A Calculus of Functional BSP Programs with Projection

Frédéric Loulergue

Laboratoire d'Informatique Fondalementale d'Orléans (LIFO)
Université d'Orléans – France
frederic.loulergue@univ-orleans.fr

Abstract

Bulk Synchronous Parallel ML (BSML) is an extension of the functional language Objective Caml to program Bulk Synchronous Parallel (BSP) algorithms. It is deterministic, deadlock free and performances are good and predictable. Parallelism is expressed with a set of 4 primitives on a parallel data structure called parallel vector. These primitives are pure functional ones: they have no side-effect. It is thus possible, and we did it, to prove the correctness of BSML programs using a proof assistant like Coq. The BSλ-calculus is an extension of the λ-calculus which models the core semantics of BSML. Nevertheless some principles of BSML are not well captured by this calculus. This paper presents a new calculus, with a projection primitive, which provides a better model of the core semantics of BSML.

1. Introduction

Very often concurrent programming is used to program massively parallel algorithms. An imperative programming language is used with a message passing communication library such as MPI (*Message Passing Interface*). This approach is of course very general since it allows to define any parallel algorithm but also the details of its communications protocols. Nevertheless the freedom is not free and the development of such programs is very difficult because they may contain non-determinism and deadlocks. This is confirmed by the high complexity of related validation problems [3]. The semantics of a concurrent program being in general very complex, the time required to run it (related to its operational semantics) is also difficult to determine, which hinders the portability of performances.

The BSP model [24] (*Bulk Synchronous Parallelism*) aims at maximizing the portability of performances by adding a notion of explicit processes to data parallelism [22]. A BSP program is explicitly written for a static number of processors. The BSP execution model separates synchronization and communication and obliges both to be collective operations. It proposes a simple and accurate cost model (in this context, cost means the estimate of parallel execution time) making it possible to predict performances in a realistic and portable way. The theory of the proof of BSP programs [15] is it also close in complexity to the sequential case. The BSP model was used successfully for a broad variety of problems.

The previous approaches are however all imperative and do not allow the writing of parallel algorithms like functional programs. However the functional programming languages offers abstraction mechanisms such as higher order functions, polymorphism, pattern matching, which ease the writing of programs.

An intermediate approach is that of *algorithmic skeletons* [7, 9] in which only a finite set of operations are run in parallel. Their functional semantics is explicit but their parallel operational semantics is implicit. From the point of view of the programmer, the programming style is to use combiners which effect on parallelization is defined outside the formal semantics. The advantages compared to the concurrent extensions are of course that non-determinism and deadlocks are avoided. Compared to the I-structures and Caml Flight, a compositional semantics is preserved.

The skeletons are not in general any parallel operations, but try to capture the essence of well-known techniques of parallel programming such as parallel pipeline, master-slave algorithms, the application of a function to distributed collections, *etc.*

The designer of libraries of skeletons must find a balance between two opposite objectives:

- To provide the most complete and the most expressive possible set of skeletons. Indeed this set

1-4244-0054-6/06/$20.00

being finite, all the possible parallel programs cannot be written. Thus the designer should propose a set which is not too restricted and which is usable in practice.

- To provide the smallest possible set of skeletons because it is necessary: (a) on the one hand to efficiently implement each skeleton on each target parallel machine, (b) on the other hand that the user of the library of skeletons is not lost in a plethora of skeletons possible to use to solve his particular problem.

To help the user there are methodologies of transformation of programs which make it possible to replace a skeleton by a more efficient one or a composition of skeletons by only one skeleton. These methodologies rely on models of performances but often:

- either the model of performances is abstract thus applicable to any target architecture but is not realistic (such as for example the PRAM model);
- or the model of performances is too detailed and thus there is a different model of performances for each target architecture. The transformation made for an architecture could not be valid any more for a different architecture. This does not ease portability.

More recently a proposal [8] aims at integrating skeletons and more widespread practices of parallel programming such as MPI. This makes it possible to the user to program using MPI the algorithms which are not possible to implement using the provided skeletons. Of course the problems which MPI programming can bring, could appear. It seems in particular difficult to be able to give a simple and portable model of performances. Other work proposes simple and portable models of performances based on the BSP model [23, 13]. Recent work propose a set of skeletons based on Java for Grid computing [2] with a framework for performance prediction [1].

Our work is an approach similar to the intermediate position that the paradigm of the skeletons occupies. However the method was not to design an a priori finite set of operations then to design models of performances, but to set a priori a structured model of parallelism (with its model of performances) then to design a universal set of operations which allows the programming of any algorithm of this model. We chose the BSP model. On operational approach leads us to define an extension of the λ-calculus by BSP primitives [20]. Any BSP program can be written with this calculus, that is why it is said universal for the BSP model.

A library based on this calculus, the BSMLlib (we will also use BSML for "Bulk Synchronous Parallel ML") has been designed for the Objective Caml [16] (or Ocaml) language. It allows programming using a polymorphic parallel data structure. The programs are (sequential) functions written in Objective Caml but process a parallel data structure with dedicated primitives. BSML follows the BSP execution model so it is deadlock free. Performance predictability has been shown.

Being based on a confluent calculus it is deterministic. For a pure functional subset of Objective Caml, proofs of correctness of BSML programs [10] can be done using the Coq proof assistant. Nevertheless when we studied how non-functional features of Objective Caml were interacting with our parallel operations (for e.g. [11]), it appears that the BSλ-calculi do not capture the essence of BSML's behavior. Moreover the calculi contain a global conditional operation which cannot be implemented as is in BSML. On the other hand BSML contains a projection operation which cannot be safely introduced in the calculi without introducing nested parallelism which is not allowed [12].

In this paper we present a new calculus which is closer to BSML with a parallel projection but without nested parallelism. In section 2, an informal overview of BSML is given. In section 3 we give some elements of the previous BSλ-calculi in order to highlight the novelties of the BSλ_{at}-calculus with projection (section 4). Section 5 is devoted to examples and we end with related work 6 and conclusions 7.

2. Functional Bulk Synchronous Parallelism

A BSP computer contains a set of uniform *processor-memory* pairs, a *communication network* allowing inter-processor delivery of messages and a *global synchronization unit* which executes collective requests for a *synchronization barrier* (for the sake of conciseness, we refer to [24] for more details). A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjointed phases: (a) Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes; (b) the network delivers the requested data transfers; (c) a global synchronization barrier occurs, making the transferred data available for the next super-step.

The performance of the machine is characterized by 3 parameters expressed as multiples of the local processing speed s: p is the number of processor-memory pairs, L is the time required for a global synchroniza-

bsp_p: unit→ int
bsp_g: unit→ float
bsp_l: unit→ float

mkpar: (int → α) → α **par**
apply: (α → β) **par** → α **par** → β **par**
put: (int→ α option) **par** → (int→ α option) **par**
at: α **par** → int → α

Figure 1. Primitives

tion and g is the time for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation in time $g \times h$ for any arity h. The execution time of a super-step is thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time.

There is no full implementation of a Bulk Synchronous Parallel ML or BSML language but an implementation as a library for Objective Caml [16]. This library is based on the primitives given at figure 1. It is to notice that these primitives are not the same than the primitives of the first BSML proposal and implementation. They are closer to the semantics which in turn evolves to be closer to the implementation.

It gives access to the BSP parameters of the underling architecture. In particular, **bsp_p**() is p, the *static* number of processes. This value is constant during execution. **bsp_g**() and **bsp_l**() give the BSP parameters g and L.

BSML offers a parallel data structure called parallel vector. Its type is α **par**. It indicates than we have p values (one per processor) of type α. This type α should not be a type containing an occurrence of **par** (this point is discussed in details in [12]). BSML programs are written as usual Caml program but using the four parallel primitives for the creation and manipulation of parallel vectors. A BSML program is thus a kind of sequential program on a parallel data structure. This is very different from SPMD programming (Single Program Multiple Data) where the programmer must use a sequential language and a communication library (such as MPI). A parallel program is then the 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* and may contain *deadlocks*.

The asynchronous computation phase can be programmed with the two primitives **mkpar** and **apply**. Two successive calls to these primitives do not need any synchronization barrier.

The first one allows to create a parallel vector from a function. (**mkpar** f) will give the following parallel vector: $\boxed{v_0 \mid \cdots \mid v_i \mid \cdots \mid v_{p-1}}$ where at processor i, $(f\ i)$ is evaluated to v_i.

The second primitive applies a parallel vector of functions to a parallel vector of arguments:

$$(\textbf{apply}\ \boxed{f_0 \mid \cdots \mid f_i \mid \cdots \mid f_{p-1}}\ \boxed{v_0 \mid \cdots \mid v_i \mid \cdots \mid v_{p-1}}) = \boxed{v'_0 \mid \cdots \mid v'_i \mid \cdots \mid v'_{p-1}}$$

where at processor i, $(f_i\ v_i)$ is evaluated to v'_i.

The BSP cost of these two primitives is $\max_{0 \leq i < p} w_i$ where w_i is the time needed to evaluate at processor i the expression $(f\ i)$ for **mkpar** or the expression $(f_i\ v_i)$ for **apply**.

Unlike BSPlib [14] or PUB [5] we do not distinguish communication phase and synchronization barrier. The two primitives **put** and **at** are used to program the communication phase implicitly and immediately followed by a synchronization barrier.

put uses the type α option defined by: **type** α option = None | Some **of** α.

The argument of this primitive is a parallel vector of functions which describe the messages to be *sent*. The function f_i at processor i indicates for each destination processor j, either the data to be sent is $(f_i\ j) = \text{Some}\, v$ or the constant None which indicates that no data will be sent from i to j.

The result is also a parallel vector of functions which describe the *received* messages. The function g_j at processor j applied to i will give Some v if processor i sent the value v to processor j and will give None if i sent nothing to j.

The BSP cost for the evaluation of a **put** is the cost of a whole super-step. The messages to be sent are evaluated (asynchronous computation phase) then sent and the super-step is ended by a synchronization barrier.

The primitive **at** projects the value of a parallel vector from a given process:

$$(\textbf{at}\ \boxed{v_0 \mid \cdots \mid v_i \mid \cdots \mid v_{p-1}}\ \ i) = v_i$$

An expression containing this primitive should not be evaluated in the context of a **mkpar** primitive (this can be enforced by a type system [12]). The BSP cost of the evaluation of **at** is the cost of a direct broadcast (see below): $(p-1) \times s \times g + L$ where s is the size of value v_i.

The first BSλ-calculus contains a primitive called global conditional (see section 3). The global control

cannot depend on the value of a parallel vector at a given processor without this primitive. It is used to express algorithms like:

Repeat
Parallel Iteration
Until Max of local errors $< \epsilon$.

It is not possible to offer such a primitive as a function because Objective Caml is an eager language, but one can write: (**if** (**at** vec n) **then** e_1 **else** e_2). Furthermore, **at** can be used in other circumstances for example with pattern matching **match with**.

These four parallel functions are called *primitives* because they need lower-level libraries (C libraries wrapped to Objective Caml code) to be implemented. In the version of BSML under development the implementation of these functions rely on MPI, PUB [5], or the TCP/IP functions provided by the Unix module of Objective Caml.

Other functions are very often used when one write BSML programs. Nevertheless these functions can be defined using *only* the primitives and require no lower-level libraries. They are part of what is called the standard library of BSML. The following functions are used in the example at the end of this section:

```
let replicate x = mkpar(fun pid→ x)
let parfun f vv = apply (replicate f) vv
let noSome (Some x) = x
```

Consider the following BSML program:

```
(* val bcast: int →  α par →  α par *)
let bcast root vv =
  let tosend=mkpar(fun pid v dst→ if pid=root
                   then Some v else None) in
  let recv=put(apply tosend vec) in
  parfun noSome (apply recv (replicate root))
```

(bcast root vv) broadcast the value of the parallel vector vv at processor root to all other processors. The BSP cost of a call to this function is: $p + (p-1) \times s \times g + L$ where s is the size of the value vv at processor root.

3. BSλ-calculi

In this section we give an overview of the previous BSλ-calculi.

3.1. The BSλ-calculus

The syntax used here is not the one used in the cited publications, but is coherent with the name of the BSML primitives and with the syntax of the other semantics.

We refer to [4] for the definitions of free variables, substitutions, *etc.*, and for conventions.

We consider the disjoint sets $\dot{V}$ of *local* variables and $\mathcal{V}$ of *global* variables. Let $\dot{x}, \dot{y}, \ldots$ denote local variables and $X, Y, \ldots$ denote global variables from now on. x will denote a variable which can be either local or global. The same convention applies to terms.

The syntax of BSλ begins with *local* terms $\dot{e}$: λ-terms representing programs or values stored in a processor's local memory. The set $\mathcal{L}$ of local terms is given by the following grammar:

$$\dot{e} \quad ::= \quad \dot{x} \mid \dot{e}\,\dot{e} \mid \lambda\dot{x}.\,\dot{e}$$

where $\dot{x}$ denotes an arbitrary local variable.

Let p be the *finite* number of processor names $\mathcal{N}$, each being a closed λ-term in normal form. The BSλ-calculus is parametrized by this set. The elements of $\mathcal{N}$ will be always written $\dot{n}$ and variants. $|\cdot|$ is a bijection from the set $\mathcal{N}$ of processors names to the subset $\{0, 1, \ldots, p-1\}$ of naturals. $\dot{n}_0$ denotes any elements of $\mathcal{N}$ whereas $|0|^{-1}$ denotes the name of the processor 0.

Integers and booleans and the related operations can be coded in pure λ-calculus [4]. *For illustration* processors names will always be $0, \ldots, p-1$, coded for example as Church numeral (but one can also add new constants and δ-rules).

The set $\mathcal{G}$ of global terms is given by the following grammar:

$$\begin{array}{rcl} E & ::= & X \mid E\,E \mid E\,\dot{e} \mid \lambda X.\,E \mid \lambda\dot{x}.\,E \\ & \mid & (\mathbf{mkpar}\,\dot{e}) \mid (\mathbf{apply}\,E\,E) \mid (\mathbf{get}\,E\,E) \\ & \mid & (\mathbf{if}\,E\,\mathbf{at}\,\dot{e}\,\mathbf{then}\,E\,\mathbf{else}\,E) \end{array}$$

and has the following meaning.

Global terms denote maps from $\mathcal{N}$ to local values, functions between them or functions from local values to such maps. In particular the denotation of $(\mathbf{mkpar}\,\dot{e})$ has for value at processor $\dot{n}$ the value (normal form) of $(\dot{e}\ \dot{n})$. The forms $(\mathbf{apply}\,E_1\,E_2)$ and $(\mathbf{get}\,E_1\,E_2)$ are called point-wise parallel application and *get* respectively.

get represents the communication phase of a BSP super-step: a collective data exchange with a barrier synchronization. In $(\mathbf{get}\,E_1\,E_2)$, the resulting parallel vector contains values from E_1 taken at processor names defined in E_2. The exact meanings of **apply** and **get** are defined by the BSλ rules.

The last form of global terms define synchronous conditional expressions. The meaning of $(\mathbf{if}\,E_1\,\mathbf{at}\,\dot{e}\,\mathbf{then}\,E_2\,\mathbf{else}\,E_3)$ is that of E_2 (resp. E_3) if the parallel vector denoted by E_1 has value T (resp. F) at the processor name denoted by $\dot{e}$ T and F denotes the coding (or new constants) of the boolean values.

$$\frac{\forall \dot{m} \in \mathcal{N} \,.\, \dot{e}_2\, \dot{m} \;\rightarrow_{\mathrm{BS}\lambda}\; \dot{n} \in \mathcal{N}}{(\mathbf{get}\,(\mathbf{mkpar}\,\dot{e}_1)\,(\mathbf{mkpar}\,\dot{e}_2)) \;\rightarrow_{\mathrm{BS}\lambda}\; (\mathbf{mkpar}\,\lambda\dot{x}.\; \dot{e}_1(\dot{e}_2\;\dot{x}))} \tag{1}$$

$$(\mathbf{apply}\,(\mathbf{mkpar}\,\dot{e}_1)\,(\mathbf{mkpar}\,\dot{e}_2)) \;\rightarrow_{\mathrm{BS}\lambda}\; (\mathbf{mkpar}\,\lambda\dot{x}.\; (\dot{e}_1\;\dot{x})(\dot{e}_2\;\dot{x})) \tag{2}$$

$$\frac{\dot{e}\;\dot{e}' \;\rightarrow_{\mathrm{BS}\lambda}\; T \qquad \dot{e}' \in \mathcal{N}}{(\mathbf{if}\,(\mathbf{mkpar}\,E)\,\mathbf{at}\,\dot{e}'\,\mathbf{then}\,E_1\,\mathbf{else}\,E_2) \;\rightarrow_{\mathrm{BS}\lambda}\; E_1} \qquad \frac{\dot{e}\;\dot{e}' \;\rightarrow_{\mathrm{BS}\lambda}\; F \qquad \dot{e}' \in \mathcal{N}}{(\mathbf{if}\,(\mathbf{mkpar}\,E)\,\mathbf{at}\,\dot{e}'\,\mathbf{then}\,E_1\,\mathbf{else}\,E_2) \;\rightarrow_{\mathrm{BS}\lambda}\; E_2} \tag{3}$$

Figure 2. BSλ-calculus

get is not one of the BSML primitives of section 2. It is possible to have a **put** primitive for the BSλ-calculus. Nevertheless the principles of the calculus remains unchanged but with more complicated rules.

The one-step reduction of local terms is simply:

$$(\lambda\dot{x}.\;\dot{e})\dot{e}' \;\rightarrow_{\mathrm{BS}\lambda}\; \dot{e}[\dot{x} \leftarrow \dot{e}'] \tag{4}$$

it could be applied in any context (context rules are omitted, we refer to [20]).

The reduction of global terms is defined by syntax-directed rules and context rules which determine the applicability of the former. Each rule defines a reduction on global terms.

First, there are axioms for global beta-reduction:

$$(\lambda X.\; E)E' \quad \rightarrow_{\mathrm{BS}\lambda} \quad E[X \leftarrow E'] \tag{5}$$

$$(\lambda\dot{x}.\; E)\dot{e}' \quad \rightarrow_{\mathrm{BS}\lambda} \quad E[\dot{x} \leftarrow \dot{e}'] \tag{6}$$

As terms $(\lambda\dot{x}.E_1)\,E_2$ are well formed terms, but that substitution $E_1[\dot{x} \leftarrow E_2]$ is not part of the syntax, the two rules (5) and (6) are necessary.

There are also axioms for the interaction of the parallel vector constructor **mkpar** with the other BSP operations.

Rules (1) and (2) encode the meaning of the BSP operations on parallel vectors. In particular, **get** is functional composition inside the parallel vector construction. The value of $(\mathbf{get}\,(\mathbf{mkpar}\,\dot{e}_1)\,(\mathbf{mkpar}\,\dot{e}_2))$ at processor name $\dot{n}$ is $(\dot{e}_1\,(\dot{e}_2\,\dot{n}))$, i.e. the value of $(\mathbf{mkpar}\,\dot{e}_1)$ at processor name $(\dot{e}_2\,\dot{n})$.

Next, the global conditional is defined by two rules whose numerators refer to local computations. The two rules (3) generate the following bulk-synchronous computation: first a pure computation phase where all processors evaluate local term $\dot{e}'$ yielding value $\dot{n}$; then processor $\dot{n}$ evaluates $(\dot{e}\,\dot{n})$ into value $\dot{v}$; if $v = T$ (resp. F) then processor $\dot{n}$ broadcasts the order for global evaluation of E_1 (resp. E_2); otherwise the computation fails.

3.2. The BSλ_p-calculus

The parallel interpretation of a term $(\mathbf{mkpar}\,\dot{e})$ is:

$\dot{e}\,\lvert 0\rvert^{-1}$	$\cdots$	$\dot{e}\,\lvert i\rvert^{-1}$	$\cdots$	$\dot{e}\,\lvert p-1\rvert^{-1}$

and each of these p local terms can be reduced.

An intentional description $\dot{e}$ is replaced by p local terms. If we want to consider a proofs of equivalence of a given strategy of the BSλ-calculus with a semantics close to the implementation in SPMD style, we have to face the opposite transformation from p local terms to an intentional term $(\mathbf{mkpar}\,\dot{e})$ with the additional constraint that this transformation should be compatible with reduction. This is a problem. To avoid it we designed the BSλ_p-calculus [18].

The $(\mathbf{mkpar}\,\dot{e})$ construct is suppressed and replaced by a enumerated parallel vector construct:

$$\langle\; \dot{e}\;, \ldots,\; \dot{e}\;, \ldots,\; \dot{e}\;\rangle$$

It is always possible to write programs using the **mkpar** operation. But it is no more a primitive but a function defined by:

$$\mathbf{mkpar} \equiv \lambda\dot{e}.\langle\; \dot{e}\,\lvert 0\rvert^{-1}\;, \ldots,\; \dot{e}\,\lvert i\rvert^{-1}\;, \ldots,\; \dot{e}\,\lvert p-1\rvert^{-1}\;\rangle$$

Rules (4), (5) and (6) belongs to the BSλ_p-calculus and the other rules become simpler [18].

4. A New Calculus with Projection

The parallel interpretation of a reduction of the BSλ-calculus [20] gives informally a BSP cost to each reduction. This is done using an evaluation of intentional parallel vectors. The results looks partially like an ancestor of the BSλ_p-calculus.

For the BSλ_p-calculus the parallel interpretation is more direct [18]. Cost can be represented as a parallel vector of numbers and it is possible to associate to each rule of the calculus on global term a rule on cost vectors. The only problem concerns the contexts: for the local rule, it has an unitary cost in the context of a parallel vector but it adds one unit of time at each component of the cost vector in other cases. This phenomenon appears clearly with *distributed evaluation*.

$$
\begin{array}{rcll}
(\lambda \dot{x}.\dot{e}_1)\,\dot{e}_2 & \rightarrow_{\mathrm{at}} & \dot{e}_1[\dot{x} \leftarrow \dot{e}_2] & (7)\\
(\lambda \bar{x}.\bar{e}_1)\,\bar{e}_2 & \rightarrow_{\mathrm{at}} & \bar{e}_1[\bar{x} \leftarrow \bar{e}_2] & (8)\\
(\lambda \bar{x}.E_1)\,\bar{e}_2 & \rightarrow_{\mathrm{at}} & E_1[\bar{x} \leftarrow \bar{e}_2] & (9)\\
(\lambda X.E_1)\,E_2 & \rightarrow_{\mathrm{at}} & E_1[X \leftarrow E_2] & (10)\\
(\mathbf{at}\ \langle\ \dot{e}_0\ ,\ldots,\ \dot{e}_i\ ,\ldots,\ \dot{e}_{p-1}\ \rangle\ \bar{e}) & \rightarrow_{\mathrm{at}} & \mathcal{U}(\dot{e}_{|\mathcal{D}(\bar{e})|}) \text{ if } \forall i.\mathrm{FV}(\dot{e}_i) = \emptyset & (11)\\
(\mathbf{replicate}\ \bar{e}) & \rightarrow_{\mathrm{at}} & \langle\ \mathcal{D}(\bar{e})\ ,\ldots,\ \mathcal{D}(\bar{e})\ ,\ldots,\ \mathcal{D}(\bar{e})\ \rangle \text{ if } \mathrm{FV}(\bar{e}) = \emptyset & (12)\\
(\mathbf{apply}\ \langle\ \dot{e}^1_0\ ,\ldots,\ \dot{e}^1_{p-1}\ \rangle\ \langle\ \dot{e}^2_0\ ,\ldots,\ \dot{e}^2_{p-1}\ \rangle) & \rightarrow_{\mathrm{at}} & \langle\ \dot{e}^1_0\,\dot{e}^2_0\ ,\ldots,\ \dot{e}^1_i\,\dot{e}^2_i\ ,\ldots,\ \dot{e}^1_{p-1}\,\dot{e}^2_{p-1}\ \rangle & (13)\\
(\mathbf{get}\ \langle\ \dot{e}^1_0\ ,\ldots,\ \dot{e}^1_{p-1}\ \rangle\ \langle\ \dot{e}^2_0\ ,\ldots,\ \dot{e}^2_{p-1}\ \rangle) & \rightarrow_{\mathrm{at}} & \langle\ \dot{e}^1_{|\dot{e}^2_0|}\ ,\ldots,\ \dot{e}^1_{|\dot{e}^2_i|}\ ,\ldots,\ \dot{e}^1_{|\dot{e}^2_{p-1}|}\ \rangle & (14)
\end{array}
$$

Figure 3. The BSλ_{at}-calculus

The programming model corresponds to the BSλ-calculi: it is a kind of sequential composition of primitives on a parallel data structure. The execution model (of the implementation) corresponds to a parallel composition of sequential programs. Going from one model to the another is very important [6].

The distributed evaluation [19] offers an SPMD view of the evaluation of the terms obtained by a kind of compilation of BSλ_p terms. Starting from a BSλ_p term we obtain p terms by projection. Each term can be evaluated independently by one processor until a communication primitive has to be evaluated. To obtain these terms only one component (the same for one projection) of parallel vectors is preserved. The evaluation of the communication primitives can only be performed on p terms at the same time: there are synchronous operations.

The grammars are as follows: local terms remain unchanged, global terms are defined by a grammar similar to the grammar of the global terms of the BSλ_p-calculus, $\langle\ \dot{e}\ ,\ldots,\ \dot{e}\ \rangle$ being replaced by $\langle\dot{e}\rangle$. New terms called distributed terms are also used, they represent the whole SPMD program obtained by parallel composition of the p copies:

$$E_D ::= \langle\langle\ E\ ,\ldots,\ E\ ,\ldots,\ E\ \rangle\rangle \mid \langle\langle\ \dot{e}\ ,\ldots,\ \dot{e}\ ,\ldots,\ \dot{e}\ \rangle\rangle$$

For example the reduction,

$\langle\ \dot{e}_0\ ,\ldots,\ \dot{e}_i\ ,\ldots,\ \dot{e}_{p-1}\ \rangle \rightarrow_{\mathrm{BS}\lambda} \langle\ \dot{e}_0\ ,\ldots,\ \dot{e}'_i\ ,\ldots,\ \dot{e}_{p-1}\ \rangle$
becomes

$\langle\langle\ \langle\dot{e}_0\rangle\ ,\ldots,\ \langle\dot{e}_i\rangle\ ,\ldots,\ \langle\dot{e}_{p-1}\rangle\ \rangle\rangle$
$\rightarrow_{\mathrm{SD}}$ $\langle\langle\ \langle\dot{e}_0\rangle\ ,\ldots,\ \langle\dot{e}'_i\rangle\ ,\ldots,\ \langle\dot{e}_{p-1}\rangle\ \rangle\rangle$
in the distributed evaluation (unitary cost for one processor).

But reduction $E\,\dot{e} \rightarrow_{\mathrm{BS}\lambda} E\,e'$ becomes in the distributed evaluation:

$\langle\langle\ E\,\dot{e}\ ,\ldots,\ E\,\dot{e}\ ,\ldots,\ E\,\dot{e}\ \rangle\rangle$
$\rightarrow_{\mathrm{SD}}$ $\langle\langle\ E\,\dot{e}'\ ,\ldots,\ E\,\dot{e}\ ,\ldots,\ E\,\dot{e}\ \rangle\rangle \rightarrow_{\mathrm{SD}} \ldots$
$\rightarrow_{\mathrm{SD}}$ $\langle\langle\ E\,\dot{e}'\ ,\ldots,\ E\,\dot{e}'\ ,\ldots,\ E\,\dot{e}'\ \rangle\rangle$
(unitary cost for each processor).

One can observe that local terms are handled very differently depending on the context. This is a problem as soon as non-functional features are introduced [11].

Another problem is that until now the calculi propose the global conditional whereas the BSMLlib library proposes the primitive **at** which of course can be used to define a global conditional but which also allows to do other very useful things in practice.

All this makes us design a new calculus which avoid these problems. We consider now three kinds of terms:

- *local* term each processor: $\dot{e} ::= \dot{x} \mid \dot{e}\,\dot{e} \mid \lambda\dot{x}.\dot{e}$
- *replicated* terms which look like local terms if we think in terms of type *à la* ML but which are in fact copied on each processor of the parallel machine. They correspond of the second case presented above: $\bar{e} ::= \bar{x} \mid \bar{e}\,\bar{e} \mid \lambda\bar{x}.\bar{e} \mid (\mathbf{at}\ E\ \bar{e})$

  The last term allows to obtain a replicated term from a global term by taking the value of this global term at a given processor.
- *global* terms (note that the global conditional disappears):

$$\begin{array}{rl} E & ::= X \mid E\,E \mid E\,\bar{e} \mid \lambda X.E \mid \lambda\bar{x}.E\\ & \mid (\mathbf{replicate}\ \bar{e}) \mid (\mathbf{apply}\ E\ E)\\ & \mid \langle\ \dot{e}\ ,\ldots,\ \dot{e}\ ,\ldots,\ \dot{e}\ \rangle \mid (\mathbf{get}\ E\ E) \end{array}$$

Rules are given in figure 3 where e and variants denote either a local, replicated or global term and where:

$$\begin{cases} \mathcal{U}(\dot{x}) & = \bar{x}\\ \mathcal{U}(\dot{e}_1\,\dot{e}_2) & = \mathcal{U}(\dot{e}_1)\mathcal{U}(\dot{e}_2)\\ \mathcal{U}(\lambda\dot{x}.\dot{e}) & = \lambda\bar{x}.\mathcal{U}(\dot{e}) \end{cases} \qquad \begin{cases} \mathcal{D}(\bar{x}) & = \dot{x}\\ \mathcal{D}(\bar{e}_1\,\bar{e}_2) & = \mathcal{D}(\bar{e}_1)\,\mathcal{D}(\bar{e}_2)\\ \mathcal{D}(\lambda\bar{x}.\bar{e}) & = \lambda\dot{x}.\mathcal{D}(\bar{e}) \end{cases}$$

When the functions used in rules are undefined then the rule is considered to be inapplicable. Rules are applicable in any context using the contexts and contexts rules of figure 4 where $\Gamma_l[\dot{e}]$ is the term obtained by replacing $[]_l$ by the local term $\dot{e}$ in context Γ_l ; $\Gamma_r[e]$

is the term obtained by replacing $[]_r$ (resp. $[]_g$) by the replicated term (resp. global) e in context Γ_r ; $\Gamma_g[e]$ is the term obtained by replacing $[]_l$ (resp. $[]_g$) by the local term (resp. global) e in context Γ_g.

Theorem 1 (Confluence) *Let be e, e_1 and e_2 terms such as: $e \rightarrow^*_{at} e_1$ and $e \rightarrow^*_{at} e_2$. Then there exists a term e_3 such as: $e_1 \rightarrow^*_{at} e_3$ and $e_2 \rightarrow^*_{at} e_3$.*

5. Examples

Direct broadcast presented in section 2 can be written as the following BSλ_{at} term:

$$\textbf{bcast} \equiv \lambda\bar{r}.\lambda V.(\textbf{get}\ V\ (\textbf{replicate}\ \bar{r}))$$

We assume that :

- the set $\mathcal{N}$ is the set of the p first Church naturals ($p > 1$)
- $|\cdot|$ maps a Church natural to its corresponding natural
- $\bar{\mathbf{1}}$ is the replicated form of Church natural $\mathbf{1}$: $\mathcal{U}(\lambda\dot{f}.\lambda\dot{x}.(\dot{f}\,\dot{x}))$
- $\dot{e}_i$ are closed terms

A possible reduction is given in figure 5.

With the new primitive **at** it is also possible to write: $\textbf{bcast}' \equiv \lambda\bar{r}.\lambda V.(\textbf{replicate}\ (\textbf{at}\ V\ \bar{r}))$

The reduction is then:

$$\begin{array}{ll} & (\textbf{bcast}'\ \bar{\mathbf{1}}\ \langle\ \dot{e}_0\ ,\ldots,\ \dot{e}_{p-1}\ \rangle) \xrightarrow{(9)} \\ \xrightarrow{(10)} & (\textbf{replicate}\ (\textbf{at}\ \langle\ \dot{e}_0\ ,\ldots,\ \dot{e}_{p-1}\ \rangle\ (\lambda\bar{f}\,\bar{x}.(\bar{f}\,\bar{x})))) \\ & \text{as } |\mathcal{D}(\lambda\bar{f}\,\bar{x}.(\bar{f}\,\bar{x}))| = 1 \text{ we obtain,} \\ \xrightarrow{(11)} & (\textbf{replicate}\ \bar{e}_1) \text{ where } \bar{e}_1 = \mathcal{U}(\dot{e}_1) \\ \xrightarrow{(12)} & \langle\ \dot{e}_1\ ,\ldots,\ \dot{e}_1\ ,\ldots,\ \dot{e}_1\ \rangle \end{array}$$

As for the BSλ_p-calculus, the **mkpar** is not a primitive of the BSλ_{at}-calculus but it can be defined as a function :

$$\begin{array}{rcl} \textbf{this} & \equiv & \langle\ |0|^{-1}\ ,\ldots,\ |i|^{-1}\ ,\ldots,\ |p-1|^{-1}\ \rangle \\ \textbf{mkpar} & \equiv & \lambda\bar{f}.(\textbf{apply}\ (\textbf{replicate}\ \bar{f})\ \textbf{this}) \end{array}$$

This suggests that BSML could contain **replicate** and **this** as primitives *instead* of **mkpar** with **mkpar** implemented as: **let mkpar** f = **apply** (**replicate** f) **this**.

In this case **replicate** is particularly efficient since it is implemented as identity. this would also be more efficient but it is less interesting because it is not used very often in BSML programs. The next release of BSML will include this two primitives.

6. Related Work

BSλ-calculi adapt the notion of data fields [17] to direct mode BSP: parallel data structures are "flat" and correspond directly to processors. This difference could seem unimportant but it is not: the evaluation of BSML expressions does not need any flattening and the programmer has a total control over communications and synchronizations.

The first work on a formal semantics for BSP is [15] and there are a lot of other papers since then In all cases a set of algebraic rules for imperative sequential languages is extended with new constructs and rules for parallelism, like parallel composition and communications through shared variables. The solved problems are for example how to handle the different number of synchronization barriers or determinism problems.

More recently, BSPA [21] is a process algebra with which one can describe BSP programs, with a notion of enumerated parallel vector and global synchronization. It also preserves the point-to-point communications of CCS. This allows in particular to describe the coordination of several BSP machines. A BSP cost model is compatible with strong bissimulation.

7. Conclusions and Future Work

We have design a calculus of BSP functional programs with a projection primitive. The formalism has the same advantages as our previous work BSλ [20] and BSλ_p [18] but it is closer to the practical language, Bulk Synchronous Parallel ML, and the parallel interpretation is more direct by the introduction of a new class of terms called replicated terms.

The last formal step will be a distributed abstract machine that will be correct w.r.t this calculus (using an intermediate distributed evaluation). We will then obtain a complete formal basis for the design of a complete programming environment containing: a polymorphic strongly typed parallel functional language (BSML), tools for performance prediction, tools to help to prove programs correction and tools to derive programs, the derivation being driven by the costs. This is the program of the Propac project (http://wwwpropac.free.fr).

Acknowledgments. This work is supported by the Propac project funded by the French National Program for Young Researchers. I wish to thank the anonymous referees for their comments.

$$
\begin{array}{lcl}
\Gamma_l & ::= & []_l \mid \lambda\dot{x}.\Gamma_l \mid \Gamma_l\,\dot{e} \mid \dot{e}\,\Gamma_l \\
\Gamma_r & ::= & []_r \mid \lambda\bar{x}.\Gamma_r \mid \Gamma_r\,\bar{e} \mid \bar{e}\,\Gamma_r \mid (\mathbf{at}\ \Gamma_g\ \bar{e}) \mid (\mathbf{at}\ E\ \Gamma_r) \\
\Gamma_g & ::= & []_g \mid \lambda x.\Gamma_g \mid \Gamma_g\,e \mid E\,\Gamma_l \mid E\,\Gamma_g \mid (\mathbf{get}\ \Gamma_g\ E) \mid (\mathbf{get}\ E\ \Gamma_g) \mid (\mathbf{apply}\ \Gamma_g\ E) \mid (\mathbf{apply}\ E\ \Gamma_g) \\
 & \mid & \langle\, \Gamma_l\,, \ldots,\, \dot{e}\,, \ldots,\, \dot{e}\,\rangle \mid \ldots \mid \langle\, \dot{e}\,, \ldots,\, \Gamma_l\,, \ldots,\, \dot{e}\,\rangle \mid \ldots \mid \langle\, \dot{e}\,, \ldots,\, \dot{e}\,, \ldots,\, \Gamma_l\,\rangle
\end{array}
$$

$$
\frac{\dot{e} \rightarrow_{\mathrm{at}} \dot{e}'}{\Gamma_l[\dot{e}] \rightarrow_{\mathrm{at}} \Gamma_l[\dot{e}']} \qquad \frac{e \rightarrow_{\mathrm{at}} e'}{\Gamma_r[e] \rightarrow_{\mathrm{at}} \Gamma_r[e']} \qquad \frac{e \rightarrow_{\mathrm{at}} e'}{\Gamma_g[e] \rightarrow_{\mathrm{at}} \Gamma_g[e']} \tag{15}
$$

Figure 4. Contexts and context rules

$$
\begin{array}{ll}
 & (\mathbf{bcast}\ \bar{\mathbf{1}}\ \langle\, \dot{e}_0\,, \ldots,\, \dot{e}_{p-1}\,\rangle) \xrightarrow{(9)}\xrightarrow{(10)} (\mathbf{get}\ \langle\, \dot{e}_0\,, \ldots,\, \dot{e}_{p-1}\,\rangle\ (\mathbf{replicate}\ (\lambda\bar{f}.\lambda\bar{x}.(\bar{f}\,\bar{x})))) \\
\xrightarrow{(12)} & (\mathbf{get}\ \langle\, \dot{e}_0\,, \ldots,\, \dot{e}_{p-1}\,\rangle\ \langle\, \lambda\dot{f}.\lambda\dot{x}.(\dot{f}\,\dot{x})\,, \ldots,\, \lambda\dot{f}.\lambda\dot{x}.(\dot{f}\,\dot{x})\,, \ldots,\, \lambda\dot{f}.\lambda\dot{x}.(\dot{f}\,\dot{x})\,\rangle) \text{ as } |\lambda\dot{f}.\lambda\dot{x}.(\dot{f}\,\dot{x})| = 1 \\
\xrightarrow{(14)} & \langle\, \dot{e}_1\,, \ldots,\, \dot{e}_1\,, \ldots,\, \dot{e}_1\,\rangle
\end{array}
$$

Figure 5. Example: bcast

References

[1] M. Alt, H. Bischof, and S. Gorlatch. Program Development for Computational Grids Using Skeletons and Performance Prediction. *Parallel Processing Letters*, 12(3):157–174, 2002.

[2] M. Alt and S. Gorlatch. A prototype grid system using Java and RMI. In V. E. Malyshkin, editor, *PaCT*, LNCS 2763, pages 401–414, Springer, 2003.

[3] K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag, 2nd ed. edition, 1997.

[4] H. Barendregt. *The Lambda Calculus, Its Syntax and Semantics*. North-Holland, 1986.

[5] O. Bonorden, B. Juurlink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[6] L. Bougé. Le modèle de programmation à parallélisme de données: une perspective sémantique. *RAIRO Technique et Science Informatiques*, 12(5), 1993.

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

[8] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[9] J. Darlington et al. Parallel programming using skeleton functions. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, LNCS 694, Springer, 1993.

[10] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.

[11] F. Gava and F. Loulergue. Semantics of a Functional BSP Language with Imperative Features. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *Proceedings of the 10th ParCo Conference*, pages 95–102, North Holland, 2004.

[12] F. Gava and F. Loulergue. A Static Analysis for BSML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5):665–671, 2005.

[13] Y. Hayashi and M. Cole. BSP-based Cost Analysis of Skeletal Programs. In G. Michaelson, P. Trinder, and H.-W. Loidl, editors, *Trends in Functional Programming*, pages 20–28, Intellect, 2000.

[14] J. Hill, W. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.

[15] H. Jifeng, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, LNCS 1123–1124, Springer, 1996.

[16] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.09, 2005. web pages at www.ocaml.org.

[17] B. Lisper. Data parallelism and functional programming. In *École de Printemps sur le Data-Parallélisme*, Springer, 1996.

[18] F. Loulergue. BSλ_p: Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, LNCS 1940, pages 355–363, Springer, 2000.

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

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

[21] A. Merlin and G. Hains. A generic cost model for concurrent and data-parallel meta-computing. In *Draft proceedings of AVOCS'04*, 2004.

[22] J. Sipelsten and G. Blelloch. Collection-oriented languages. In *Proceedings IEEE*, volume 79, pages 504–23, 1991.

[23] D. B. Skillicorn, M. Danelutto, S. Pelagatti, and A. Zavanella. Optimising data-parallel programs using the BSP cost model. In *Europar'98*, LNCS 1470, pages 698–715, Springer, 1998.

[24] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.