Minimally Synchronous Parallel ML with Parallel Composition

Radia Benheddi and Frédéric Loulergue

University of Orléans
Laboratoire d'Informatique Fondamentale d'Orléans,
Bâtiment IIIA, rue Léonard de Vinci, B.P. 6759
F-45067 Orleans cedex 2, France
{benheddi,floulerg}@univ-orleans.fr

Abstract

This paper presents the semantics of a new primitive for parallel composition in a parallel functional language called *Minimally Synchronous Parallel ML* (MSPML). MSPML is a parallel functional language based on a small number of primitives on a parallel data structure. The programs are written like usual ML programs by using this small set of functions. MSPML is deterministic and without deadlock. The execution time of the programs can be estimated. It has an asynchronous semantics, i.e. without global synchronisation barrier. The new parallel composition primitive allows to write parallel divide-and-conquer algorithms which are common in the literature.

Keywords
Parallelism, parallel composition, semantics, execution model.

1 INTRODUCTION

Very often concurrent programming is used to write massively parallel algorithms by combining a sequential programming language (usually C or Fortran) with a message passing communication library such as MPI [34] or PVM [20]. This approach is indeed extremely general because it allows to define both the parallel algorithm and all the details of its realisation by communication protocols. Nevertheless the development of such programs is difficult because of non-determinism and deadlocks. This difficulty is confirmed by the high complexity of the associated validation problems [1]. As the semantics of a concurrent program is in general complicated, its execution time (related to the operational semantics) can hardly be predicted. This can hinder the portability of performances.

We developed libraries for the Objective Caml [24, 15] language to ease the writing, performance prediction and validation of parallel programs. BSML or Bulk Synchronous Parallel ML [27], whose semantics is given by a confluent extension of the λ-calculus [2, 28], allows to implement Bulk Synchronous Parallel (BSP) [37, 33, 9] algorithms using a small set of primitives on a polymorphic data-structure. The complexity of BSML programs proofs [18] is identical to that of the sequential case [8].

For efficiency reasons, it is not possible to use BSML to program meta-computers (clusters of parallel machines). We designed a two-tiered model and language called Departmental Meta-computing ML or DMML [19]: each node is seen as a BSP machine and programmed using BSML and an additional level, minimally synchronous, is used for the coordination. The concepts introduced for this level can also be used independently in a parallel language. We call it Minimally Synchronous Parallel ML or MSPML. The primitives of MSPML are identical or similar to BSML ones but the execution model of the two languages are different. In BSML communications are followed by a synchronisation barrier involving all the processors of the parallel machine. In MSPML synchronisations occur only among processors involved in exchanges of data.

MSPML is currently *flat*: it is not possible to divide the machine and use independently two subsets of the processors to evaluate two expressions. There are many parallel algorithms in the literature which are parallel divide-and-conquer algorithms. It is possible to flatten and implement them in MSPML but the flatten programs are far less readable and more difficult to design than the original parallel recursive programs.

In this paper we present the semantics of a new primitive for parallel composition in MSPML, called juxtaposition, which allows to directly implement divide-and-conquer parallel algorithms. In section 2 we present informally flat MSPML and the juxtaposition primitive as well as the underlying communication mechanism. In section 3 we present a semantics which formalise the execution model of MSPML with parallel composition. We end with related work (section 4), conclusions and future work (section 5).

2 MSPML: INFORMAL PRESENTATION

In this section, a brief presentation of the MPM model and the core MSPML language without parallel composition juxtaposition is given. Then the new parallel composition primitive and the underlying mechanisms are presented.

2.1 The Message Passing Machine Model

The Message Passing Machine or MPM model [32] proposes an execution and cost model for programs run on distributed memory parallel machines. A MPM program is a sequence of *m-steps*. At each m-step, each processor performs a computation phase followed by a communication phase. During the communication phase, the processors concerned by the communication exchange the data the need for the next m-step. There is no synchronisation barriers but only synchronisation between processors which exchange data. For a given processor i and a m-step s, $\Omega_{s,i}$ is the set containing i and the processors, called incoming partners, which send messages to processor i.

The parallel machine is characterised by the following parameters: p the number of processors, g the communication gap, and L the network latency.

The execution time of a MPM program is bounded by :

$$\Psi = \max\{\Phi_{R,j} | j \in \{0,1,\ldots,p-1\}\}$$

$\Phi_{s,i}$ is inductively defined by :

$$\begin{cases} \Phi_{1,i} = \max\{w_{1,j} | j \in \Omega_{1,i}\} + (g \times h_{1,i} + L) \\ \Phi_{s,i} = \max\{\Phi_{s-1,j} + w_{s-1,j} | j \in \Omega_{s,i}\} + (g \times h_{s,i} + L) \end{cases}$$

where $h_{s,i} = \max\{h^+_{s,i}, h^-_{s,i}\}$ for $i \in \{0,\ldots,p-1\}$ and $s \in \{2,\ldots,R\}$ where $i \in \{0,\ldots,p-1\}$ and $s \in \{2,\ldots,R\}$ where R is the number of m-steps in the program and $w_{s,i}$ and $h_{s,i}$ are respectively, the local computation time of processor i during the m-step s and $h_{s,i} = max\{h^+_{s,i}, h^-_{s,i}\}$ where $h^+_{s,i}$ (respectively $h^-_{s,i}$) is the number of the received words (respectively sent words) by the processor i during the m-step s. The experiences done show that this model is suited to MSPML [26].

2.2 The Core Library

The MSPML library is based on the following primitives:

p : unit $\rightarrow$ int
g : unit $\rightarrow$ float
l : unit $\rightarrow$ float
mkpar : (int $\rightarrow$ α) $\rightarrow$ α **par**
apply : (α $\rightarrow$ β) **par** $\rightarrow$ α **par** $\rightarrow$ β **par**
get : α **par** $\rightarrow$ int **par** $\rightarrow$ α **par**
mget : (int $\rightarrow$ α) **par** $\rightarrow$ (int $\rightarrow$ bool) **par** $\rightarrow$ (int $\rightarrow$ α option) **par**
at : α **par** $\rightarrow$ int $\rightarrow$ α

They give access to the parameters of the parallel machine. In particular, the function **p** return the static number of processors of the parallel machine. This value do not change during the execution, as long as the parallel composition is not introduced.

There is also a polymorphic abstract type α **par** which represents the type of the parallel vectors composed of p expressions of type α, one expression per processor. The nesting of parallel expressions is not allowed and can be avoided by a type system.

The parallel constructors operate on parallel vectors. These parallel vectors are created by the primitive **mkpar**. The expression (**mkpar** f) is a parallel or global expression. For example, **mkpar**(**fun** $i \rightarrow i$) will be evaluated to the parallel vector $\langle\, 0\,, \ldots,\, p-1\,\rangle$.

In the MPM model, an algorithm is written as a combination of asynchronous local computation phases and communication phases. The asynchronous phases are programed with **mkpar** and **apply**.

The expression **apply**(**mkpar** f)(**mkpar** e) will be evaluated to $(f\ i)(e\ i)$ at processor i. The communication phases are programmed using **get** and **mget**. The semantics of **get**, where % is the modulo, is given by:

$$\mathbf{get}\langle\, v_0 \,,\ldots,\, v_{p-1} \,\rangle\langle\, i_0 \,,\ldots,\, i_{p-1} \,\rangle = \langle\, v_{i_0\%p} \,,\ldots,\, v_{i_{p-1}\%p} \,\rangle$$

The function **mget** is a generalisation of **mget**. It allows to have different data send to different processors during one m-step. The complete language contains also a global conditional which is necessary to take into account data computed locally for the global control. For the sake of conciseness these two primitives are omitted here.

2.3 Costs

The execution time of a MSPML program is represented by a cost vector of execution time on each processor: $\langle\, c_0 \,,\ldots,\, c_{p-1} \,\rangle$.

If the evaluation of a ML expression e outside a **mkpar**, require time w then its evaluation as a MSPML program adds w to each component of the cost vector: $\langle c_0 + w, \ldots, c_{p-1} + w\rangle$.

The evaluation of a **mkpar** expression requires w_i at processor i, the time required to evaluate $(f\ i)$. For **apply** time required is that of the evaluation of the arguments giving the vectors $\langle\, f_0 \,,\ldots,\, f_{p-1} \,\rangle$ and $\langle\, v_0 \,,\ldots,\, v_{p-1} \,\rangle$, then for processor i the time w_i required for the evaluation of $(f_i\ v_i)$.

The execution time of a communication primitive **get**, if the vector of costs after the evaluation of its arguments is $\langle\, c_0 \,,\ldots,\, c_{p-1} \,\rangle$, is given by the cost vector $\langle\, c'_0 \,,\ldots,\, c'_{p-1} \,\rangle$. It is defined by:

- if $\langle\, v_0 \,,\ldots,\, v_{p-1} \,\rangle$ is the first argument, $\#v_i$ is the size of value v_i,
- if $\langle\, i_0 \,,\ldots,\, i_{p-1} \,\rangle$ is the second argument of the **get**,
- if Ω_k is the set of the incoming partners of processor k:

$$\Omega_k = \{j | i_j = k\} \cup \{k\}$$

- then $c'_k = max\{c_i | i \in \Omega_k\} + max\{\sum_{j\in\Omega_k\setminus\{k\}} \#v_j, \#v_{j_k} \text{ if } j_k \neq k\} \times g + L$.

2.4 Examples

We present here some examples which form a part of the standard MSPML library.

The standard library of MSPML contains functions which are defined using only the primitives. For example, the function **replicate** creates a parallel vector which contains the same value everywhere. The primitive **apply** can be used only in the case the functions of the parallel vector of functions take only one argument. The function **apply2** is required to handle functions with two arguments.

```
(∗ val replicate: α → α par ∗)
let replicate x = mkpar(fun pid→ x)
```

(∗ val apply2:(α → $\beta \rightarrow$ γ) par → α par → β par → γ par ∗)
let apply2 f v1 v2 = **apply**(**apply** f v1) v2

It is also very convenient to apply the same sequential function to all the components of a parallel vector. That can be done by using the functions **parfun**. The difference between them is the number of their arguments:

(∗ val parfun: (α → β)par→ α par→ β par ∗)
let parfun f v = **apply** (replicate f) v
(∗ val parfun2: (α → $\beta \rightarrow$ γ) par → α par → β par → γ par ∗)
let parfun2 f v1 v2 = **apply**(parfun f v1) v2

The semantics of the total exchange is given by:

$$\text{totex}\langle\, v_0\,, \ldots,\, v_{p-1}\,\rangle = \langle\, f\,, \ldots,\, f\,, \ldots,\, f\,\rangle \text{ where } \forall i.(0 \leq i < p-1) \Rightarrow\ (f\ i) = v_i$$

The code is presented below where **noSome** is a function which removes the constructor **Some** and **compose** is the functional composition:

(∗ val totex: α par → (int → α) par ∗)
let totex vv = parfun (compose noSome)
  (**mget** (parfun(**fun** v i→ v)vv) (replicate(**fun** i → **true**)))

Its parallel cost is $(p-1) \times s \times g + L$, where s is the size in words of the biggest value v held by a processor. From this function we can obtain a total exchange function which returns a parallel vector of lists instead of a parallel vector of functions where procs() = [0;...;**p**()-1]:

(∗ val totex_list: α par → α list par ∗)
let totex_list v = parfun2 List.map (totex v) (replicate(procs()))

The set of bcast functions broadcast the value of a parallel vector at a given processor to all the other processor. Their semantics is given by:

$$\text{bcast}\ \langle\, v_0\,, \ldots,\, v_{p-1}\,\rangle\ r \;=\; \langle\, v_{r\%p}\,, \ldots,\, v_{r\%p}\,\rangle$$

The bcast_direct function is one possible implementation, using only one m-step. It could be written as follows:

(∗ bcast_direct: int → α par → α par ∗)
let bcast_direct root vv = **get** vv (replicate root)

Its parallel cost is $(p-1) \times s \times g + L$, where s is the size in words of the value v_n at processor n. Another possible implementation is the broadcast function which is evaluated in $\log p$ m-steps (figure 1)

The standard library of MSPML contains a collection of such functions to ease the writing of parallel programs. Thus, its same to write MSPML programs or programs with data-parallel skeletons. But if the MSPML standard library does not provide the required function it is possible to write new skeletons as higher order functions, using the primitives only.

```
let bcast_logp root vv =
  let newi i = natmod (i+(p())−root) (p()) in
  let from n = mkpar(fun i→ let j = newi i in
                          if (n/2<=j)&&(j<n) then i−(n/2) else i) in
  let rec aux n vv = if n<1 then vv else get (aux (n/2) vv) (from n)
  in aux (p()) vv
```

FIGURE 1. Broadcast Function

2.5 A New Primitive for Parallel Composition

The spatial parallel composition, called *juxtaposition*, allows to divide the parallel machine into two sub-machines. It allows the evaluation of two independent parallel programs on the same machine. The evaluation of the expression $(\textbf{juxta}\ m\ E_1\ E_2)$ proceeds as follows. The m first processors evaluate the expression E_1 and the $p-m$ remainder evaluate E_2. These $p-m$ processors are however renamed, the processor m becoming 0 and the processor $(p-1)$ becoming $(p-1)-m$. The juxtaposition does not modify the MPM cost model.

The result of evaluation of the parallel juxtaposition is:

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

In order to avoid the evaluation of the parallel arguments before the call to the juxtaposition, the type of the parallel composition is:

$$\textbf{juxta}\text{: int} \to (\text{unit} \to\ \alpha\ \textbf{par}) \to (\text{unit} \to\ \alpha\ \textbf{par}) \to\ \alpha\ \textbf{par}.$$

The following program is a divide-to-conquer version for prefix sum. Its semantics is:

$$\text{scan}\ \oplus\ \langle\ v_0\ ,\ldots,\ v_{p-1}\ \rangle\ =\ \langle\ v_0\ ,\ldots,\ v_0 \oplus v_1 \oplus \ldots \oplus v_{p-1}\ \rangle$$

where $\oplus$ is an associative binary operation.

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

The network is divided into two parts and the function scan is recursively applied to the two parts. The value at the last processor of the first part is broadcast to all the processors of the second part. Then this value and the local values computed by the recursive call are combined with the operation **op** on each processor of the second part.

2.6 The Communication Mechanism

The asynchronous nature of MSPML led us to design a communication mechanism which relies on the storage of values to be potentially requested by other processors in a data structure called communication environment. Each processor has its own communication environment and a value is stored per m-step.

During the execution of a MSPML program, for each process i, the system has a variable $mstep_i$ containing an integer indicating the current m-steps. For flat MSPML, all the processors perform the same number of m-steps. When the expression **(get vv vi)**, is evaluated at a given process i:

1. mstep_i is increased by one;
2. The value that this process holds in parallel vector vv is stored with the value of mstep_i in the communication environment;
3. the value j that this process holds in parallel vector vi is the process number from which the process i wants to receive a value. Thus process i sends a request to process j: it asks for the value at m-step mstep_i. When process j receives the request (threads are dedicated to handle requests, so the work of process j is not interrupted by the request), there are two cases:
   - $\text{mstep}_j \geq \text{mstep}_i$: it means that process j has already reached the same m-step than process i. Thus process j looks in its communication environment for the value associated with m-step mstep_i and sends it to process i;
   - $\text{mstep}_j < \text{mstep}_i$: nothing can be done until process j reaches the same m-step than process i.

If $i = j$, the third step is not performed.

Without juxtaposition all processes execute the same number of m-steps. It is not the case when juxtaposition is introduced. For example, in the following expression:

$$\textbf{let}\ this = \textbf{mkpar}\,(\textbf{fun}\ i \rightarrow\ i)\ \textbf{in juxta}\ 2\ (\textbf{get}\ this\ this)\ this$$

the two first processors of the network are the only ones to increase their m-step counters by evaluating the **get** primitive. Thus we cannot rely on a simple numbering of m-steps by naturals in order to exchange correctly messages among processors.

To distinguish the various messages from different sub-machines we introduce the following m-step numbering:

$$\text{step} ::= (n,m)\ \mid\ \text{step.R}(n,m)\ \mid\ \text{step.L}(n,m) \text{ where } n \text{ and } m \text{ are naturals.}$$

Each time a processor calls the juxtaposition primitive, its m-step counter grows: L (resp. R) indicates that the processor belongs to the sub-machine which evaluates the first (resp. second) parallel expression of a juxtaposition. The natural n

is a counter used to know how many m-steps (use of **get**, **mget** or **at**) have been performed *in the given sub-network*.

When the call to the juxtaposition ends the last pair of the m-step counter and the sub-machine indicator are removed. But it is possible to have successive non-nested uses of the juxtaposition:

$$\textbf{let}\ e_1 = (\textbf{juxta}\ m\ \ldots\ \ldots)\ \textbf{in}\ \textbf{let}\ e_2 = (\textbf{juxta}\ m\ \ldots\ \ldots)\ \textbf{in}\ \textbf{let}\ e_3 = (\textbf{juxta}\ m\ \ldots\ \ldots)\ \textbf{in}$$

Thus if we count only the number of m-steps in a give call to the juxtaposition, two values of two successive non-nest calls to the juxtaposition could have the same m-step counter, which would lead to an incorrect mechanism. Thus the second natural in the pairs (n,m) gives the number of successive non-nested calls to the juxtaposition *in the given sub-network*.

The figure 2 illustrates the m-steps numbering on an example.

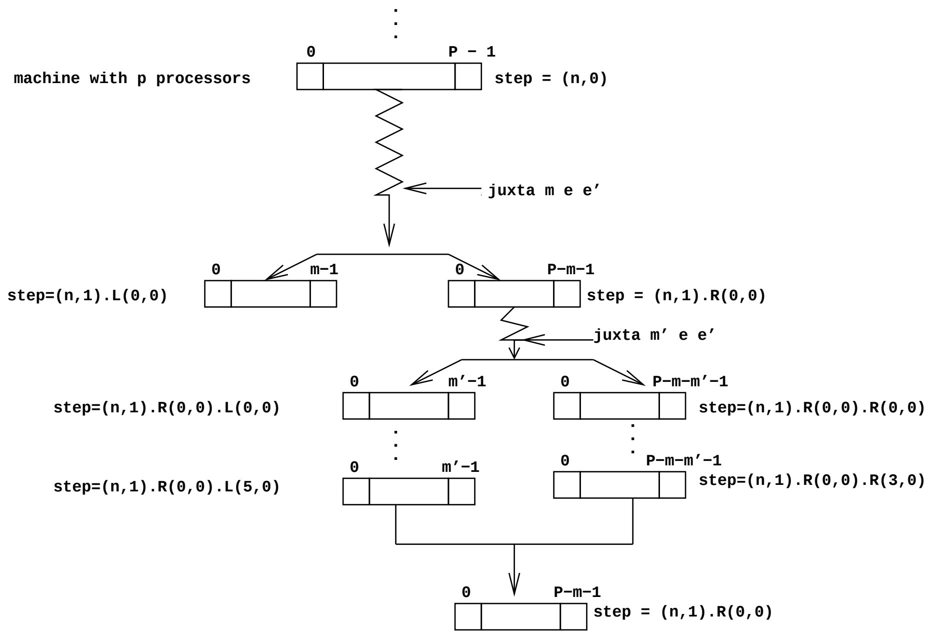

FIGURE 2. M-steps Numbering

3 DISTRIBUTED SEMANTICS OF MSPML WITH JUXTAPOSITION

Parallel languages have usually two models: a programming model which is the semantics given to the programmer and an execution model which is the semantics of what really happens during the execution of a program on a parallel machine. In the case of MSPML, in the informal semantics we presented a MSPML program as a sequential program on a parallel data structure. The execution of a MSPML program on an actual parallel machine is in fact the execution of p sequential communicating programs on p processors. The parallel data structure does not exist: it is just a logical view of p values on p different processors.

MSPML has two formal semantics. The first one formalises the programming model and is a big steps semantics. We omit it here. In this semantics the parallel primitives seem to be synchronous. A MSPML program is seen like a classical Objective Caml sequential program with the use from time to time of operations on parallel vectors. From this point of view, the MSPML programming model is quite similar to that of BSML, the essential difference being the parallel costs associated to each primitive.

The second semantics formalises the execution model. A MSPML program is in this case as a SPMD program: it is in fact p copies of the same program, one by processor of the parallel machine, which works only on one component (a slice) of parallel vectors. These copies communicate by message passing. This semantics provides a specification for the implementation of MSPML.

3.1 Syntax

Here we do not distinguish local and global expressions in the same way it is done in the BSλ-calculus, i.e. in the syntax. We just distinguish local and global variables. A typing system [6], omitted here, gives the distinction between local and global expressions. In addition, this system avoids parallel nesting.

$$
\begin{array}{lllll}
e ::= & x & \mid c & \mid op \\
 & \mid \mathbf{let}\ x:\tau\ \mathbf{in}\ e & \mid \mathbf{fun}\ x:\tau \to e & \mid (e\ e) \\
 & \mid (e,e) & \mid \mathbf{if}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e & \mid \mathbf{fst}\ e \\
 & \mid \mathbf{snd}\ e & \mid \mathbf{mkpar}\ e & \mid \mathbf{apply}\ e\ e \\
 & \mid \mathbf{get}\ e\ e & \mid \mathbf{request}\ e\ e & \mid \mathbf{juxta}\ e\ e\ e \\
 & \mid \mathbf{if}\ e\ \mathbf{at}\ e\ \mathbf{then}\ e\ \mathbf{else}\ e & \mid \overrightarrow{e} & \mid \|e_d\|
\end{array}
$$

FIGURE 3. MSPML Syntax

We found in the syntax presented in figure 3: the classical functional expressions, the parallel operations presented in the previous section, **juxta** $e\ e\ e$ the parallel juxtaposition, $\overrightarrow{e}$ the local expression of the parallel vector at the processor which contains it, **request** $e\ e$ the request for a value from a given processor (first argument) at a given m-step (second argument), and $\|e_d\|$ indicates that the expression e is evaluated in a sub-machine.

3.2 Evaluation

The distributed evaluation is defined here in two steps:

- local reduction (performed by one process i) $\rightharpoonup^{i}_{\mathrm{p}}$;
- global reduction $\rightharpoonup_{\mathrm{i}}$ of distributed terms which allows the evaluation of communication requests (for **get**, **mget at**).

The values of the local reduction are as follows:

$$v \quad ::= \quad \textbf{fun}\, x : \tau \rightarrow\ e \ \mid\ c \ \mid\ op \ \mid\ (v,\, v) \ \mid\ \overrightarrow{v}$$

request and $\|v_d\|$ are not a values.

Locally, with the parallel juxtaposition, the number of processors and processors identifiers are not any more static values. To manage this dynamism each processor contains two stacks: $\mathcal{E}_p$ and $\mathcal{E}_{pid}$ which contain respectively on top of the stack the number of processors of the current network and the identifier of the processor in the current network. The environment $\mathcal{E}_p$ (resp.$\mathcal{E}_{pid}$) is initialised by the number of processors (resp. absolute processor identifier).

The rules have the form $(e,s,\mathcal{E}_c,\mathcal{E}_p,\mathcal{E}_{pid}) \ \rightharpoonup^{i}_{\mathrm{p}}\ (e',s',\mathcal{E}'_c,\mathcal{E}'_p,\mathcal{E}'_{pid})$ that we read as follows:

> "At processor $\mathrm{hd}(\mathcal{E}_{pid})$[1] in the network of $\mathrm{hd}(\mathcal{E}_p)$ processors, whose absolute *pid* is i, and at the m-step s the expression e in the communication environment $\mathcal{E}_c$ is reduced *locally* to the expression e' with possible changes in the m-step counter, the stacks and the communication environment $\mathcal{E}'_c$".

The local reduction is divided in three groups of rules: contexts and the rule of context, the functional reduction which corresponds to a classical semantics of a sequential functional programming language and the reduction of the parallel operations specific to MSPML.

The head reduction cannot be applied in any context. We follow a weak call-by-value strategy. Therefore, the contexts 3.2 define the order of evaluation of the arguments for each term. The contexts force the evaluation of the arguments (from left to right) before allowing the reduction. The contexts are applied by using the context rule:

$$\frac{(e_i,s_i,\mathcal{E}_{c_i},\mathcal{E}_{p_i},\mathcal{E}_{pid_i}) \ \rightharpoonup^{i}_{\mathrm{p}}\ (e'_i,s'_i,\mathcal{E}'_{c_i},\mathcal{E}'_{p_i},\mathcal{E}'_{pid_i})}{(\Gamma(e_i),s_i,\mathcal{E}_{c_i},\mathcal{E}_{p_i},\mathcal{E}_{pid_i}) \ \rightharpoonup^{i}_{\mathrm{p}}\ (\Gamma(e'_i),s'_i,\mathcal{E}'_{c_i},\mathcal{E}'_{p_i},\mathcal{E}'_{pid_i})}$$

Rules of functional reduction, change only the first component of the tuple. The full set of rules is presented in [6]. We present some rules as examples. For binding we have:

$$((\textbf{let}\, x : \tau = v \textbf{ in } e),s,\mathcal{E}_c,\mathcal{E}_p,\mathcal{E}_{pid}) \quad \rightharpoonup^{i}_{\mathrm{p}} \quad (e[x \leftarrow v],s,\mathcal{E}_c,\mathcal{E}_p,\mathcal{E}_{pid})$$

[1]the environment $\mathcal{E}_{pid}$ (resp. $\mathcal{E}_p$) is a stack that top is given by $\mathrm{hd}(\mathcal{E}_{pid})$ (resp. $\mathrm{hd}(\mathcal{E}_p)$)

$$\begin{array}{lllllll}
\Gamma & ::= & [] & \mid & \Gamma\ e & \mid & v\ \Gamma \\
 & \mid & \mathbf{fst}\ \Gamma & \mid & \mathbf{snd}\ \Gamma & \mid & \mathbf{let}\ x:\tau=\Gamma\ \mathbf{in}\ e \\
 & \mid & \overrightarrow{\Gamma} & \mid & \mathbf{if}\ \Gamma\ \mathbf{then}\ e\ \mathbf{else}\ e & \mid & \mathbf{mkpar}\,\Gamma \\
 & \mid & \mathbf{apply}\,\Gamma\,e & \mid & \mathbf{apply}\,v\,\Gamma & \mid & \mathbf{get}\,\Gamma\,e \\
 & \mid & \mathbf{get}\,v\,\Gamma & \mid & \mathbf{if}\,\Gamma\,\mathbf{at}\,e\,\mathbf{then}\,e\,\mathbf{else}\,e & \mid & \mathbf{if}\,v\,\mathbf{at}\,\Gamma\,\mathbf{then}\,e\,\mathbf{else}\,e \\
 & \mid & \mathbf{juxta}\ \Gamma\ e\ e & \mid & \|\Gamma\| & &
\end{array}$$

FIGURE 4. Evaluation contexts of the distributed semantics

The following rule allows the creation of the enumerated parallel vectors. $(\overrightarrow{v\ i'})$ is the part of the parallel enumerated vector at processor i.

$$(\mathbf{mkpar}\,v, s, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup^i_{\mathrm{p}} \quad ((\overrightarrow{v\ i'}), s, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \text{ with } i' = hd(\mathcal{E}_{pid})$$

The **apply** rule is a classical parallel rule which performs the point-wise parallel application of a parallel vector of functions to a parallel vector of arguments:

$$(\mathbf{apply}\,\overrightarrow{v_1}\,\overrightarrow{v_2}, s, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup^i_{\mathrm{p}} \quad (\overrightarrow{v_1\ v_2}, s, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid})$$

The communication rules concern the primitive **get** and **at**. The rules are similar and we present the rules for the **get** primitive only. When a **get** evaluated, the value of the first argument is stored in the communication environment together with the current value of m-step counter. Then the **get** becomes a **request**, if the destination processor is not the source processor:

$$\begin{array}{c}
(\mathbf{get}\,\overrightarrow{v}\ \overrightarrow{j}, s, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \ \rightharpoonup^i_{\mathrm{p}} \ (\overrightarrow{\mathbf{request}\,s'\ j'}, s', (s', v) :: \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \\
\text{if } hd(\mathcal{E}_{pid}) \neq j \text{ and } s' = \mathrm{inc}_c(s) \text{ with } j' = (i - hd(\mathcal{E}_{pid})) + (j \% hd(\mathcal{E}_p))
\end{array}$$

The name of destination processor j in the current sub-network is translated into the absolute name j' of this processor in the whole parallel machine.

If the destination and source processors are the same, then the first argument of the get is simply put the communication environment together with the current value of the m-step counter:

$$(\mathbf{get}\,\overrightarrow{v}\ \overrightarrow{i}, s, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup^i_{\mathrm{p}} \quad (\overrightarrow{v}, \mathrm{inc}_c(s), (\mathrm{inc}_c(s), v) :: \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid})$$

The incrementation of the m-step counter inc_c is defined by :

$$\begin{cases}
\mathrm{inc}_c((n,m)) & = (n+1, m) \\
\mathrm{inc}_c(s.\mathrm{D}(n,m)) & = s.\mathrm{D}(n+1, m) \text{ where } D = L \text{ or } D = R
\end{cases}$$

To do the exchanges, the evaluation must be considered globally and not locally, which is done by the global rules.

The evaluation of **juxta** *m e*1 *e*2, the current machine is divided into two sub-machines. If the processor identifier in the sub-machine is less than the first argument of the juxtaposition we have:

$$(\textbf{juxta}\ m\ e_1\ e_2, s, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \quad \rightharpoonup^{i}_{\mathrm{p}} \quad \left(\|e_1\|, \mathrm{inc}_j(s).L(0,0), \mathcal{E}_c, m :: \mathcal{E}_p, pid :: \mathcal{E}_{pid}\right)$$
$$\text{with } pid = hd(\mathcal{E}_{pid}) \text{ and if } pid < m$$

inc_j has a same effect of inc_c but on the second component of the pair.

If the processor identifier in the current machine is greater or equal to *m*, a similar rule is performed with the following changes: L in the m-step counter value is replaced by R, e_1 by e_2, m by $p-m$, and $hd(\mathcal{E}_{pid})$ by $hd(\mathcal{E}_{pid})-m$.

When the evaluation of the expression inside $\|\cdot\|$ is done, the following rule restores the parameters of the super machine:

$$(\|v\|, s.D(n,m), \mathcal{E}_c, p' :: \mathcal{E}_p, pid :: \mathcal{E}_{pid}) \ \rightharpoonup^{i}_{\mathrm{p}} \ (v, s, \mathcal{E}_c, \mathcal{E}_p, \mathcal{E}_{pid}) \text{ where } D \in \{L, R\}$$

The global reduction $\rightarrow$ is a relation on distributed vectors:

$$\langle\!\langle\, (e_0, s_0, \mathcal{E}^c_0, \mathcal{E}^p_0, \mathcal{E}^{pid}_0), \ldots,\ (e_{p-1}, s_{p-1}, \mathcal{E}^c_{p-1}, \mathcal{E}^p_{p-1}, \mathcal{E}^{pid}_{p-1})\,\rangle\!\rangle$$

The global reduction is defined by two rules. The first one allows a local reduction to be considered at the global level:

$$\frac{(e_i, s_i, \mathcal{E}^c_i, \mathcal{E}^p_i, \mathcal{E}^{pid}_i) \ \rightharpoonup^{i}_{\mathrm{p}} \ (\overline{e}_i, \overline{s}_i, \overline{\mathcal{E}}^c_i, \overline{\mathcal{E}}^p_i, \overline{\mathcal{E}}^{pid}_i)}{\begin{array}{c}\langle\!\langle\, (e_0, s_0, \mathcal{E}^c_0, \mathcal{E}^p_0, \mathcal{E}^{pid}_0), \ldots, (e_i, s_i, \mathcal{E}^c_i, \mathcal{E}^p_i, \mathcal{E}^{pid}_i), \ldots, (e_{p-1}, s_{p-1}, \mathcal{E}^c_{p-1}, \mathcal{E}^p_{p-1}, \mathcal{E}^{pid}_{p-1})\,\rangle\!\rangle \\ \rightarrow \\ \langle\!\langle\, (e_0, s_0, \mathcal{E}^c_0, \mathcal{E}^p_0, \mathcal{E}^{pid}_0), \ldots, (\overline{e}_i, \overline{s}_i, \overline{\mathcal{E}}^c_i, \overline{\mathcal{E}}^p_i, \overline{\mathcal{E}}^{pid}_i), \ldots, (e_{p-1}, s_{p-1}, \mathcal{E}^c_{p-1}, \mathcal{E}^p_{p-1}, \mathcal{E}^{pid}_{p-1})\,\rangle\!\rangle\end{array}}$$

The second rule allows the exchange of messages between two processors (the reduction of a request). This rules formalises the third phase of the communication mechanism (section 2.6):

$$\frac{(e_i = \Gamma[\textbf{request}\, n\ \ j]) \text{ and } (n, v) \in \mathcal{E}_{c_j}}{\begin{array}{c}\langle\!\langle\, (e_0, s_0, \mathcal{E}^c_0, \mathcal{E}^p_0, \mathcal{E}^{pid}_0), \ldots, (e_i, s_i, \mathcal{E}^c_i, \mathcal{E}^p_i, \mathcal{E}^{pid}_i), \ldots, (e_{p-1}, s_{p-1}, \mathcal{E}^c_{p-1}, \mathcal{E}^p_{p-1}, \mathcal{E}^{pid}_{p-1})\,\rangle\!\rangle \\ \rightarrow \\ \langle\!\langle\, (e_0, s_0, \mathcal{E}^c_0, \mathcal{E}^p_0, \mathcal{E}^{pid}_0), \ldots, (\Gamma[v], s_i, \mathcal{E}^c_i, \mathcal{E}^p_i, \mathcal{E}^{pid}_i), \ldots, (e_{p-1}, s_{p-1}, \mathcal{E}^c_{p-1}, \mathcal{E}^p_{p-1}, \mathcal{E}^{pid}_{p-1})\,\rangle\!\rangle\end{array}}$$

The confluence of the semantics of MSPML with juxtaposition, the correctness of the new numbering of m-steps and the safety of the typing are proved. The theorems, propositions, and proofs are given in detail in [7].

4 RELATED WORK

[3] showed that NESL [11, 10] is more effective when the vectors size is constant. Even if it isn't the case, the majority of the operations of NESL can be implemented in MSPML. In particular the nested lists can be implemented like in [23]. From

this point of view MSPML can seem lower level than NESL. But MSPML offers a high level functions whereas it is not the case for NESL.

The parallel functional language Caml-Flight [14] is based on the *vague* mechanism. The primitive **sync** is used to indicate which processors can exchange messages when the **get** primitive is used. This **get** is very different from the our : it is a question here of asking the distant evaluation of an expression. This mechanism is more complex than that of MSPML and there is no purely functional semantics of Caml-Flight [17]. Moreover programs Caml-Flight are SPMD and thus more difficult to write and read that MSPML programs. The type system used to avoid incorrect nested parallelism is also complex [36].

[31, 30] describes the mechanism of the *structural clocks* which allows the execution of data-parallel programs written in a small imperative language SPMD. The difficulty within this framework is that the number of communication phases can be different on each processor because there is a parallel composition. Our m-step numbering is thus similar to structural clocks.

[35] 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 our BSML parallel *superposition* [25], several BSP threads use the whole network. The same author advocates in [29] 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.

[38] 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 [16] which allows subset synchronisation. We follow [21] to reject such a possibility in BSML.

In the BSPlib library [22] subset synchronisation is not allowed as explained in [33]. The PUB library [12] is another implementation of the BSPlib standard proposal. It offers additional features with respect to the standard which follows the BSP* model [4] and the D-BSP model [16]. Minimum spanning trees nested BSP algorithms [13] have been implemented using these features.

5 CONCLUSIONS AND FUTURE WORK

Adding the juxtaposition to MSPML allows to write easily divide-and-conquer parallel programs. An implementation was developed, taking the semantics presented in this article as a specification. It remains to experiment the new possibilities offered by the juxtaposition, as well as to validate the cost model.

The presented semantics is a semantics of the execution. It is a semantics for the implementer of MSPML. This is not the semantics which is presented to the programmer. The latter called the programming model formalises the informal presentation of MSPML given in section 2. In order to guarantee the correctness of our system, we have to prove the equivalence of the programming model and the execution model.

In the current implementation, the management of the communication environments requires a global synchronisation from time to time. In the case of MSPML without parallel composition, a new mechanism was proposed which removes these global synchronisations [5]. We should adapt it for MSPML with juxtaposition.

Acknowledgements

This work is supported by the french ministry of research through the ACI Young Researchers program under the project “Certified Parallel Programming” (Propac, http://wwwpropac.free.fr).

REFERENCES

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

[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] M. Bamha. L’implémentation d’un langage portable à parallélisme emboîté en processus statiques. Master thesis, University of Orléans, LIFO, 1996.

[4] W. Bäumker, A. adn Dittrich and F. Meyer auf der Heide. Truly effi cient parallel algorithms: *c*-optimal multisearch for an extension of the BSP model. In 3rd *European Symposium on Algorithms (ESA)*, pages 17–30, 1995.

[5] A. Belbekkouche. MSPML : Environnements de communication et tolérance aux pannes. Master thesis, University of Orléans, LIFO, 2005.

[6] R. Benheddi. Composition parallèle pour MSPML: sémantiques et implantation. Master thesis, University of Orléans, LIFO, 2005.

[7] R. Benheddi and F. Loulergue. Semantics and Implementation of a Parallel Composition Primitive for Minimally Synchronous Parallel ML. Technical Report 2006-09, University of Orleans, LIFO, 2006. In preparation.

[8] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.

[9] R. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

[10] G. Blelloch, S. Chatterjee, J. Hardwick, J. Sipelstein, and M. Zagha. Implementation of a portable nested data-parallel language. *Journal of Parallel and Distributed Computing*, 21(1):4–14, 1994.

[11] G.E. Blelloch. NESL: A Nested Data-Parallel Language (version 2.6). Technical Report CMU-CS-93-129, School of Computer Science, Carnegie Mellon University, April 1993.

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

[13] O. Bonorden, F. Meyer auf der Heide, and R. Wanka. Composition of Effi cient Nested BSP Algorithms: Minimum Spanning Tree Computation as an Instructive Example.

In *International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, 2002.

[14] E. Chailloux and C. Foisy. A Portable Implementation for Objective Caml Flight. *Parallel Processing Letters*, 13(3):425–436, 2003.

[15] E. Chailloux, P. Manoury, and B. Pagano. *Développement d'applications avec Objective Caml*. O'Reilly France, 2000. freely available in english at http://caml.inria.fr/oreilly-book/index.html.

[16] P. de la Torre and C. P. Kruskal. Submachine locality in the bulk synchronous setting. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, number 1123–1124 in Lecture Notes in Computer Science, Lyon, August 1996. LIP-ENSL, Springer.

[17] C. Foisy, J. Vachon, and G. Hains. DPML: de la sémantique à l'implantation. In P. Cointe, C. Queinnec, and B. Serpette, editors, *Journées Francophones des Langages Applicatifs*, volume 11 of *Collection Didactique*, Noirmoutier, Février 1994. INRIA.

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

[19] F. Gava and F. Loulergue. A Functional Language for Departmental Metacomputing. *Parallel Processing Letters*, 15(3):289–304, 2005.

[20] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam. *PVM Parallel Virtual Machine. A User's Guide and Tutorial for Networked Parallel Computing*. Scientifi c and Engineering Computation Series. MIT Press, 1994.

[21] G. Hains. Subset synchronization in BSP computing. In H.R.Arabnia, editor, *PDPTA'98 International Conference on Parallel and Distributed Processing Techniques and Applications*, volume I, pages 242–246, Las Vegas, July 1998. CSREA Press.

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

[23] Z. Hu, T. Takahashi, H. Iwasaki, and M. Takeichi. Segmented Diffusion Theorem. In *IEEE International Conference on Systems, Man and Cybernetics (SMC 02)*. IEEE Press, October 6-9 2002.

[24] 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.

[25] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part I*, number 2659 in LNCS, pages 223–232. Springer Verlag, june 2003.

[26] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics and Implementation of Minimally Synchronous Parallel ML. *International Journal of Computer and Information Science*, 5(3):182–199, 2004.

[27] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In Vaidy S. Sunderam, G. Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *International Conference on Computational Science, Part II*, number 3515 in LNCS, pages 1046–1054. Springer, 2005.

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

[29] J. M. R. Martin and A. Tiskin. BSP modelling a two-tiered parallel architectures. In B. M. Cook, editor, *WoTUG'99*, pages 47–55, 1999.

[30] E. Melin, B. Raffi n, X. Rebeuf, and B. Virot. A Structured Synchronization and Communication Model Fitting Irregular Data Accesses. *Journal of Parallel and Distributed Computing*, 50:3–27, 1998.

[31] X. Rebeuf. *Un modèle de coût symbolique pour les programmes parallèles asynchrones à dépendances structurées*. PhD thesis, Université d'Orléans, LIFO, 2000.

[32] J. L. Roda, C. Rodríguez, D. G. Morales, and F. Almeida. Predicting the execution time of message passing models. *Concurrency: Practice and Experience*, 11(9):461–477, 1999.

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

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

[35] A. Tiskin. A New Way to Divide and Conquer. *Parallel Processing Letters*, (4), 2001.

[36] J. Vachon. Une analyse statique pour le contrôle des effets de bords en Caml-Flight beta. In C. Queinnec, V. V. Donzeau-Gouge, and P. Weis, editors, *JFLA*, number 13. INRIA, Janvier 1995.

[37] L. G. Valiant. A bridging model for parallel computation. *Comm. of the ACM*, 33(8):103, 1990.

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