Systematic Development of Correct Bulk Synchronous Parallel Programs

Louis Gesbert*, Zhenjiang Hu†, Frédéric Loulergue‡, Kiminori Matsuzaki§ and Julien Tesson¶

*MLstate, Paris, France, louis.gesbert@mlstate.com

†National Institute of Informatics, Tokyo, Japan, hu@nii.ac.jp

‡LIFO, Université d'Orléans, France, Frederic.Loulergue@univ-orleans.fr

§Kochi University of Technology, Kochi, Japan, matsuzaki.kiminori@kochi-tech.ac.jp

¶LIFO, Université d'Orléans, France, Julien.Tesson@univ-orleans.fr

***Abstract*— With the current generalisation of parallel architectures arises the concern of applying formal methods to parallelism. The complexity of parallel, compared to sequential, programs makes them more error-prone and difficult to verify. Bulk Synchronous Parallelism (BSP) is a model of computation which offers a high degree of abstraction like PRAM models but yet a realistic cost model based on a structured parallelism. We propose a framework for refining a sequential specification toward a functional BSP program, the whole process being done with the help of the Coq proof assistant. To do so we define BH, a new homomorphic skeleton, which captures the essence of BSP computation in an algorithmic level, and also serves as a bridge in mapping from high level specification to low level BSP parallel programs.**

I. Introduction

With the current generalisation of parallel architectures and increasing requirement of parallel computation arises the concern of applying formal methods, which allow specifications of parallel and distributed programs to be precisely stated and the conformance of an implementation to be verified using mathematical techniques. However, the complexity of parallel programs, compared to sequential ones, makes them more error-prone and difficult to verify. This calls for a strongly structured form of parallelism [16], which should not only be equipped with an abstraction or model that conceals much of the complexity of parallel computation, but also provide a systematic way of developing such parallelism from specifications for practically nontrivial examples.

The Bulk Synchronous Parallel (BSP) model is a model for general-purpose, architecture-independent parallel programming [7]. The BSP model consists of three components, namely a set of processors each with a local memory, a communication network, and a mechanism for globally synchronising the processors. A BSP program proceeds as a series of super-steps. In each super-step, a processor may operate only on values stored in local memory. Values sent through the communication network are guaranteed to arrive at the end of a super-step. Although the BSP model is simple and concise, it remains as a challenge to systematically develop efficient and correct BSP programs that meet given specifications.

To see this clearly, consider the following tower-building problem, which is an extension of the known line-of-sight problem [3]. Given a list of locations (position x_i and height h_i) along a line in the mountain (see Fig. 1):

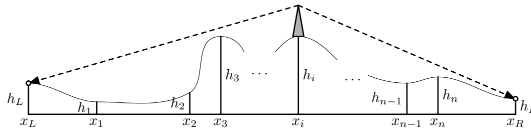

Fig. 1. Tower-Building Problem

$$[(x_1, h_1), \ldots, (x_i, h_i), \ldots, (x_n, h_n)]$$

and two special points (x_L, h_L) and (x_R, h_R) on the left and right of these locations along the same line, the problem is to find all locations from which one can see the two points after building a tower of height h. If we do not think about efficiency and parallelism, this problem can be easily solved by considering for each location (x_i, h_i), whether it can be seen from both (x_L, h_L) and (x_R, h_R). The tower with height h at location (x_i, h_i) can be seen from (x_L, h_L) if for any $k = 1, 2, \ldots, i-1$ the inequality

$$\frac{h_k - h_L}{x_k - x_L} < \frac{h + h_i - h_L}{x_i - x_L}$$

holds. Similarly, it can be seen from (x_R, h_R) means that for any $k = i+1, \ldots, n$, the inequality

$$\frac{h_k - h_R}{x_R - x_k} < \frac{h + h_i - h_R}{x_R - x_i}$$

holds. While the specification is clear, its BSP parallel program is rather complicated, even though there are libraries for BSP programming such as BSML [14], one in the functional language Objective Caml. This gap makes it difficult to verify that the implementation is correct with respect to the specification.

In this paper, we propose the first general framework (in Sect. II), as far as we are aware, for systematic development of certified functional BSP parallel programs. More specifically, (1) we introduce a novel algorithmic skeleton (Sect. III), BSP Homomorphism (or *BH* for short), which can not only capture the essence of BSP computations at an algorithmic level, but also serve as a bridge by mapping high level specification to low level BSP parallel programs; (2) we develop a set of useful theories (Sect. IV) in Coq for systematic and formal derivation of programs from specification to *BH*, and we

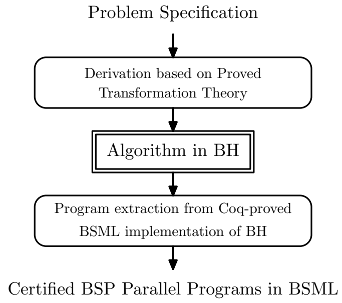

Fig. 2. An Overview of our Framework for Developing Correct BSP Programs

provide a certified parallel implementation (Sect. V) of *BH* in BSML so that a certified BSP parallel program can be automatically extracted; and (3) we demonstrate how we can extract certified BSP parallel programs in our new framework and report experiment results (Sect. VI).

II. An Overview

Figure 2 depicts an overview of our framework. We insert a new layer, called "algorithm in *BH*", between problem specifications and certified BSP parallel programs, so as to divide the development process into two easier steps: A formal derivation of algorithms from specification to *BH* and a proof of correctness of a BSML implementation of *BH*.

In our framework, a specification is described in Coq [2], allowing the user to be confident in its correctness without concern of parallelism. We chose to take Coq definitions as specifications for reasons of simplicity of our system, and for giving access to the full strength of the Coq assistant to prove initial properties of the algorithms (the system will then provide a proof that these properties are preserved throughout the transformations). In the first step, we rewrite the specification into a program using the *BH* skeleton, in a semi-automated way. To do so, we provide a set of Coq theories over *BH* and tools to make this transformation easier. This transformation is implemented in Coq, and proved to be correct, *i.e.* preserving the semantics of the initial specification. Thus, this step converts the original specification into a program (using BH) that is proved equivalent. In the second step, we replace the calls to the skeleton *BH* in the algorithm with a parallel implementation (in BSML) that is proved correct. By using the program extraction features of Coq on the rewritten algorithm, we get a parallel program that implements the algorithm of the specification, and that is proved correct.

III. A BSP Homomorphism

Homomorphisms play an important role in both formal derivation of parallel programs [6], [10] and automatic parallelisation [15]. Function h is said to be a *homomorphism*, if it is defined recursively in the form of

$$\left\{\begin{array}{lcl} h\ [\,] & = & \iota_{\odot} \\ h\ [a] & = & f\ a \\ h\ (x +\!\!+\, y) & = & (h\ x) \odot (h\ y) \end{array}\right.$$

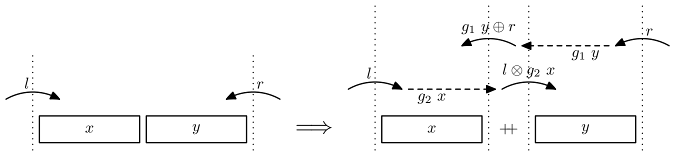

Fig. 3. Information Propagation for *BH*.

where applying a function f to an expression x is written $f\ x$, $[x_1, \ldots, x_n]$ denotes the list containing the elements x_1 to x_n, $+\!\!+$ denotes list concatenation, $\iota_{\odot}$ denotes the identity unit of $\odot$. Since h is uniquely determined by f and $\odot$, we will write $h = (\![\odot, f]\!)$.

Though being general, different parallel computation models would require different specific homomorphisms together with a set of specific derivation theories. For instance, the distributed homomorphism [8] is introduced to treat the hypercube style of parallel computation, and the accumulative homomorphism [11] is introduced to treat the skeleton parallel computation.

Our BH (BSP Homomorphism) is a specific homomorphism carefully designed for systematic development of certified BSP algorithms. The key point is that we formalise "data waiting" and "synchronisation" in the super-step of the BSP model by gathering necessary information for computation around for each element of a list and then perform computation independently to update each element.

Definition 1 (BSP Homomorphism): Given function k, two homomorphisms g_1 and g_2, and two associative operators $\oplus$ and $\otimes$, a function bh is said to be a BSP Homomorphism or *BH*, if it is defined in the following way.

$$\left\{\begin{array}{lcl} bh\ [a]\ l\ r & = & [k\ a\ l\ r] \\ bh\ (x +\!\!+\, y)\ l\ r & = & bh\ x\ l\ (g_1\ y \oplus r)\ +\!\!+ \\ & & bh\ y\ (l \otimes g_2\ x)\ r \end{array}\right.$$

The above bh defined with functions k, g_1, g_2, and associative operators $\oplus$ and $\otimes$ is denoted as

$$bh = BH(k, (g_1, \oplus), (g_2, \otimes)).$$

Function bh is a *higher-order* homomorphism, which takes a list as input and returns a new list of the same length. In addition to the input list, bh has two additional parameters, l and r, which contain necessary information to perform computation on the list. The information of l and r, as defined in the second equation and shown in Fig. 3, is propagated from left and right with functions $(g_2, \otimes)$ and $(g_1, \oplus)$ respectively.

It is worth remarking that BH is powerful; it cannot only describe super-steps of BSP computation, but is also powerful enough to describe various computation including all homomorphisms (map and reduce) (Sect. IV), scans, as well as the BSP algorithms in [7].

IV. Deriving Algorithms in BH

In this section, we show how to derive correct algorithms in terms of BH from problem specifications. The specification gives a direct solution to the problem, where one does not need to think about low level parallel computation issues, such as layout of processors, task distribution, data communication. This specification will be transformed into an equivalent algorithm in terms of BH based on a set of transformation theorems.

A. Specification

Coq functions are used to write specification, from which an algorithm in BH is to be derived. Recursions and the well-known collective operators (such as map, fold, and scan) can be used in writing specification. To ease description of computation using data around, we introduce a new collective operator $mapAround$.

The $mapAround$, compared to map, describes more interesting independent computation on each element of lists. Intuitively, $mapAround$ is to map a function to each element (of a list) but is allowed to use information of the sublists in the left and right of the element, e.g.,

$$\begin{array}{l} mapAround\ f\ [x_1, x_2, \ldots, x_n] = \\ \quad [\ f\ ([], x_1, [x_2, \ldots, x_n]), f\ ([x_1], x_2, [x_3, \ldots, x_n]), \\ \qquad \ldots, f\ ([x_1, x_2, \ldots, x_{n-1}], x_n, [])\ ]. \end{array}$$

In addition, we provide a set of communication functions such as $permute$, $shiftL$, $shiftR$ to redistribute list elements. They are designed to be not only useful for reconstructing lists in the specification level but also equipped with low lever certified BSP implementations.

Example 1 (Specification of the Tower-Building Problem): Recall the tower-building problem in the introduction. We can solve it directly using $mapAround$, by computing independently on each location and using informations around to decide whether a tower should be built at this location. So our specification can be defined straightforwardly as follows.

$$\begin{array}{l} tower\ (x_L, h_L)\ (x_R, h_R)\ xs = mapAround\ visibleLR\ xs \\ \quad \textbf{where}\ visibleLR\ (ls, (x_i, h_i), rs) = \\ \qquad\qquad visibleL\ ls\ x_i \wedge visibleR\ rs\ x_i \\ \qquad visibleL\ ls\ x_i = maxAngleL\ ls < \frac{h+h_i-h_L}{x-x_L} \\ \qquad visibleR\ rs\ x_i = maxAngleR\ rs < \frac{h+h_i-h_R}{x_R-x} \end{array}$$

The inner function $maxAngleL$ is to decide whether the left tower can be seen, and is defined as follows (where $a \uparrow b$ returns the bigger of a and b).

$$\begin{array}{lcl} maxAngleL\ [\,] & = & -\infty \\ maxAngleL\ ([(x,h)] +\!\!+ xs) & = & \frac{h-h_L}{x-x_L} \uparrow maxAngleL\ xs \end{array}$$

and the function $maxAngleR$ can be similarly defined. □

B. Theorems for Deriving BH

Since our specification is a simple combination of collective functions and recursive functions, derivation of a certified BSP parallel program can be reduced to derivation of certified BSP parallel programs for all these functions, because the simple combination is easy to be implemented by composing super-steps in BSP.

While simple communication functions may be mapped directly to certified BSP parallel programs, it is more difficult for the collective functions and recursive functions, which may be parametrised with other functions and have more flexible computation structure. Our idea is to map these functions into *BH*, and then show how *BH* can be mapped to a certified BSP parallel programs.

First, let us see how to deal with collective functions. The central theorem for this purpose is the following theorem.

Theorem 1 (Parallelisation $mapAround$ with BH*):* For a function

$$h = mapAround\ f$$

if we can decompose f as $f\ (ls, x, rs) = k\ (g_1\ ls, x, g_2\ rs)$, where k is any function and g_i is a composition of a function p_i with a homomorphism $h_i = (\![\oplus_i, k_i]\!)$, then

$$h\ xs = BH(k', (h_2, \oplus_2), (h_1, \oplus_1))\ xs\ \iota_{\oplus_1}\ \iota_{\oplus_2}$$

$$\text{where} \begin{cases} k'\ (l, x, r) = k(p_1\ l, x, p_2\ r) \text{ holds,} \\ \iota_{\oplus_1} \text{ is the (left) unit of } \oplus_1, \\ \iota_{\oplus_2} \text{ is the (right) unit of } \oplus_2\,. \end{cases}$$

Proof: This has been proved by induction on the input list of h with Coq (available in [18]). ■

Theorem 1 is general and powerful in the sense that it can parallelise not only $mapAround$ but also other collective functions to *BH*.

For instance, the useful $scan$ computation

$$scan\ (\oplus)\ [x_1, x_2, \ldots, x_n] = [x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n]$$

is a special $mapAround$: $scan\ (\oplus) = mapAround\ f$ where $f(ls, x, rs) = first\ ((\![\oplus, id]\!)\ ls, x, [])$ and $first$ returns the first component of a triple.

What is more important is that any homomorphism can be parallelised with *BH*, which allows us to utilise all the theories [6], [10], [15] that have been developed for derivation of homomorphism.

Corollary 1 (Parallelisation Homomorphism with BH*):* Any homomorphism $(\![\oplus, k]\!)$ can be implemented with a *BH*. *Proof:* Notice that $(\![\oplus, k]\!) = last \circ mapAround\ f$ where $f(ls, x, rs) = ((\![\oplus, k]\!) xs) \oplus (k\, x)$ and $last$ returns the last element of a list. It follows from Theorem 1 that the homomorphism can be parallelised by a *BH*. ■

Now we consider how to deal with recursive functions. This can be done in two steps. We first use the existing theorems [9], [10], [15] to obtain homomorphisms from recursive definitions, and then use Corollary 1 to get *BH* for the derived homomorphisms.

It is worth noting that homomorphisms are very important in all our derivations of *BH*, not only because *BH* itself is a specific homomorphism, but also because many of our derivations go along with derivations of homomorphisms. For example, in Theorem 1, our main theorem, we have to derive homomorphisms so that function f can be defined in a way that the theorem can be applied.

Example 2 (Derivation for the Tower-Building Problem): From the specification given before, we can see that

Theorem 1 is applicable with

$$\begin{aligned}
&\mathit{visibleLR}\ (ls, x, rs) = k\ (g_1\ ls, x, g_2\ rs)\\
&\quad \textbf{where}\ g_1 = \mathit{maxAngleL}\\
&\qquad\quad g_2 = \mathit{maxAngleR}\\
&\qquad\quad k\ (maxl, (x_i, h_i), maxr) =\\
&\qquad\qquad (maxl < \tfrac{h+h_i-h_L}{x-x_L}) \wedge (maxr < \tfrac{h+h_i-h_R}{x_R-x})
\end{aligned}$$

provided that g_1 and g_2 can be defined in terms of homomorphisms.

By applying the theorems in [9], [10], [15], we can easily obtain the following two homomorphisms (the detailed derivation is beyond the scope of this paper).

$$\begin{aligned}
&\mathit{maxAngleL} = (\!|\uparrow, k_1|\!) \quad \textbf{where}\ \uparrow = max;\ k_1\ (x, h) = \tfrac{h-h_L}{x-x_L}\\
&\mathit{maxAngleR} = (\!|\uparrow, k_2|\!) \quad \textbf{where}\ \ k_2\ (x, h) = \tfrac{h_R-h}{x_R-x}
\end{aligned}$$

Therefore, applying Theorem 1 yields the following result in *BH*.

$$\begin{aligned}
&\mathit{tower}\ (x_L, h_L)\ (x_R, h_R)\ xs =\\
&\quad BH(k, (\mathit{maxAngleL}, \uparrow), (\mathit{maxAngleR}, \uparrow))\ xs\ (-\infty)\ (-\infty)\\
&\quad \textbf{where}\ k\ (maxl, (x_i, h_i), maxr)\\
&\qquad\qquad = (maxl < \tfrac{h+h_i-h_L}{x-x_L}) \wedge (maxr < \tfrac{h+h_i-h_R}{x_R-x}) \quad \square
\end{aligned}$$

C. Theorem Implementation in Coq

The Coq proof assistant [2] is based on the calculus on inductive construction. This calculus is a higher-order typed λ-calculus. Theorems are types and their proofs are terms of the calculus. The Coq systems helps the user to build the proof terms and offers a language of tactics to do so. Coq is also a functional programming language.

We use the Coq proof assistant to prove correct the derivations from a functional specification to a BH skeleton instantiation. Derivations and parallel term retrieving is automated using a newly introduced feature in Coq: Type classes[17]. The full code of our Coq developments can be downloaded [18].

V. BH to BSML: Certified Parallelism

We have until now supposed a certified parallel implementation of BH on which the algorithms rely. This implementation is realized using Bulk Synchronous Parallel ML (BSML) [14], an efficient BSP [7] parallel language based on Objective Caml [13] and with formal bindings and definitions in Coq. In Coq, we prove the equivalence of the natural specification of BH with its implementation in BSML, therefore being able to translate the previous BH certified versions of the algorithms to a parallel, BSML version. We also analyse the parallel performances of this implementation.

A. Bulk Synchronous Parallel ML: An Overview

a) Bulk synchronous parallelism: A BSP machine can be thought as a homogeneous distributed memory machine with a unit able to synchronise all the processors. 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 synchronisation barrier occurs, making the transferred data available for the next super-step.

The performance of a BSP machine is characterised by 3 parameters including p the number of processor-memory pairs. These parameters are the basis of the BSP performance model, omitted here for lack of space.

b) BSML parallel vectors: BSML designed as a full language, is currently implemented as a library for the Objective Caml language. BSML offers an access to the parameters of the underlying BSP architecture including the constant bsp_p (an integer).

A BSML program is not written in the Single Program Multiple Data (SPMD) style as many parallel programs are, but offers a *global view* of the parallel program. It is a usual OCaml program plus operations on a parallel data structure. This structure is called parallel vector and it has an abstract polymorphic type 'a par. A parallel vector has a fixed width bsp_p equals to the constant number of processes in a parallel execution. We will informally write $\langle x_0, \ldots, x_{n-1}\rangle$ for a parallel vector of size n, which contains the value x_i at processor i. The nesting of parallel vectors is forbidden (this can be enforced by a type system). BSML provides four primitives for the manipulation of parallel vectors. For each of these primitives, we will give its signature, its informal semantics, examples of use and its formalisation in Coq.

In the Coq developments of our framework, all the modules related to parallelism are functors that take as argument a module which provides a realization of the semantics of BSML. This semantics is modelled in a module type called BSML_SPECIFICATION. A module type or module signature in Coq is a set of definitions, parameters and axioms, the latter being types without an associated proof terms.

The module type BSML_SPECIFICATION contains : the definition processor of processor names, and associated axioms; the type par of parallel vectors; the axioms which define the semantics of the four parallel primitives of BSML.

A natural processor_max is assumed to be defined. The total number of processor, the BSP parameter bsp_p is the successor of this natural. The type processor is defined as:

```
Definition processor := { pid: nat | pid < bsp_p }.
```

A term of type {x:A | P x} is a value of type A and a proof that this value verify the property P. A value of type processor is thus a pair: A natural and a proof that this natural is lower than bsp_p.

The type of parallel vectors is an opaque type

```
Parameter par: Set→Set.
```

This means that par could take as argument a type and returns a new type: It is a polymorphic type thus it has as argument the type of the values contained in the vectors. For example, the type for parallel vectors of string could be written par string (and is written string par in BSML).

In the informal semantics above, a parallel vector is an enumeration of p values. However in our Coq formalisation, the type par is opaque: We do not know how its values are built and one cannot access directly a parallel vector component. Thus we assume that the type par comes with a function get that can access any component of the parallel vector:

```
Parameter get : ∀A: Set, par A →processor →A.
```

Given a type T, a parallel vector vec of type par T and a processor i, (get T vec i) is the value held by processor i in parallel vector vec.

c) Parallel vector creation: The primitive mkpar is used to create parallel vectors. It takes a function f as argument and creates a parallel vector that contains the value of $(f\ i)$ at processor i, $0 \le i <$ bsp_p. The signature of this primitive is: mkpar: (int→ 'a) → 'a par, and informally its semantics could be written as:

$$\text{mkpar } f = \langle f\ 0, \ldots, f\ (\text{bsp\_p} - 1)\rangle$$

In the interactive loop of BSML [1] on a machine with 6 processors, one could have the following result:

```
# let this = mkpar (fun i → i);;
val this : int Bsml.par = <0, 1, 2, 3, 4, 5>
```

is the prompt inviting us to write an expression to evaluate. The expression **let** this =... is used to define the value this by applying the primitive mkpar to the (anonymous) identity function. In the second line, the top-level answers that a value (**val**) called this has been defined, that it is a parallel vector of integers (int Bsml.par) and its value is $\langle 0, 1, 2, 3, 4, 5\rangle$.

A useful function is replicate:

```
# let replicate = fun x → mkpar(fun _ → x);;
val replicate : 'a → 'a Bsml.par = <fun>

# let vx = replicate "PDCAT";;
vx : string Bsml.par = <"PDCAT", "PDCAT", "PDCAT",
                        "PDCAT", "PDCAT", "PDCAT">
```

The evaluation of an application of mkpar to a function f is done without any communication (any usual Objective Caml value is replicated on all the processors, hence the function f), and is done during the asynchronous computation phase of a BSP super-step.

The semantics of the parallel primitives of BSML in Coq are specified using the get function. It is a quite straightforward translation of the informal semantics. Instead of giving the result parallel vector as a whole it is described by giving the values of its components. A Coq formalisation of the mkpar primitive is thus:

```
Parameter mkpar_specification : ∀(A:Set) (f: processor →A),
  { X: par A | ∀i: processor, get X i = f i }.
```

d) Point-wise parallel application: As the type par and the primitive mkpar are polymorphic, it is possible to create parallel vectors of functions. For example:

```
# let vf = mkpar(fun i→  fun x → x ^ (string_of_int i));;
val vf : (string → string) Bsml.par = ≪ fun>, ..., <fun≫
```

[1] In case the reader wants to try the examples in the interactive loop (command `bsml`) while reading the paper, she needs to write **open** Bsml;; at the beginning of the session.

Thus we may need to apply a parallel vector of functions to a parallel vector of values. Such an application is not a usual functional application: We need a primitive to perform it. This primitive is called apply. Its signature is: apply:('a→ 'b)par→ 'a par→ 'b par.

For example, we can apply the parallel vector of functions vf to the parallel vector of values vx:

```
# let v2 = apply vf vx;;
val v2 : string Bsml.par = <"PDCAT0", "PDCAT1", "PDCAT2",
                            "PDCAT3", "PDCAT4", "PDCAT5">
```

The informal semantics of apply could be written as:

$$\text{apply } \langle f_0, ..., f_{p-1}\rangle\ \langle x_0, ..., x_{p-1}\rangle = \langle f_0\ x_0, ..., f_{p-1}\ x_{p-1}\rangle$$

The evaluation of an application of apply requires no communication.

In Coq, apply is specified as follows:

```
Parameter apply_specification :
  ∀(A B: Set) (vf: par (A →B)) (vx: par A),
  { X: par B | ∀i: processor, get X i = (get vf i) (get vx i) }.
```

Very useful functions which are part of the BSML standard library could be implemented from mkpar and apply such as:

```
(* parfun: ('a → 'b) → 'a par → 'b par *)
let parfun = fun f v → apply (replicate f) v
```

There exist variants parfunN of parfun for point-wisely applying a sequential function with N arguments to N parallel vectors.

e) Communications: The two last primitives of BSML require both communication and synchronisation.

The primitive put is used to exchange data between processes. It takes and returns a parallel vector of functions. At each processor the input function describes the messages to be sent to other processors and the output function describes the messages received from other processors. Its signature is put:(int→ 'a)par→ (int→ 'a)par and it informal semantics follows:

$$\text{put } \langle f_0, \ldots, f_{p-1}\rangle = \langle \lambda i.f_i 0, \ldots, \lambda i.f_i(p-1)\rangle$$

At processor i, the function f_i encodes the bsp_p possible messages to be sent to other processors. Some values, as the empty list or the first constructor without parameters in a sum type, are considered to mean "no message". For example if we want to broadcast a value at a specific processor in a parallel vector to all other processors, we will use two functions: One that will send this value to every processor, one that will send nothing to every processor. The first of these functions should be used by the processor that is the root of the broadcast, and the second function by all other processors:

```
let choice root pid value =
  let toall = fun dst → [value]
  and nothing = fun dst → [] in
  if pid=root then toall else nothing
```

After the function choice is used to build a parallel vector of functions describing the communications we want to perform, this parallel vector is given as argument to put. Then we need to retrieve the sent values. Here again the bsp_p possible received values by a processor are encoded as a function. To know the value sent by a processor i to a processor j, one has to apply, at processor j, the obtained function to i. In the case of the broadcast, all processors need to apply the function to the processor root:

```
# let broadcast root vx =
    let msg = apply (mkpar(choice root)) vx in
    parfun List.hd (apply (put msg) (replicate root));;
val broadcast : int → 'a Bsml.par → 'a Bsml.par = <fun>

# let v3 = broadcast 3 v2;;
val v3 : string Bsml.par = <"PDCAT3", "PDCAT3", "PDCAT3",
                             "PDCAT3", "PDCAT3", "PDCAT3">
```

The evaluation of an application of the put primitive requires a full super-step: First the messages are computed from the parallel vector of functions describing the messages to send; then the messages are exchanged; and a global synchronisation ends the super-step in order to allow the functions describing the received messages to be built.

The Coq specification of put is:

```
Parameter put_specification :
  ∀(A:Set) (vf: par (processor →A)),
  { X: par (processor →A) | ∀i j: processor, get X i j = get vf j i }.
```

The last primitive proj:'a par→ (int→ 'a) is the inverse of mkpar (for functions defined on the domain of processor names). It also requires a full super-step, we omit the details for the sake of conciseness.

B. Proving Correct BSML Implementation of BH in Coq

From the axioms presented in subsection V-A above, we obtain four Coq functions that verify the BSML specifications. These functions and their properties are used when we prove the correctness of a parallel version of BH with respect to definition 1.

The main theorem is:

```
Theorem bh_bsml_bh lst: ∀(L A R B:Set) (k:L→A→R→B)
  (gl: list A→L) (opl:L→L→L) (gr:list A→R) (opr:R→R→R)
  (gl_hom : is_homomorphism A L gl opl)
  (gr_hom : is_homomorphism A R gr opr)
  (lst:list A),
  list_of_par_list(bh_bsml_comp (gl nil) (scatter lst) (gr nil))) =
  bh_comp k gl opl gr opr (gl nil) lst (gr nil).
```

It states the equivalence of bh_comp and the parallel version bh_bsml_comp. The bh_bsml_comp function is quite similar to the direct implementation of BH in BSML [18]. This function takes a distributed list (or parallel vector of lists) as input (type 'a list par in BSML and par(list A) in Coq), and also returns a distributed list whereas bh_comp takes as input a list and returns a list. Thus some conversions are needed. scatter takes a list and cuts it into pieces that are distributed. list_of_par_list does the inverse: it takes a parallel vector of lists, converts it to a function from natural to lists (using a variant of the proj primitive) and eventually merges the lists into one list.

In order to prove this theorem, two intermediate results are necessary. The proofs are technical and use several steps where sub-lists are cut and combined: They are done by considering the processor list as being the list (l1++p::nil)++l2 and by reasoning by induction on l1 and on l2 (full proofs available in [18]).

VI. Program Extraction and Experiments

Let us summarise the different steps towards a proved correct parallel implementation of the tower building problem: (a) First we specify the problem as an instance of mapAround; (b) using theorem 1, we prove that the problem is an instance of BH; (c) using theorem bh_bsml_bh, we prove that the problem is an instance of the parallel version of BH.

From this latter proof, we can extract an implementation of the tower building problem in BSML. The resulting code is of course similar in structure of the code of the direct implementation of BH in BSML [18]. The main differences are on the sequential data structures. The lists type are the one defined inductively in Coq, not the optimised ones defined in Ocaml. The BSML primitives in Coq manipulate processor and nat which rely on a Peano encoding of naturals. Thus the extracted code contains: **type** nat = | O | S **of** nat which is very inefficient for computations.

To test the differences of efficiency between extracted and non-extracted programs, we experimented with two different tower building programs: The direct implementation and the implementation extracted from the derivation in Coq. The experiments were conducted on the "Centre de Calcul Scientifique de la Région Centre (CCSC)" which is a 42 nodes cluster of IBM blades with 2 quad-core Xeon E5450 and 8 Gb of memory per blade.

We had to use only one core per CPU due to a dramatic loss of performance in MPI communications when multiple cores of the same CPU try to access to the network, and we were able to book up to 18 nodes of the cluster.

In order to avoid the garbage collector of OCaml to be triggered too often, we grew the minor heap size to 1 Gb. We performed a garbage collection after the computation and took its time into account in our benchmark. Indeed, when the data are to big to fit in the minor heap, the recurrent calls to the garbage collector dramatically hinder the overall performance.

Figure 4 shows, for 12 processors the computation time for different sizes of data. We can see that the programs execution time (and the garbage collection time) grows linearly with the amount of data. The extracted version of the program is slower than the direct implementation with a time factor between 1 and 2.5. As said earlier, this come most probably from the difference in data structure encoding. The Figure 5 show the computation time for a fixed amount of data, with an increasing number of processor. We give the time for computation and garbage collection of each implementation. We also experimented that the speedup of both implementation is linear with the number of processors, for a fixed amount of data.

VII. Related Work

Our framework combines two strength: constructive algorithmic and proved correct bulk synchronous parallel language.

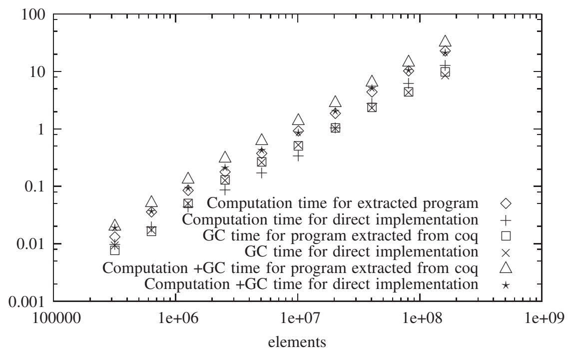

Fig. 4. Tower Building Timings (in s)

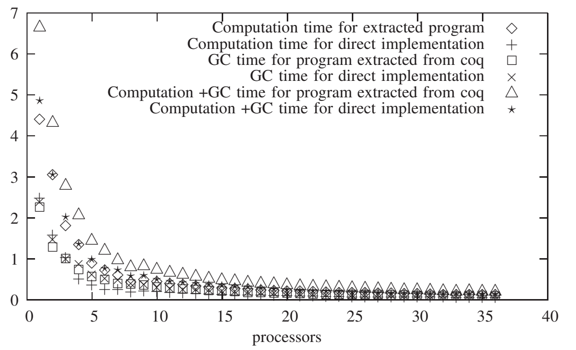

Fig. 5. Timings of direct and extracted implementations (5.12M elements)

In this paper we focus on the semantics of the programming model of Bulk Synchronous Parallel ML but as it is traditional in data-parallel languages, we also provide the semantics of an execution model which describes the parallel implementation of BSML programs as SPMD programs. We even propose a semantics which is even closer to the real implementation: a parallel abstract machine. All these semantics have been proved equivalent. Thus proving the correctness of a BSML program using the semantics of the programming model of BSML ensures *the parallel execution* of this program will also be correct.

Constructive algorithmic provides various ways to help the programmer of parallel applications to systematically derive efficient parallel algorithms [6], [9]. However there is a gap between the final composition of algorithmic skeletons obtain by derivation and its implementation. Usually in skeletal parallelism, the semantics of the execution model remains informal. Exceptions are [4], [1]. Several researchers worked on formal semantics for BSP computations, for example [12]. But to our knowledge none of these semantics were used to generate programs as the last step of a systematic development. On the contrary LOGS [5] is a semantics of BSP programs and was used to generate C programs. Compared to our approach, there is a gap between the primitives of the semantics and the implementation in C.

VIII. Conclusion

In this paper, we report our first attempt of combining the theory of constructive algorithmic and proved correct BSML parallel programs for systematic development of certified BSP parallel programs, and demonstrate how it can be useful to develop certified BSP parallel programs. Our newly proposed framework for the certified development of programs includes the new theory for the BH homomorphism, and an integration of Coq (for specification and development interaction), the BH homomorphism and BSML programming. All the certification of the transition from specification to algorithms in BH and to certified BSML parallel programs is done with the Coq proof assistant. We prove, in Coq, theorems validating the transformations of a simple, sequential specification into a more detailed and complex parallel specification. Then, using the program-extraction features of Coq yields a certified parallel program.

Acknowledgements

This work was supported by the French Ministry of Research and the Japanese Society for the Promotion of Science.

References

[1] M. Aldinucci and M. Danelutto. Skeleton-based parallel programming: Functional and parallel semantics in a single shot. *Computer Languages, Systems and Structures*, 33(3-4):179–192, 2007.
[2] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development*. Springer, 2004.
[3] G. E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, 1989.
[4] A. Cavarra, E. Riccobene, and A. Zavanella. A formal model for the parallel semantics of p3l. In *SAC '00: Proceedings of the 2000 ACM symposium on Applied computing*, pages 804–812. ACM, 2000.
[5] Y. Chen and J. W. Sanders. Logic of global synchrony. *ACM Transaction on Programming Languages and Systems*, 26(2):221–262, 2004.
[6] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.
[7] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.
[8] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In *Euro-Par'96. Parallel Processing*, LNCS 1124, pages 401–408. Springer-Verlag, 1996.
[9] S. Gorlatch. Systematic extraction and implementation of divide-and-conquer parallelism. In *Proc. Conference on Programming Languages: Implementation, Logics and Programs*, LNCS 1140, pages 274–288. Springer-Verlag, 1996.
[10] Z. Hu, H. Iwasaki, and M. Takeichi. Formal derivation of efficient parallel programs by construction of list homomorphisms. *ACM Transactions on Programming Languages and Systems*, 19(3):444–461, 1997.
[11] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *11st European Symposium on Programming (ESOP 2002)*, LNCS 2305, pages 83–97. Springer, 2002.
[12] H. Jifeng, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In *Euro-Par'96. Parallel Processing*, LNCS 1123–1124. Springer, 1996.
[13] X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml System release 3.11. http://caml.inria.fr, 2009.
[14] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In *International Conference on Computational Science*, LNCS 3515, pages 1046–1054. Springer, 2005.
[15] K. Morita, A. Morihata, K. Matsuzaki, Z. Hu, and M. Takeichi. Automatic inversion generates divide-and-conquer parallel programs. In *ACM SIGPLAN 2007 Conference on Programming Language Design and Implementation (PLDI 2007)*, pages 146–155. ACM Press, 2007.
[16] D. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.
[17] M. Sozeau. *Coq 8.2 Reference Manual*, chapter Type Classes. INRIA TypiCal, 2008.
[18] The SDPP Development Team. Systematic Development of Parallel Programs. https://traclifo.univ-orleans.fr/SDPP.