A Parallel Categorical Abstract Machine for Bulk Synchronous Parallel ML

Frédéric Gava, Frédéric Loulergue and Frédéric Dabrowski

Laboratory of Algorithms, Complexity and Logic, Créteil, France

{gava,loulergue}@univ-paris12.f, fdabrowski@lspf.org

Abstract

We have designed a functional data-parallel language called BSML for programming bulk-synchronous parallel (BSP) algorithms in so-called direct mode. In a direct-mode BSP algorithm, the physical structure of processes is made explicit. The execution time can then be estimated and dead-locks and indeterminism are avoided. The BSMLlib library has been implemented for the Objective Caml language. But there is currently no full implementation of such a language and an abstract machine is needed to have a certified implementation. Our approach is based on a byte-code compilation to a parallel abstract machine performing exchange of data and synchronous requests derived from the abstract machine of the Caml language.

1. Introduction

Our previous work aimed at the design of a parallel functional language based on formal semantics. Bulk Synchronous Parallel ML or BSML is such a language. It is an extension of ML for programming direct-mode parallel Bulk Synchronous Parallel algorithms as functional programs. Bulk-Synchronous Parallel (BSP) computing is a parallel programming model introduced by Valiant [23] to offer a high degree of abstraction like PRAM models and yet allow portable and predictable performance on a wide variety of architectures. A BSP algorithm is said to be in *direct mode* [10] when its physical process structure is made explicit (the programmer retains control of memory allocation for each process unlike shared memory style). Such algorithms offer predictable and scalable performances. BSML expresses them with a small set of primitives taken from the *confluent* BSλ calculus [17]: a constructor of parallel vectors, asynchronous parallel function application, synchronous global communications and a synchronous global conditional.

This is no full implementation of the BSML language but our `BSMLlib` library implements (with slight restrictions) the BSML primitives using Objective Caml [13] and MPI (Message Passing Interface) [22]. It is efficient and its performance follows curves predicted by the BSP cost model [15]. The BSλ-calculus describes the primitives at a high level. We can see a BSλ term as a program which applies *sequentially* the primitives to parallel vectors. At the implementation level, the parallel vector does not exist. Each processor works on "sequential" values. Thus it is not obvious that the implementation is correct with respect to the calculus. We designed several semantics from the BSλ-calculus to a formal presentation of the implementation as a parallel abstract machine: the distributed evaluation [14] and the BS-SECD machine [19] each level of semantics has been proved correct with respect to the previous level. Nevertheless, the SECD machine [12] is a very old abstract machine for the execution of function languages and is no longer used in implementations. Merlin and Hains [18] designed a parallel abstract machine used for the execution of BSML programs and based on the Categorical Abstract Machine or CAM, which is the abstract machine of the runtime system of the CAML language.

We are now involved in the CARAML project (`www.caraml.org`) which it aims at using Objective Caml for Grid computing with, for example, applications to parallel databases and molecular simulation. In this project, we desire to design a safe environment for the compilation and execution of BSML programs and a methodology for proving the correctness of BSML programs. A "safe environment" means that we want to have a parallel virtual machine described formally and correct with respect to the BSλ-calculus. Thus, the implementation of the parallel abstract machine and the compilation schema has to be proved correct with respect to their formal descriptions.

In such a context, we could use the abstract machine designed by Merlin and Hains but it cannot be easily extended to distributed high performance computing because *the number of processes must be known at compilation time*. In the framework of Grid computing we want to have "parallel agents", compiled in byte-code, which would move to the fronted of a parallel machine and then unfold on the available nodes. In this case, of course, the num-

ber of available processes is known only when the agent arrives at the front-end of the parallel machine. The compilation and the abstract machine must be independent of the number of processes available at runtime.

Thus, we need to design a new parallel abstract machine (based on a sequential machine already used in implementations of functional languages) which may be extended easily (or which may be a substantial subpart of a more complex abstract machine) for Grid computing. This paper presents such a machine. We first present the BSP model and give an informal presentation of BSML through the `BSMLlib` programming library (section 2). Then we define a bulk synchronous parallel categorical abstract machine and the compilation of BSML to this parallel abstract machine (section 3) and conclude (section 4).

2. Functional Bulk Synchronous Parallelism

The Bulk Synchronous Parallel (BSP) model [23, 21] describes: an abstract parallel computer, a model of execution and a cost model. 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 performance of the BSP computer is characterized by three parameters (expressed as multiples the local processing speed): the number of processor-memory pairs p ; the time l required for a global synchronization ; the time g for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation (communication phase where every processor receives/sends at most h words) in time $g \times h$. Those parameters can easily be obtained using benchmarks.

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjointed phases: (a) Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes; (b) the network delivers the requested data transfers; (c) a global synchronization barrier occurs, making the transferred data available for the next super-step. The execution time of a super-step is, thus the sum of the maximal local processing time, of the data delivery time and of the global synchronization time. The execution time of a BSP program is, therefore the sum of the execution time of its super-steps.

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation as a library for Objective Caml. The so-called `BSMLlib` library is based on the following elements. It gives access to the BSP parameters of the underling architecture. In particular, it offers the function `bsp_p:unit->int` such that the value of `bsp_p()` is p, the static number of processes of the parallel machine. 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. A type system enforces this restriction [8]. The BSML parallel constructs operates on parallel vectors which are created by:

```
mkpar: (int -> 'a) -> 'a par
```

so that `(mkpar f)` stores `(f i)` on process i for i between 0 and $(p-1)$.

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a super-step) and phases of global communication (second phase of a super-step) with global synchronization (third phase of a super-step). Sequential functional languages are based on particular strategies for the evaluation of expressions. In our framework we keep the strategy of the sequential language: the parallelism comes only from our new primitives. Thus the sequentiality which comes from the evaluation strategy remains and is used to identify the different phases in a BSML program. Asynchronous phases are programmed with `mkpar` and with:

```
apply: ('a -> 'b) par
                -> 'a par -> 'b par
```

`apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. The distinction between a communication request and its realization at the barrier is ignored. `put` expresses communication and synchronization phases:

```
put: (int->'a option) par
           -> (int->'a option) par
```

where `'a option` is defined by: `type 'a option = None | Some of 'a`.

Consider: `put(mkpar(fun i->fs`$_i$`))` $(*)$. To send a value `v` from process `j` to process `i`, the function `fs`$_j$ at process `j` must be such that `(fs`$_j$ `i)` evaluates to `Some v`. To send no value from process `j` to process `i`, `(fs`$_j$ `i)` must evaluate to `None`. Expression $(*)$ 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`.

The full language would also contain a synchronous conditional operation:

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

such that `ifat (v,i,v1,v2)` will evaluate to `v1` or `v2` 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 `BSMLlib` 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)`. Global conditional is necessary of express algorithms like:

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

This framework is a good tradeoff for parallel programming because: we defined a *confluent calculus* so we designed a purely functional parallel language from it. Without side-effects, programs are easier to prove, and to reuse. An eager language allows good performances ; this calculus is based on BSP operations, so programs are easy to port, their costs can be predicted and are also portable because they are parametrized by the BSP parameters of the target architecture.

3. A BSP Abstract Machine for BSML

3.1. Introduction

3.1.1. Abstract machines for the λ-calculus To calculate the values of the λ-calculus, a lot of abstract machines have been designed. The first was the SECD machine [12] which was used for the first implementation of the LISP language. It uses environment (a list of values) for the closure and four stacks for the calculus. But it is an old and not optimized machine. In the same spirit, [2] presents the functional abstract machine (FAM). The FAM optimizes access to the environment by using arrays (so with a constant cost access). It is to be noticed that for functional languages with a call by name strategy, [20] designs the G-machine with its graph reduction. But we have an eager language so those techniques are not suitable for us.

The CAM, categorical abstract machine, was introduced and used by Curien and Cousineau to implement the CAML language [4]. The CAM [3] is an environment machine derived from categorical combinators of Curien [5] and has its roots from equational and denotational semantics of the λ-calculus.

3.1.2. Abstract machines for the BSλ-calculus For the BSλ-calculus, [19] modified the SECD. But this new machine still has the same problems as the original one: slowness, difficulty to have real instruction machine and optimize it, notably for the exchange of closures.

To remedy to these problems, [18] introduced a modification of the CAM for BSP. But this machine has two problems:

- the number of processors of the machine which will execute the program has to be known at compilation. Using an abstract machine eases portability but statically defining the number of processors for compilation is against it. Moreover it is not at all adapted for Grid computing.
- the instruction of exchange of values is difficult to translate to real code, especially in an efficient way, because this instruction adds instructions to the code during the execution.

The first problem is specific to [18] but the second problem is shared with the BSP SECD machine. We give here a suitable abstract machine which is an extension of the CAM for BSP computing without these problems. We first recall the execution model of the original sequential CAM and after present its new BSP extensions. Then we explain a technique allowing to compile our core language to this abstract machine.

3.2. The BSP CAM

The BSP CAM has two kinds of instructions: sequential and parallel ones. Its corresponds to the two structures of the original calculus: the BSλ-calculus [17].

3.2.1. Sequential CAM The CAM machine is a very simple machine where categorical terms can be considered as code acting on a graph of values. Instructions are few in number and quite close to real machine instructions. The machine state has two components:

1. a code pointer C called *program counter* representing the code being executed as a sequence of instructions.
2. a stack S (a sequence of machine values) called *stack pointer* holding function arguments, intermediate results and function return contexts.

Each of them holds a pointer in a real implementation; however, for simplicity, we will describe them as containing respectively a list of instructions (program counter) and a list of stack elements (stack pointer). The top of the stack corresponds to a term (a structured value) which have been computed by the CAM. It may be viewed as a *register*. The CAM supposes a de-curryfied version of the calculus. This is why all our operators (and specially the `apply` operator) use pairs (noted (s, t)).

The values stored in this stack are constants, closures, pairs of values which may in turn be pairs so that trees may be constructed. The CAM uses closures $[C, s]$ and recursive closures $[C, s]_r$ where C is a fragment of CAM code and s is a value meant to denote an environment for representing functional values in a natural way since its structure is induced by categorical combinator's properties (optimize closure building and environment sharing instead of optimizing access to values). Environments, as in the natural semantics, are trees of values coding by pairs. Environments give the values of free variables in closure for the CAM machine which could be used with special instructions. Predefined operations (such as addition, subtraction, etc.) may be added to the instructions.

BEFORE		AFTER		
stack	**code**	**stack**	**code**	
$(t_1,t_2)::S$	**Fst**;C	$t_1::S$	C	
$(t_1,t_2)::S$	**Snd**;C	$t_2::S$	C	
$t::S$	**Quote**(c);C	$c::S$	C	
$t::S$	**Cur**(C_1);C	$[C_1,t]::S$	C	
$t::S$	**CurRec**(C_1);C	$[C_1,t]_r::S$	C	
$s::S$	**Push**;C	$s::s::S$	C	
$t_1::t_2::S$	**Swap**;C	$t_2::t_1::C$	C	
$t_1::t_2::S$	**Cons**;C	$(t_2,t_1)::S$	C	
$([C_1,s],t)::S$	**App**;C	$(s,t)::C::S$	C_1	
$([C_1,s]_r,t)::S$	**App**;C	$((s,[C_1,s]_r,t))::C::S$	C_1	
$t::C_1::S$	**Return**;C	$t::S$	C_1	
$(t_1,t_2)::S$	**Add**;C	$(t_1 + t_2)::S$	C	
$(t_1,t_2)::S$	**Sub**;C	$(t_1 - t_2)::S$	C	
$(t_1,t_2)::S$	**Equal**;C	**true**::S	C	if $t_1 = t_2$
$(t_1,t_2)::S$	**Equal**;C	**false**::S	C	else
true::S	**Branch**(C_1,C_2);C	S	C_1;C	
false::S	**Branch**(C_1,C_2);C	S	C_2;C	
nc::S	**Isnc**;C	**true**::S	C	
$t::S$	**Isnc**;C	**false**::S	C	

Figure 1. Asynchronous instructions

The traditional machine is summarized in Figure 1. It must be read as: *executing the instruction when the machine is in this state takes it to a new state*. Evaluating a CAM program begins with an empty stack and ends with a value in the register that is the result of the program and an empty program counter. We briefly give the meaning of the instructions:

Push : duplicates the register.

Swap : swaps the tops of the stack.

Cons : makes a pair on the tops of the stack.

Fst : expects a term (s,t) and replaces it by s (resp. t for the **Snd** instruction).

Cur : replaces the register s by the closure $[C,s]$ where C is in the code encapsulated by the **Cur** instruction.

CurRec : replaces the register s by the recursive closure $[C,s]_r$ where C is in the code encapsulated by the **CurRec** instruction. This is a cyclic closure.

App : expects the register $([C,s],t)$ (resp. $([C,s]_r,t)$) replaces it by (s,t) (resp. $((s,[C,s]),t)$) prefixed the rest of the code and the program counter is C.

Quote : replaces the register by the encapsulated constants (int, bool or the special constant of non-communication).

Return : returns the context program counter.

Branch : removes the register and according to whether it is **true** or **false**, executed C_1 or C_2.

Add : primitive operator of the addition. Takes a pair on the register and gives the addition of the two components. (Idem for **Sub**, **Equal** and another arithmetic operations).

Isnc : primitive operator which tests if the register is the non-communication constant or not.

We have an instruction **Cur** for closure and **CurRec** for recursive one. The recursivity of the closure appears in the **App** rule when the closure is used again in the environment. The **Branch** instruction is used for the **if then else** (and **at**) constructor. It was also added to the CAM by [5]. Environments are not mere lists but full binary trees. The categorical combinators **Fst** and **Snd** are precisely access functions into those binary trees. The compiling process uses a pattern that is the formal image of the environment. **Fst** and **Snd** are also used to represented the **fst** and **snd** primitive operators. We refer to [5] and [3] for more details on the CAM and the theory of categorical combinators.

Those instructions which are the traditional instructions of the CAM correspond to the asynchronous steps of the BSP model. In the next section, we add special instructions (which could be seen as categorical combinators) for

	stack	code
Before	$\langle\, n{::}t_0{::}S_0, \ldots, n{::}t_{p-1}{::}S_{p-1}\,\rangle$	$\langle\, \textbf{At};C_0, \ldots, \textbf{At};C_{p-1}\,\rangle$
After	$\langle\, t_n{::}S_0, \ldots, t_n{::}S_{p-1}\,\rangle$	$\langle\, C_0, \ldots, C_{p-1}\,\rangle$
Before	$\langle \ldots, t :: S, \ldots\rangle$	$\langle \ldots, \textbf{Nprocs};C, \ldots\rangle$
After	$\langle \ldots, p{::}S, \ldots\rangle$	$\langle \ldots, C, \ldots\rangle$
Before	$\langle \ldots, \overbrace{S}^{i}, \ldots\rangle$	$\langle \ldots, \overbrace{\textbf{Pid};C}^{i}, \ldots\rangle$
After	$\langle \ldots, \overbrace{i{::}S}^{i}, \ldots\rangle$	$\langle \ldots, \overbrace{C}^{i}, \ldots\rangle$
Before	$\langle\ (t_0^{p-1}, (t_0^{p-2}, (\ldots, t_0^0)\ldots)){::}S_0, \ldots, (t_{p-1}^{p-1}, (t_{p-1}^{p-2}, (\ldots, t_{p-1}^0)\ldots)){::}S_{p-1}\ \rangle$	$\langle\, \textbf{Send};C_0, \ldots, \textbf{Send};C_{p-1}\,\rangle$
After	$\langle\ (t_{p-1}^0, (t_{p-2}^0, (\ldots, t_0^0)\ldots)){::}S_0, \ldots, (t_{p-1}^{p-1}, (t_{p-2}^{p-1}, (\ldots, t_0^{p-1})\ldots)){::}S_{p-1}\ \rangle$	$\langle\, C_0, \ldots, C_{p-1}\,\rangle$

Figure 2. BSP CAM instructions

the parallel machine and for the synchronous steps of the model.

3.2.2. Parallel extensions A BSP CAM is simply the duplication, on each process, of the sequential CAM extended by some primitives. To express the BSP super-steps we need two kinds of instructions: sequential instructions and synchronous ones. For the first phase of the BSP model (asynchronous calculus), we also need the number of processes and the name (number) of each process (in the spirit of SPMD programming) given by new sequential instructions:

Nprocs : replaces the register by the number of processes (p). The actual value is defined *at the beginning of execution* and not at compilation. Thus a program compiled for our abstract machine is totally portable.

Pid : adds on the stack the number of the process.

The **Pid** instruction is needed for the construction of the parallel vectors by giving the name of the process (i). Then, to express the synchronization and communication phases of the BSP super-step, we need to add two special instructions to the BSP CAM: **At** and **Send**. They are the only instructions which need BSP synchronization between the sequential CAMs. **At** (with a **Branch** instruction) is used for the global conditional. **Send** is an instruction for the primitive synchronous **put** operator, used for the exchange of values between the processes. We now give the actions of these instructions on the BSP CAM:

At : reads the value n of the register and pops it, then replaces the (new) register by the register of the n^{th} process.

Send : replaces the register, supposed to be a "recursive" pair (see next section) by another pair which is the result of the exchange of values between the p processes.

The instructions of the BSP CAM are given in Figure 2 for a p processors machine. **Nprocs** and **Send** are the only instructions which depend on the number of processes.

3.3. Compilation of BSML

We shall now consider the problem of compiling our core language to the machine. The BSML language uses real identifiers and not De Bruijn indices [6] (they transform an identifier to the number of λ-abstraction which are included between the identifier and the λ-abstraction that binds it; this method was used to solve the problem of binding variables); our compiling function will have to deal with the translation of a variable to some access code that will find at run time its value in the environment. Thus the compiling function has an extra parameter which gives the position of the free variables of the expression to be compiled in the environment. We note $[\![e]\!]_P$ the function of compilation of an expression e with the environment P. The compiling function is defined by induction on the expression and begins with an empty environment $()$. We suppose that the expressions are well-typed and the nested of parallel vectors is rejected by the type checker of the BSML language.

3.3.1. Environments We have seen that environments are binary trees. Thus the access function will transform variables on **Fst** and **Snd** instructions to access the environment:

$$\begin{aligned}
[\![x]\!]_{()} &= \text{raise } \textbf{fail}\\
[\![x]\!]_{(P,x)} &= \textbf{Snd}\\
[\![x]\!]_{(x,P)} &= \textbf{Fst}\\
[\![x]\!]_{(P_1,P_2)} &= (\textbf{Snd}; [\![x]\!]_{P_2})?(\textbf{Fst}; [\![x]\!]_{P_1})
\end{aligned}$$

Application

$[\![\textbf{op}\ e]\!]_P = [\![e]\!]_P; Code-Operator(\textbf{op});$

$[\![(e_1\ e_2)]\!]_P = \textbf{Push}; [\![e_1]\!]_P; \textbf{Swap}; [\![e_2]\!]_P; \textbf{Cons}; \textbf{App};$

Pair and Abstraction

$[\![(e_1, e_2)]\!]_P = \textbf{Push}; [\![e_1]\!]_P; \textbf{Swap}; [\![e_2]\!]_P; \textbf{Cons};$

$[\![(\textbf{fun}\ x \rightarrow e)]\!]_P = \textbf{Cur}([\![e]\!]_{(P,x)}; \textbf{Return};);$

Sequential construction

$[\![\textbf{if}\ e_1\ \textbf{then}\ e_2\ \textbf{else}\ e_3]\!]_P = \textbf{Push}; [\![e_1]\!]_P; \textbf{Branch}([\![e_2]\!]_P; \textbf{Return};, [\![e_3]\!]_P; \textbf{Return};);$

$[\![\textbf{let}\ x = e_1\ \textbf{in}\ e_2]\!]_P = \textbf{Push}; [\![e_1]\!]_P; \textbf{Cons}; [\![e_2]\!]_{(P,x)};$

$[\![\textbf{let rec}\ f\ x = e_1\ \textbf{in}\ e_2]\!]_P = \textbf{Push}; \textbf{CurRec}([\![e_1]\!]_{((P,f),x)}; \textbf{Return};); [\![e_2]\!]_{(P,f)}; \textbf{Cons};$

Parallel construction

$[\![\textbf{if}\ e_1\ \textbf{at}\ e_2\ \textbf{then}\ e_3\ \textbf{else}\ e_4]\!]_P = \textbf{Push}; \textbf{Push}; [\![e_1]\!]_P; \textbf{Swap}; [\![e_2]\!]_P; \textbf{At}; \textbf{Branch}([\![e_3]\!]_P; \textbf{Return};, [\![e_4]\!]_P; \textbf{Return};);$

$[\![\textbf{put}]\!]_P = Insert\_Send([\![put\_function]\!]_P)$

$[\![\textbf{put}\ e]\!]_P = \textbf{Push}; Insert\_Send([\![put\_function]\!]_P); \textbf{Swap}; [\![e]\!]_P; \textbf{Cons}; \textbf{App};$

Figure 3. Compilation

$[\![\ ]\!]?[\![\ ]\!]$ means that during the compilation if the first branch of the function fails then the second is used.

3.3.2. Sequential mini-BSML expressions Constants are trivially compiled to **Quote** and **Nprocs** instructions:

$$\begin{aligned} [\![i]\!]_P &= \textbf{Quote}(i); \quad \text{if } i \in \mathbb{N} \\ [\![b]\!]_P &= \textbf{Quote}(b); \quad \text{if } b \in \mathbb{B} \\ [\![\textbf{nc}]\!]_P &= \textbf{Quote}(\textbf{nc}); \\ [\![\textbf{bsp\_p}]\!]_P &= \textbf{Nprocs}; \end{aligned}$$

For the primitive operators we use a function $Code\_Operator$ which gives the instruction of each operator and we have: $[\![\textbf{op}]\!]_P = \textbf{Cur}(\textbf{Snd}; Cod\_Op(\textbf{op}););$ where:

$$\begin{aligned} Cod\_Op(\texttt{+}) &= \textbf{Add} & Cod\_Op(\texttt{-}) &= \textbf{Sub} \\ Cod\_Op(\texttt{=}) &= \textbf{Equal} & Cod\_Op(\textbf{fst}) &= \textbf{Fst} \\ Cod\_Op(\textbf{snd}) &= \textbf{Snd} & Cod\_Op(\textbf{isnc}) &= \textbf{Isnc} \end{aligned}$$

Applications, abstractions, pairs, let binding and recursive binding are classically compiled (see [5, 1]). For some optimizations, we use a trick of [5] which compiles differently applications and primitive operators applied to their arguments (figure 3).

3.3.3. Parallel operators For the primitive operations, we used what the semantics suggests: the parallel operator **mkpar** is compiled to the application of the expression to the "pid" (or name) of the processes and **apply** is simply the application because the first value is supposed to be a closure (or recursive) from an abstraction or an operator. Thus we add to the Cod_Op the two new cases:

$$Cod\_Op(\textbf{op}) = \begin{cases} \textbf{Pid}; \textbf{Cons}; \textbf{App} \text{ where } \textbf{op} = \textbf{mkpar} \\ \textbf{App} \text{ where } \textbf{op} = \textbf{apply} \end{cases}$$

The global conditional is compiled like the traditional conditional but with another argument and by adding the **At** instructions before the **Branch** to make the synchronous and communication running of the BSP model (figure 3).

To compile the **put** operator, a first way presented by [18] used at compilation the static number of processes and two special instructions: one adds code during the execution to calculate all the values to be sent and a second exchanges those values and generates code to read them. Clearly, in a real implementation with real machine codes this is neither easy nor efficient to add a lot of machine code. Moreover the length of this code was dependent on the static number of processes of the parallel machine.

To remedy to this problem, we use a program which recursively computes the values to be sent. The source of this program is independent on the number of processes be it execution depends on it since this program uses **bsp_p**.

To do this, we can add a special closure name *put_function* to iterate the calculus. We can write it in our functional language to directly have the code generated by our compiler (Figure 4).

To have the compilation of the **put** primitive operator, we use $Insert\_Put$ (figure 2): informally $Insert\_Put$ adds the **Send** instruction in the code generated (by our compilation schema) from the *put_function* between the end of the construction of the last pair and the call of the read function (the compiled code is not shown here for the sake of conciseness).

Let us explain informally what happens at a given process. `create` recursively computes the values to be sent. `f` is here the function given as argument (at a given process) to the `put` primitive in a BSML program. It is applied to all the process numbers from the last one to the

```
let put_function = fun f ->
  let rec create n =
    if n=0 then f n else ((f n),(create (n-1))) in
  let construct_one_case = fun g -> fun pid -> fun value -> fun n ->
      if n=pid then value else g n in
  let rec read couple = let counter=(fst couple) in
      let value=(snd couple) in
        if counter=0 then fun n -> (if n=0 then value else nc)
         else (construct_on_case(read(counter-1,snd value))counter)(fst value)
  in read ((bsp_p-1),(create(bsp_p-1)))
```

Figure 4. `put_function`

first one: we then have p values to be sent. They are in a data structure similar to a list but built using pairs only: $(v_{p-1}, (v_{p-2}, \ldots (v_1, v_0)))$.

The **Send** instruction exchanges the values between the different processes. At the end of the exchange, each process has again p values organized as described before.

Then `read` and `construct_one_case` build recursively the function (in the case of the machine a closure) which is the result of the **put** primitive of BSML (at a given process). At the first step the function produced is `fun n -> if n=p-1 then` $\mathtt{v}_{p-1}$ `else nc`. When the recursion ends we obtain a closure which represents the following function:

`fun n->if n=p-1 then` $\mathtt{v}_{p-1}$
`else` $\ldots$`if n=0 then` $\mathtt{v}_0$ `else nc`

3.4. Example

We developed a sequential implementation of the BSP CAM. To illustrate, we compiled and ran the following trivial expression (see Figure 5) with two processes `mkpar (fun pid -> pid+1)`. We also implemented a parallel version of the BSP CAM with the `BSMLlib` library. BSML programs are compiled to lists of instructions of the BSP CAM. Thus it cannot attain the efficiency of an abstract machine written in a low level language and with a real compiler to byte-code, but the first experiments show that the execution times follow the costs predicted by the BSP model with of course very large values for g and l and a low value for the "processor" speed (which in this case represents the speed of the non-optimized CAM).

4. Conclusions and Future Works

The Bulk Synchronous Parallel Categorical Abstract Machine presented here provides a detailed formal description of a parallel runtime system for Bulk Synchronous Parallel ML. It has two advantages with respect to the BSP-SECD machine and the BSP-CAM of [18]: the number of processes of the parallel machine has not to been known at compilation, thus improving the portability; the communication operation does not add instructions at execution, making the implementation both simpler and more classic. We already developed a parallel prototype of this machine using the `BSMLlib` library. Timings show that the BSP model holds. A complementary side is our work on proofs of BSML program using the Coq Proof Assistant [7]. The next phases will be:

- the proof of correctness of this machine with respect to a version of the BSλ-calculus with explicit substitution [16]. We will use the work of Hardin and et. [11] on the proof of sequential abstract machines using a λ-calculus of explicit substitutions.
- extensions of the machine to be able to compile the full Caml language extended by parallel operations: imperative features such as assignment (we already studied at a higher level, the interaction of our parallel operations with the imperative features of Caml [9]) and exceptions. This is an ongoing work. The main problem is to design an exception mechanism to deal with exceptions raised into a **mkpar** (at the local level) but caught only at the global level.
- the development of an efficient and certified parallel implementation of this abstract machine. We will use Spark (`www.sparkada.com`).

Acknowledgments This work is supported by a grant from the French Ministry of Research and the ACI Grid program, under the project CARAML.

5. References

[1] S. Boutin. Proving correctness of the translation from mini-ml to the cam with the coq proof development system. Technical Report 2536, INRIA, 1995.

[2] L. Cardelli. Compiling a functional language. In *Conference Record of the 1984 ACM Symposium on*

	Processor 0		Processor 1
()::[]	**Cur**(C);**Pid**;**Cons**;**App**;	()::[]	**Cur**(C);**Pid**;**Cons**;**App**;
([C:()])::[]	**Pid**;**Cons**;**App**;	[C:()]::[]	**Pid**;**Cons**;**App**;
0::[C:()]::[]	**Cons**;**App**;	1::[C:()]::[]	**Cons**;**App**;
([C:()],0)::[]	**App**;	([C:()],0)::[]	**App**;
((),0)::;::[]	C	((),1)::;::[]	C
((),0)::((),0)::;::[]	**Snd**;**Swap**;**Quote** 1;**Cons**;**Add**;**Return**;	((),1)::((),1)::;::[]	**Snd**;**Swap**;**Quote** 1;**Cons**;**Add**;**Return**;
0::((),0)::;::[]	**Swap**;**Quote** 1;**Cons**;**Add**;**Return**;	1::((),1)::;::[]	**Swap**;**Quote** 1;**Cons**;**Add**;**Return**;
((),0)::0::;::[]	**Quote** 1;**Cons**;**Add**;**Return**;	((),1)::1::;::[]	**Quote** 1;**Cons**;**Add**;**Return**;
1::0::;::[]	**Cons**;**Add**;**Return**;	1::1::;::[]	**Cons**;**Add**;**Return**;
(0,1)::;::[]	**Add**;**Return**;	(1,1)::;::[]	**Add**;**Return**;
1::;::[]	**Return**;	2::;::[]	**Return**;
1::[]	;	2::[]	;

where C=**Push**;**Snd**;**Swap**;**Quote** 1;**Cons**;**Add**;**Return**;
and where i is the name of the process and p the number of processes

Figure 5. BSP CAM running example

Lisp and Functional Programming, pages 208–217, Austin, Texas, August 1984. ACM.

[3] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. *Science of Computer Programming*, 8:173–202, 1987.

[4] G. Cousineau and G. Huet. The CAML primer (version 2.6.1). Technical report, INRIA-Rocquencourt, 1990.

[5] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming*. Birkhäuser, Boston, second edition, 1993.

[6] N.G. De Bruijn. Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation, whith application to the Church-Rosser theorem. *Indag. Math.*, 34:381–392, 1972.

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

[8] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In *PaCT 2003*, number 2763 in LNCS, pages 215–229. Springer, 2003.

[9] F. Gava and F. Loulergue. Semantics of a Functional Bulk Synchronous Parallel Language with Imperative Features. In *ParCo 2003*, Dresden, Germany, 2003. to appear.

[10] A. V. Gerbessiotis and L. G. Valiant. Direct Bulk-Synchronous Parallel Algorithms. *Journal of Parallel and Distributed Computing*, 22:251–267, 1994.

[11] T. Hardin, L. Maranget, and L. Pagano. Functional runtime systems within the lambda-sigma calculus. *Journal of Functional Programming*, 8(2):131–176, 1998.

[12] P. J. Landin. The mechanical evaluation of expressions. *The Computer Journal*, 4(6):308–320, 1964.

[13] Xavier Leroy. The Objective Caml System 3.06, 2002. web pages at www.ocaml.org.

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

[15] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14*th IASTED PDCS Conference*, pages 452–457. ACTA Press, 2002.

[16] F. Loulergue. A Calculus of Functional BSP Programs with Explicit Substitution. In *ParCo 2003*, Dresden, Germany, 2003. to appear.

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

[18] A. Merlin and G. Hains. La Machine Abstraite Catégorique BSP. In *Journées Francophones des Langages Applicatifs*. INRIA, 2002.

[19] A. Merlin, G. Hains, and F. Loulergue. A SPMD Environment Machine for Functional BSP Programs. In *Proceedings of the Third Scottish Functional Programming Workshop*, 2001.

[20] S. L. Peyton Jones. *The Implementation of Functional Programming Languages*. Prentice-Hall, 1987.

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

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

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