Semantics and Implementation of Minimally Synchronous Parallel ML

Frédéric Loulergue[a] Frédéric Gava[a] Myrto Arapinis[a] Frédéric Dabrowksi[a]
University Paris Val-de-Marne, France

Abstract

This paper presents a new functional parallel language: Minimally Synchronous Parallel ML (MSPML). The execution time can be estimated, dead-locks and indeterminism are avoided. Programs are written as usual ML programs but using a small set of additional functions. Provided functions are used to access the parameters of the parallel machine and to create and operate on a parallel data structure. It follows the execution and cost model of the Message Passing Machine model (MPM). It shares with Bulk Synchronous Parallel ML its syntax and high-level semantics but it has a minimally synchronous distributed semantics. Experiments have been run on a cluster of PC using an implementation of the Diffusion algorithmic skeleton.

Keywords: Asynchronous Parallelism, Functional Programming, Deterministic Semantics, Cost Model.

1. Introduction

Bulk Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [61, 57] to offer a high degree of abstraction in the same way as PRAM [23] models and yet allow portable and predictable performance on a wide variety of architectures. Bulk synchronous parallelism (and the Coarse-Grained Multicomputer model, CGM, which can be seen as a special case of the BSP model) has been used for a large variety of applications [59]: scientific computing [9, 32], genetic algorithms [12] and genetic programming [19], neural networks [54], parallel databases [4], constraint solvers [28], *etc.*. 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” [17].

The main advantages of the BSP model are:

- deadlocks are avoided, indeterminism can be either avoided or restricted to very specific cases. For example in the BSPlib [31], indeterminism can only occur when using the direct remote memory access operation **put**: two processes can write different values in the same memory address of a third process
- portability and performance predictability [36].

Nevertheless the majority of parallel programs written are not BSP programs. There are two main arguments against BSP. First the global synchronization barrier is claimed to be expensive. Second the BSP model is claimed to be too restrictive. All parallel algorithms are not fitted to its structured parallelism.

Divide-and-conquer parallel algorithms are a class of algorithms which seem to be difficult to write using the BSP model and several models derived from the BSP model which allow subset synchronization have been proposed [16, 56]. Another approach is to offer a two tiered architecture [45]. We showed that divide-and-conquer algorithms can be written using extensions [41, 40] of our Bulk Synchronous Parallel ML language (BSML) [44, 39]. The execution of such programs even follow the pure BSP model. Anyway the BSP model and these extensions are efficient enough only under some conditions dependent on the BSP parameters and the problem.

Thus we decided to investigate semantics of a new functional parallel language, without synchronization barriers, called Minimally Synchronous Parallel ML (MSPML). As a first phase we aimed at having (almost) the same source language and high level semantics (programming view) than BSML (in particular to be able to use with MSPML work done on type system [25] and proof of parallel BSML programs [24]), but with a different (and more efficient for unbalanced programs) low-level semantics and implementation.

With this new language we would like to:

- have a functional semantics and a deadlock free language but a simple cost model is no more mandatory ;
- compare the efficiencies of BSML and MSPML as the comparisons of BSP and other parallel paradigms were done with classical imperative languages (C, Fortran) ;
- investigate the expressiveness of MSPML for non BSP-like algorithms.

MSPML will also be our framework to investigate extensions which are not suitable for BSML, such as the nesting of parallel values or which are not intuitive enough in BSML, such as spatial parallel composition.

[a]Laboratory of Algorithms, Complexity and Logic
University Paris Val-de-Marne, Cr´eteil, France
http://mspml.free.fr

MSPML and BSML have been mixed to obtain a functional language [26] for departmental meta-computing. Several BSML programs are run on several parallel machines and are coordinated by a MSPML-like program.

This paper is organized as follows. We first present the cost and execution model used for flat (without parallel composition) MSPML. Then we describe informally the main constructs of MSPML, first from the application programmer's point of view (section 3) and then from the implementor's point of view (section 4). Formal semantics of MSPML are given in section 5. Section 6 presents some experiments done on a cluster of PC with a small application which uses the Diffusion algorithmic skeleton. Section 7 is devoted to related work. We end with conclusions and future work (section 8).

2. Cost and Execution Model

The Bulk Synchronous Parallel model [61, 46, 57, 8] represents a parallel computation on a parallel machine with p processors, a communication network and synchronization unit, as a sequence of *super-steps* (Figure 1) consisting of an alternate of computation phases (p asynchronous computations) and communications phases (data exchanges between processors) with global synchronization. The BSP *cost* model estimates execution times by a simple formula. A computation super-step takes as long as its longest sequential process, a global synchronization takes a fixed, system-dependent time L and a communication super-step is completed in time proportional to the arity h of the data exchange: the maximal number of words sent or received by a processor during that super-step. The system-dependent constant g, measured in time/word, is multiplied by h to obtain the estimated communication time. It is useful to measure times in multiples of a Flop so as to normalize g and L w.r.t. the sequential speed of processor nodes.

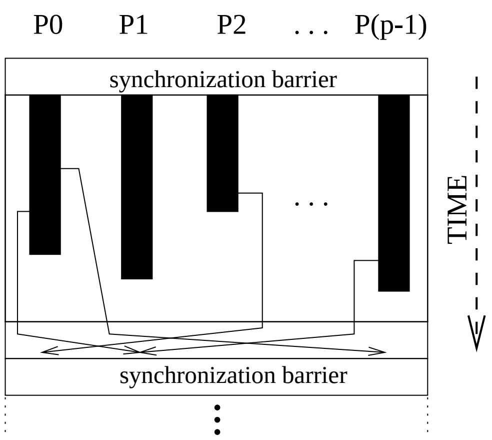

Figure 1. A BSP Super-step

2.1 BSPWB: BSP Without Barrier

BSPWB, for *BSP Without Barrier* [53], is a model directly inspired by the BSP model. It proposes to replace the notion of super-step by the notion of m-step defined as: at each m-step, each process performs a sequential computation phase then a communication phase. During this communication phase the processes exchange the data they need for the next m-step. The parallel machine in this model is characterized by three parameters (expressed as multiples of the processors speed):

- the number of processes p,
- the latency L of the network,
- the time g which is taken to one word to be exchanged between two processes.

The time needed *for a process* i to execute a m-step s, is $t_{s,i}$ bounded by T_s the time needed for the execution of the m-step s *by the parallel machine*. T_s is defined inductively by:

$$\begin{cases} T_1 = max\{w_{1,i}\} + \max\{g \times h_{1,i} + L\} \\ T_s = T_{s-1} + max\{w_{s,i}\} + max\{g \times h_{s,i} + L\} \end{cases}$$

where $i \in \{0,\ldots,p-1\}$ and $s \in \{2,\ldots,R\}$ where R is the number of m-steps of the program and $w_{s,i}$ and $h_{s,i}$ respectively denote the local computation time at process i during m-step s and $max\{h^+_{s,i}, h^-_{s,i}\}$ where $h^+_{s,i}$ (resp. $h^-_{s,i}$) is the number of words sent (resp. received) by process i during m-step s. This model could be applied but it will be not accurate enough because the bounds are too coarse.

2.2 MPM: Message Passing Model

A better bound $\Phi_{s,i}$ is given by the Message Passing Machine (MPM) model [52]. The parameters of the Message Passing Machine are the same than those of the BSPWB model. The model uses the set $\Omega_{s,i}$ for a process i and a m-step (Figure 2) s defined as:

$$\Omega_{s,i} = \left\{ \begin{array}{l} j/\text{process } j \text{ sends a message} \\ \text{to process } i \text{ at m-step } s \end{array} \right\} \bigcup \{i\}$$

Processes included in $\Omega_{s,i}$ are called "incoming partners" of process i at m-step s. $\Phi_{s,i}$ is inductively defined as:

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

where $h_{s,i} = \max\{h^+_{s,i}, h^-_{s,i}\}$ for $i \in \{0,\ldots,p-1\}$ and $s \in \{2,\ldots,R\}$. Execution time for a program is thus bounded by: $\Psi = \max\{\Phi_{R,j}/j \in \{0,1,\ldots,p-1\}\}$.

The MPM model takes into account that a process only synchronizes with each of its incoming partners and is therefore more accurate. The preliminary experiments (section 6) done with our implementation showed that the model applies well to MSPML.

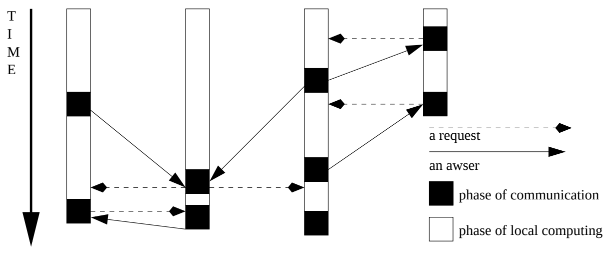

Figure 2. The MPM m-step

3. Minimally Synchronous Parallel ML

There is no implementation of a full Minimally Synchronous Parallel ML (MSPML) language but rather a partial implementation as a library for the functional programming language Objective Caml [51, 14, 38].

A MSPML program can be seen as an usual sequential program which manipulates a parallel data structure (called parallel vector). This is very different from SPMD programming (Single Program Multiple Data) where the programmer must use a sequential language and a communication library (like MPI [58]). 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* or may contain *deadlocks*.

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

3.1 The Core Library

The so-called MSPML library is based on the elements given in figure 3.

It gives access to the parameters of the underling architecture which is considered as a Message Passing Machine (MPM). In particular, it offers the function **p** such that the value of **p**() is p, the static number of processes of the parallel machine. The value of this variable does not change during execution. There is also 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. This can be ensured by a type system [25].

The parallel constructs of MSPML 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 **fun** pid $\rightarrow$ e for f to show that the expression e may be different on each process. This expression e is said to be *local*: it is inside a **mkpar** function and its value depends on the location (process) it is. The expression (**mkpar** f) is a parallel object and it is said to be *global*. For example the expression **mkpar**(**fun** pid $\rightarrow$ pid) will be evaluated to the parallel vector $\langle\ 0\ , \ldots,\ p-1\ \rangle$.

In the MPM model, an algorithm is expressed as a combination of asynchronous local computations and phases of communication. Asynchronous phases are programmed with **mkpar** and with **apply**. It is such as the expression **apply** (**mkpar** f) (**mkpar** e) stores (f i) (e i) on process i.

The communication phases are expressed by **get** and **mget**. The semantics of **get** is given by:

$$\begin{aligned} &\textbf{get}\,\langle v_0,\ldots,v_{p-1}\rangle\,\langle i_0,\ldots,i_{p-1}\rangle \\ =\ &\langle\ v_{(i_0\%p)}\ ,\ldots,\ v_{(i_{(p-1)}\%p)}\ \rangle \end{aligned}$$

where $\%$ is modulo.

The **mget** function is a generalization which allows to request data from several processes during the same m-step and to deliver different messages to different processes. In the type of **mget** in Figure 3, α option is defined by **type** α option = None | Some **of** α . Its semantics is:

$$\begin{aligned} &\textbf{mget}\,\langle f_0,\ldots,f_{p-1}\rangle\,\langle b_0,\ldots,b_{p-1}\rangle = \langle g_0,\ldots,g_{p-1}\rangle \\ &\text{where } g_i = \textbf{fun}\ j\ \rightarrow \textbf{if}\ (b_i\ j)\ \textbf{then}\ \text{Some}\ (f_j\ i)\ \textbf{else}\ \text{None} \end{aligned}$$

The full language would also contain a synchronous conditional:

$$\textbf{if}\ e\ \textbf{at}\ n\ \textbf{then}\ e_1\ \textbf{else}\ e_2$$

It evaluates to v_1 (the value obtained by evaluating e_1) or v_2 (the value obtained by evaluating e_2) depending on the value of the parallel vector of booleans e at process given by the integer n. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core MSPML library contains the function **at** to be used in the constructions:

- **if** (**at** e n) **then** ... **else** ...
- **match** (**at** e n) **with** ...

at expresses communication phase. Global conditional is necessary to express algorithms like:

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

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

3.2 Examples

We will now present some examples which are part of the MSPML standard library and which are called in the Diffusion algorithmic skeleton program used for experiments in section 6.

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

Figure 3. MSPML Core Library

3.2.1 Very Often Used Functions

Some useful functions can be 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 for a parallel vector of functions which take only one argument. To deal with functions which take two arguments we need to define the apply2 function.

```
let replicate x = mkpar (fun pid → x)
let apply2 vf v1 v2 = apply (apply vf v1) v2
```

It is also very common to apply the same sequential function at each process. It can be done using the parfun functions: they differ only in the number of arguments of the function to apply:

```
let parfun f v = apply (replicate f) v
let parfun2 f v1 v2 = apply (parfun f v1) v2
let parfun3 f v1 v2 v3 = apply (parfun2 f v1 v2) v3
```

(applyat n f_1 f_2 v) applies function f_1 at process n and function f_2 at other processes:

```
let applyat n f1 f2 v =
  apply (mkpar(fun i → if i=n then f1 else f2)) v
```

parpair_of_pairpar transforms a parallel vector of pairs in a pair of parallel vectors:

```
let fst (x,y) = x and snd (x,y) = y
let parpair_of_pairpar vv = (parfun fst vv,parfun snd vv)
```

3.2.2 Communication Functions

The semantics of the total exchange function is given by:

$$\text{totex}\ \langle v_0, \ldots, v_{p-1} \rangle \ = \ \langle\ f\ , \ldots,\ f\ , \ldots,\ f\ \rangle$$

where $\forall i.(0 \leq i < p) \Rightarrow (f\ i) = v_i$. The code is as follows where noSome just removes the Some constructor and compose is usual function 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 denotes the size of the biggest value v held at some process n in words.

From it we can obtain a version which returns a parallel vector of lists:

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

where

$$\left\{\begin{array}{l} (*\textit{val List.map: } (\ \alpha\ \rightarrow\ \beta\ ) \rightarrow\ \alpha\ \textit{list} \rightarrow\ \beta\ \textit{list} *) \\ \text{List.map}\ f\ [v_0; \ldots; v_n] \ = \ [(f\ v_0); \ldots; (f\ v_n)] \\ \text{procs()} \ = \ [0; \ldots; \mathbf{p}()\text{-}1]. \end{array}\right.$$

The semantics of the broadcast is:

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

The direct broadcast function which realizes the broadcast can be written as:

```
(* 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 denotes the size of the value v_n held at process n in words.

The standard library of MSPML contains a collection of such functions which ease the writing of programs. In this respect, it is quite similar to write MSPML programs and to write programs using data-parallel skeletons. But with MSPML it is possible to write one's own skeletons as higher order functions if the standard library does not provide the ones needed.

Some of the standard library functions are recursive functions. For example, there exists a broadcast function which is evaluated in $\log p$ m-steps instead of 1 mstep:

```
let bcast logp root vv =
  let from n =
    mkpar(fun i → let j=natmod (i+(p())−root) (p()) 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
```

3.3 Comparison with BSML

Bulk Synchronous Parallel ML has the same operations than MSPML, but with one difference. The communications (followed immediately by a synchronization barrier) are done using the **put** primitive. It has the following type:

put:(int $\rightarrow$ α option) **par** $\rightarrow$ (int $\rightarrow$ α option) **par**

Consider the expression:

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

To send a value v from process j to process i, the function fs_j at process j must be such as $(\text{fs}_j$ 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.

This primitive can be used to program **get** and **mget** functions. Nevertheless these functions would be far less efficient than the primitives of MSPML: they need two BSP super-steps and thus two synchronization barriers.

Thus for the moment the main difference between BSML programming and MSPML programming is that get-style communications are used in MSPML whether put-style ones are used in BSML. Specific MSPML programming style needs further investigation.

For example the broadcast of the previous section could be written in BSML:

```
let bcast_direct root vv =
  let mkmsg = mkpar(fun i v dst → if i=root
  then Some v else None) in parfun noSome
    (apply (put (apply mkmsg vv)) (replicate root))
```

A BSML and MSPML programs which use only the bcast_direct function for communication would be identical, but their costs will be different. It is thus rather easy to rewrite BSML programs to obtain MSPML programs. The only difficulty is when the BSML program use directly the **put** primitive, instead of functions from the standard library: some more complicated rewriting is needed.

It is not yet possible to compare directly execution times since BSML relies on MPI and MSPML on TCP/IP communications. But both implementations are currently ported using both MPI and TCP/IP to allow thus comparisons. We will send rewrite in MSPML some of the small applications written in BSML: a prototype implementation of the BSP Categorical Abstract Machine [27], a set of functions for tree manipulation [30], as well as the BSML standard library.

4. Overview of the MSPML Implementation

MSPML is implemented as a library for Objective Caml. It is divided in two main parts: the core library which contains the primitives only, and the standard library implemented using the core library. The current core uses only Objective Caml libraries. The communications are done using the TCP/IP protocol available through the Unix module of Objective Caml and the values are serialized using the Marshal module. The current MSPML (0.2) implementation is distributed under a LGPL license and is available at http://mspml.free.fr.

The primitives described in section 3.1 are contained in the Mspml module. They are of course not directly implemented using the Unix and Marshal modules. The Mspml module uses our Tcpip module which offers a very small set of functions similar to MPI's ones (Figure 4) implemented using the Unix module of Objective Caml. The Mspml module is thus written in SPMD style. The Tcpip module is not available for the application programmer who can only use the Mspml module which offer only the functions presented in section 3.1. Using the Mspml module, the programmer cannot write SPMD-like programs and hence avoids the problems which can occur with this programming paradigm.

4.1 The Tcpip Module

p and **pid** respectively gives the total number of processes and the process identifier. **initialize** and **finalize** are similar to MPI_Init and MPI_Finalize. At the beginning of a MSPML execution there are two threads per process: one which corresponds to the local reduction of the distributed semantics (Section 5), used for the "main" program, and one which is created by **initialize** and which is in charge of the communications. This second one answered to requests made by the other processes.

MSPML follows the MPM model (section 2). Thus the execution is a sequence of computation phases and communications phases. Each couple of computation-communication phase constitutes a m-step. During the execution of a MSPML program, at each process i the system has a variable $\mathbf{mstep}_i$ containing the number of the current m-step.

At each m-step each process stores a value in what is called its *communication environment*. A communication environment can be seen as an association list which relates m-step numbers with values. The storage is done using the **store** function. Another process can obtain this value using the **request** function. The argument of the **request** function is the process identifier of the destination of the request.

The size of the communication environments is of course bounded. This size is a parameter given by the user (using the mspmlrun script). The current implementation enforces a global synchronization in order to allow to empty the communication environments when they are full. The **reset_mstep** function increments the $\mathbf{mstep}_i$ variable and checks if the maximal value is reached. If it is the case a global synchronization occurs, the communication environments are emptied and $\mathbf{mstep}_i$ is reset to its initial value. Of course unless a very large amount of data is exchanged at each m-step the maximal value could be rather large which make the synchronization barriers rare. The next version of the library will implement a new management of the communication environments which avoids synchronization barriers [42].

4.2 The Mspml Module

The implementation of the core MSPML library follows the SPMD programming style. Thus for one process the type of parallel vectors is defined by: **type** α **par** = α . Using such an abstract type allows to avoid the problem of data-parallel SPMD programming which does not discriminate between variables which are dependent on the process identifier and those which are not.

p : unit → int
pid : unit → int
finalize : unit → unit
initialize : unit → unit
store_mode = Get | Mget
store : α → **store_mode** → unit
request : int → α
reset_mstep : unit → unit

Figure 4. The Tcpip Module

Then it is easy to implement the primitives which does not need communications:

let mkpar f = f (pid())
let apply f v = f v

The implementation of the **get** primitive could be roughly described as follows. Each time the expression **get** vv vi is evaluated, at a given process i:

1. $\mathbf{mstep}_i$ is increased by one;
2. the value this process holds in parallel vector vv is stored together with the value of $\mathbf{mstep}_i$ in the communication environment;
3. the value j 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 $\mathbf{mstep}_i$. 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:
    - $\mathbf{mstep}_j \geq \mathbf{mstep}_i$: 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 $\mathbf{mstep}_i$ and sends it to process i;
    - $\mathbf{mstep}_j < \mathbf{mstep}_i$: nothing can be done until process j reaches the same m-step than process i.

If $i = j$ the step 2 is of course not performed.

The implementation of **mget** follows the same principles but it uses threads. At a given process i the function of values (first argument of **mget**) is stored in the communication environment. Then the function from integers to booleans (second argument) is applied to integers between 0 and $p-1$. Each time the result is **true**, a new thread is created to request a value from the given process. When all requests are answered, the result function is created. When a process i receives a request from a process j for a value stored by a **mget**, if the m-step of i is equal or greater than the one of j then the value stored in the communication environment of i is deserialized and applied to j to yield to a value v. This value is then serialized and sent to j.

4.3 Nested Parallelism Issues

All this can work only if all processes call the same number of times and in the same order **get**. Incorrect programs could be written when nested parallelism is used:

let this = **mkpar**(**fun** i → i) **in**
mkpar(**fun** i → **if** i<(**p**()/2) **then** this **else get** this this)

It breaks the model because one part of the parallel machine will evaluate an expression with communications and another half will evaluate an expression without communication: the numbering of steps will be no more consistent between processes. This is why it is currently forbidden. A type system can enforce this restriction [25] but currently the programmer is responsible to avoid such nesting. They are easy to detect since the type of the previous expression is int **par par**.

It is also possible to use the **at** functions in other situations than the ones given in section 3.1. But one should avoid the (hidden) nesting of parallel vectors. For example the following expression:

let this = **mkpar**(**fun** i → i)
and com () = **get** this this **in**
mkpar(**fun** i → **if** i<(**p**()/2) **then at** (com()) 0 **else** 1)

is not a correct program (you can write it and compile it with the MSPML library but the execution will fail and the type-checking by our type system [25] fails) because the parallel expression com() would be evaluated *inside* a **mkpar**. As previously it breaks the model, for the same reason: the numbering of steps will not be consistent between processes. For a detailed discussion about these problems (for BSML but it applies also to MSPML), see [25].

It is usually not difficult to obtain a correct program. For example the following program can be safely executed because e is already evaluated when it is used inside the **mkpar**:

let this = **mkpar**(**fun** i → i)
and com () = **get** this this **in**
let e = **at** (com()) 0 **in**
mkpar(**fun** i → **if** i<(**p**()/2) **then** e **else** 1)

5. Formal Semantics

This section is devoted to the formal semantics of MSPML. We first give a high level semantics for MSPML. It is similar to the high level semantics of BSML (but the **get** operator is here a primitive whereas it can be defined in BSML using the **put** primitive). Then we give the distributed minimally synchronous semantics (which is close to the implementation) of MSPML. To simplify the presentation we will only consider **get** and **ifat** communications.

5.1 High Level Semantics

5.1.1 Syntax

Reasoning on the complete definition of a functional and parallel language such as MSPML, would have been complex and tedious. In order to simplify the presentation and to ease the formal reasoning, this section introduces a core language. It is an attempt to trade between integrating the principal features of a functional MPM language and being simple. The syntax of the core MSPML is given by the grammar given in figure 5.

In this grammar, x ranges over a countable set of identifiers. The form $(e'_1\ e'_2)$ stands for the application of a function or an operator e'_1 to an argument e'_2. In the following, we wrote $x{:}\tau$ to say that this identifier could be global ($x{:}\mathcal{G}$, i.e., outside a parallel vector) or local ($x{:}\mathcal{L}$, i.e., inside a parallel vector), i.e, $\tau = \mathcal{L} \mid \mathcal{G}$. Term $\mathbf{fun}\ x : \tau \to e'$ is the functional abstraction, the function whose parameter is x (local or global) and result is given by the value of e'. Constants c are integers, the booleans and the MPM parameters, i.e., p, g, l. The set of operators *op* contains arithmetic operators and fixpoint **fix**, used to defined natural iteration functions and have more expressiveness. **mkpar**, **apply**, **get** and **ifat** are the parallel operators presented in the previous section. **fst** and **snd** are pair operators.

There is one semantics per value of p, the number of processors of the parallel machine (constant during execution). In the following $\forall i$ means $\forall i \in \{0, 1, \ldots, p{-}1\}$. The previous grammar is extended by enumerated parallel vectors:

$$e \quad ::= \quad \ldots \mid \langle e, e, \ldots, e\rangle \quad \text{(parallel vector)}$$

The programmer does not use this new syntax, but the syntax of figure 5, because enumerated parallel vectors are created during evaluation.

Before presenting the dynamic semantics of the language, i.e., how the expressions of MSPML are computed to *values*, we present the values themselves. Those values are defined by the following grammar:

$$\begin{array}{lll} v ::= & c & \text{(constants)} \\ & \mid \mathit{op} & \text{(operators)} \\ & \mid \mathbf{fun}\ x{:}\mathcal{L} \to e & \text{(local functional value)} \\ & \mid \mathbf{fun}\ x{:}\mathcal{G} \to e & \text{(global functional value)} \\ & \mid (v, v) & \text{(pairs of values)} \\ & \mid \langle v, v, \ldots, v\rangle & \text{(enumerated parallel vector)} \end{array}$$

5.1.2 Static analysis

In these syntaxes we do separate local and global identifiers with the $\mathcal{L}$ and $\mathcal{G}$ annotations, contrasting with the BSλ-calculus where the syntax separates local, global identifiers and expressions. To obtain the same distinction for MSPML expressions we introduce in figure 6 and figure 7 a type system which types only well-formed expressions, i.e., a global expressions will never be inside a parallel vector after evaluation.

This is needed to avoid the nesting of parallel vectors which could break the m-step mechanism as explained in section 4.3. The system used here would not be very convenient in practice since the programmer should define several times almost the same function. For example there are several identity functions depending on the sort of the input:

$$\begin{cases} \textbf{let}\ \mathrm{id}_{\mathcal{L}} = & \textbf{fun}\ \mathrm{x}{:}\mathcal{L} \to \mathrm{x} \\ \textbf{let}\ \mathrm{id}_{\mathcal{G}} = & \textbf{fun}\ \mathrm{x}{:}\mathcal{G} \to \mathrm{x} \end{cases}$$

But in practice we rely on the polymorphic type system described in [25] to avoid nesting of parallel values without annotated identifiers, which allow for example to write only one identity function usable on local and global expressions. Nevertheless this polymorphic type system is rather complex and is outside the scope of the present article.

The rules of figures 6 and 7 contains judgments of the form $E \vdash e : \tau$ which means "in the typing environment E, the expression e has type τ". The environment E simply gives the types of the free variables in e. $\{x : \tau\}$ means that the variable x has type τ in this environment. In the following we write $e : \tau$ for $\emptyset \vdash e : \tau$.

To avoid hidden nesting, it is not possible to obtain a local value starting from a global value. In the rules we use the following order on the two kind of types:

$$\mathcal{L} < \mathcal{L} \qquad \mathcal{L} < \mathcal{G} \qquad \mathcal{G} < \mathcal{G}$$

So for example, rule (Fun) enforces the type of the result of the function τ_2 to be greater than the type τ_1 of the input: thus it is impossible to have a global argument (type $\mathcal{G}$) with a local result (type $\mathcal{L}$).

In the following, we note $e_1[x \leftarrow e_2]$ the substitution of the free occurrences of x in e_1 by e_2.

5.1.3 Evaluation rules

First come the rules for the constants, operators and functions:

$$c \rhd c \qquad \mathit{op} \rhd \mathit{op} \qquad (\mathbf{fun}\ x{:}\tau \to e) \rhd (\mathbf{fun}\ x{:}\tau \to e)$$

where τ is $\mathcal{L}$ or $\mathcal{G}$. Then rules for application, binding and pairs:

$$\frac{e_1 \rhd (\mathbf{fun}\ x{:}\tau \to e) \quad e_2 \rhd v_2 \quad e[x \leftarrow v_2] \rhd v \quad e_2 : \tau}{(e_1\ e_2) \rhd v}$$

$$\frac{e_1 \rhd v_1 \quad e_2[x \leftarrow v_1] \rhd v}{\mathbf{let}\ x{:}\tau = e_1\ \mathbf{in}\ e_2 \rhd v} \qquad \frac{e_1 \rhd v_1 \quad e_2 \rhd v_2}{(e_1, e_2) \rhd (v_1, v_2)}$$

Rules for conditional, projection, arithmetic operators and fix-point are also rules which can be found in the semantics of sequential functional programming languages:

$$\frac{e_1 \rhd + \quad e_2 \rhd (n_1, n_2) \quad n = n_1 + n_2}{(e_1\ e_2) \rhd n}$$

$$\frac{e_1 \rhd \mathbf{fix}\ \ e_2 \rhd (\mathbf{fun}\ x{:}\tau \to e_3)\ \ e_3[x \leftarrow \mathbf{fix}(e_2)] \rhd v\ \ e_2 : \tau}{(e_1\ e_2) \rhd v}$$

$$\frac{e_1 \rhd \mathbf{fix} \quad e_2 \rhd \mathit{op}}{(e_1\ e_2) \rhd \mathit{op}}$$

$$\frac{e_1 \rhd \mathbf{true} \quad e_2 \rhd v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v} \qquad \frac{e_1 \rhd \mathbf{false} \quad e_3 \rhd v}{\mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 \rhd v}$$

$$\frac{e \rhd (v_1, v_2)}{\mathbf{fst}\ e \rhd v_1} \qquad \frac{e \rhd (v_1, v_2)}{\mathbf{snd}\ e \rhd v_2}$$

$$\begin{array}{lll|ll}
e' ::= & x & \text{(variables)} & c & \text{(constants)} \\
\mid & \mathbf{fun}\ x{:}\mathcal{L} \to e' & \text{(local abstraction)} & \mathbf{fun}\ x{:}\mathcal{G} \to e' & \text{(global abstraction)} \\
\mid & \mathbf{let}\ x{:}\mathcal{L} = e'\ \mathbf{in}\ e' & \text{(local binding)} & \mathbf{let}\ x{:}\mathcal{G} = e'\ \mathbf{in}\ e' & \text{(global binding)} \\
\mid & op & \text{(operators)} & (e'\ e') & \text{(application)} \\
\mid & (e', e') & \text{(pairs)} & \mathbf{if}\ e'\ \mathbf{then}\ e'\ \mathbf{else}\ e' & \text{(conditional)} \\
\mid & \mathbf{fst}\ e' & \text{(first of a pair)} & \mathbf{snd}\ e' & \text{(second of a pair)} \\
\mid & \mathbf{mkpar}\ e' & \text{(parallel vector)} & \mathbf{apply}\ e'\ e' & \text{(parallel application)} \\
\mid & \mathbf{get}\ e'\ e' & \text{(communication)} & \mathbf{if}\ e'\ \mathbf{at}\ e'\ \mathbf{then}\ e'\ \mathbf{else}\ e' & \text{(global conditional)}
\end{array}$$

Figure 5. Syntax of the core language

$$\frac{}{E \vdash x : E(x)}(Var) \qquad \frac{}{E \vdash c : \mathcal{L}}(Const) \qquad \frac{}{E \vdash op : \mathcal{L}}(Op) \qquad \frac{E \vdash e : \tau}{E \vdash (\mathbf{fix}\, e) : \tau}(Fix)$$

$$\frac{E + \{x : \tau_1\} \vdash e : \tau_2 \quad \text{if } \tau_1 < \tau_2}{E \vdash (\mathbf{fun}\ x{:}\tau_1 \to e) : \tau_2}(Fun)$$

$$\frac{E \vdash e_1 : \tau_1 \qquad E \vdash e_2 : \tau_2 \quad \text{if } \tau_2 < \tau_1}{E \vdash (e_1\ e_2) : \tau_1}(App)$$

$$\frac{E \vdash e_1 : \tau_2 \qquad E + \{x : \tau_1\} \vdash e_2 : \tau_3 \quad \text{if } \tau_1 = \tau_2 \text{ and } \tau_2 < \tau_3}{E \vdash \mathbf{let}\ x{:}\tau_1 = e_1\ \mathbf{in}\ e_2 : \tau_3}(Let)$$

$$\frac{E \vdash e_1 : \tau_1 \qquad E \vdash e_2 : \tau_2 \quad \text{with } \tau_3 = \tau_2 \text{ if } \tau_1 < \tau_2 \text{ else } \tau_3 = \tau_2}{E \vdash (e_1, e_2) : \tau_3}(Pair)$$

$$\frac{E \vdash e_1 : \tau_1 \qquad E \vdash e_2 : \tau_2 \quad \text{if } \tau_2 < \tau_1}{E \vdash \mathbf{fst}\ (e_1, e_2) : \tau_1}(Fst) \qquad \frac{E \vdash e_1 : \tau_1 \qquad E \vdash e_2 : \tau_2 \quad \text{if } \tau_1 < \tau_2}{E \vdash \mathbf{snd}\ (e_1, e_2) : \tau_2}(Snd)$$

$$\frac{E \vdash e_1 : \mathcal{L} \qquad E \vdash e_2 : \tau_2 \qquad E \vdash e_3 : \tau_3 \quad \text{if } \tau_2 = \tau_3}{E \vdash \mathbf{if}\ e_1\ \mathbf{then}\ e_2\ \mathbf{else}\ e_3 : \tau_2}(If)$$

Figure 6. Type system for the functional part

$$\frac{E \vdash e : \mathcal{L}}{E \vdash \mathbf{mkpar}\ e : \mathcal{G}}(Vector)$$

$$\frac{E \vdash e_1 : \mathcal{G} \qquad E \vdash e_2 : \mathcal{G} \qquad \text{where } \mathbf{Par\_op} = \mathbf{apply} \text{ or } \mathbf{get}}{E \vdash \mathbf{Par\_op}\ e_1\ e_2 : \mathcal{G}}(Par)$$

$$\frac{E \vdash e_1 : \mathcal{G} \qquad E \vdash e_2 : \mathcal{L} \qquad E \vdash e_3 : \mathcal{G} \qquad E \vdash e_4 : \mathcal{G}}{E \vdash \mathbf{if}\ e_1\ \mathbf{at}\ e_2\ \mathbf{then}\ e_3\ \mathbf{else}\ e_4 : \mathcal{G}}(IfAt)$$

$$\frac{\forall i (E \vdash e_i : \mathcal{L})}{E \vdash \langle e_0, \ldots, e_{p-1} \rangle : \mathcal{G}}(EVector)$$

Figure 7. Type system for the parallel operators

The unusual rules are for the parallel operators (Fig. 8). They are simply the formalization of the informal description given in section 3.

We have the following results for this semantics.

Proposition 1 (Safety of the type system) *Let e be a MSPML expression and E a typing environment. If $E \vdash e : \tau$ and $e \triangleright v$ then $E \vdash v : \tau$*

Proposition 2 (Confluence of the semantics) *Let e be a closed well-formed MSPML expression. If $e \triangleright v_1$ and $e \triangleright v_2$ then $v_1 = v_2$.*

The proofs are done by induction on terms. The full text is given in [43].

5.2 Distributed semantics

The high-level semantics does not give the steps of the computation but only the result. Thus all parallel operators seem to be synchronous in this semantics. To show how desynchronizing is handled in MSPML, a distributed semantics, which gives the steps of a reduction towards a value, is needed.

Distributed evaluation $\rightarrow$ can be defined in two steps:

1. local reduction (performed by one process i) $\rightharpoonup_i$
2. global reduction $\rightharpoonup$ of distributed terms which allows the evaluation of communication requests (for **get** and **ifat**).

5.2.1 Syntax

For the programmer, the syntax is almost the same as the syntax of the previous section, but it is to notice that each process will hold the same program (or that the program for the parallel machine is built with p copies of the same program) whereas in the previous section it was a program for the whole parallel machine. This syntax and semantics is thus closer to the implementor's view. As in the previous section we need to define new terms which may be created during evaluation:

$$\begin{array}{rcl} e_d & ::= & x \mid c \mid op \mid (e_d\, e_d) \mid (e_d, e_d) \\ & \mid & \textbf{fun}\ x{:}\mathcal{L} \rightarrow e_d \mid \textbf{fun}\ x{:}\mathcal{G} \rightarrow e_d \\ & \mid & \textbf{let}\ x{:}\mathcal{L} = e_d\ \textbf{in}\ e_d \mid \textbf{let}\ x{:}\mathcal{G} = e_d\ \textbf{in}\ e_d \\ & \mid & \textbf{fst}\ e_d \mid \textbf{snd}\ e_d \mid \textbf{if}\ e_d\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\ & \mid & \textbf{mkpar}\ e_d \mid \textbf{apply}\ e_d\, e_d \mid \textbf{get}\ e_d\, e_d \\ & \mid & \textbf{if}\ e_d\ \textbf{at}\ e_d\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\ & \mid & \textbf{request}\ e_d\, e_d \mid \overrightarrow{e_d} \end{array}$$

The syntax is almost the same with two differences: a new **request** operation is used and the enumerated parallel vector is replaced by $\overrightarrow{e_d}$ which is simply the projection on one process of an enumerated parallel vector, ie the value held by this process in this parallel vector.

The type system could be slightly modified to be applicable to the e_d terms: **request** is typed like **get** and $\overrightarrow{e_d}$ like **mkpar**. We omit this type system.

5.2.2 Rules

The distributed semantics follows the SPMD paradigm. For example at process i the expression **mkpar** f will be reduced to $f\ i$. **request** is used to allow the evaluation of the **get** operation without having a global synchronization. At each step of communication (a call to **get** or **ifat**), called a *m-step*, each process stores the number of the m-step (each process performs the same number of m-steps thus this numbering can be done locally) and the value it holds: for **get** this value is the first argument of **get** and also for **ifat**. Those pairs are stored into a *communication environment* (one per process) $\mathcal{E}_C$. Those environments can be thought as associative lists. Those environments evolved asynchronously during execution and to know at which m-step is a process, we will use the **mstep** function defined by:

$$\begin{cases} \textbf{mstep}([]) = 0 \\ \textbf{mstep}((n, v_d) :: \mathcal{E}_C) = n. \end{cases}$$

Now when a process i evaluates **get** $v\ j$, it adds the pair $(\textbf{mstep}(\mathcal{E}_C) + 1\ ,\ v)$ to the communication environment $\mathcal{E}_C$ and then it asks the value held by the communication environment of process j at the current m-step ($n = \textbf{mstep}(\mathcal{E}_C) + 1$). This asking is formally written: **request** $n\ j$. The local reduction can create **request** expressions but it cannot make them disappear: this can be done only at the global level. The values for local reduction are:

$$\begin{array}{rcl} v_d & ::= & c \mid op \mid (v_d, v_d) \mid \overrightarrow{v_d} \\ & & \mid \textbf{fun}\ x{:}\mathcal{L} \rightarrow e_d \mid \textbf{fun}\ x{:}\mathcal{G} \rightarrow e_d \end{array}$$

request expressions are *not* values.

Local reduction is a relation between pairs of expressions e_d and communication environments. First we begin with axioms for head reduction $(e_d, \mathcal{E}_C) \overset{\varepsilon}{\rightharpoonup}_i (e'_d, \mathcal{E}'_C)$. It can be read as "Expression e_d in communication environment $\mathcal{E}_C$ is reduced to expression e'_d in environment $\mathcal{E}'_C$, at process i". Figure 9 presents the head reduction for the functional part of the language and Figure 10 for the parallel one.

Those rules cannot be applied in any context. To have a weak (which means that no evaluation is allowed under an abstraction) call-by-value strategy, the following contexts are needed:

$$\begin{array}{rcl} \Gamma & ::= & \bullet \mid \Gamma\, e_d \mid v_d\, \Gamma \mid (\Gamma, e_d) \mid (v_d, \Gamma) \mid \overrightarrow{\Gamma} \\ & \mid & \textbf{fst}\ \Gamma \mid \textbf{snd}\ \Gamma \mid \textbf{let}\ x{:}\tau = \Gamma\ \textbf{in}\ e_d \\ & \mid & \textbf{if}\ \Gamma\ \textbf{then}\ e_d\ \textbf{else}\ e_d \mid \textbf{mkpar}\ \Gamma \\ & \mid & \textbf{apply}\ \Gamma\, e_d \mid \textbf{apply}\ v_d\, \Gamma \\ & \mid & \textbf{get}\ \Gamma\, e_d \mid \textbf{get}\ v_d\, \Gamma \\ & \mid & \textbf{if}\ \Gamma\ \textbf{at}\ e_d\ \textbf{then}\ e_d\ \textbf{else}\ e_d \\ & \mid & \textbf{if}\ v_d\ \textbf{at}\ \Gamma\ \textbf{then}\ e_d\ \textbf{else}\ e_d \end{array}$$

$\bullet$ is a "hole" which may be filled by any distributed expression. These contexts are used together with the context rule:

$$\frac{(e_d, \mathcal{E}_C) \overset{\varepsilon}{\rightharpoonup}_i (e'_d, \mathcal{E}'_C)}{(\Gamma[e_d], \mathcal{E}_C) \rightharpoonup_i (\Gamma[e'_d], \mathcal{E}'_C)}$$

$$\frac{e_1 \triangleright \langle v'_1, v'_2, \ldots, v'_{p-1}\rangle \quad e_2 \triangleright \langle v''_0, v''_1, \ldots, v''_{p-1}\rangle \quad \forall i (v'_i\ v''_i) \triangleright v_i}{\textbf{apply}\ e_1\ e_2 \triangleright \langle v_0, v_1, \ldots, v_{p-1}\rangle}$$

$$\frac{e_1 \triangleright v \quad \forall i\ (v\ i) \triangleright v_i}{\textbf{mkpar}\ e_1 \triangleright \langle v_0, \ldots, v_{p-1}\rangle} \qquad \frac{e_1 \triangleright \langle v_0, v_1, \ldots, v_{p-1}\rangle \quad e_2 \triangleright \langle i_0, i_1, \ldots, i_{p-1}\rangle}{\textbf{get}\ e_1\ e_2 \triangleright \langle v_{i_0\%p}, \ldots, v_{i_{p-1}\%p}\rangle}$$

$$\frac{e_1 \triangleright \langle \ldots, \overbrace{\textbf{true}}^{n}, \ldots\rangle \quad e_2 \triangleright n \quad e_3 \triangleright v_3}{\textbf{if}\ e_1\ \textbf{at}\ e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4 \triangleright v_3} \qquad \frac{e_1 \triangleright \langle \ldots, \overbrace{\textbf{false}}^{n}, \ldots\rangle \quad e_2 \triangleright n \quad e_4 \triangleright v_4}{\textbf{if}\ e_1\ \textbf{at}\ e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4 \triangleright v_4}$$

Figure 8. Rules for parallel operators

$$\begin{array}{llll}
((\textbf{fun}\ x{:}\tau \to e_d)\ v_d, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (e_d[x \leftarrow v_d], \mathcal{E}_C) \text{ with } v_d : \tau & (\beta_{fun}) \\
((\textbf{let}\ x{:}\tau = v_d\ \textbf{in}\ e_d), \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (e_d[x \leftarrow v_d], \mathcal{E}_C) & (\beta_{let}) \\
(+\ (n_1, n_2), \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (n, \mathcal{E}_C) \text{ with } n = n_1 + n_2 & (\delta_+) \\
\textbf{fst}\ (v_{d_1}, v_{d_2}, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (v_{d_1}, \mathcal{E}_C) & (\delta_{fst}) \\
\textbf{snd}\ (v_{d_1}, v_{d_2}, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (v_{d_2}, \mathcal{E}_C) & (\delta_{snd}) \\
(\textbf{fix}\ (\textbf{fun}\ x{:}\tau \to\ e_d), \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (e_d[x \leftarrow \textbf{fix}(\textbf{fun}\ x{:}\tau \to\ e_d)], \mathcal{E}_C) \text{ with } e_d : \tau & (\delta_{fix}) \\
(\textbf{fix}\ (\textbf{op}), \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (\textbf{op}, \mathcal{E}_C) & (\delta_{fixop}) \\
(\textbf{if true then}\ e_1\ \textbf{else}\ e_2), \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (e_1, \mathcal{E}_C) & (\delta_{ift}) \\
(\textbf{if false then}\ e_1\ \textbf{else}\ e_2), \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (e_2, \mathcal{E}_C) & (\delta_{iff})
\end{array}$$

Figure 9. Functional local reduction

$$\begin{array}{llll}
(\textbf{mkpar}\ v_d, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (\overrightarrow{(v_d\ i)}, \mathcal{E}_C) & (\delta_{mkpar}) \\
(\textbf{apply}\ \overrightarrow{v_{d_1}}\ \overrightarrow{v_{d_2}}, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (\overrightarrow{(v_{d_1}\ v_{d_2})}, \mathcal{E}_C) & (\delta_{apply}) \\
(\textbf{get}\ \overrightarrow{v_d}\ \overrightarrow{j}, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (\overrightarrow{\textbf{request}\ (\textbf{mstep}(\mathcal{E}_C) + 1)\ j}, & (\delta_{get}^{dst}) \\
 & & (\textbf{mstep}(\mathcal{E}_C) + 1, v_d) :: \mathcal{E}_C) \text{ if } j \neq i & \\
(\textbf{get}\ \overrightarrow{v_d}\ \overrightarrow{i}, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (\overrightarrow{v_d}, (\textbf{mstep}(\mathcal{E}_C) + 1, v_d) :: \mathcal{E}_C) & (\delta_{get}^{loc}) \\
(\textbf{if}\ \overrightarrow{b}\ \textbf{at}\ n\ \textbf{then}\ v_1\ \textbf{else}\ v_2, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (\textbf{if}\ (\textbf{request}\ (\textbf{mstep}(\mathcal{E}_C) + 1)\ n) & (\delta_{ifat}^{dst}) \\
 & & \textbf{then}\ v_1\ \textbf{else}\ v_2, (\textbf{mstep}(\mathcal{E}_C) + 1, b) :: \mathcal{E}_C) & \\
 & & \text{if } n \neq i & \\
(\textbf{if}\ \overrightarrow{b}\ \textbf{at}\ i\ \textbf{then}\ v_1\ \textbf{else}\ v_2, \mathcal{E}_C) & \overset{\varepsilon}{\rightharpoonup}_i & (\textbf{if}\ b\ \textbf{then}\ v_1\ \textbf{else}\ v_2, & \\
 & & (\textbf{mstep}(\mathcal{E}_C) + 1, b) :: \mathcal{E}_C) & (\delta_{ifat}^{loc})
\end{array}$$

Figure 10. Parallel local reduction

Informally the contexts enforces the evaluation of the arguments before allowing the reduction, the branches of conditionals are not evaluated, and nothing is evaluated under an abstraction.

Distributed expressions are p-wide tuples of pairs of local expressions and communication environments: $\langle\langle(e_{d_0},\mathcal{E}_{C_0}),(e_{d_1},\mathcal{E}_{C_1}),\ldots,(e_{d_{p-1}},\mathcal{E}_{C_{p-1}})\rangle\rangle$.
Distributed values are:
$\langle\langle(v_{d_0},\mathcal{E}_{C_0}),(v_{d_1},\mathcal{E}_{C_1}),\ldots,(v_{d_{p-1}},\mathcal{E}_{C_{p-1}})\rangle\rangle$.

The distributed semantics consists of a predicate between p-wide tuples and another p-wide tuples defined by a set of axioms and rules called steps. This semantics describes all the steps of the language from the p-wide tuples to distributed values. We note $\rightharpoonup$ for one of these steps and $\overset{*}{\rightharpoonup}$, for the transitive closure of $\rightharpoonup$.

The rules for global reduction are given in figure 11. If process i requests the value held by process j at m-step n ($\mathbf{request}\ n\ j$) and the communication environment $\mathcal{E}_{C_j}$ of process j contains the value v_d at m-step n then the value v_d is sent to process i. Otherwise the rule cannot be applied: this means that if process j has not yet reached the n^{th} m-step, then process i must wait.

The high level semantics and the lower level one are equivalent:

Theorem 1 *(Equivalence) Let e be a closed well-formed MSPML expression.*

$$e \triangleright v \quad \textit{iff} \quad \mathcal{C}(e) \overset{*}{\rightharpoonup} \mathcal{P}(v)$$

where $\mathcal{C}(e) = \langle\langle(e_{d_0},\emptyset),(e_{d_1},\emptyset),\ldots,(e_{d_{p-1}},\emptyset)\rangle\rangle$ and $\emptyset$ is the empty communication environment and $\mathcal{P}(e) = \langle\langle\mathcal{P}_0(e),\mathcal{P}_1(e),\ldots,\mathcal{P}_{p-1}(e)\rangle\rangle$ and $e_{d_i} = \mathcal{P}_i(e)$ and where $\mathcal{P}_i$ is an trivial induction projection of our expressions to distributed expressions where $\mathcal{P}_i(\langle e_0,\ldots,e_i,\ldots,e_{p-1}\rangle) = \overrightarrow{e_i}$.

The full proof is in [43].

Example 1 *For the broadcast example, with $p = 3$, distributed evaluation of*

$$\textit{bcast}\ 2\ (\mathbf{mkpar}(\mathbf{fun}\ x{:}\mathcal{L} \rightarrow\ 2\times x))$$

begins with local reduction at each process. At process i, local reduction is given in figure 12. Then global reduction is used:

$$\begin{array}{ll} & \langle\langle \\ & \quad (\mathbf{request}\ 0\ 2,[(0,0)]), \\ & \quad (\mathbf{request}\ 0\ 2,[(0,2)]), \\ & \quad (\mathbf{request}\ 0\ 2,[(0,4)]) \\ & \rangle\rangle \\ \overset{3}{\rightharpoonup} & \langle\langle(4,[(0,0)]),(4,[(0,2)]),(4,[(0,4)]),\rangle\rangle \end{array}$$

6. Experiments

We first present additional functions from the standard library: implementations of reduction and parallel prefix algorithms. Then we use them to implement the Diffusion algorithmic skeleton [35]. This skeleton is then used to write a toy application: the smaller elements program. Execution times have been measured for this example on a cluster of PC.

6.1 Reduction and Parallel Prefix

There are several versions of parallel reduction. One can reduce a parallel vector of type α **par** but also a parallel vector of type α collection **par** where collection could for example be list or array. In figure 14, fold_direct is a parallel reduction on a parallel vector of type α **par**. It is called direct because the values are exchanged between process in a single m-step. The cost of the communication phase is $s\times(p-1)\times g\ +\ L$ where s is the size of the biggest value in the parallel vector. Depending on the parameters s, p, g and L it could be more efficient to use $\log_2 p$ phases with a cost of $s\times g + L$ (for the sake of conciseness and simplicity we will consider only(direct algorithms in this paper).

The parallel reduction of a parallel vector of lists can be easily defined using fold_direct. We could of course have defined a generic parallel reduction of parallel vector of collections which allows to use a direct fold or not, depending on the parameters:

```
let generic_wide_fold sfold fold op neutral vv =
  let local_fold = parfun (sfold op neutral) vv in
  fold op neutral local_fold
```

where sfold is a sequential reduction over the collection and fold a parallel fold of α **par** parallel vectors. Then it could be instantiated:

```
let wide_fold_list_direct op e vv=
  generic_wide_fold List.fold_left fold_direct op e vv
```

This could also be done for parallel prefix algorithms. generic_scan (Figure 14) is a generic operation which could compute the sum of prefixes of any parallel vectors of some collection of indexed elements. The collections could be lists, arrays or any indexed collection of elements (for example unary functions from integers to something else). Thus the inferred type of this function is quite complex because it is very general. In Figure 14, the type is presented in such a way that each line is the type of one of the argument, for example $((\alpha\rightarrow\beta)\rightarrow\gamma\rightarrow\alpha_1\rightarrow\alpha_2)$ is the inferred type of the argument sscan. The last line gives the type of the arguments op, neutral, vec and the type of the result.

generic_scan operates on a parallel vector of collections of indexed elements. It depends on:

- the data structure used for the collection of elements;
- the sequential scan on this collection of elements;
- the parallel prescan on vectors of this kind of elements (not parallel vector of collections). The semantics of prescan is given by:

$$\text{prescan}\ \oplus\ \langle v_0,\ldots,v_{p-1}\rangle = \langle \imath_\oplus, v_0,\ldots,\oplus_{k=0}^{p-2} v_k\rangle$$

  where $\imath_\oplus$ is the neutral for the binary operation $\oplus$;

$$\frac{(e_{d_i},\mathcal{E}_{C_i}) \rightharpoonup_i (e'_{d_i},\mathcal{E}'_{C_i})}{\langle\langle(e_{d_0},\mathcal{E}_{C_0}),\ldots,(e_{d_i},\mathcal{E}_{C_i}),\ldots,(e_{d_{p-1}},\mathcal{E}_{C_{p-1}})\rangle\rangle \rightharpoonup \langle\langle(e_{d_0},\mathcal{E}_{C_0}),\ldots,(e'_{d_i},\mathcal{E}'_{C_i}),\ldots,(e_{d_{p-1}},\mathcal{E}_{C_{p-1}})\rangle\rangle}$$

$$\frac{(e_{d_i} = \Gamma[\mathbf{request}\; n\; j]) \wedge ((n,v_d)\in\mathcal{E}_{C_j})}{\langle\langle(e_{d_0},\mathcal{E}_{C_0}),\ldots,(e_{d_i},\mathcal{E}_{C_i}),\ldots,(e_{d_{p-1}},\mathcal{E}_{C_{p-1}})\rangle\rangle \rightharpoonup \langle\langle(e_{d_0},\mathcal{E}_{C_0}),\ldots,(\Gamma[v_d],\mathcal{E}_{C_i}),\ldots,(e_{d_{p-1}},\mathcal{E}_{C_{p-1}})\rangle\rangle}$$

Figure 11. Global reduction

$$\begin{array}{ll}
 & \big(\mathbf{get}\ (\mathbf{mkpar}(\mathbf{fun}\ x{:}\mathcal{L}\to\ 2\times x))\ (\mathbf{mkpar}(\mathbf{fun}\ x{:}\mathcal{L}\to\ 2))\ \ ,\ \ []\big)\\
\rightharpoonup_i & \big(\mathbf{get}\ ((\mathbf{fun}\ x{:}\mathcal{L}\to\ 2\times x)\ i)\ (\mathbf{mkpar}(\mathbf{fun}\ x{:}\mathcal{L}\to\ 2))\ \ ,\ \ []\big)\\
\rightharpoonup_i & \big(\mathbf{get}\ 2i\ (\mathbf{mkpar}(\mathbf{fun}\ x{:}\mathcal{L}\to\ 2))\ \ ,\ \ []\big)\\
\rightharpoonup_i & \big(\mathbf{get}\ 2i\ ((\mathbf{fun}\ x{:}\mathcal{L}\to\ 2)\ i)\ \ ,\ \ []\big)\\
\rightharpoonup_i & \big(\mathbf{get}\ 2i\ 2\ \ ,\ \ []\big)\\
\rightharpoonup_i & \big(\mathbf{request}\ 0\ 2\ \ ,\ \ [(0,2i)]\big)
\end{array}$$

Figure 12. Example

- how to map on this kind of collection;
- how to take (last) and remove (cutlast) the last element of this kind of collection.

Figure 14 shows a parallel prescan but it is more simple to implement as follows:

let prescan_direct op neutral v =
  let com = get_list v (**mkpar**(**fun** i → from_to 0 (i−1))) **in**
  parfun (List.fold_left op neutral) com

using the get_list communication primitive defined by:

$$\begin{array}{ll}
\text{get\_list} & \langle v_0,\ldots,v_{p-1}\rangle\ \langle\ldots,\ [i_1^j;\ldots;i_{k_j}^j],\ \ldots\rangle\\
= & \langle\ [v_{i_1^0};\ldots;v_{i_{k_0}^0}]\ ,\ldots,\ [v_{i_1^{p-1}};\ldots;v_{i_{k_{p-1}}^{p-1}}]\ \rangle
\end{array}$$

and where:

$$\text{from\_to}\ n_1\ n_2\ =\ [n_1;n_1+1;\ldots;n_2]$$

Now let explains how generic_scan works. To ease the presentation we consider that the collections are lists. Thus the sequential functions (sscan, cutlast and last) given in argument should satisfy the equalities of figure 13 (map has been defined in the previous section).

If vv is the parallel vector:

$$\langle\ [x_1;x_2;\ldots;x_{n/p}]\ ,\ldots,\ [x_{n+1-n/p};x_{n+2-n/p};\ldots;x_n]\ \rangle$$

then local_scan is the following vector:

$$\begin{array}{l}\langle\ \ [\text{c}; \text{op2 c}\,x_1;\ldots;\text{op2}\,(\ldots)\,x_{n/p}],\ \ldots,\\ \quad[\text{c}; \text{op2 c}\,x_{n+1-n/p};\ldots;\text{op2}\,(\ldots)\,x_{n/p}]\ \rangle\end{array}$$

Each process now holds $\frac{n}{p}+1$ values. For each process (except the last one) we need to remove the last value of the collection. The tmp vector is a parallel vector of pairs. The first component is $(\text{Some}\ v_i)$ where v_i was the last element of the list and the second component is the list without this last element. At process $p-1$ the second component is the unchanged list. From this vector we obtain last_elements the parallel vector composed with the last elements and new_lists the parallel vector of lists without their last elements. values_to_add computes the partial reductions of the last_elements vector. At process i, only the first $i-1$ values will be reduced using prescan. To end the values obtained are added to the new_lists parallel vector of lists.

scanl (figure 14) is an example of instantiation of the generic parallel scan function.

6.2 The Diffusion Algorithmic Skeleton

Algorithmic skeleton languages [15, 47, 48], in which only a finite set of operations (the skeletons) are parallel, constitute from the programmer's point of view an easy approach to parallel programming. The set of algorithmic skeletons has to be as complete as possible but it is often dependent on the domain of application.

We show here than it is possible to implement algorithmic skeletons in MSPML. We choose to implement the Diffusion skeleton [1]. It is derived from the *Diffusion Theorem* [35] and is defined in terms of classical primitive skeletons **map**, **reduce** and **scan**. It provides a good abstraction of a combination of parallel primitives. Using the diffusion theorem, recursive functions defined in a specific form and under some conditions, could be expressed as an instantiation of the diffusion skeleton.

Another advantage is that under some conditions, some deforestation-like techniques could be used to replace a composition of several diffusion skeleton instantiations by only one diffusion skeleton instantiation [33].

The Diffusion Skeleton using the Bird-Meertens Formalism (BMF) [7, 55] is defined by:

$$\begin{array}{ll} & \mathbf{diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ xs\ c\\ = & \mathbf{reduce}\ (\oplus)\ (\mathbf{map}\ k\ as)\ \oplus\ g_1\ b\end{array}$$

where

$$\begin{cases} bs +\!\!\!+ [b] & = \mathbf{map}\ (c\otimes)\ (\mathbf{scan}\otimes(\mathbf{map}\ g_2\ xs))\\ as & = \mathbf{zip}\ xs\ bs\end{cases}$$

$$\begin{aligned}
\text{sscan } (\oplus)\ [x_1, x_2, \ldots, x_n] &= [\imath_\oplus; x_1; x_1 \oplus x_2; \ldots; x_1 \oplus x_2 \oplus \ldots \oplus x_n] \\
\text{cutlast } [x_1; \ldots; x_n] &= (\text{Some } x_n,\ [x_1; \ldots; x_{n-1}]) \\
\text{cutlast } [\,] &= (\text{None},\ [\,]) \\
\text{last } [x_1; \ldots; x_n] &= (\text{Some } x_n,\ [x_1; \ldots; x_n]) \\
\text{last } [\,] &= (\text{None},\ [\,])
\end{aligned}$$

Figure 13. Sequential functions needed for parallel scan

```
(* val fold_direct: ( α → β → α ) → α → β par → α par *)
let fold_direct op neutral vv =
  parfun (List.fold_left op neutral) (totex_list vv)

(* val wide_fold_list_direct: ( α → α → α ) → α → α list par → α list par *)
let wide_fold_list_direct op neutral vv =
  let local_fold = parfun (List.fold_left op neutral) vv in
  fold_direct op neutral local_fold

(* val prescan_direct : ( α → β → α ) → α → β par → α par *)
let prescan_direct op neutral v =
  let com = mget (parfun (fun v i → v) v) (mkpar(fun i j → j<i))
  and sfold = List.fold_left op neutral
  and lists = mkpar(fun i → from_to 0 (i−1)) in
  parfun sfold (parfun2 List.map (parfun (compose noSome) com) lists)

(* val generic_scan :
  (( α → β ) → γ → α1 → α2 ) →
  (( α → β ) → γ → β1 par → α par) →
  ( β → β2 → γ1 ) →
  ( α2 → β1 option * β2 ) →
  ( α2 → β1 option * β2 ) →
  ( α → β ) → γ → α1 par → γ1 par *)
let generic_scan sscan prescan map last cutlast op neutral vv =
  let local_scan = parfun (sscan op neutral) vv in
  let tmp = applyat (p()−1) last cutlast local_scan in
  let last_elements = parfun (compose noSome fst) tmp
  and new_lists = parfun snd tmp in
  let values_to_add = prescan op neutral last_elements in
    parfun2 map (parfun op values_to_add) new_lists

(* val scanl: ( α → α → α ) → α → α list par → α list par *)
let scanl op e =
  generic_scan sscan prescan_direct List.map last cutlast op e
```

Figure 14. Parallel reduction, parallel prefix

and $\oplus$ and $\otimes$ are associative operations with units, $+\!\!+$ is the concatenation of lists and where

$$\begin{array}{rl} & \textbf{reduce}\ (\oplus)\ [x_1, x_2, \ldots, x_n] \\ = & x_1 \oplus x_2 \oplus \ldots \oplus x_n \\ & \textbf{zip}\ [x_1, x_2, \ldots, x_n]\ [y_1, y_2, \ldots, y_n] \\ = & [(x_1, y_1), (x_2, y_2, \ldots, (x_n, y_n)] \end{array}$$

The implementation of the skeleton in MSPML is given in figure 15. It uses only functions described in previous sections. One could notice than we could have provided a generic diff function. The function used for the tests is in fact such a function. We present here a function similar to the instantiation of the generic function used for the experiments.

6.3 Smaller Elements

We performed some experiments with the ‘smaller elements’ program. The aim of this program is to remove from a list all the elements which are lesser than elements which are placed before them in the list.

The function which computes this list could be written using the diffusion skeleton:

```
let se_diff xs =
  let k x c = if x<c then [] else [x]
  and g1 x = []
  and g2 x = x in
  diff (@) [] max k g1 g2 xs min_int
```

All the operations used here have a constant complexity: this is obvious for k, g1, g2 and max. The concatenation of lists (@) is linear in its first argument. It is used in diff as an argument for a parallel fold: reducer. Its first argument in this case is the result of an application of k: an empty list or a list of size 1. Thus the sequential time required for the execution of se_diff depends linearly on the size of the input list xs (which is in fact a parallel vector of lists but which should be seen as the list obtained by concatenation of the lists held by all the processes).

The communications are produced first by scanl, at process i the cost is $s \times i \times g + L$ (where s is the size of one element), and by reducer, a direct fold which uses a total exchange which costs $s \times (p-1) \times g + L$. This total exchange induces a global synchronization thus the total cost is $(s \times (p-1) + p) \times g + 2 \times L$. In our experiments s increases linearly with the size.

We ran the se_diff program on a cluster of 2, 4 and 8 processes, with lists of size between 200 and 20000 *per process*. Each machine is a Pentium III (1Ghz) with 128 Mo of RAM running Linux Mandrake Clic (http://clic.mandrakesoft.com). The network is a fast Ethernet network. Each test was performed 4 times and the average was taken. Each test consisted of the execution of 10 calls to se_diff on the same list. We used the Objective Caml byte-code compiler (in order to compare with some architectures on which the Objective Caml native code compiler is unavailable).

Figure 16a shows the time in seconds for the execution on the input list of total size given in abscissa whereas figure 16b shows the execution time on input list with the size of the sublist held *by each process*.

These figures show than the obtained curves are linear ones. We obtain a super-linear speed-up when the number of processes increase. This is not a surprise since with less processes, each process holds more data and garbage collection occurs more often. What is more surprising is shown in figure 16b: for the same size per process the execution time is a bit greater when less processes are involved.

This phenomena is due to the buffering strategy of the Linux TCP/IP layer: the buffers are flushed as soon as they are full but there is a delay for sending buffers which are not full. Thus with less processors, fewer messages are sent and they may not fill the buffers: there is a delay before the messages are actually sent.

7. Related Work

As shown in the previous section, MSPML can be used to implement data-parallel skeletons. We chose the Diffusion algorithmic skeleton, but other could be implemented as for example [18]. The advantage is to have in the same language the possibility to use skeletons and to write them. [3] has demonstrated that NESL [10] is more efficient when the size of the vectors is constant. Even if it is not the case, most of the operations of NESL could be implemented in MSPML. In particular nested lists could be implemented as shown in [34]. From this point of view MSPML could seem lower-level than NESL. But MSPML offers higher-order functions while NESL is a first-order language.

There are several works on extension of the BSPlib library or libraries to avoid synchronization barrier [20, 2, 37] which rely on different kind of messages counting. To our knowledge the only extension to the BSPlib standard which offers zero-cost synchronization barriers and which is available for downloading is the PUB library [11]. The oblivious synchronization function **bsp_oblsync** takes as argument the number of messages that must be received by the process at the given super-step: when the process has received this number of message it begins the next super-step without synchronizing with other processes.

Caml-flight, a functional parallel language [21, 13], relies on the wave mechanism. A **sync** primitive is used to indicated which processes could exchange messages using a **get** primitive which is very different from ours: this primitive asks the remote evaluation of an expression given as argument. This mechanism is more complex than ours and there is no pure functional high level semantics for Caml-flight [22]. Moreover Caml-flight programs are SPMD programs which are more difficult to write and read. The type system to avoid incorrect nested parallelism is also complex [60].

[49] describes the mechanism of *structural clocks* to allow a minimally synchronous execution of data-parallel programs written in a small imperative language in SPMD style. The

```
let diff op1 op1neutral op2 k g1 g2 xs c =
  let reducer op neutral e vec =
    let local_fold = parfun (List.fold_left op neutral) vec in
    fold_direct op e local_fold in
  let bs'=scanl op2 c (parfun (List.map g2) xs) in
  let nocut l = None,l in
  let b',bs=parpair_of_pairpar(applyat (p()−1) cutlast nocut bs') in
  let (Some b)=at b' (p()−1) in
  reducer op1 op1neutral (g1 b) (parfun2 (List.map2 k) xs bs)
```

Figure 15. MSPML Implementation of the Diffusion Skeleton

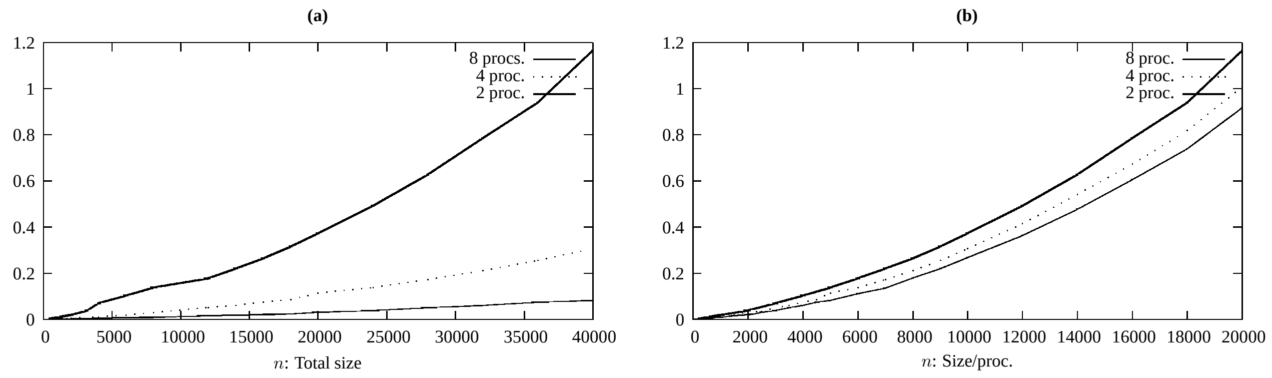

Figure 16. Smaller elements

difficulty is this framework is that the number of communication phases may be different at each process, because an operator of parallel composition is provided. We will also need a more complex m-step numbering which may be similar to the numbering used in structural clocks, when we will add parallel composition to MSPML. This operation allows to divide the parallel machine in two independent sub-machines. The high level semantics of the parallel composition for MSPML will be the same as the semantics of parallel juxtaposition for BSML [40].

8. Conclusions and Future Work

Minimally Synchronous Parallel ML is a functional parallel language which shares its syntax and high-level semantics with Bulk Synchronous Parallel ML but which has a new lower level semantics and implementation. Communications do not need global synchronization barriers. The Message Passing Machine cost model can be applied to MSPML. The first experiments with our prototype implementation show the applicability of the cost model.

Future work can be divided into four parts:

- for the moment MSPML is a library for the Objective Caml language and it uses the threads facilities and the Unix module for TCP/IP communications. We plan to write also an MPI version to compare MSPML with the BSMLlib library. The fourth version (0.2) of MSPML has been released in june 2004;
- management of the communication environments: we proposed a new mechanism for the management of communication environments which would avoid global synchronization to empty the communication environment [42]. We will prove its correctness using Abstract State Machines [29] and First Order Timed Logic [5] using model-checking tools;
- extension of the language with new constructs:
  - extension of MSPML with a parallel juxtaposition which allows to divide the machine in two distinct parallel machines which evaluate two MSPML expressions in parallel. With this primitive the number of communication phases may be different on each process. Thus a new mechanism of communication environment must be designed;
  - extension of MSPML to allow the nesting of parallel vectors;
- automatic cost estimates. Two research directions will be followed. Either in relation with our work on the verification of parallel program using the Coq proof assistant. [6] is such an approach for sequential programs and the proof assistant Nuprl. Or directly as in [50] for the sequential case.

All the work done on MSPML will be also used in Departmental Metacomputing ML (DMML) [26] designed to program clusters of parallel machines and which is based on a two-tiered model: the BSP model for each parallel unit and the MPM model for coordinating this heterogeneous set of BSP units. The upper level of DMML is similar to MSPML.

Acknowledgments This work is supported by a grant from the French Ministry of Research and by the ACI Grid program, under the project CARAML (http://www.caraml.org). The authors wish to thank the anonymous referees of this special issue, the anonymous referees and the organizers of the workshop "High-Level Approaches to Parallel and Distributed Computing" and of the 4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03).

9. References

[1] S. Adachi, H. Iwasaki, and Z. Hu. Diff: A Powerfull Parallel Skeleton. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume 4, pages 425–527. CSREA Press, 2000.

[2] R. Alpert and J. Philbin. cbsp: Zero-cost synchronization in a modified bsp model. Technical Report 97-054, NEC Research Institute, 1997.

[3] M. Bamha. L'implémentation d'un langage portable à parallélisme emboîté en processus statiques. Mémoire de DEA d'informatique, LIFO, Université d'Orléans, Septembre 1996.

[4] M. Bamha, F. Bentayeb, and G. Hains. An efficient scalable parallel view maintenance algorithm for shared nothing multi-processor machines. In T. Bench-Capon, G. Soda, and A. Min Tjoa, editors, *10th International Conference on Database and Expert Systems Applications, DEXA'99*, number 1677 in LNCS, pages 616–625. Springer-Verlag, August 30 – September 3 1999.

[5] D. Beauquier and A. Slissenko. A first order logic for specification of timed algorithms: basic properties and a decidable class. *Annals of Pure and Applied Logic*, 113, 2002.

[6] R. Benzinger. Automated higher-order complexity analysis. *Theoretical Computer Science*, 318:79–103, 2004.

[7] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

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

[9] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.

[10] 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.

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

[12] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999.

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

[14] 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.

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

[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] F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999.

[18] R. Di Cosmo and S. Pelagatti. A Calculus for Dense Array Distributions. *Parallel Processing Letters*, 13(3):377–388, 2003.

[19] D. C. Dracopoulos and S. Kent. Speeding up genetic programming: A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996.

[20] A. Fahmy and A. Heddaya. Communicable memory and lazy barriers for bulk synchronous parallelism in bspk. Technical Report BU-CS-96-012, Boston University, 1996.

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

[22] 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.

[23] S. Fortune and J. Wyllie. Parallelism in Random Access Machines. In *Proceedings of the Tenth Annual Symposium on Theory of Computing (STOC)*. ACM, May 1978.

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

[25] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

[26] F. Gava and F. Loulergue. A functional language for departmental metacomputing. In S. Gorlatch, editor, *4th Workshop on Constructive Methods for Parallel Programming*, pages 63–80. Westfälische Wilheims-Universität Münster, 2004.

[27] F. Gava, F. Loulergue, and F. Dabrowski. A Parallel Categorical Abstract Machine for Bulk Synchronous Parallel ML. In W. Dosch and R. Y. Lee, editors, *4th International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD'03)*, pages 293–300. ACIS, 2003.

[28] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press.

[29] Y. Gurevich. Evolving Algebras 1993: Lipari Guide. In E. Börger, editor, *Specification and Validation Methods*, pages 9–36. Oxford University Press, 1995.

[30] G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, pages 165–178. Nova Science Publishers, august 2002.

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

[32] Guy Horvitz and Rob H. Bisseling. Designing a BSP version of ScaLAPACK. In Bruce Hendrickson et al., editor, *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, 1999.

[33] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *European Symposium on Programming (ESOP)*, number 2305 in LNCS, pages 83–97. Springer, 2002.

[34] Z. Hu, H. Iwasaki, and M. Takeichi. Flattening Transformation for Efficient Segmented Computation - Segmented Diffusion Theorem. In *APLAS*, LNCS, pages 246–257. Springer, 2002.

[35] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating Efficient Parallel Programs. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pages 85–94. ACM Press, January 22-23 1999.

[36] S.A. Jarvis, J.M.D Hill, C.J. Siniolakis, and V.P. Vasilev. Portable and architecture independent parallel performance tuning using BSP. *Parallel Computing*, 28:1587–1609, 2002.

[37] Jin-Soo Kim, Soonhoi Ha, and Chu Shik Jhon. Relaxed barrier synchronization for the BSP model of computation on message-passing architectures. *Information Processing Letters*, 66(5):247–253, 1998.

[38] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system release 3.07. Web pages at www.ocaml.org, 2003.

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

[40] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In H. Kosch, L. Boszorményi, and H. Hellwagner, editors, *Euro-Par 2003*, number 2790 in LNCS, pages 781–788. Springer Verlag, 2003.

[41] 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 III*, number 2659 in LNCS, pages 223–232. Springer Verlag, june 2003.

[42] F. Loulergue. Management of Communication Environments for Minimally Synchronous Parallel ML. In Z. Juhasz, P. Kacsuk, and D. Kranzimuller, editors, *Distributed and Parallel Systems (DAPSYS 2004)*, pages 285–292. Springer, 2004.

[43] F. Loulergue, F. Gava, M. Arapinis, and F. Dabrowski. Semantics of Minimally Synchronous Parallel ML. Technical Report 2004-07, University of Paris 12, LACL, 2004. in preparation.

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

[45] J. M. R. Martin and A. Tiskin. BSP modelling a two-tiered parallel architectures. In B. M. Cook, editor, *WoTUG'90*, pages 47–55, 1999.

[46] W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996.

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

[48] F. A. Rabhi and S. Gorlatch, editors. *Patterns and Skeletons for Parallel and Distributed Computing*. Springer, 2003.

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

[50] B. Reistag and D. K. Gifford. Static Dependent Costs for Estimating Execution Time. In *ACM conference on Lisp and Functional Programming*. ACM Press, 1994.

[51] D. Rémy. Using, Understanding, and Unravellling the OCaml Language. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 413–536. Springer, 2002.

[52] 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.

[53] C. Rodriguez, J.L. Roda, F. Sande, D.G. Morales, and F. Almeida. A new parallel model for the analysis of asynchronous algorithms. *Parallel Computing*, 26:753–767, 2000.

[54] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6):409–424, 1998.

[55] D. B. Skillicorn. *Foundations of Parallel Programming*. Number 6 in International Series on Parallel Computation. Cambridge University Press, 1994.

[56] D. B. Skillicorn. Multiprogramming BSP programs. Department of Computing and Information Science, Queen's University, Kingston, Canada, October 1996.

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

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

[59] A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms*. PhD thesis, Oxford University Computing Laboratory, 1998.

[60] 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.

[61] L. G. Valiant. A bridging model for parallel computation. *Communications of the ACM*, 33(8):103–111, August 1990.

Frédéric Loulergue obtained his PhD in Computer Science from the University of Orléans in 2000. He is currently an associate professor at the University Paris Val-de-Marne. He is a member of the Laboratory of Algorithms, Complexity and Logic and head of the "High Level Parallel Programming" group. His research interest is high-level parallel programming: semantics, implementation of parallel languages as well as the certification of parallel programs and compilers. He organized the second international workshop on High-Level Parallel Programming in 2003 and the first international workshop on Practical Aspects of High-Level Parallel Programming in 2004 (hlpp.free.fr).

Frédéric Gava obtained his Master in Computer Science from the University of Paris 7 in 2002. He is currently a PhD student at the Laboratory of Algorithms, Complexity and Logic. His research topics include the design and implementation of functional languages for parallel and meta computing. He also works on the verification of parallel programs, using the Coq proof assistant. A new topic is the automation of cost estimates.

Myrto Arapinis obtained her Master in Computer Science from the University Paris 7 in 2004. She worked in the group "High-Level Parallel Programming". She is currently a PhD student at the Laboratory of Algorithms, Complexity and Logic in the team "Algorithms and Verification". The topic of her master thesis was the design of a language for the representation of tactics based on patterns.

Frédéric Dabrowski obtained his Master in Computer Science from the University Paris 7 in 2003. He is currently a PhD student at INRIA Sophia-Antipolis, France in the Mimosa team. He is a member of the CRISS project which aims at the design of secured synchronous languages.