc}()$ where

Figure 4: Parallel δ-rules

Local store δ-rules:

$$\mathbf{ref}(v)\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} \ell\ /\ s_i + \{\ell \mapsto v\} \quad \text{if } \ell \notin Dom(s_i) \qquad (\delta^i_{ref})$$

$$!(\ell)\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} s_i(\ell)\ /\ s_i \quad \text{if } \ell \in Dom(s_i) \qquad (\delta^i_{!})$$

$$:=(\ell, v)\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} ()\ /\ s_i + \{\ell \mapsto v\} \quad \text{if } \ell \in Dom(s_i) \qquad (\delta^i_{:=})$$

Global store δ-rules:

$$\mathbf{ref}(v)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} \ell^{\bowtie}\ /\ S' \qquad (\delta^{\bowtie}_{ref})$$
where $S' = [s_0 + \{\ell^{\bowtie}_0 \mapsto v\}, \ldots, s_{p-1} + \{\ell^{\bowtie}_{p-1} \mapsto v\}]$ and $\forall i.\ell^{\bowtie}_i \notin Dom(s_i)$

$$!(\ell^{\bowtie})\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} v\ /\ S \qquad (\delta^{\bowtie}_{!})$$
where $\exists v\ \forall i\ s_i(\ell^{\bowtie}_i) = v$ and $\ell^{\bowtie}_i \in Dom(s_i)$

$$:=(\ell^{\bowtie}, v)\ /\ S \xrightarrow[\delta_{\bowtie}]{\varepsilon} ()\ /\ S' \qquad (\delta^{\bowtie}_{:=})$$
where $S' = [s_0 + \{\ell^{\bowtie}_0 \mapsto v\}, \ldots, s_{p-1} + \{\ell^{\bowtie}_{p-1} \mapsto v\}]$ and $\forall i.\ell^{\bowtie}_i \in Dom(s_i)$

$$\ell^{\bowtie}\ /\ s_i \xrightarrow[\delta_i]{\varepsilon} \ell^{\bowtie}_i\ /\ s_i \qquad (\delta^{\bowtie}_{\mathrm{proj}})$$

Figure 5: "Store" δ-rules

This rule is only valid outside a **mkpar**. But a reference created outside a **mkpar** can be affected and deferenced inside an **mkpar**. For affectation, the value can be different on each process. To allow this, we need to introduce a rule ($\delta^{\bowtie}_{\text{proj}}$) (figure 5) which transforms (only inside an **mkpar** and at process i) the common name $l^{\bowtie}$ into its projection $l^{\bowtie}_i$. Notice that the affection or the deferencing of a location $l^{\bowtie}$ cannot be done inside a **mkpar** with rules ($\delta^i_{:=}$) and ($\delta^i_!$) since the condition $l \in Dom(s_i)$ does not hold. The use of the rule ($\delta^{\bowtie}_{\text{proj}}$) is first needed.

The deferencing of $l^{\bowtie}$ outside a **mkpar** can only occur if the values held by its projections at each process contain the same value. This verification is done by rule ($\delta^{\bowtie}_!$).

The complete definitions of our two kinds of reductions are:

$$\begin{aligned} \overset{i}{\rightharpoonup} &= \underset{i}{\overset{\varepsilon}{\rightharpoonup}} \cup \underset{\delta_i}{\overset{\varepsilon}{\rightharpoonup}} \\ \overset{\bowtie}{\rightharpoonup} &= \underset{\bowtie}{\overset{\varepsilon}{\rightharpoonup}} \cup \underset{\delta_{\bowtie}}{\overset{\varepsilon}{\rightharpoonup}} \end{aligned}$$

3.2.1 Contexts

It is easy to see that we cannot always make a head reduction. We have to reduce in depth in the sub-expression. To define this deep reduction, we define two kinds of contexts (an expression with a "hole" noted []) that have the following abstract syntax:

Γ	::=	[]	head evaluation
	|	$\Gamma\ e$	right application evaluation
	|	$v\ \Gamma$	left application evaluation
	|	**let** $x = \Gamma$ **in** e	let evaluation
	|	(Γ, e)	left pair evaluation
	|	(v, Γ)	right pair evaluation

Γ^j_l	::=	$\Gamma^j_l\ e$	
	|	$v\ \Gamma^j_l$	
	|	**let** $x = \Gamma^j_l$ **in** e	
	|	(Γ^j_l, e)	
	|	(v, Γ^j_l)	
	|	$\langle e, \ldots, \overbrace{\Gamma}^{j}, e, \ldots, e\rangle$	j^{th} component

Now we can reduce in depth in the sub-expression. To define this deep reduction, we use the following inference rules:

$$\frac{e\ /\ s_j \overset{j}{\rightharpoonup} e'\ /\ s'_j}{\Gamma^j_l(e)\ /\ S \rightharpoonup \Gamma^j_l(e')\ /\ S'} \qquad \text{(Local context rule)}$$

where $S = [s_0, \ldots, s_j, \ldots, s_{p-1}]$ and $S' = [s_0, \ldots, s'_j, \ldots, s_{p-1}]$

$$\frac{e\ /\ S \overset{\bowtie}{\rightharpoonup} e'\ /\ S'}{\Gamma(e)\ /\ S \rightharpoonup \Gamma(e')\ /\ S'} \qquad \text{(Global context rule)}$$

where $S = [s_0, \ldots, s_{p-1}]$ and $S' = [s'_0, \ldots, s'_{p-1}]$.

So we can reduce into the parallel vectors and the context gives the name of the processor where the expression is reduced. We can remark that the contexts give an order to evaluate an expression but not for the parallel vectors. It is not a problem because the BSλ-calculus is confluent [18]. We can also remark that our two kinds of contexts are discriminated each other by construction because the "hole" in a Γ^i_l context is always in a component of a parallel vector and never for a Γ one. Thus, we have a rule and its context to reduce global expressions and a rule to reduce usual expressions (in the parallel vectors).

3.3 Examples

3.3.1 Adding a natural in a parallel vector

We ran the following expression (see figure 6):

```
(1) let a=ref(1) in
(2)   mkpar (fun j ->
(3)     let b=ref(j) in  b:=!b+!a;!b );;
```

on a network with 2 processors ($\mathcal{N} = \{0, 1\}$).

$$
\begin{array}{lll}
 & \textbf{let } a = \textbf{ref}(1) \textbf{ in } \textbf{mkpar}(\textbf{fun } j \to \textbf{let } b = \textbf{ref}(j) \textbf{ in } b :=!b+!a;!b) & / \emptyset_{\bowtie} \\
\rightharpoonup & \textbf{let } a = \ell^{\bowtie} \textbf{ in } \textbf{mkpar}(\textbf{fun } j \to \textbf{let } b = \textbf{ref}(j) \textbf{ in } b :=!b+!a;!b) & / \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \textbf{mkpar}(\textbf{fun } j \to \textbf{let } b = \textbf{ref}(j) \textbf{ in } b :=!b+!\ell^{\bowtie};!b) & / \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle \textbf{let } b = \textbf{ref}(0) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle \textbf{let } b = \ell' \textbf{ in } b :=!b+!\ell^{\bowtie};!b \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle \ell' :=!\ell'+!\ell^{\bowtie};!\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle \ell' := 0+!\ell^{\bowtie};!\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle \ell' := 0+!\ell_0^{\bowtie};!\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle \ell' := 0+1;!\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle \ell' := 1;!\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle !\ell' \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle 1 \ , \ \textbf{let } b = \textbf{ref}(1) \textbf{ in } b :=!b+!\ell^{\bowtie};!b \rangle & / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \ldots & \\
\rightharpoonup & \langle 1, 2 \rangle \qquad / \{\{\ell_0^{\bowtie} \mapsto 1, \ell'_0 \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1, \ell'_1 \mapsto 2\}\} &
\end{array}
$$

Figure 6: Evaluation of an expression with an extended syntax

3.3.2 Global conditional

Considering the running of the expression of figure 2. We have one location per store initialized to 0 and after the creation of the parallel vector, we have that the stored value is different on each processor (equal to i on the i^{th} name processor). The evaluation of this expression is now stopped because the stored value is different in all the processor. In a `BSMLlib` library implementation of this semantics, an exception will be raised. The test of equality for the $(\delta^{\bowtie}_{!}$-rule has an additional parallel cost. It need at least a full super-step:

each processor send its value to the other processors (second and third phase) and test if they are all equal (beginning of first phase of the next super-step). It allows to do not have a crash of the machine, but the cost model is no longer compositional because an affectation outside the scope of a **mkpar** does not need communications and synchronization. We now present a simple modification of our semantics to have a compositional cost model.

3.4 Cost model preserving semantics

In order to avoid the comparison of the values held by projections of a $l^{\bowtie}$ location in rule $(\delta_!^{\bowtie})$ we can forbid the affectation of such a location inside a **mkpar**. This can be done by suppressing rule $(\delta_{\text{proj}}^{\bowtie})$. But in this case, deferencing inside a **mkpar** is not longer allowed. Thus we need to add a new rule:

$$!(\ell^{\bowtie}) \ / \ s_i \quad \underset{\delta_i}{\overset{\varepsilon}{\rightharpoonup}} \quad s_i(\ell_i^{\bowtie}) \ / \ s_i \quad \text{if } \ell_i^{\bowtie} \in Dom(s_i) \quad (\delta_!^{i\bowtie})$$

and modify the rule $(\delta_!^{\bowtie})$ to suppress the comparison:

$$!(\ell^{\bowtie}) \ / \ S \quad \underset{\delta\bowtie}{\overset{\varepsilon}{\rightharpoonup}} \quad s_i(l_i^{\bowtie}) \ / \ S \quad \text{if } \ell_i^{\bowtie} \in Dom(s_i) \quad (\delta'^{\bowtie}_!)$$

This rule is not deterministic but since affectation of a $l^{\bowtie}$ location is not allowed inside a **mkpar** the projections of a $l^{\bowtie}$ location always contain the same value and we have:

Theorem 1 *if* $e/S \overset{*}{\rightharpoonup} v_1/S_1$ *and* $e/S \overset{*}{\rightharpoonup} v_2/S_2$ *then* $v_1 = v_2$ *and* $S_1 = S_2$.

The calculus is not deterministic (parallel calculus came from the context rules) but is confluent. The cost model is now compositional since the new $(\delta'^{\bowtie}_!)$ does not need communications and synchronization. In the implementation an exception will be raised if the program tries to affect inside a **mkpar** a reference created outside a **mkpar**.

For example we ran the following expression (see Figure 7):

```
(1) let a=ref(0) in
(2)  a:=!a+1;
(3)  mkpar(fun i -> i+!a);;
```

on a network with 2 processors ($\mathcal{N} = \{0, 1\}$).

4 Conclusions and Future Work

The Bulk Synchronous Parallel ML allows direct mode Bulk Synchronous Parallel (BSP) programming. The semantics of BSML [18, 14, 15] were pure functional semantics. Nevertheless, the current implementation of BSML is the `BSMLlib` library [17] for the Objective Caml language which offers imperative features.

We have presented in this paper semantics of the interaction of our bulk synchronous operations with imperative features. The safe communication of references has been investigated, and for this particular point, the presented semantics is conform to the implementation. To

$$
\begin{array}{lll}
 & \textbf{let } a = \textbf{ref}(0) \textbf{ in } a{:=}!a + 1; \textbf{mkpar}(\textbf{fun } i \to i + !a) & / \quad \emptyset_{\bowtie} \\
\rightharpoonup & \textbf{let } a = \ell^{\bowtie} \textbf{ in } a{:=}!a + 1; \textbf{mkpar}(\textbf{fun } i \to i + !a) & / \quad \{\{\ell_0^{\bowtie} \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 0\}\} \\
\rightharpoonup & \ell^{\bowtie}{:=}!\ell^{\bowtie} + 1; \textbf{mkpar}(\textbf{fun } i \to i + !\ell^{\bowtie}) & / \quad \{\{\ell_0^{\bowtie} \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 0\}\} \\
\rightharpoonup & \ell^{\bowtie}{:=}0 + 1; \textbf{mkpar}(\textbf{fun } i \to i + !\ell^{\bowtie}) & / \quad \{\{\ell_0^{\bowtie} \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 0\}\} \\
\rightharpoonup & \ell^{\bowtie}{:=}1; \textbf{mkpar}(\textbf{fun } i \to i + !\ell^{\bowtie}) & / \quad \{\{\ell_0^{\bowtie} \mapsto 0\}, \{\ell_1^{\bowtie} \mapsto 0\}\} \\
\rightharpoonup & \textbf{mkpar}(\textbf{fun } i \to i + !\ell^{\bowtie}) & / \quad \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle 0 + !\ell^{\bowtie}, 1 + !\ell^{\bowtie} \rangle & / \quad \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle 0 + 1, 1 + !\ell^{\bowtie} \rangle & / \quad \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle 0 + 1, 1 + 1 \rangle & / \quad \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle 0 + 1, 2 \rangle & / \quad \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\} \\
\rightharpoonup & \langle 1, 2 \rangle & / \quad \{\{\ell_0^{\bowtie} \mapsto 1\}, \{\ell_1^{\bowtie} \mapsto 1\}\}
\end{array}
$$

Figure 7: Evaluation using the cost preserving semantics

ensure safety, communications may be needed in case of affectation (but in this case the cost model is no longer compositional) or references may contain additional information used dynamically to insure that dereferencing of references pointing to local value will give the same value on all processes.

We are currently working on the typing of effects [26] to avoid this problem statically and to a new implementation that follows the cost model preserving semantics.

Acknowledgments

This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAML.

References

[1] Arvind and R. Nikhil. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.

[2] O. Ballereau, F. Loulergue, and G. Hains. High-level BSP Programming: BSML and BSλ. In G. Michaelson and Ph. Trinder, editors, *Trends in Functional Programming*, pages 29–38. Intellect Books, 2000.

[3] G.-H. Botorog and H. Kuchen. Efficient high-level parallel programming. *Theoretical Computer Science*, 196:71–107, 1998.

[4] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[5] M. Cole. Bringing Skeletons out of the Closet. *submitted to Parallel Computing*, 2002.

[6] M. Danelutto, R. Di Cosmo, X. Leroy, and S. Pelagatti. Ocamlp3l a functional parallel programming system. Liens-98-1, ENS, 1998.

[7] J. Darlington, A. J. Field, P. G. Harrison, P. Kelly, D. Sharp, Q. Wu, and R. While. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, Munich, June 1993. Springer.

[8] C. Foisy and E. Chailloux. Caml Flight: a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.

[9] P. Fradet and J. Mallet. Compilation of a specialized functional language for massively parallel computers. *Journal of Functional Programming*, 10(6):561–605, 2000.

[10] G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, number 694 in LNCS, pages 56–67. Springer, 1993.

[11] Y. Hayashi. *Shaped-based Cost Analysis of Skeletal Parallel Programs.* PhD thesis, University of Edinburgh, 2001.

[12] H. Kuchen. A Skeleton Library. In *Europar 2002*, LNCS. Springer-Verlag, 2002.

[13] Xavier Leroy. The Objective Caml System 3.06, 2002. web pages at www.ocaml.org.

[14] F. Loulergue. BSλ_p: Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, number 1940 in Lecture Notes in Computer Science, pages 355–363. Springer, October 2000.

[15] F. Loulergue. Distributed Evaluation of Functional BSP Programs. *Parallel Processing Letters*, (4):423–437, 2001.

[16] F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.

[17] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14^{th} *IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.

[18] F. Loulergue, G. Hains, and C. Foisy. A Calculus of Functional BSP Programs. *Science of Computer Programming*, 37(1-3):253–277, 2000.

[19] W. F. McColl. Universal computing. In L. Bouge and al., editors, *Proc. Euro-Par '96*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[20] A. Merlin, G. Hains, and F. Loulergue. A SPMD Environment Machine for Functional BSP Programs. In *Proceedings of the Third Scottish Functional Programming Workshop*, august 2001.

[21] P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science. Springer, 1996.

[22] S. Pelagatti. *Structured Development of Parallel Programs.* Taylor & Francis, 1998.

[23] J. Schwarz and F.A. Rabhi. A skeleton-based implementation of iterative transformation algorithms using functional languages. In M. Kara et al., editor, *Abstract Machine Models for Parallel and Distributed Computing*, pages 119–133. IOS Press, April 1996.

[24] J. Serot. Embodying Parallel Functional Skeletons: An Experimental Implementation on Top of MPI. In C. Lengauer, M. Griebl, and S. Gorlatch, editors, *Euro-Par'97 Parallel Processing*, number 1300 in Lecture Notes in Computer Science, pages 629–633, Passau, Germany, August 1997. Springer.

[25] M. Snir and W. Gropp. *MPI the Complete Reference.* MIT Press, 1998.

[26] Jean-Pierre Talpin and Pierre Jouvelot. The Type and Effect Discipline. *Information and Computation*, 111(2):245–296, June 1994.

[27] Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103, August 1990.

[28] A. Zavanella. *Skeletons and BSP : Performance Portability for Parallel Programming.* PhD thesis, Universita degli studi di Pisa, 1999.