Bulk Synchronous Parallel ML: Semantics and Implementation of the Parallel Juxtaposition

F. Loulergue[1], R. Benheddi[1], F. Gava[2], and D. Louis-Régis[1]

1: Laboratoire d'Informatique Fondamentale d'Orléans
Université d'Orléans, France
{floulerg,rbenhedd}@univ-orleans.fr

2: Laboratory of Algorithms, Complexity and Logic
University Paris XII, France
gava@univ-paris12.fr

1 Introduction

The design of parallel programs and parallel programming languages is a trade-off. On one hand the programs should be efficient. But the efficiency should not come at the price of non portability and unpredictability of performances. The portability of code is needed to allow code reuse on a wide variety of architectures and to allow the existence of legacy code. The predictability of performances is needed to guarantee that the efficiency will always be achieved, whatever is the used architecture.

Another very important characteristic of parallel programs is the complexity of their semantics. Deadlocks and indeterminism often hinder the practical use of parallelism by a large number of users. To avoid these undesirable properties, a trade-off has to be made between the expressiveness of the language and its structure which could decrease the expressiveness.

Bulk Synchronous Parallelism [22, 20] (BSP) is a model of computation which offers a high degree of abstraction like PRAM models but yet a realistic cost model based on a structured parallelism: deadlocks are avoided and indeterminism is limited to very specific cases in the BSPlib library [13]. BSP programs are portable across many parallel architectures.

Over the past decade, Bulk Synchronous Parallelism (and the Coarse-Grained Multicomputer or CGM which can be seen as a special case of the BSP model) have been used for a large variety of applications. It is to notice that "A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures, between the late eighties and the time from the mid nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain" [8].

Our research aims at combining the BSP model with functional programming. We obtained the Bulk Synchronous Parallel ML language (BSML) based on a *confluent* extension of the λ-calculus. Thus BSML is deadlock free and deterministic. Being a high-level language, programs are easier to write, to reuse and to compose. It is even possible to *certify* the correctness of BSML programs [9] with the help of the Coq proof assistant [2]. The performance prediction of BSML programs is

possible. BSML has been extended in many ways throughout the years and the papers related to this research are available at http://bsml.free.fr.

One direction for the extension of BSML was to offer new primitives for the programming of divide-and-conquer Bulk Synchronous Parallel algorithms. Two new primitives have been designed :

- the parallel superposition [17, 10] which creates two parallel threads whose communication and synchronization phases are fused ;
- the parallel juxtaposition [16] which divides the parallel machine in two independent sub-machines while preserving the Bulk Synchronous Parallel model.

[16] presents the programming model of BSML with juxtaposition. This model presents a global view to the programmer, easier to understand than what actually happens when a BSML program is run on a parallel machine. Nevertheless to implement BSML with juxtaposition we need a *distributed semantics* (section 3) which specifies the execution model i.e. what actually happens on a parallel machine. Using this specification we implemented (section 4) the juxtaposition using the parallel superposition and *imperative features*.

We begin the paper with an overview of BSML with juxtaposition (section 2). Related work and conclusions end the paper (sections 5 and 6).

2 Bulk Synchronous Parallel ML with Juxtaposition: an Overview

There is currently no implementation of a full BSML language but rather a partial implementation as a library for Objective Caml language [14, 6]. BSML follows the Bulk Synchronous Parallel (BSP) model which offers a model of architecture, a model of execution and a cost model.

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 [3] for more details). In this model, a parallel computation is divided in *super-steps*, at the end of which a the routing of the messages and barrier synchronization are performed. Hereafter all requests for data which have been posted during a preceding super-step are fulfilled.

The performance of the machine is characterized by 3 parameters expressed as multiples of the local processing speed r: p is the number of processor-memory pairs, l is the time required for a global synchronization 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. The execution time of a program is the sum of the execution time of its super-steps.

BSML does not rely on SPMD programming. Programs are usual "sequential" Objective Caml programs but work on a parallel data structure. Some of the advantages is a simpler semantics and a better readability: the execution order follows (or at least the results is such as the execution order seems to follow) the reading order.

The core of the BSMLlib library is based on the following elements:

bsp_p: unit→int
mkpar: (int →α) →α **par**
apply: (α →β) **par** →α **par** →β **par**
type α option = None | Some **of** α
put: (int→α option) **par** →(int→α option) **par**
proj: α option **par** →int →α option

It gives access to the BSP parameters of the underling architecture. In particular, **bsp_p**() is p, the *static* number of processes. There is an abstract polymorphic type α **par** which represents the type of p-wide parallel vectors of objects of type α one per process. The nesting of **par** types is prohibited. Our type system enforces this restriction [11].

The BSML parallel constructs operate on parallel vectors. Those parallel vectors are created by **mkpar** 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 and phases of global communication with global synchronization.

Asynchronous phases are programmed with **mkpar** and **apply**. The expression (**apply** (**mkpar** f) (**mkpar** e)) stores ((f i)(e i)) on process i.

Let consider the following expression:

let vf = **mkpar**(**fun** i→(+) i) **and** vv = **mkpar**(**fun** i→2∗i+1) **in**
apply vf vv

The two parallel vectors are respectively equivalent to:

fun x→x+0	**fun** x→x+1	$\cdots$	**fun** x→x+(**p**−1)

and

0	3	$\cdots$	$2 \times (p-1)+1$

The expression **apply** vf vv is then evaluated to:

0	4	$\cdots$	$2 \times (p-1)+2$

Readers familiar with BSPlib [13] 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**. Consider the expression:

$$\textbf{put}(\textbf{mkpar}(\textbf{fun}\ \text{i}\rightarrow \text{fs}_i)) \tag{1}$$

To send a value v from process j to process i, the function fs_j at process j must be such that (fs_j i) evaluates to Some v. To send no value from process j to process

i, (fs_j i) must evaluate to None. Expression (1) evaluates to a parallel vector containing a function fd_i of delivered messages on every process. At process i, (fd_i 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.

BSML also contains a synchronous projection operation **proj** whose detailed presentation is omitted here. It is necessary of express algorithms like:

Repeat Parallel Iteration **Until** Max of local errors $<$ ϵ

The projection should not be evaluated inside the scope of a **mkpar**. This is enforced by our type system [11].

To evaluate two parallel programs on the same machine, one can partition it into two sub-machines and evaluate each program independently on each partition. Nevertheless in this case the BSP cost model is lost since for example a global synchronization of each sub-machine would no more cost L. To preserve the BSP model, which is the best solution [12], synchronization barriers need to remain global for the whole machine. In a first definition of parallel composition [15], it was possible to compose two programs whose evaluations need the same number of super-steps. It is of course restrictive and the programmer was responsible to write programs which fulfill this constraint. A new version called *parallel juxtaposition* removes this constraint [16]. It is the version that we present in this paper.

Consider the expression (**juxta** m E_1 E_2). It means that the m first processors will evaluate E_1 and the others will evaluate E_2. From the point of view of E_1 the network will have m processors named $0, \ldots, m-1$. From the point of view of E_2 the network will have $p - m$ processors (where p is the number of processors of the current network) named $0, \ldots, (p - m - 1)$ (processor m is renamed 0, *etc.*). The value of **bsp_p**() is also changed. Otherwise the evaluation of the expressions is the same, on each sub-machine, than without parallel juxtaposition, but the evaluation of **put** and **at** need the whole machine for the global synchronization. A problem occurs when the evaluation of E_1 and the evaluation of E_2 need a different number of super-steps. That is why a new primitive is necessary. The **sync** primitive is a loop of synchronization barrier calls. It loops until a synchronization barrier call is made by **sync** on the whole machine.

In case of the evaluation of E_1 needs one more super-step than the evaluation of E_2, the evaluation of (**sync** (**juxta** m E_1 E_2)) can be described as follows:

- at the beginning, each synchronization barrier request for the evaluation of E_1 matches a synchronization barrier request for the evaluation of E_2;
- then the evaluation of E_2 ends. E_2 requests one more synchronization barrier for its last super-step. The second sub-machine has finished the evaluation of E_2 so it evaluates **sync**: the synchronization barrier request of **sync** will match the request of the first sub-machine;
- each sub-machine has finished the evaluation of its expressions and they both request a synchronization barrier from a **sync**. As this request concerns the whole machine the evaluation of **sync** ends.

Evaluation result of a parallel juxtaposition is a parallel vector:

$$(\textbf{juxta}\ m\ \langle\ v_0\ ,\ldots,\ v_{m-1}\ \rangle\ \langle\ v'_0\ ,\ldots,\ v'_{p-1-m}\ \rangle) = \langle v_0,\ldots,v_{m-1},v'_0,\ldots,v'_{p-1-m}\rangle$$

From the functional point of view, the **sync** function is identity.

In the BSML library, the fact that Objective Caml is a language with a weak call-by-value evaluation strategy must be taken into account. To avoid the evaluation of the two last arguments of the function **juxta** and the argument of the function **sync**, these arguments should be functions:

juxta: int →(unit →α **par**) →(unit →α **par**) →α **par**
sync: (unit→α **par**) →α **par**

The following example is a divide-and-conquer version of the scan program which is defined by scan $\oplus\ \langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle = \langle\ v_0\ ,\ldots,\ v_0 \oplus v_1 \oplus \ldots \oplus v_{p-1}\ \rangle$ where $\oplus$ is an associative binary operation.

```
let rec scan op vec =
 if bsp_p()=1 then vec
 else
  let mid=bsp_p()/2 in
  let vec'=juxta mid (fun ()→scan op vec) (fun ()→scan op vec) in
  let msg vec=apply (mkpar(fun i v→
   if i=mid−1
   then fun dst→ if dst>=mid then Some v else None
   else fun dst→ None)) vec
  and parop=parfun2(fun x y→match x with None→y|Some v→op v y)in
  parop (apply(put(msg vec'))(mkpar(fun i→mid−1))) vec'
```

The juxtaposition divides the network into two parts the scan is recursively applied to each part. The value held by the last processor of the first part is broadcast to all the processors of the second part, then this value and the value held locally are combined by the operator op on each processor of the second part.

To use this function at top-level, it must be put into a **sync** primitive:

(**sync** (**fun** () →scan (+) this))

3 Distributed Semantics

High level semantics corresponds to the programming model. Distributed semantics corresponds to the execution model. In the former, all the parallel operations seem synchronous, even those which do not need communication. In the latter, the operations without communication are asynchronous and the operations with communications are synchronous.

The distributed evaluation can be defined in two steps:

1. local reduction (performed by one process) ;
2. global reduction of distributed terms which allows the evaluation of communications.

3.1 Syntax

We consider here only a small subset of the Ocaml language with our parallel primitives. We first consider the *flat* part of the language, i.e. without parallel juxtaposition :

e	::=	x	(variable)
	|	c	(constant)
	|	**bsp_p**	BSP parameter p
	|	(**fun** $x \rightarrow e$)	(abstraction)
	|	op	(operator)
	|	(e e)	(application)
	|	(**let** $x = e$ **in** e)	(binding)
	|	(**if** e **then** e **else** e)	(conditional)
	|	(**mkpar** e)	(parallel vector)
	|	(**apply** e e)	(parallel application)
	|	(**get** e e)	(communication primitive)
	|	(**if** e **at** e **then** e **else** e)	(global conditional)
	|	$\langle e \rangle$	(enumerated parallel vector)
	|	(**sync** e')	(sync primitive)

The use of the juxtaposition is only allowed in the scope of a **sync** primitive :

$$e' ::= x \mid c \mid \textbf{bsp\_p} \mid (\textbf{fun}\ x \rightarrow e') \mid op \mid (e'\ e') \mid (\textbf{let}\ x = e'\ \textbf{in}\ e') \mid (\textbf{if}\ e'\ \textbf{then}\ e'\ \textbf{else}\ e') \mid (\textbf{mkpar}\ e') \mid (\textbf{apply}\ e'\ e') \mid (\textbf{get}\ e'\ e') \mid (\textbf{if}\ e'\ \textbf{at}\ e'\ \textbf{then}\ e'\ \textbf{else}\ e') \mid \langle e' \rangle \mid (\textbf{juxta}\ m\ e'\ e') \mid \|e'\|$$

For the sake of conciseness, we use the **get** and **if at** constructs instead of the more general **put** and **proj** functions. There is no fundamental differences, but the semantics is simpler. We also omit in the remaining of the paper to distinguish expressions e and e'. Most of the rules are valid for both. We also omit a simple type system (with explicit typing of variables with two possible annotations: local or global) which allows to avoid the nesting of parallel values.

The user is not supposed to write enumerated parallel vectors $\langle e \rangle$. These expressions are created during the evaluation of a **mkpar** expressions. $\|e'\|$ indicates that the expression e is a branch of a juxtaposition.

Values are given by the following grammar :

$$v ::= (\textbf{fun}\ x \rightarrow e) \mid c \mid op \mid \langle v \rangle$$

3.2 Local reduction

The distributed evaluation is an SPMD semantics. Each processor will evaluate one copy of the BSML program. As long as the expression is not an expression which requires communications, the evaluation can proceed asynchronously on each processor.

When the juxtaposition primitive is used two sub-machines are considered. For a given process it means that the process identifier and the number of processes can change. Nevertheless these values are constant for the actual parallel machine. Thus we choose to put these parameters on the arrow. $\longrightarrow_p^i$ is

the local reduction at processor whose absolute process identifier is i on a parallel machine of p processors. The absolute process identifier of the first process and number of processes of the sub-machine a process belong to are stored in two stacks : $\mathcal{E}^{\mathrm{f}}$ and $\mathcal{E}^{\mathrm{p}}$.

The location reduction is a relation on tuples of one expression, and two stacks. It is defined by the rules of figure 1 (the set of rules for predefined sequential operators is omitted[1]) plus the following contexts and context rule :

$$\begin{array}{l} \Gamma \ := [] \ \mid \ (\Gamma\ e) \ \mid \ (v\ \Gamma) \ \mid \ (\textbf{let}\ x = \Gamma\ \textbf{in}\ e) \ \mid \ (\textbf{if}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e) \\ \quad \mid \ (\textbf{mkpar}\ \Gamma) \ \mid \ (\textbf{apply}\ \Gamma\ e) \ \mid \ (\textbf{apply}\ v\ \Gamma) \ \mid \ (\textbf{get}\ \Gamma\ e) \ \mid \ (\textbf{get}\ v\ \Gamma) \\ \quad \mid \ (\textbf{if}\ e\ \textbf{at}\ \Gamma\ \textbf{then}\ e\ \textbf{else}\ e) \ \mid \ (\textbf{if}\ \Gamma\ \textbf{at}\ v\ \textbf{then}\ e\ \textbf{else}\ e) \\ \quad \mid \ (\textbf{juxta}\ \Gamma\ e\ e) \ \mid \ \langle\Gamma\rangle \ \mid \ \|\Gamma\| \ \mid \ (\textbf{sync}\ \Gamma) \end{array}$$

$$\frac{\left(\ e_1\ \mathcal{E}_1^{\mathrm{f}},\ \mathcal{E}_1^{\mathrm{p}}\ \right) \ \longrightarrow_p^i \ \left(\ e_2,\ \mathcal{E}_2^{\mathrm{f}},\ \mathcal{E}_2^{\mathrm{p}}\ \right)}{\left(\ \Gamma(e_1),\ \mathcal{E}_1^{\mathrm{f}},\ \mathcal{E}_1^{\mathrm{p}}\ \right) \ \longrightarrow_p^i \ \left(\ \Gamma(e_2),\ \mathcal{E}_2^{\mathrm{f}},\ \mathcal{E}_2^{\mathrm{p}}\ \right)} \tag{2}$$

The four first rules of figure 1 are usual rules of a functional language. Rule (7) returns the head of the stack of number of processors. For the stacks we use "::" for adding a value to the stack. The function h is defined by $\mathrm{h}(v :: \mathcal{E}) = v$. If the stack is empty, then if it is the $\mathcal{E}^{\mathrm{f}}$ then the hd function returns 0, if it is the $\mathcal{E}^{\mathrm{p}}$ stack the hd function return the value if p given by the $\longrightarrow_p^i$ arrow.

The two next rules formalize the informal semantics of the BSML primitives **mkpar** and **apply**, but as opposed as section 2, we consider here only what happens at process i. For example for rule (8), the processor i has (in the current sub-machine) the process identifier $i - \mathrm{h}(\mathcal{E}^{\mathrm{f}})$. Thus for it evaluating **mkpar** f is evaluating $(\mathrm{f}\ (i - \mathrm{h}(\mathcal{E}^{\mathrm{f}})))$.

The three last rules are devoted to the juxtaposition :

- the two first are used to choose which branch is evaluated by the given processor, depending on its identifier. New values of the process identifier of the first processor and the number of processor of the sub-machine are push on top of the respective stacks.
- the last one is used to restore the values of the process identifier of the first processor and the number of processor of the larger machine at the end of the evaluation of the branch.

3.3 Global reduction

The global reduction $\rightarrow$ concerns the whole parallel machine. A distributed expression is thus p tuples manipulated by the local reduction. We use the following syntax for distributed expressions : $\left\langle\ (e_0, \mathcal{E}_0^{\mathrm{f}}, \mathcal{E}_0^{\mathrm{p}})\ , \ldots,\ (e_{p-1}, \mathcal{E}_{p-1}^{\mathrm{f}}, \mathcal{E}_{p-1}^{\mathrm{p}})\ \right\rangle$

[1] this set includes rules for the **fix** operator used for recursion

$$\left(\ ((\textbf{fun}\ x \to e)\ v),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \left(\ e[x \leftarrow v],\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \tag{3}$$

$$\left(\ (\textbf{let}\ x = v\ \textbf{in}\ e),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \left(\ e[x \leftarrow v],\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \tag{4}$$

$$\left(\ (\textbf{if true then}\ e_1\ \textbf{else}\ e_2),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \left(\ e_1,\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \tag{5}$$

$$\left(\ (\textbf{if false then}\ e_1\ \textbf{else}\ e_2),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \left(\ e_1,\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \tag{6}$$

$$\left(\ \textbf{bsp\_p},\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \left(\ \mathrm{h}(\mathcal{E}^{\mathrm{p}}),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \tag{7}$$

$$\left(\ (\textbf{mkpar}\ v),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \left(\ (v\ (i - \mathrm{h}(\mathcal{E}^{\mathrm{f}}))),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \tag{8}$$

$$\left(\ (\textbf{apply}\ \langle v_1\rangle\ \langle v_2\rangle),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \left(\ \langle (v_1\ v_2)\rangle,\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \tag{9}$$

$$\left(\ (\textbf{juxta}\ m\ e_1\ e_2),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \begin{array}{l}\left(\ \|e_1\|,\ \mathrm{h}(\mathcal{E}^{\mathrm{f}}) :: \mathcal{E}^{\mathrm{f}},\ m :: \mathcal{E}^{\mathrm{p}}\ \right)\\ \quad \text{if } 0 \le m < \mathrm{h}(\mathcal{E}^{\mathrm{p}})\\ \quad \text{and } (i - \mathrm{h}(\mathcal{E}^{\mathrm{f}})) < m\end{array} \tag{10}$$

$$\left(\ (\textbf{juxta}\ m\ e_1\ e_2),\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \begin{array}{l}\left(\ \|e_2\|,\ (\mathrm{h}(\mathcal{E}^{\mathrm{f}})+m) :: \mathcal{E}^{\mathrm{f}},\ (\mathrm{h}(\mathcal{E}^{\mathrm{p}})-m) :: \mathcal{E}^{\mathrm{p}}\ \right)\\ \quad \text{if } 0 \le m < \mathrm{h}(\mathcal{E}^{\mathrm{p}})\\ \quad \text{and } (i - \mathrm{h}(\mathcal{E}^{\mathrm{f}})) \ge m\end{array} \tag{11}$$

$$\left(\ \|v\|,\ f :: \mathcal{E}^{\mathrm{f}},\ p' :: \mathcal{E}^{\mathrm{p}}\ \right) \longrightarrow_p^i \left(\ v,\ \mathcal{E}^{\mathrm{f}},\ \mathcal{E}^{\mathrm{p}}\ \right) \tag{12}$$

Fig. 1. Local reduction

The first rule takes into account the local reduction :

$$\frac{(e_i, \mathcal{E}^{\mathrm{f}}, \mathcal{E}^{\mathrm{p}}) \ \longrightarrow_p^i \ (e'_i, \mathcal{E'}^{f}_{\ i}, \mathcal{E'}^{p}_{\ i})}{\left\langle \ldots,\ (e_i, \mathcal{E}^{\mathrm{f}}_i, \mathcal{E}^{\mathrm{p}}_i), \ldots \right\rangle \to \left\langle \ldots,\ (e'_i, \mathcal{E'}^{f}_{\ i}, \mathcal{E'}^{p}_{\ i}), \ldots \right\rangle} \tag{13}$$

The second one is used for communications and synchronisation. The p processors are partitioned into $1 \le k \le p$ parts, each part containing one of more successive processor. Two processors belongs to the same part n ($1 \le n \le k$) if the values at the top of their $\mathcal{E}^{\mathrm{f}}$ stacks are equal. In this case they also have the same value on top of $\mathcal{E}^{\mathrm{p}}$ which we note p_n.

We note $(e_{n,i}, \mathcal{E}^{\mathrm{f}}_{n,i}, \mathcal{E}^{\mathrm{p}}_{n,i})$ the process i *in the* n^{th} *part*. This processor has process identifier $i + \mathrm{h}(\mathcal{E}^{\mathrm{f}})$ in the whole parallel machine. We want the reduction :

$$\begin{array}{c}\left\langle \begin{array}{l}(e_{1,0}, \mathcal{E}^{\mathrm{f}}_{1,0}, \mathcal{E}^{\mathrm{p}}_{1,0}), \ldots, (e_{1,p_1-1}, \mathcal{E}^{\mathrm{f}}_{1,p_1-1}, \mathcal{E}^{\mathrm{p}}_{1,p_1-1}),\\ (e_{2,0}, \mathcal{E}^{\mathrm{f}}_{2,0}, \mathcal{E}^{\mathrm{p}}_{2,0}), \ldots, (e_{k,p_k-1}, \mathcal{E}^{\mathrm{f}}_{k,p_k-1}, \mathcal{E}^{\mathrm{p}}_{k,p_k-1})\end{array} \right\rangle\\ \to\\ \left\langle \begin{array}{l}(e'_{1,0}, \mathcal{E'}^{f}_{1,0}, \mathcal{E'}^{p}_{1,0}), \ldots, (e'_{1,p_0-1}, \mathcal{E'}^{f}_{1,p_0-1}, \mathcal{E'}^{p}_{1,p_0-1}),\\ (e'_{2,0}, \mathcal{E'}^{f}_{2,0}, \mathcal{E'}^{p}_{2,0}), \ldots, (e'_{k,p_k-1}, \mathcal{E'}^{f}_{k,p_k-1}, \mathcal{E'}^{p}_{k,p_k-1})\end{array} \right\rangle\end{array}$$

We have either :

– all processors are evaluating a **sync**, i.e. $\forall n.\forall i.(1 \le n \le k)\&(0 \le i < p_n) \Rightarrow e_{n,i} = \Gamma(\textbf{sync}\ v_{n,i})$ then $\forall n.\forall i.(1 \le n \le k)\&(0 \le i < p_n) \Rightarrow e'_{n,i} = \Gamma(v_{n,i})$

– at least one part is evaluating a primitive of communication. For each part we have to evaluate the corresponding primitive. For all n such that $1 \leq n \leq k$, we have either :

get: $\begin{cases} \text{we have} & \forall i.0 \leq i < p_n \Rightarrow e_{n,i} = \Gamma(\textbf{get}\ \langle v_i \rangle\ \langle n_i \rangle) \\ \text{then} & \forall i.0 \leq i < p_n \Rightarrow e'_{n,i} = \Gamma(\langle v_{n_i} \rangle) \end{cases}$

if at: we have $\forall i.0 \leq i < p_n \Rightarrow e_{n,i} = \Gamma(\textbf{if}\ \langle v_i \rangle\ \textbf{at}\ \langle m \rangle\ \textbf{then}\ e_i^1\ \textbf{else}\ e_i^2)$ then:

- if $0 \leq m < p_n$ and $v_m = \textbf{true}$ then $\forall i.0 \leq i < p_n \Rightarrow e'_{n,i} = \Gamma(e_i^1)$.
- if $0 \leq m < p_n$ and $v_m = \textbf{false}$ then $\forall i.0 \leq i < p_n \Rightarrow e'_{n,i} = \Gamma(e_i^2)$.

sync: $\begin{cases} \text{we have} & \forall i.0 \leq i < p_n \Rightarrow e_{n,i} = \Gamma(\textbf{sync}\ v_i) \\ \text{then} & \forall i.0 \leq i < p_n \Rightarrow e'_{n,i} = \Gamma(\textbf{sync}\ v_i) \end{cases}$

Theorem 1. $\rightarrow$ *is confluent.*

4 Implementation

The semantics given in the previous section is a specification for a distributed SPMD implementation. The current implementation of BSML is modular [18]. The module of primitives is a function which takes as argument a module of lower-level communications. Several such modules are provided and built on top of MPI, PVM, PUB, and also directly TCP/IP.

There are two implementations of the parallel juxtaposition. One which needs to extend the lower-level module interface. This solution adds very little sequential computation overhead. The drawback is that each lower-level module should be modified.

The second one implements the juxtaposition using the superposition. The advantage is that the implementation of the superposition is independent from the lower-level module. Moreover the implementation of the juxtaposition using the superposition is quite simple. and it is moreover very close to the semantics. This implementation adds more useless sequential computations, but there is no need for the **sync** additional synchronization barrier.

The superposition [16] allows two parallel expressions to be concurrently evaluated by two parallel threads running on the whole parallel machine. The results is a pair of parallel vectors of size p. The outline of the implementation of **juxta** m e1 e2 is as follows, where two stacks – one for the current number of processors p of the current machine and one for the number (in the previous machine) of the first processor f of the current machine – are used. The implementation o evaluate :

1. check if m is not greater than the number of processors
2. push the current values of f and p on the stacks
3. superpose the following expressions:
   (a) set p to m and evaluate e1()
   (b) set f to $m + f$ and p to $p - m$ and evaluate e2()
   the pair (va,vb) of parallel vectors is obtained

4. restore the values of f and p by popping them from the stacks
5. return a parallel vector in which:
   - the m first values come from va (from index f to $f + m - 1$)
   - the $p - m$ next values come from vb (from index $f + m$ to $f + p - 1$)

We did some experiments with the scan program shown in section 2. We run it on parallel vectors of polynomial, the operation being the addition. The tests were done on a 10 nodes Pentium IV cluster with Giga-bit Ethernet network (figure 2).

We compared the version with juxtaposition which requires $\log p$ super-steps with a direct version without superposition in 1 super-step. In the latter case, p polynomials are received by the last processor. In the former case, at each step, a processor receives at most 2 polynomial. As the polynomial are quite big, the $\log p$ version performs better than the direct version. Of course for smaller polynomials the direct version is better. It also depends on the BSP parameters of the parallel machine.

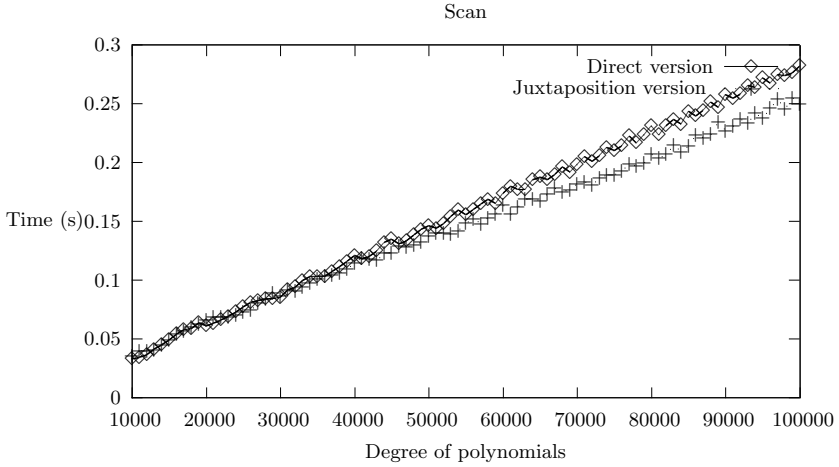

Fig. 2. Experiments with the scan programs

5 Related work

[21] presents another way to divide-and-conquer in the framework of an object-oriented language. There is no formal semantics and no implementation from now on. The proposed operation is similar to the parallel *superposition*, several BSP threads use the whole network. The same author advocates in [19] a new extension of the BSP model in order to ease the programming of divide-and-conquer BSP algorithms. It adds another level to the BSP model with new parameters to describe the parallel machine.

[23] is an algorithmic skeletons language based on the BSP model and offers divide-and-conquer skeletons. Nevertheless, the cost model is not really the BSP model but the D-BSP model [7] which allows subset synchronization. We follow [12] to reject such a possibility.

In the BSPlib library [13] subset synchronization is not allowed as explained in [20]. The PUB library [4] is another implementation of the BSPlib standard proposal. It offers additional features with respect to the standard which follows the BSP* model [1] and the D-BSP model [7]. Minimum spanning trees nested BSP algorithms [5] have been implemented using these features.

6 Conclusion and Future Work

We have presented a distributed semantics – which formalizes the execution model – and an implementation of the parallel juxtaposition primitive for Bulk Synchronous Parallel ML. This primitive allows to write parallel divide-and-conquer BSP algorithms.

The programming model, and its formalization, of BSML with juxtaposition has been presented in a previous paper [16]. We need now to prove that the programming model and the execution model are equivalent i.e. that their formalizations are equivalent semantics.

The parallel juxtaposition is in fact an imperative extension of BSML. It has for example a side effect on the number **bsp_p**() of processors of the current parallel machine. Thus the method presented in [9] used to prove the correctness of BSML programs with the Coq proof assistant cannot be used. Another direction of research is thus to provide a transformation from a program with parallel juxtaposition to an *equivalent pure functional* program without parallel juxtaposition. The equivalence is in this case a semantic equivalence, the performance of the two programs being different. This transformation should also be proved correct. The correctness of the original program can then be ensured by proving, using Coq, the correctness of the transformed pure functional program.

Acknowledgments The authors wish to thank the anonymous referee for their comments. This work is supported by the "ACI Jeunes chercheurs" program from the French ministry of research under the project "Programmation parallèle certifiée" PROPAC (http://wwwpropac.free.fr).

References

1. W. Bäumker, A. adn Dittrich and F. Meyer auf der Heide. Truly efficient parallel algorithms: c-optimal multisearch for an extension of the BSP model. In 3^{rd} *European Symposium on Algorithms (ESA)*, pages 17–30, 1995.
2. Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
3. R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

4. O. Bonorden, B. Juurlink, I. von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.
5. O. Bonorden, F. Meyer auf der Heide, and R. Wanka. Composition of Efficient Nested BSP Algorithms: Minimum Spanning Tree Computation as an Instructive Example. In *Proceedings of PDPTA*, 2002.
6. E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly France, 2000.
7. P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In L. Bougé et al., eds., *Euro-Par'96*, LNCS 1123–1124, Springer, 1996.
8. F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999.
9. F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.
10. F. Gava. *Approches fonctionnelles de la programmation parallèle et des méta-ordinateurs. Sémantiques, implantations et certification.* PhD thesis, University Paris Val-de-Marne, LACL, 2005.
11. F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5):665–671, 2005.
12. G. Hains. Subset synchronization in BSP computing. In H.R.Arabnia, ed., *Proceedings of PDPTA*, vol. I, pages 242–246, CSREA Press, 1998.
13. J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.
14. 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.
15. F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, ed., *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.
16. F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In H. Kosch et al., eds., *Euro-Par 2003*, LNCS 2790, pages 781–788, Springer, 2003.
17. F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot et al., eds., *Proceedings of ICCS 2003, Part I*, LNCS 2659, pages 223–232, Springer, 2003.
18. F. Loulergue, F. Gava, and D. Billiet. BSML: Modular Implementation and Performance Prediction. In Vaidy S. Sunderam et al., eds., *Proceedings of ICCS 2005, Part II*, LNCS 3515, pages 1046–1054, Springer, 2005.
19. J. M. R. Martin and A. Tiskin. BSP modelling a two-tiered parallel architectures. In B. M. Cook, ed., *WoTUG'99*, pages 47–55, 1999.
20. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
21. A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, (4), 2001.
22. Leslie G Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.
23. A. Zavanella. *Skeletons and BSP : Performance Portability for Parallel Programming.* PhD thesis, Universita degli studi di Pisa, 1999.