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):

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

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
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

\[
\begin{align*}
\text{h} \, [], & \quad = \, \epsilon \\
\text{h} \, [a], & \quad = \, h \, a \\
\text{h} \, (x \, ++ \, y), & \quad = \, (h \, x) \circ (h \, y)
\end{align*}
\]

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, \(\epsilon\) denotes the identity unit of \(\circ\). Since h is uniquely determined by f and \(\circ\), we will write \(h = ([\epsilon, f])\).

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 hyper-cube 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 \(\circ\), a function bh is said to be a BSP Homomorphism or BH, if it is defined in the following way.

\[
\begin{align*}
\text{bh} \, [a] \, l \, r & \quad = \, [k \, a \, l \, r] \\
\text{bh} \, (x \, ++ \, y) \, l \, r & \quad = \, \text{bh} \, x \, l \, (g_1 \, y \, \oplus \, r) \, \oplus \, \text{bh} \, y \, (l \, \circ \, g_2 \, x) \, r
\end{align*}
\]

The above bh defined with functions \(k\), \(g_1\), \(g_2\), and associative operators \(\oplus\) and \(\circ\) is denoted as \(bh = BH (k, (g_1, \circ), (g_2, \oplus))\).

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 \((g_2, \circ)\) 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 $\text{mapAround}$.

The $\text{mapAround}$, compared to $\text{map}$, describes more interesting independent computation on each element of lists. Intuitively, $\text{mapAround}$ 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.,

$$\text{mapAround } f \left[ x_1, x_2, \ldots, x_n \right] = \left[ f \left[ \emptyset, x_1, \left[ x_2, \ldots, x_n \right] \right], f \left[ \left[ x_1 \right], x_2, \left[ x_3, \ldots, x_n \right] \right], \ldots, f \left[ \left[ x_1, x_2, \ldots, x_{n-1} \right], x_n, \emptyset \right] \right].$$

In addition, we provide a set of communication functions such as $\text{permute}$, $\text{shiftL}$, $\text{shiftR}$ 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 $\text{mapAround}$, 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.

$$\text{tower } (x_L, h_L) (x_R, h_R) \, \text{xs} = \text{mapAround} \, \text{visibleLR} \, \text{xs}$$

where

$$\text{visibleLR } (ls, (x, h), rs) = \begin{cases} 
\text{visibleL } ls \, x \, \land \, \text{visibleR } rs \, x, & \text{if } \text{visibleL } ls \, x, \\
\text{visibleL } ls \, x = \text{maxAngleL } ls < \frac{h + h_L - h_R}{x_R - x_L}, & \text{otherwise}
\end{cases}$$

The inner function $\text{maxAngleL}$ 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 $b$).

$$\text{maxAngleL } [] = -\infty$$

$$\text{maxAngleL } \left[ (\left[ x, h \right]) \uplus \text{xs} \right] = \frac{h - h_L}{x_R - x_L} \uparrow \text{maxAngleL } \text{xs}$$

and the function $\text{maxAngleR}$ 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 $\text{mapAround}$ with BH):** For a function

$$h = \text{mapAround } f$$

if we can decompose $f$ as $f \left( (l, x, rs) = k (g_1 l, x, g_2 rs) \right)$, where $k$ is any function and $g_i$ is a composition of a function $p_i$ with a homomorphism $h_i = (\oplus, k_i)$, then

$$h \, \text{xs} = \text{BH} \left( k', (h_2, \oplus_2), (h_1, \oplus_1) \right) \, \text{xs} \, t_{\oplus_1} \, t_{\oplus_2}$$

where

$$\begin{align*}
\oplus_1 &: \text{the (left) unit of } \oplus_1, \\
\oplus_2 &: \text{the (right) unit of } \oplus_2.
\end{align*}$$

**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 $\text{mapAround}$ but also other collective functions to BH.

For instance, the useful scan computation

$$\text{scan } (\oplus) \left[ x_1, x_2, \ldots, x_n \right] = \left[ x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus \cdots \oplus x_n \right]$$

is a special $\text{mapAround}$; $\text{scan } (\oplus) = \text{mapAround } f$ where $f \left( (l, x, rs) = \text{first } (\left[ \oplus, \text{id} \right] l, x, \left[ \right] \right)$ and $\text{first}$ 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 $(\oplus, k) = \text{last } \circ \text{mapAround } f$ where $f \left( (l, x, rs) = (\left[ \oplus, k \right] l) x, \left[ \right] \right)$ and $\text{last}$ 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 theories [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

\[
\text{visibleLR } (ls, x, rs) = k \ (g_1 \ ls, x, g_2 \ rs)
\]

where \( g_1 = \text{maxAngleL} \)

\[
g_2 = \text{maxAngleR}
\]

\[
k \ (\text{maxl}, (x_1, h_1), \text{maxr}) =
\begin{cases}
\text{maxl} < \frac{h_1 - h_r}{x - x_l} & \text{if } \text{maxl} < \frac{h_1 - h_r}{x - x_l} \\
\text{maxr} < \frac{h_1 - h_r}{x - x_l} & \text{if } \text{maxr} < \frac{h_1 - h_r}{x - x_l}
\end{cases}
\]

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).

\[
\text{maxAngleL} = (\uparrow, k_1) \quad \text{where} \quad \uparrow = \text{max}; \quad k_1 \ (x, h) = \frac{h_h - h_r}{x - x_l}
\]

\[
\text{maxAngleR} = (\uparrow, k_2) \quad \text{where} \quad k_2 \ (x, h) = \frac{h_h - h_r}{x - x_l}
\]

Therefore, applying Theorem 1 yields the following result in BH:

\[
tower \ (x_L, h_L) \ (x_R, h_R) \ (x, h) \ (\infty) \ (\infty) =
\]

\[
\text{BH} \ (k, (\text{maxAngleL}, \uparrow), (\text{maxAngleR}, \uparrow)) \ (x, h) \ (\infty) \ (\infty)
\]

where \( k \ (\text{maxl}, (x_1, h_1), \text{maxr}) =
\begin{cases}
\text{maxl} < \frac{h_1 - h_r}{x - x_l} & \text{if } \text{maxl} < \frac{h_1 - h_r}{x - x_l} \\
\text{maxr} < \frac{h_1 - h_r}{x - x_l} & \text{if } \text{maxr} < \frac{h_1 - h_r}{x - x_l}
\end{cases}
\]

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 \( \lambda \)-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 core 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 \textit{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 \( \text{bsp\_p} \) (an integer).

A BSML program is not written in the Single Program Multiple Data (SPMD) style as many parallel programs are, but offers a \textit{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 \( \text{bsp\_p} \) 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 \( \text{bsp\_p} \) is the successor of this natural. The type processor is defined as:

\[
\text{Definition} \quad \text{processor} := \{ \text{pid} : \text{nat} | \text{pid} < \text{bsp\_p} \}.
\]

A term of type \( \{x \ A \mid P \ x\} \) 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 \( \text{bsp\_p} \).

The type of parallel vectors is an opaque type

\[
\text{Parameter} \quad \text{par} : \text{Set} \rightarrow \text{Set}.
\]

This means that \( \text{par} \) 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 \( \text{par string} \) (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 \( \text{par} \) 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 \( \text{par} \) comes with a function \( \text{get} \) that can access any component of the parallel vector:

\[
\text{Parameter } \text{get} : \forall t : \text{Set}, \text{par } t \rightarrow \text{processor } \rightarrow t.
\]

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

\( c) \) Parallel vector creation: The primitive \( \text{mkpar} \) 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 \( i \), \( 0 \leq i < \text{bsp.p} \). The signature of this primitive is: \( \text{mkpar} : (\text{int} \rightarrow \text{a}) \rightarrow \text{a } \text{par} \)

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 \( \text{apply} \). Its signature is: \( \text{apply} : (\text{a } \rightarrow \text{b}) \text{par } \rightarrow (\text{a } \text{par} ) \rightarrow \text{b } \text{par} \)

For example, we can apply the parallel vector of functions \( \text{vf} \) to the parallel vector of values \( \text{vx} \):

\[
\# \text{let } \text{vf} = \text{vf }; \text{val } \text{vf} : \text{string } \text{Bsml.par} = \langle "\text{PDCA T0}" , "\text{PDCA T1}" , "\text{PDCA T2}" , "\text{PDCA T3}" , "\text{PDCA T4}" , "\text{PDCA T5}" >
\]

The informal semantics of apply could be written as:

\[
\text{apply} (f_0 , \ldots , f_{p-1}) (x_0 , \ldots , x_{p-1}) = (f_0 x_0 , \ldots , f_{p-1} x_{p-1})
\]

The evaluation of an application of \( \text{mkpar} \) 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 \( \text{get} \) 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 \( \text{mkpar} \) primitive is thus:

\[\text{Parameter } \text{mkpar.specification} : \forall (A : \text{Set}) (f : \text{processor } \rightarrow A), \{ X : \text{par } A \mid \forall : \text{processor}, \text{get } X i = f i \} .\]

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

\[
\# \text{let } \text{vf} = \text{mkpar} (\text{fun } i \rightarrow \text{fun } x \rightarrow x \times \text{int } i);; \\
\text{val } \text{vf} : (\text{string } \rightarrow \text{string}) \text{Bsml.par} = \langle \text{fun} \rangle , \ldots , \langle \text{fun} \rangle
\]

In case the reader wants to try the examples in the interactive loop (command \text{bsml}) while reading the paper, she needs to write \text{open Bsml}; at the beginning of the session. 

\[1\text{In case the reader wants to try the examples in the interactive loop (command \text{bsml}) while reading the paper, she needs to write \text{open Bsml};; at the beginning of the session.}\]
need to retrieve the sent values. Here again the bsp procedure 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 function:

```plaintext
# 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>
```

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:

```plaintext
Parameter putSpecification :  
  ∀ (A:Set) (vf: par (processor → A)),  
  ( X: par (processor → A) | ∀ 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:

```plaintext
Theorem bh_bsmlLph lst: ∀(L A R B:Set) (k:l→A→R→B)  
  (gl: list A → L) (opL→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 (1+(p::nil)+2 and by reasoning by induction on n1 and on n2 (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_nh, 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.
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 semantics which is even closer to the real implementation: the execution model which describes the parallel implementation of Bulk Synchronous Parallel ML but as it is traditional theory of constructive algorithmic and proved correct BSML implementation in C.

Computation +GC time for program extracted from coq
Computation time for extracted program
GC time for program extracted from coq

Fig. 4. Tower Building Timings (in s)

Computation +GC time for direct implementation
Computation time for direct implementation
GC time for direct implementation

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

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**


