inria-00465838, version 2 - 23 Mar 2010

4 rue Léonard de Vinci
BP 6759
F-45067 Orléans Cedex 2
FRANCE
http://www.univ-orleans.fr/lifo

Rapport de Recherche

Systematic Development of Functional Bulk Synchronous Parallel Programs

Julien Tesson, Zhenjiang Hu,
Kiminori Matsuzaki, Frédéric Loulergue,
and Louis Gesbert

Rapport n^o **RR-2010-01**

inria-00465838, version 2 - 23 Mar 2010

inria-00465838, version 2 - 23 Mar 2010

Systematic Development of Functional Bulk Synchronous Parallel Programs

Julien Tesson
LIFO, University of Orléans
julien.tesson@univ-orleans.fr

Zhenjiang Hu
National Institute of Informatics
hu@nii.ac.jp

Kiminori Matsuzaki
IPL, University of Tokyo
kmatsu@ipl.t.u-tokyo.ac.jp

Frédéric Loulergue
LIFO, University of Orléans
frederic.loulergue@univ-orleans.fr

Louis Gesbert
LACL, University Paris-Est
gesbert@univ-paris12.fr

March 23, 2010

Abstract

With the current generalization of parallel architectures arises the concern of applying formal methods to parallelism, which allows specifications of parallel programs to be precisely stated and the correctness of an implementation to be verified. However, the complexity of parallel, compared to sequential, programs makes them more error-prone and difficult to verify. This calls for a strongly structured form of parallelism, which should not only ease programming by providing abstractions that conceal much of the complexity of parallel computation, but also provide a systematic way of developing practical programs from specification.

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 a proof assistant. The main technical contributions of this paper are as follows: 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 ; We develop a set of useful theories in Coq for systematic and formal derivation of programs in BH from specification, and we provide a certified parallel implementation of BH in the parallel functional language Bulk Synchronous Parallel ML so that a certified BSP parallel program can be automatically extracted from the proof ; We demonstrate with an example that our new framework can be very useful in practice to develop certified BSP parallel programs.

Contents

1 Introduction **3**

2 *BH*: A BSP Homomorphism **6**

3 Deriving Algorithms in BH **7**
3.1 Specification 7
3.2 Theorems for Deriving *BH* 7
3.3 Theorem Implementation in Coq 8
3.3.1 Coq Type Classes. 9
3.3.2 BH in Coq. 9
3.3.3 Derivations in Coq. 9

4 BH to BSML: Certified Parallelism **10**
4.1 An Informal Presentation of Bulk Synchronous Parallel ML 10
4.2 Bulk Synchronous Parallel ML in Coq 11
4.3 Proving Correct BSML Implementation of BH in Coq 13

5 Cost of BSML implementation **14**

6 Program Extraction and Experiments **15**

7 Related Work **17**

8 Conclusion **18**

inria-00465838, version 2 - 23 Mar 2010

inria-00465838, version 2 - 23 Mar 2010

1 Introduction

With the current generalization of parallel architectures and increasing requirement of parallel computation arises the concern of applying formal methods [12], 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, compared to sequential, programs makes them more error-prone and difficult to verify. This calls for a strongly structured form of parallelism [20, 26], 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 specification for practically nontrivial examples.

The Bulk Synchronous Parallel (BSP) model is a model for general-purpose, architecture-independent parallel programming [11]. 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 synchronizing the processors. A BSP program proceeds as a series of supersteps. In each superstep, a processor may operate only on values stored in local memory. Value sent through the communication network are guaranteed to arrive at the end of a superstep. Although the BSP model is simple and concise, it remains as a challenge to systematically develop efficient and correct BSP programs that meet given specification.

To see this clear, consider the following tower-building problem, which is an extension of the known line-of-sight problem [5]. Given a list of locations (its x position and its height) along a line in the mountain (see Figure 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 aline the same line, the problem is to find all locations from which we 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 of h at location (x_i, h_i) can be seen from (x_L, h_L) means that for any $k = 1, 2, \ldots, i-1$ the equation

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

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

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

holds. While the specification is clear, its BSP parallel program in BSML [10] (a library for BSP programming in the functional language Objective Caml) is rather complicated (Figure 2). This gap makes it difficult to verify that the implementation is correct with respect to the specification.

In this paper, we propose, as far as we are aware, the first general framework for systematic development of certified BSP parallel programs, by introducing a novel algorithmic skeleton, BSP Homomorphism (or *BH* for short). This new algorithmic skeleton plays an important role in bridging the gap between the high level specification and the low level parallel execution. Figure 3 depicts an overview of our framework. We insert a new layer, called "algorithm in *BH*", in between the specification and the certified BSP parallel programs, so as to divide the development process into two easily tacking steps: a formal derivation of algorithms in *BH* from specification and a proof of correctness of BSML implementation of *BH*.

inria-00465838, version 2 - 23 Mar 2010

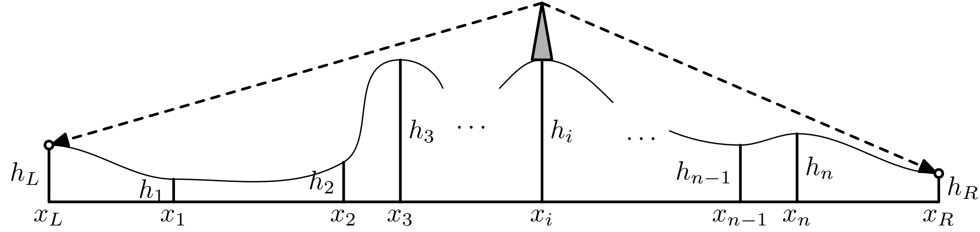

Figure 1: Tower-Building Problem

```
let sequence n1 n2 =
  let rec aux acc n2 =
    if (n1>n2) then acc else aux (n2::acc) (n2−1) in
  aux [] n2

let compose f g x = f (g x)

let procs_left = mkpar(fun i→ sequence 0 (i − 1))

let procs_right = mkpar(fun i→ sequence (i + 1) (bsp_p − 1))

let tower_building (xl,hl) lv (xr,hr) htower =
  let tan_l (x,h) = (h −. hl) /. (x −. xl) in
  let tan_r (x,h) = (h −. hr) /. (xr −. x) in
  let maxtans_l =
    parfun (fold_left (fun (m,acc) p →
                              let m = max (tan_l p) m in
                                m, m :: acc) (neg_infinity,[])) lv in
  let maxtans_r =
    parfun (fun l → fold_right (fun p (m,acc) →
                                        let m = max (tan_r p) m in
                                          m, m :: acc) l (neg_infinity,[])) lv in
  let maxtan_l,maxtanlist_l =
    parfun fst maxtans_l, parfun (compose rev snd) maxtans_l in
  let maxtan_r,maxtanlist_r =
    parfun fst maxtans_r, parfun snd maxtans_r in
  let comms = put(apply2 (mkpar(fun i ml mr dest →
                                        if dest < i then mr
                                        else if dest > i then ml
                                        else 0.)) maxtan_l maxtan_r) in
  let fold comms procs =
    fold_left (fun acc j → max acc (comms j)) neg_infinity procs in
  let maxl = parfun2 fold comms procs_left
  and maxr = parfun2 fold comms procs_right in
    apply3(parfun2 (fun maxl maxr→
                map3 (fun tan_l (x,h) tan_r →
                        (h +. htower −. hl) /. (x −. xl) > max tan_l maxl &&
                          (h +. htower −. hr) /. (xr −. x) > max tan_r maxr))
              maxl maxr)
      maxtanlist_l lv maxtanlist_r
```

Figure 2: A BSP Program in BSML for Solving the Tower-Building Problem

inria-00465838, version 2 - 23 Mar 2010

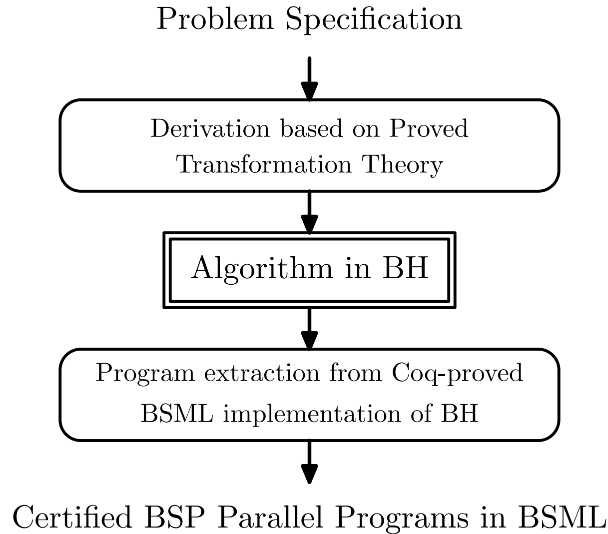

Figure 3: An Overview of our Framework for Developing Certified BSP Programs

More specifically, in our framework, specification is described as Coq definitions, which is simple for the user to be confident in its correctness without concern of parallelism. We chose to take specification as Coq definitions, for reasons of simplicity of our system, and because this also allows 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 this specification into a program using the specific *BH* skeleton, in a semi-automated way. To do so, we provide a set of Coq theories on *BH* and tools to make this transformation easier. This transformation is implemented in Coq, and proved to be correct, *i.e.* preserving the results of the initial specification. Thus, this step converts the original specification into an algorithm using our skeletons that is proved equivalent. In the second step, with a certified parallel implementation of our skeleton *BH*, we rewrite the calls to the skeleton in the algorithm. 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.

The main technical contributions of this paper can be summarized as follows.

- We define *BH*, a new homomorphic skeleton, which can not only capture the essence of BSP computation in an algorithmic level, but also serve as a bridge in mapping from high level specification to low level BSP parallel programs.
- We develop a set of useful theories in Coq for systematic and formal derivation of programs in *BH* from specification, and we provide a certified parallel implementation of *BH* in BSML so that a certified BSP parallel program can be automatically extracted from the proof.
- We demonstrate with examples that our new framework can be very useful to develop certified BSP parallel programs for solving various nontrivial problems.

The organization of this paper is as follows. We start by reviewing the basic concepts of homomorphism and defining the BSP homomorphism, one of the important computation skeleton in this paper in Section 2. We then show how to derive *BH* from specification in Section 3, and how to give its certified parallel implementation in BSML in Section 4. We demonstrate our system with several examples and report some experimental results in Section 6. We discuss the related work in Section 7 and conclude the paper in Section 8.

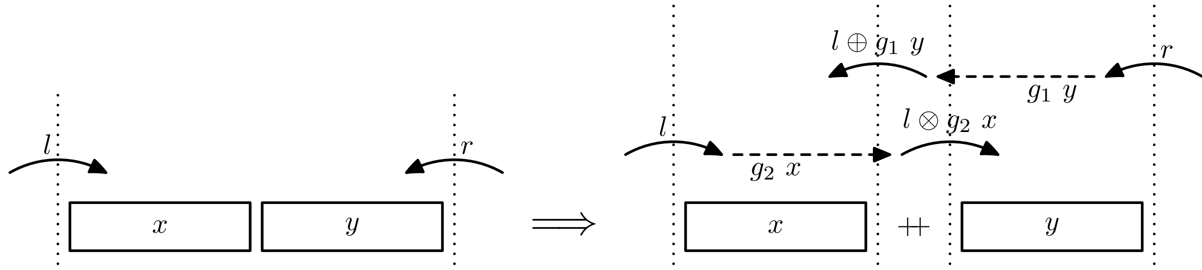

Figure 4: Information Propagation for *BH*.

2 *BH*: A BSP Homomorphism

Homomorphisms play an important role in both formal derivation of parallel programs [8, 14, 16, 18] and automatic parallelization [24]. Function h is said to be a *homomorphism*, if it is defined[1] recursively in the form of

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

where $id_\odot$ 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 [13] is introduced to treat the hyper-cube style of parallel computation, and the accumulative homomorphism [17] 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 formalize "data waiting" and "synchronization" in the superstep of the BSP model by computation of gathering necessary information around for each element of a list and then perform computation independently to update each element.

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

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

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, in the sense that 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. 4, 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 supersteps of BSP computation, but is also powerful enough to describe various computation including all homomorphisms (map and reduce) (Sect. 3), scans, as well as the BSP algorithms in [11].

[1] Our notations are basically based on the functional programming language Haskell.

inria-00465838, version 2 - 23 Mar 2010

inria-00465838, version 2 - 23 Mar 2010

3 Deriving Algorithms in BH

3.1 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.,

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

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.

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

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).

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

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

3.2 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 supersteps in BSP.

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

Theorem 1 (Parallelization *mapAround* **with** ***BH*****)** *For a function*

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

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

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

where $k'\ (l, x, r) = k(p_1\ l, x, p_2\ r)$ *holds, where* $\iota_{\oplus_1}$ *is the (left) unit of* $\oplus_1$ *and* $\iota_{\oplus_2}$ *is the (right) unit of* $\oplus_2$.
Proof. *This has been proved by induction on the input list of h with Coq (available in [1]).* □

Theorem 1 is general and powerful in the sense that it can parallelize 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 $f(ls, x, rs) = first\ ((\![\oplus, id]\!)\ ls, x, id\ rs)$.

What is more important is that any homomorphism can be parallelized with *BH*, which allows us to utilize all the theories [8, 14, 16, 18, 24] that have been developed for derivation of homomorphism.

Corollary 1 (Parallelization Homomorphism with *BH*) *Any homomorphism* $(\![\oplus, k]\!)$ *can be implemented with a* BH. **Proof.** *Notice that* $(\![\oplus, k]\!) = last \circ mapAround\ f$ *where* $f(ls, x, rs) = ((\![\oplus, k]\!) xs) \oplus (k\, x)$. *It follows from Theorem 1 that the homomorphism can be parallelized 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 [14, 16, 24] to obtain homomorphisms from recursive definitions, and then use Corollary 1 to get *BH* for the derived homomorphisms.

Example 2: Derivation of *BH* for the Tower-Building Problem. From the specification given before, we can see that Theorem 1 is applicable with

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

provided that g_1 and g_2 can be defined in terms of homomorphisms. By applying the theorems in [14, 16, 24], we can easily obtain the following two homomorphisms (the detailed derivation is beyond the scope of this paper.).

$$\begin{array}{l} maxAngleL = (\![\uparrow, k_1]\!) \quad \textbf{where}\ \uparrow = max;\ k_1\ (x, h) = \frac{h-h_L}{x-x_L} \\ maxAngleR = (\![\uparrow, k_2]\!) \quad \textbf{where}\ k_2\ (x, h) = \frac{h_R-h}{x_R-x} \end{array}$$

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

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

3.3 Theorem Implementation in Coq

We use the Coq proof assistant [2, 4] in our developments, particularly a recently introduced and still experimental feature: type classes.

inria-00465838, version 2 - 23 Mar 2010

3.3.1 Coq Type Classes.

The type class mechanism [28, 29] enables function overloading by specifying a set of functions for a parametric type (in a class) which are defined only for some types (those for which an instance of the class exists).

The example of a type class defining addition for a parametric type and its instantiation for types nat and Z follows:

```
Class Addition (A:Set) :={
  add : A→A→A;
  add_comm: ∀a b:A, add a b=add b a
}.
```

```
Instance add_nat: Addition nat :=
{ add:= plus; add_comm := plus_comm }.
Instance add_Z : Addition Z :=
{ add:= Zplus; add_comm := Zplus_comm }.
```

Once an instance of the class is constructed for a given type, all properties and functions declared by the class are available for this type. So, now we can apply add to two nat or to two Z elements. In fact add is a function that has a parameter of type Addition, it will automatically select the right operation by looking for an instance of Addition in a hint dedicated for this class. A Coq hint is a base of term available for helping certain automatic tactics. A hints for a type class can contain an instance parametrized by a type class, in this case, a matching instance will be searched.

3.3.2 BH in Coq.

Definition 1 says a *BH* function h is $BH(k, (g_1, \oplus_1), (g_2, \oplus_2))$. In fact this refers to an algorithmic skeleton which defines a program computing the function h using k, g_1, $\oplus_1$, g_2 and $\oplus_2$.

First, we have implemented such a skeleton in Coq as a higher-order function named bh_comp, and proved that for any function k and any homomorphisms $(\![g_1, \oplus_1]\!)$ and $(\![g_2, \oplus_2]\!)$, bh_comp has the properties of a *BH*.

Then we give a type class BH which is parametrized by h, k, g_1, $\oplus_1$, g_2, $\oplus_2$ and contains fields specifying that $(\![g_1, \oplus_1]\!)$ and $(\![g_2, \oplus_2]\!)$ must be homomorphisms and that bh_comp applied to k, g_1, $\oplus_1$, g_2, $\oplus_2$ is extensionally equivalent to h. The class has also a field BH_parallelisation containing a term which is a parallel implementation of h automatically constructed using the parallel implementation of the *BH* skeleton (see 4.3).

For a given function f that can be defined as a *BH*, to construct a correct parallel implementation of f we only have to build an instance of BH with f as parameter. The parallel implementation will be the field BH_parallelisation of this instance.

Homomorphism property is itself coded as a type class. So, once an instance of Homomorphism is build for $(\![g, \oplus]\!)$, it will be found automatically when trying to build an instance of BH with g and $\oplus$ as parameter.

3.3.3 Derivations in Coq.

We prove that *mapAround* is parallelizable with *BH* using Coq type classes. We first define a class specifying necessary condition for a specification using mapAround to be a *BH*. This class, MapAround_BH (Fig. 5) has as parameters the functions g_1, g_2, $\oplus_1$, $\oplus_2$ and f, and specifies that (g_1,$\oplus_1$) and (g_2,$\oplus_2$) must be homomorphisms, that f is extensionally equivalent to the function fun l ⇒mapAround f' l. Then we prove that any instance of MapAround_BH is an instance of BH.

The proof relates MapAround_BH parameters to BH ones, uses MapAround_BH homomorphisms specification to prove those of BH, and then a proof of equivalence of fun l ⇒mapAround f' l with bh_comp for any l is made. this proof is done by case on l, depending on whether l is empty, is

inria-00465838, version 2 - 23 Mar 2010

```
Class MapAround_BH {L A R B:Set} {k:L→A→R→B} {gl:list A→L} {opl:L→L→L}
  {idl:L} {gr:list A→R} {opr:R→R→R} {idr:R} (f:list A→list B) : Prop :=
{
  MA_BH_f' : (list A)*A*(list A) →B;
  f_is_mapAround : ∀l, f l = mapAround MA_BH_f' l;
  MA_BH_f'_to_k : (∀ ls x rs, MA_BH_f' (ls,x,rs) = k (gl ls) x (gr rs));
  MA_BH_right_identity :> Unit opr idr;
  MA_BH_left_identity :> Unit opl idl;
  MA_BH_left_homomorphism :> Homomorphism opl gl ;
  MA_BH_right_homomorphism :> Homomorphism opr gr
}.
```

Figure 5: The MapAround_BH Type Class

a singleton or is a list with at least two elements (so that it can be spitted in two non-empty lists). This proof use the homomorphisms properties.

As we have added MapAround_BH to the hint of BH instances, when we declare an instance of MapAround_BH for a specification f, BH properties are accessible for this function. Indeed, in the hints of BH we can find a function that takes an instance of MapAround_BH as parameter and returns an instance of BH. As there is an instance related to f in MapAround_BH hint, it can be found automatically.

To parallelize the tower-building problem we give a specification tower modular against the type being used for positions and heights description, and against functions for calculating angles.

An instance of MapAround_BH tower is build. This instance needs a proof that *maxAngleL* and *maxAngleR* are homomorphisms, then, as tower building is already specified as a mapAround, there is almost nothing to do. We just have to provide k which is very close to the *visibleLR* definition except that it take right and left maximums instead of lists of heights.

We can now build a parallel program semantically equivalent to tower like this :

```
Definition Parallel_tower := BH_parallelisation (f:=tower).
```

4 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) [21], an efficient BSP parallel language based on Objective Caml 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.

4.1 An Informal Presentation of Bulk Synchronous Parallel ML

Bulk Synchronous Parallel ML or BSML is a parallel functional language which allows the writing of Bulk Synchronous Parallel (BSP) programs. 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-

inria-00465838, version 2 - 23 Mar 2010

inria-00465838, version 2 - 23 Mar 2010

mkpar $f = \langle f0, \ldots, f(p-1) \rangle$
apply $\langle f_0, \ldots, f_{p-1} \rangle \ \langle x_0, \ldots, x_{p-1} \rangle = \langle f_0 \ x_0, \ldots, f_{p-1} \ x_{p-1} \rangle$
put $\langle f_0, \ldots, f_{p-1} \rangle = \langle \lambda i. f_i 0, \ldots, \lambda i. f_i (p-1) \rangle$
proj $\langle x_0, \ldots, x_{p-1} \rangle = \lambda i. x_i$

Figure 6: Bulk Synchronous Parallel ML Primitives

step. The BSP model comes with a performance model which we omit here due to the lack of space.

A BSML program is not written in the Single Program Multiple Data (SPMD) style as many programs are. It is a usual ML 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 p equals to the constant number of processes in a parallel execution. The nesting of parallel vectors is forbidden.

BSML gives access to the BSP parameters of the underlying machine by four constants. It also provides four primitives for the manipulation of parallel vectors. The informal semantics of these primitives is shown in Fig. 6.

mkpar:(int→'a)→'a par creates a parallel vector.

apply:('a→'b)par→'a par→'b par applies a parallel vector of functions to a parallel vector of values. The evaluation of applications of these functions do not require communications nor synchronization.

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

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

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

The two last primitives require both communication and synchronization.

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

The primitive **proj**:'a par→(int→'a) is the inverse of **mkpar** (for functions defined on the domain of processor names)

As an example we present in Fig. 7 a BSML implementation of the BH algorithmic skeleton where sequence n1 n2 returns the list [n1 ; ... ; n2], bh_seq is a sequential implementation of BH and type of communicated data is **type** ('l,'r) comm_type = Lcx **of** 'l | Rcx **of** 'r | Ncx.

Given a list that is split into local chunks in the order of the processors, the preliminary local computation (applying gl and gr) can be done with the **apply** primitive; then a **put** is used to send the computed r value leftwards to every other processor, and the l value rightwards. **apply** is then sufficient to locally compute BH on the local chunks using the sequential version of BH.

4.2 Bulk Synchronous Parallel ML 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 modeled in a module type called BSML_SPECIFICATION. A module type or module

inria-00465838, version 2 - 23 Mar 2010

```
let bh_par k gl opl gr opr = fun (l: 'l) (lst: 'a list par) (r: 'r) →
  let accl = parfun gl lst
  and accr = parfun gr lst in
  let comms = put(apply(
                     apply(
                        mkpar(fun i accr accl receiver →
                           if i < receiver then Lcx accl
                           else if i > receiver then Rcx accr
                           else Ncx)) accl)accr) in
  let lv = parfun2
          (fun c pl →
             fold_left
                (fun acc i →opl acc (match c i with Lcx x→x))
                l
                pl)
           comms
           (mkpar(fun pid→sequence 0 (pid−1)))
  and rv = parfun2
           (fun c pr →
             fold_right
                (fun i acc →opr (match c i with Rcx x→x) acc)
                pr
                r)
            comms
            (mkpar(fun pid→sequence (pid+1) (bsp_p−1))) in
    parfun3 (bh_seq k gl opl gr opr) lv lst rv
```

Figure 7: An Implementation of BH in BSML

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 opaque 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 }.
```

The type of parallel vectors is an opaque type **Parameter** par: **Set**→**Set**. It is a polymorphic type thus it has as argument the type of the values contained in the vectors. To access the members of a parallel vector, i.e. the local values, a function get having the following specification is assumed to be defined:

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

The semantics of the parallel primitives of BSML are then specified using the get function. It is a quite straightforward translation of the informal semantics presented in Fig. 6. Instead of giving the result parallel vector as a whole it is described by giving the values of its components as follows:

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

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) }.

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

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

A term of type {x:A | P x} is a value of type A and a proof that this value verify the property P. Thus from the axioms presented in above we obtain four functions that verify the specifications. These functions and their properties are used when we prove the correctness of the parallel version of BH with respect with its definition.

4.3 Proving Correct BSML Implementation of BH in Coq

The main theorem of this part 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 given in Fig. 7. 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

inria-00465838, version 2 - 23 Mar 2010

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. They state that at a given processor, the Coq formalization of vl and vr in Fig. 7 are respectively equal to applying gl (resp. gr) to the sub-list of elements being on the left (resp. the right) of the local sub-list. The proofs are technical and use several steps where sub-lists are cut and combined. The proofs are done by considering the processor list as being the list (l1++p::nil)++l2 and by reasoning by induction on l1 for vl and on l2 for vr.

5 Cost of BSML implementation

The BSP programming model offers a cost model to predict program performance. For one super-step over p processor, the cost is

$$Max_{i=0}^{p-1}\,(w_i) + Max_{i=0}^{p-1}\,(Max(h_i^+, h_i^-)) \times g + L$$

where, at processor i, w_i is the local sequential work performed during the computation phase, h_i^+ is the size of data sent to other processor, and h_i^- the received data. g is the bandwidth of the network, L its latency.

In the following we will use $\overline{f}$ to denote the function giving the time complexity of the function f. For example, $\overline{map}\ x$ is $\mathcal{O}(length\ x)$.

$|f|$ will denote the size of the term f after reduction: For a list of fixed-size elements l, $|map\ id\ l|$=$|l|$=$\mathcal{O}(n)$ where n is the number of elements of l ; for a fixed-size element x (integer, float, double, ...) $|x|$ is $\mathcal{O}(1)$.

$\overbrace{f\ x}$ will denote the reduction of the term $f\ x$, i.e. the result of the application of f to x.

BSML implementation of BH (with parameters $k\ gl\ \oplus_l\ gr\ \otimes_r\ lst$) has the following complexity: The computation is done in one and a half BSP step which starts from a parallel computation, performs a communication phase (with synchronization); then, an other parallel computation follows. So the cost is

$$seq_1 + Max_{i=0}^{p-1}\,(Max(h_i^+, h_i^-)) \times g + seq_2$$

with $seq_1 = Max_{i=0}^{p-1}\,(\overline{gl}\ lst_i + \overline{gr}\ lst_i)$.

Processor i dealing with a part lst_i of lst sends $\overbrace{gl\ lst_i}$ to processors at its left (with smaller pid) and $\overbrace{gr\ lst_i}$ to processors at its right (with greater pid). So communicated data size is:

$$\begin{aligned}
h_i^+ &= (p-1-i) \times |gl\ lst_i| + (i \times |gr\ lst_i|) \\
h_i^- &= \sum_{j=0}^{i} |gl\ lst_j| + \sum_{j=i+1}^{p-1} |gr\ lst_j|
\end{aligned}$$

and

$$\begin{aligned}
seq_2 = \ & Max_{i=0}^{p-1}\,(\\
& \quad \overline{fold_left}\ \oplus_l\ l\ [\overbrace{gl\ lst_0}; \ldots; \overbrace{gl\ lst_{i-1}}] \\
& + \overline{fold_right}\ \otimes_r\ [\overbrace{gr\ lst_{i+1}}; \ldots; \overbrace{gr\ lst_{p-1}}]\ r \\
& + \overline{Bh_seq}\ \oplus_l\ \otimes_r\ gl\ gr\ k\ l\ r\ lst_i \\
&)
\end{aligned}$$

inria-00465838, version 2 - 23 Mar 2010

inria-00465838, version 2 - 23 Mar 2010

For a local list $lst = [e_0; \ldots; e_n]$, the sequential BH computation has the following complexity:

$\overline{Bh_seq}\ \oplus_l\ \ \otimes_r\ \ gl\ gr\ k\ l\ r\ lst_i\ =$

$$
\begin{array}{ll}
 & \sum_{e \in lst} \overline{gl}\ [e] \\
+ & \sum_{e \in lst} \overline{gr}\ [e] \\
+ & \overline{fold_left}\ \oplus_l\ \ l\ [\overline{gl\ e_0}; \ldots; \overline{gl\ e_n}] \\
+ & \overline{fold_right}\ \otimes_r\ \ [\overline{gr\ e_0}; \ldots; \overline{gr\ e_k}]\ r \\
+ & \mathcal{O}(lengh\ lst_i) \\
+ & \sum_{i=0}^{n}(\\
 & \overline{k} \\
 & \quad \overline{(fold_left\ \oplus_l\ \ l\ [gl\ e_0; \ldots; gl\ e_i])} \\
 & \quad e_n \\
 & \quad \overline{fold_right\ \otimes_r\ \ [gr\ e_0; \ldots; gr\ e_n]\ r}
\end{array}
$$

If $\oplus_l$ and $\otimes_r$ have constant complexity, the equation is greatly simplified:

$$
\begin{array}{ll}
seq_2 = Max_{i=0}^{p-1}\ (& \\
 & \mathcal{O}(length\ lst_i) \\
 + & \sum_{e \in lst_i} gl\ e + \sum_{e \in lst_i} gr\ e \\
 + & \sum_{n=0}^{length\ lst_i}(\\
 & \overline{k} \\
 & \quad \overline{(fold_left\ \oplus_l\ \ l\ [gl\ e_0; \ldots; gl\ e_n])} \\
 & \quad e_n \\
 & \quad \overline{fold_right\ \otimes_r\ \ [gr\ e_0; \ldots; gr\ e_n]\ r} \\
 & \quad) \\
)
\end{array}
$$

Tower Building Complexity

With the example of tower building, $\oplus_l = \otimes_r = \uparrow$ has constant complexity providing that we can do a comparison in constant time. *maxAngleL* and *maxAngleR* have the same complexity: $\overline{maxAngleL}\ lst = \overline{maxAngleR}\ lst = length\ lst$. So extracted Tower Building has complexity $Max_{i=0}^{p-1}\ (\mathcal{O}(length\ lst_i))) + g * \mathcal{O}(p) + L + Max_{i=0}^{p-1}\ (\mathcal{O}(length\ lst_i))$. Thus, if the list is equally distributed the complexity is

$$\mathcal{O}(length\ lst/p) + g * \mathcal{O}(p) + L + \mathcal{O}(length\ lst/p)$$

6 Program Extraction and Experiments

Let us summarize the different steps towards a proved correct parallel implementation of the tower building problem:

- First we specify the problem as an instance of mapAround;

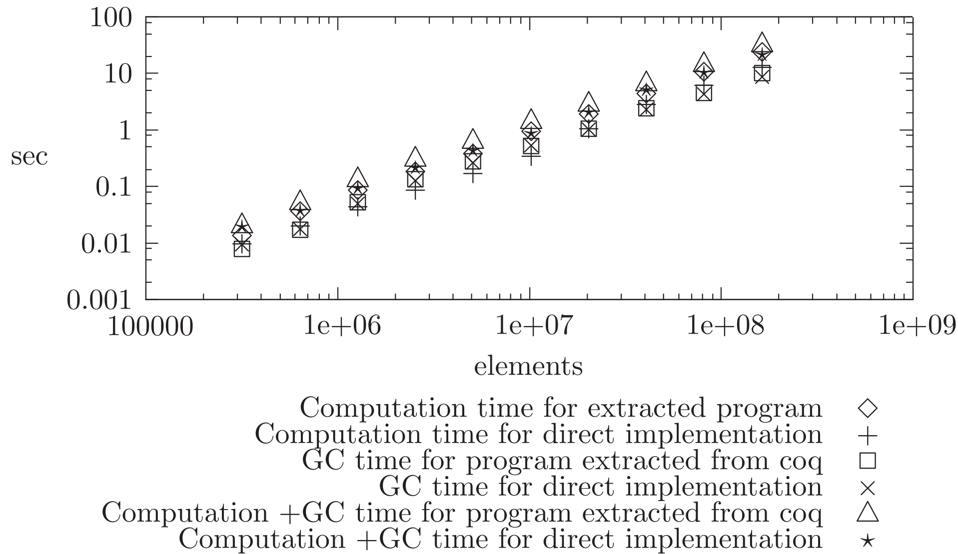

Figure 8: Timings (in s) of direct implementation and extracted one over 12 CPU

- using theorem 1, we prove that the problem is an instance of BH;
- 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 (Fig. 7). The main differences are on the sequential data structures. The lists type are the one defined inductively in coq, not the optimized 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 implementationand the implementation extracted from the derivation in Coq. The experiments were conducted on the "Centre de Calcul Scientifique de la rgion Centre (CCSC)" which is a 42 nodes cluster of IBM blade 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 communication when multiple core of the same cpu try to access to the network, and we where able to book only 18 nodes of the cluster.

In order to avoid the garbage collector of OCaml to be triggered to 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 8 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 9 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 im-

inria-00465838, version 2 - 23 Mar 2010

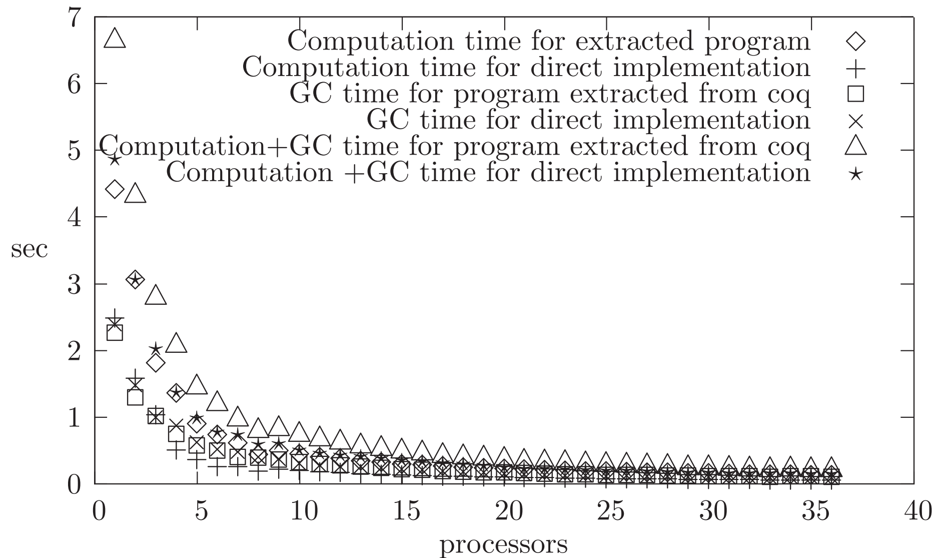

Figure 9: Timings (in s) of direct implementation and extracted one over 5120000 elements

plementation. As shown by figure10, the speedup of both implementation is linear with the number of processors.

7 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, which was first express as an extension of the λ-calculus [22]. It is possible to implement this semantics as a sequential program, for example by realizing parallel vectors by lists or arrays. 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 [9]. 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 (up to the correctness of the implementation of the parallel abstract machine).

But BSML relies, for communications, on the `put` operation which is not very easy to use. On the contrary, constructive algorithmic provides various ways to help the programmer of parallel applications to systematically derive efficient parallel algorithms [8, 14]. 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. Exception are [3, 6]. Several researchers worked on formal semantics for BSP computations, for example [19, 27, 30]. But to our knowledge none of these semantics was used to generate programs as the last step of a systematic development.

On the contrary LOGS [7] is a semantics of BSP programs and was used to generate C programs [32]. Compared to our approach, there is a gap between the primitives of the semantics and the implementation in C. Our programs are extracted from Coq proof terms.

There exists work on bulk synchronous parallel algorithmic skeletons [15, 31]: In these approaches the derivation or optimization process is guided by the BSP cost model, but the gap

inria-00465838, version 2 - 23 Mar 2010

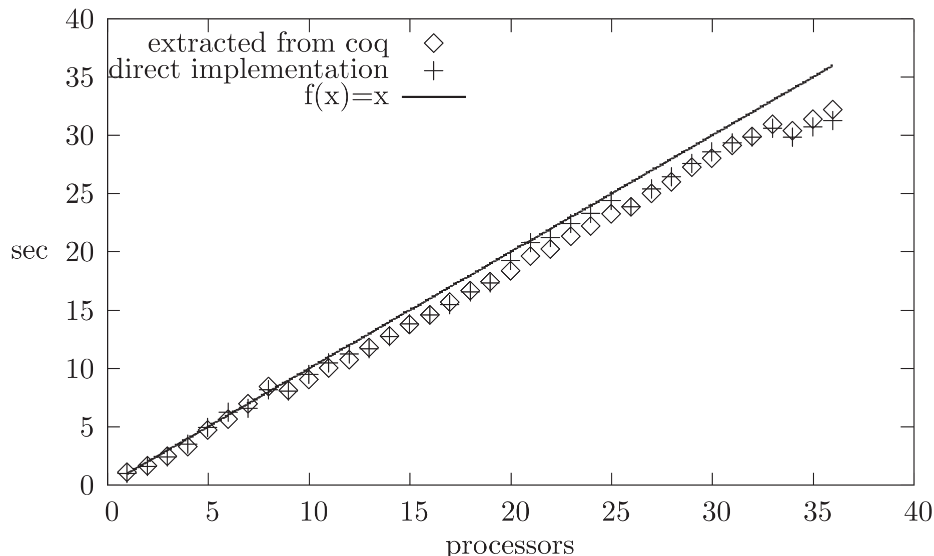

Figure 10: speedup of direct implementation and extracted implementation over 5120000 elements (calculated from computation + GC timings)

between the final composition of skeletons and the implementation is still there.

More recently, BSPA [23] is a process algebra with which one can describe BSP programs, with a notion of enumerated parallel vector and global synchronization. It also preserves the point-to-point communications of CCS. This allows in particular to describe the coordination of several BSP machines. An extended BSP cost model is proposed, compatible with strong bisimulation. The main motivation of this work is to reason formally about performance in this extended setting rather than to derive programs.

It is worth noting that the idea of using formal approaches for transforming abstract specifications into concrete implementations was also proposed as abstract parallel machines in [25]. Compared with the study, the contribution of the paper is that we proposed and realized a concrete framework based on the Coq proof assistant and the BSP model.

8 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.

Acknowledgments. Louis Gesbert was partly supported by the French Ministry of Research, Louis Gesbert and Frdric Loulergue were partly supported by the ACI Program, project "Certified Parallel Programming" and the Japanese Society for the Promotion of Science. Julien

inria-00465838, version 2 - 23 Mar 2010

inria-00465838, version 2 - 23 Mar 2010

Tesson is supported by a grant from the French Ministry of Research. This work was partly supported by the University of Orléans. The experiments were done using the machines of the *Centre de Calcul Scientifique du Centre* funded by the *Région Centre*.

References

[1] Systematic Development of Parallel Programs. `https://traclifo.univ-orleans.fr/SDPP`.

[2] The Coq Proof Assistant. http://coq.inria.fr.

[3] 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.

[4] Y. Bertot. Coq in a hurry, 2006. http://hal.inria.fr/inria-00001173.

[5] G. E. Blelloch. Scans as primitive operations. *IEEE Trans. on Computers*, 38(11):1526–1538, November 1989.

[6] 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.

[7] Y. Chen and J. W. Sanders. Logic of global synchrony. *ACM Transaction on Programming Languages and Systems*, 26(2):221–262, 2004.

[8] M. Cole. Parallel programming with list homomorphisms. *Parallel Processing Letters*, 5(2):191–203, 1995.

[9] F. Gava. *Approches fonctionnelles de la programmation paralléle et des méta-ordinateurs. Sémantiques, implantations et certification.* PhD thesis, University Paris Val-de-Marne, LACL, 2005.

[10] F. Gava and F. Loulergue. A Parallel Virtual Machine for Bulk Synchronous Parallel ML. In P. M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part I*, number 2657 in LNCS, pages 155–164. Springer Verlag, june 2003.

[11] A. V. Gerbessiotis and L. G. Valiant. Direct bulk-synchronous parallel algorithms. *Journal of Parallel and Distributed Computing*, 22(2):251–267, 1994.

[12] G. Gopalakrishnan and R. M. Kirby. Formal Methods for MPI Programs. *Electronic Notes in Theoretical Computer Science*, 193:19–27, 2007. doi:10.1016/j.entcs.2007.10.005.

[13] S. Gorlatch. Systematic efficient parallelization of scan and other list homomorphisms. In L. Bouge, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, Lecture Notes in Computer Science 1124, pages 401–408. Springer-Verlag, 1996.

[14] 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.

[15] Y. Hayashi and M. Cole. Automated cost analysis of a parallel maximum segment sum program derivation. *Parallel Processing Letters*, 12(1):95–111, 2002.

inria-00465838, version 2 - 23 Mar 2010

[16] 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.

[17] Z. Hu, H. Iwasaki, and M. Takeichi. An accumulative parallel skeleton for all. In *11st European Symposium on Programming (ESOP 2002)*, pages 83–97, Grenoble, France, April 2002. Springer, LNCS 2305.

[18] Z. Hu, M. Takeichi, and W.-N. Chin. Parallelization in calculational forms. In *25th ACM Symposium on Principles of Programming Languages (POPL'98)*, pages 316–328, San Diego, California, USA, January 1998. ACM Press.

[19] H. Jifeng, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96. Parallel Processing*, number 1123–1124 in LNCS, Lyon, August 1996. LIP-ENSL, Springer.

[20] R. Loogen, Y. Ortega-Mallén, R. Peña, S. Priebe, and F. Rubio. Parallelism Abstractions in Eden. In F. A. Rabhi and S. Gorlatch, editors, *Patterns and Skeletons for Parallel and Distributed Computing*, volume 2011 of *LNCS*. Springer, 2002.

[21] F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In V. S. Sunderam, G. D. van Albada, P. M. A. Sloot, and J. Dongarra, editors, *International Conference on Computational Science, Part II*, number 3515 in LNCS, pages 1046–1054. Springer, 2005.

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

[23] A. Merlin and G. Hains. A bulk synchronous process algebra. *Computer Languages, Systems and Structures*, 33:111–133, 2007.

[24] 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, June 2007.

[25] J. O'Donnell and G. Rünger. Abstract Parallel Machines. *Computers and Artificial Intelligence*, 19(2), 2000.

[26] D. Skillicorn. *Foundations of Parallel Programming*. Cambridge University Press, 1994.

[27] D. Skillicorn. Building BSP Progams Using the Refinement Calculus. In D. Merry, editor, *Workshop on Formal Methods for Parallel Programming: Theory and Applications*, pages 790–795. Springer, 1998.

[28] M. Sozeau. *Coq 8.2 Reference Manual*, chapter Type Classes. INRIA TypiCal, 2008.

[29] M. Sozeau and N. Oury. First-Class Type Classes. In O. A. Mohamed, C. Muñoz, and S. Tahar, editors, *Theorem Proving in Higher Order Logics (TPHOLs)*, volume LNCS 5170, pages 278–293. Springer, 2008.

[30] A. Stewart, M. Clint, and J. Gabarró. Barrier synchronisation: Axiomatisation and relaxation. *Formal Aspects of Computing*, 16(1):36–50, 2004.

[31] A. Zavanella. Skeletons, BSP and performance portability. *Parallel Processing Letters*, 11(4):393–405, 2001.

[32] J. Zhou and Y. Chen. Generating C code from LOGS specifications. In *2nd International Colloquium on Theoretical Aspects of Computing (ICTAC'05)*, number 3407 in LNCS, pages 195–210. Springer, 2005.

inria-00465838, version 2 - 23 Mar 2010