Semantics of Minimally Synchronous Parallel ML

Myrto Arapinis, Frédéric Loulergue, Frédéric Gava and Frédéric Dabrowski
Laboratory of Algorithms, Complexity and Logic, Créteil, France
`http://f.loulergue.free.fr`

Abstract

This paper presents a new functional parallel language: Minimally Synchronous Parallel ML. The execution time can then be estimated and dead-locks and indeterminism are avoided. It shares with Bulk Synchronous Parallel ML its syntax and high-level semantics but it has a minimally synchronous distributed semantics. 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 cost model of the Message Passing Machine model (MPM).

1. Introduction

Bulk Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [32, 25, 31] to offer a high degree of abstraction in the same way as PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP computer has three components: a homogeneous set of processor-memory pairs, a communication network allowing inter processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. A wide range of actual architectures can be seen as BSP computers.

The BSP execution model represents a parallel computation on p processors as an alternating sequence of computation *super-steps* (p asynchronous computations) and communications super-steps (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.

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 domains: scientific computing [3, 17], genetic algorithms [5] and genetic programming [8], neural networks [30], parallel databases [2], constraint solvers [13], *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" [7].

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 [15], 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 [14, 18].

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. [16] for example shows the efficiency of the BSPlib against other libraries. A more recent work [19] also points out the advantages of the BSP model over MPI for VIA (a lightweight protocol) nets in particular using a scheduling of messages which can be done at the synchronization barrier (using a latin square) in order to avoid sequentialization of the receipt of messages.

Second the BSP model is claimed to be too restrictive. All parallel algorithms are not fitted to its structured parallelism. This argument is not false but is more limited than the opponent of the BSP model think. BSP algorithms which have no relation with older algorithms but which compute the same thing can be found. The performance predictability of the BSP model even allows to design algorithms which cannot be imagined using unstructured parallelism (for example [2]). 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 and allowing subset synchronization have been proposed. We showed that divide-and-conquer algorithms can be written using extensions [23, 22] of our framework for functional bulk synchronous parallel programming [24, 21]. The execution of such programs even follow the pure BSP model.

As we faced those criticisms in our previous work on Bulk Synchronous Parallel ML (BSML), 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 [12] and proof of parallel BSML programs [11]), but with a different lower 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 efficiency of BSML with respect to 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. We could also mix MSPML and BSML for distributed supercomputing. Several BSML programs could run on several parallel machines and being coordinated by a MSPML-like program.

We first present informally MSPML (section 2.1), then give the semantics of MSPML (section 2.2). Predictability being one of our concern, we looked after cost models which could be applied to MSPML. The MPM model (section 2.3) is such a model. Section 3 is devoted to related work. We end with conclusions and future work (section 4).

2. Flat Minimally Synchronous Parallel ML

2.1. Informal presentation

There is currently no implementation of a full Minimally Synchronous Parallel ML (MSPML) language but rather a partial implementation: a library for Objective Caml [27, 6] (using TCP/IP for communications). The so-called MSPML library is based on the following elements.

It gives access to the parameters of the underling architecture which is considered as a Message Passing Machine (MPM) [28] (and section 2.3). In particular, it offers the function `p:unit->int` 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 `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. This can be ensured by a type system [12].

The parallel constructs of MSPML operate on parallel vectors. Those parallel vectors are created by: `mkpar: (int -> 'a) -> 'a par` so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$. We usually write `fun pid->e` for `f` to show that the expression `e` may be different on each processor. This expression `e` is said to be *local*. The expression `(mkpar f)` is a parallel object and it is said to be *global*. For example the expression `mkpar(fun pid->pid)` will be evaluated to the parallel vector $\langle 0, 1, \ldots, p-1\rangle$.

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` whose type is `('a -> 'b) par-> 'a par -> 'b par`. It is such as `apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i.

The communication phases are expressed by:

```
get:  'a par->int par->'a par
```

The semantics of this function is given by:

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

The full language would also contain:

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

the parallel conditional operation such that $\texttt{ifat}(\texttt{v}, \texttt{i}, \texttt{v}_1, \texttt{v}_2)$ will evaluate to $\texttt{v}_1$ or $\texttt{v}_2$ depending on the value of `v` at process `i`. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core MSPML library contains the function: `at:bool par->int->bool` to be used only in the construction: `if (at vec pid) then... else...` where `(vec:bool par)` and `(pid:int)`. `if at` expresses communication phases. Without it, the global control cannot take into account data computed locally. Global conditional is necessary to express algorithms like :

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

We end with small examples of functions used in the next sections. `bcast` is a direct broadcast program.

```
let replicate x = mkpar(fun pid ->x)
let bcast n vec = get vec (replicate n)
```

2.2. 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.

2.2.1. High Level Semantics

Syntax The syntax of the core of MSPML is given by the grammar given in figure 1.

In this grammar, x is an identifier, expression $(e\ e')$ corresponds to the application of a function or an operator e to an argument e'. Term **fun** $x \to e$ is the functional abstraction, the function whose parameter is x and result is given by the value of e. Constants c are integers and booleans. The set of operators op contains arithmetic operators, fix-point ($\mathbf{fix}$). $\mathbf{mkpar}$, $\mathbf{apply}$, $\mathbf{get}$ and $\mathbf{ifat}$ are the parallel operators presented in the previous section.

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

The programmer does not use this new syntax, but the syntax of figure 1, because enumerated parallel vectors are created during evaluation. In these syntaxes we do not separate local and global expression as in the BSλ-calculus. We rely on the type system describes in [12] to avoid nesting of parallel values.

The semantics says how we obtain *values* from expressions. The values of MSPML are defined by the following grammar:

$$
\begin{array}{rlll}
v ::= & & \textbf{fun } x \to e & \text{(functional value))} \\
 & \mid & c & \text{(constants)} \\
 & \mid & op & \text{(operators)} \\
 & \mid & (v, v) & \text{(pairs)} \\
 & \mid & \langle v, v, \ldots, v \rangle & \text{(enumerated parallel vector)}
\end{array}
$$

We note $e_1[x \leftarrow e_2]$ the substitution of the free occurrences of x in e_1 by e_2.

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

$$c \triangleright c \qquad op \triangleright op \qquad (\textbf{fun } x \to e) \triangleright (\textbf{fun } x \to e)$$

Then rules for application, binding and pairs:

$$\frac{e_1 \triangleright (\textbf{fun } x \to e) \quad e_2 \triangleright v_2 \quad e[x \leftarrow v_2] \triangleright v}{(e_1\ e_2) \triangleright v}$$

$$\frac{e_1 \triangleright v_1 \quad e_2[x \leftarrow v_1] \triangleright v}{\textbf{let } x = e_1 \textbf{ in } e_2 \triangleright v} \qquad \frac{e_1 \triangleright v_1 \quad e_2 \triangleright v_2}{(e_1, e_2) \triangleright (v_1, v_2)}$$

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

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

$$\frac{e_1 \triangleright \mathbf{fix} \quad e_2 \triangleright (\textbf{fun } x \to e_3) \quad e_3[x \leftarrow \mathbf{fix}(e_2)] \triangleright v}{(e_1\ e_2) \triangleright v}$$

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

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

$$\frac{e_1 \triangleright \textbf{fst} \quad e_2 \triangleright (v_1, v_2)}{(e_1\ e_2) \triangleright v_1} \qquad \frac{e_1 \triangleright \mathbf{snd} \quad e_2 \triangleright (v_1, v_2)}{(e_1\ e_2) \triangleright v_2}$$

The unusual rules are for the parallel operators (Fig. 2).

Example 1 *We now evaluate as example the application of the broadcast program (Fig. 3). We assume that v evaluates to $\langle v_0, v_1, \ldots v_{p-1} \rangle$.*

2.2.2. 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 desynchronization is handled in MSPML, a distributed semantics, which gives the steps of a reduction towards a value, is needed.

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

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

Syntax For the programmer, the syntax is 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

$$
\begin{array}{llll}
e' ::= x & \text{(variables)} & \mid c & \text{(constants)} \\
\mid op & \text{(operators)} & \mid \textbf{fun } x \to e' & \text{(abstraction)} \\
\mid (e'\ e') & \text{(application)} & \mid \textbf{let } x = e' \textbf{ in } e' & \text{(binding)} \\
\mid (e', e') & \text{(pairs)} & \mid \textbf{if } e' \textbf{ then } e' \textbf{ else } e' & \text{(conditional)} \\
\mid \textbf{mkpar } e' & \text{(parallel vector)} & \mid \textbf{apply } e'\ e' & \text{(parallel application)} \\
\mid \textbf{get } e'\ e' & \text{(communication)} & \mid \textbf{if } e' \textbf{ at } e' \textbf{ then } e' \textbf{ else } e' & \text{(global conditional)}
\end{array}
$$

Figure 1. Syntax

$$
\frac{e_1 \rhd \langle v_1^{'}, v_2^{'}, \ldots, v_{p-1}^{'} \rangle \quad e_2 \rhd \langle v_0^{''}, v_1^{''}, \ldots, v_{p-1}^{''} \rangle \quad \forall i (v_i^{'}\ v_i^{''}) \rhd v_i}{\textbf{apply } e_1\ e_2 \rhd \langle v_0, v_1, \ldots, v_{p-1} \rangle}
\qquad
\frac{e_1 \rhd \langle v_0, v_1, \ldots, v_{p-1} \rangle \quad e_2 \rhd \langle i_0, i_1, \ldots, i_{p-1} \rangle}{\textbf{get } e_1\ e_2 \rhd \langle v_{i_0 \% p}, \ldots, v_{i_{p-1} \% p} \rangle}
$$

$$
\frac{e_1 \rhd v \quad \forall i\ (v\ i) \rhd v_i}{\textbf{mkpar } e_1 \rhd \langle v_0, \ldots, v_{p-1} \rangle}
\qquad
\frac{e_1 \rhd \langle \ldots, \overbrace{\textbf{true}}^{n}, \ldots \rangle \quad e_2 \rhd n \quad e_3 \rhd v_3}{\textbf{if } e_1 \textbf{ at } e_2 \textbf{ else } e_3 \textbf{ then } e_4 \rhd v_3}
\qquad
\frac{e_1 \rhd \langle \ldots, \overbrace{\textbf{false}}^{n}, \ldots \rangle \quad e_2 \rhd n \quad e_4 \rhd v_4}{\textbf{if } e_1 \textbf{ at } e_2 \textbf{ else } e_3 \textbf{ then } e_4 \rhd v_4}
$$

Figure 2. Rules for parallel operators

program for the parallel machine. As in the previous section we need to define new terms which may be created during evaluation:

$$
\begin{array}{lll}
e_d & ::= & x \mid c \mid op \mid \textbf{fun } x \to e_d \\
& \mid & (e_d\ e_d) \mid \textbf{let } x = e_d \textbf{ in } e_d \mid (e_d, e_d) \\
& \mid & \textbf{if } e_d \textbf{ then } e_d \textbf{ else } e_d \\
& \mid & \textbf{mkpar } e_d \mid \textbf{apply } e_d\ e_d \mid \textbf{get } e_d\ e_d \\
& \mid & \textbf{if } e_d \textbf{ at } e_d \textbf{ then } e_d \textbf{ else } e_d \mid \textbf{request } e_d\ e_d
\end{array}
$$

The distributed semantics follows the SPMD paradigm. For example at process i the expression $\textbf{mkpar } f$ 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 $\textbf{get } v\ j$, it adds the pair $(\textbf{mstep}(\mathcal{E}_C) + 1\ ,\ v)$ to the communication environment[1] $\mathcal{E}_C$ 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: $\textbf{request } n\ j$. 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:

$$
v_d ::= \textbf{fun } x \to e_d \mid c \mid op \mid (v_d, v_d)
$$

request expressions are *not* values.

Local reduction (figure 4) 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".

Those rules cannot be applied in any context. To have a weak call by value strategy, the following contexts are needed ($\bullet$ is a "hole" which may be filled by any expression):

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

together with the context rule:

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

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

[1] In this implementation when a MSPML program is ran, the user must specify the asynchronicity depth, i.e. the maximum size of the communication environments in order to avoid memory leak. When this size is reached, a global synchronization occur and the communication environments are emptied.

$$\dfrac{\dfrac{\dfrac{\cdots}{v \triangleright \langle v_0, v_1, \ldots v_{p-1}\rangle} \qquad \dfrac{\mathbf{fun}\ x \to j \triangleright \mathbf{fun}\ x \to j \qquad \forall i \dfrac{\mathbf{fun}\ x \to j \triangleright \mathbf{fun}\ x \to j \quad j \triangleright j \quad j[x \leftarrow i] \triangleright j}{((\mathbf{fun}\ x \to j)\ i) \triangleright j}}{(\mathbf{mkpar}\ (\mathbf{fun}\ x \to j)) \triangleright \langle j, j, \ldots, j\rangle}}{\mathbf{get}\ v\ (\mathbf{mkpar}\ (\mathbf{fun}\ x \to j)) \triangleright \langle v_{j\%p}, v_{j\%p}, \ldots v_{j\%p}\rangle}}{\text{bcast}\ j\ v \triangleright \langle v_{j\%p}, v_{j\%p}, \ldots, v_{j\%p}\rangle}$$

Figure 3. Example

$((\mathbf{fun}\ x \to e_d)\ v_d, \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(e_d[x \leftarrow v_d], \mathcal{E}_C)$	(β_{fun})
$((\mathbf{let}\ x = v_d\ \mathbf{in}\ e_d), \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(e_d[x \leftarrow v_d], \mathcal{E}_C)$	(β_{let})
$(+(n_1, n_2), \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(n, \mathcal{E}_C)$ with $n = n_1 + n_2$	(δ_+)
$(\mathbf{fst}(v_{d_1}, v_{d_2}), \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(v_{d_1}, \mathcal{E}_C)$	(δ_{fst})
$(\mathbf{snd}(v_{d_1}, v_{d_2}), \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(v_{d_2}, \mathcal{E}_C)$	(δ_{snd})
$(\mathbf{fix}(\mathbf{fun}\ x \to\ e_d), \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(e_d[x \leftarrow \mathbf{fix}(\mathbf{fun}\ x \to\ e_d)], \mathcal{E}_C)$	(δ_{fix})
$(\mathbf{fix}(\mathbf{op}), \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(\mathbf{op}, \mathcal{E}_C)$	(δ_{fixop})
$(\mathbf{if\ true\ then}\ e_1\ \mathbf{else}\ e_2), \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(e_1, \mathcal{E}_C)$	(δ_{ift})
$(\mathbf{if\ false\ then}\ e_1\ \mathbf{else}\ e_2), \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(e_2, \mathcal{E}_C)$	(δ_{iff})
$(\mathbf{mkpar}\ v_d, \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(v_d\ i, \mathcal{E}_C)$	(δ_{mkpar})
$(\mathbf{apply}\ v_{d_1}\ v_{d_2}, \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(v_{d_1}\ v_{d_2}, \mathcal{E}_C)$	(δ_{apply})
$(\mathbf{get}\ v_d\ j, \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(\mathbf{request}\ (\mathbf{mstep}(\mathcal{E}_C) + 1)\ j,$ $(\mathbf{mstep}(\mathcal{E}_C) + 1, v_d) :: \mathcal{E}_C)$ if $j \neq i$	(δ_{get}^{dst})
$(\mathbf{get}\ v_d\ i, \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(v_d, (\mathbf{mstep}(\mathcal{E}_C) + 1, v_d) :: \mathcal{E}_C)$	(δ_{get}^{loc})
$(\mathbf{if}\ b\ \mathbf{at}\ n\ \mathbf{then}\ v_1\ \mathbf{else}\ v_2, \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(\mathbf{if}\ (\mathbf{request}\ (\mathbf{mstep}(\mathcal{E}_C) + 1)\ n)$ $\mathbf{then}\ v_1\ \mathbf{else}\ v_2, (\mathbf{mstep}(\mathcal{E}_C) + 1, b) :: \mathcal{E}_C)$ if $n \neq i$	(δ_{ifat}^{dst})
$(\mathbf{if}\ b\ \mathbf{at}\ i\ \mathbf{then}\ v_1\ \mathbf{else}\ v_2, \mathcal{E}_C)$	$\xrightarrow{\varepsilon}_i$	$(\mathbf{if}\ b\ \mathbf{then}\ v_1\ \mathbf{else}\ v_2,$ $(\mathbf{mstep}(\mathcal{E}_C) + 1, b) :: \mathcal{E}_C)$	(δ_{ifat}^{loc})

Figure 4. Local reduction

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

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

Figure 5. Global reduction

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 rules for global reduction are given in figure 5 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.

Example 2 *For the broadcast example, with $p = 3$, distributed evaluation of*

$$\textit{bcast}\ 2\ (\mathbf{mkpar}(\mathbf{fun}\ x \to\ 2 \times x))$$

begins with local reduction at each process. At process i, local reduction is given in figure 6. Then global reduction is used:

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

2.3. Cost model

2.3.1. BSPWB: BSP Without Barrier BSPWB, for *BSP Without Barrier*[29], 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 to MSPML but it will be not accurate enough because the bounds are too coarse.

2.3.2. MPM: Message Passing Machine A better bound $\Phi_{s,i}$ is given by the Message Passing Machine (MPM) model [28]. 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 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 done with our prototype implementation of MSPML showed that the model applies well to MSPML. For example, the parallel cost of the direct broadcast is $(p-1) \times s \times g + L$, where s denotes the size of the value v_n held at process n in words. Preliminary experiments showed that the actual performance of `bcast` follows this cost formula.

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

Figure 6. Example

3. Related Work

There are several works on extension of the BSPlib library or libraries to avoid synchronization barrier [9, 1, 20] 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 [4]. 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 [10], relies on the wave mechanism. This mechanism is more complex than ours and there is no pure functional high level semantics for Caml-flight.

[26] 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 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 juxtaposition to MSPML. The high level semantics of the parallel juxtaposition for MSPML will be the same as the one for BSML [22].

4. 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 MPSML. The first experiments with our prototype implementation show the accuracy of the cost model.

Future work can be divided into three parts:

- work on this implementation and experiments with the cost model. 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 first public version of MSPML will be released in october 2003.
- extension of MSPML with a parallel juxtaposition which allows to divide the machine in two distinct parallel machines which evaluate two MSPML expression 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.

5. References

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

[2] M. Bamha and G. Hains. Frequency-adaptive join for shared nothing machines. *Parallel and Distributed Computing Practices*, 2(3):333–345, 1999.

[3] 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.

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

[5] 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.

[6] 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.

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

[8] 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.

[9] 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.

[10] 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.

[11] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 2003. to appear.

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

[13] 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.

[14] J. M. D. Hill, P. I. Crumpton, and D. A. Burgess. Theory, practice, and a tool for BSP performance prediction. 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.

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

[16] Jonathan M. D. Hill and David Skillicorn. Lessons learned from implementing BSP. *Journal of Future Generation Computer Systems*, April 1998.

[17] 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.

[18] 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.

[19] Y. Kee and S. Ha. An Efficient Implementation of the BSP Programming Library for VIA. *Parallel Processing Letters*, 12(1):65–77, 2002.

[20] 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.

[21] 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.

[22] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In Harald Kosch, editor, *Euro-Par 2003*, number 2790 in LNCS, 2003.

[23] 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. Springer Verlag, june 2003.

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

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

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

[27] 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.

[28] 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.

[29] 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.

[30] 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.

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

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