Implementing Powerlists with Bulk Synchronous Parallel ML

Frédéric Loulergue*, Virginia Niculescu†, Julien Tesson‡
*Univ Orléans, INSA Centre Val de Loire, LIFO EA 4022, Orléans, France,
Frederic.Loulergue@univ-orleans.fr
†Faculty of Mathematics and Computer Science, Babeş-Bolyai University, Cluj-Napoca, Romania
vniculescu@cs.ubbcluj.ro
‡Université Paris Est Créteil, LACL, Créteil, France
Julien.Tesson@lacl.fr

***Abstract*—Tools and methods able to simplify the development process of parallel software, but also to assure a high level of performance and robustness, are necessary. Powerlists and their variants are data structures that can be successfully used in a simple, provably correct, functional description of parallel programs, which are divide-and-conquer in nature. The paper presents how programs defined based on powerlists could be implemented in the functional language OCaml plus calls to the parallel functional programming library Bulk Synchronous Parallel ML. BSML functions follow the BSP model requirements, and so its advantages are introduced in OCaml parallel code. In order to write powerlist programs in BSML we provide a data-type for powerlists and a set of skeletons (higher-order functions implemented in parallel) to manipulate them. Examples are given and concrete experiments for their executions are conducted.**

Keywords*—*Parallel recursive structures; Functional parallel programming; Bulk synchronous parallelism

I. Context and Motivation

The latest developments of the computation systems lead to an increase of the requirements in using parallel computation. Still, for many years, parallel computation has been considered difficult and error-prone. This imposes using tools and methodologies able to simplify the development process of parallel software, but also to assure a high level of performance and robustness.

This calls for a strongly structured form of parallelism [1], [2], [3], which should not only be based on 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. Since correctness is very important in this context, high-level algebraic theories are appropriate to be used as fundamentals. Among them at least two seem suitable for parallel programming: the theory of lists [4] and parallel recursive structures such as powerlists [5].

Powerlists and their variants are data structures that can be successfully used in a simple, provably correct, functional description of parallel programs, which are divide and conquer in nature [5], [6]. For each data structure, theories based on algebras and structural induction principles have been specified, which make them well suited to formally define recursive, data-parallel algorithms. These theories can be considered together a base for a model of parallel computation with a very high level of abstraction [7].

In order to be useful, a model of parallel computation must also address very carefully issues such as efficient implementation and costs evaluation. BSP model [2] is famous especially because it provides a very accurate cost analysis, and a rigorous development that could provide robustness. From these, the idea of using BSP development methodology in the process of implementing powerlists programs came natural.

Powerlist programs are defined in a very high-level way. While their divide-and-conquer nature makes them suitable for parallelism, providing an efficient implementation that preseves a high-level style is not an easy task. A full framework for the development of parallel programs using powerlists should provide a way for a user to simply express his/her algorithms in a high-level way, and transformations to obtain more efficient versions of the programs. Actually using the powerlist algebraic properties, Achatz et al. [8] proposed a method to do so, with the aim of running the optimised versions of powerlists programs on SIMD architectures. Their method transforms programs written using a set of input patterns (or skeletons) into equivalent programs using a set of output patterns. The output patterns are more efficient to run on SIMD architectures than the input patterns. While Achatz et al. transformations are "pen-and-paper" transformations, and thus error prone, our ultimate goal is to automate variants of these transformations in the Coq proof assistant [9], to be able to extract functional parallel programs.

In order to do so, we need to have a set of output patterns written in a parallel functional parallel language that could be a target of Coq extraction mechanism: OCaml [10], [11] plus calls to the parallel functional programming library Bulk Synchronous Parallel ML (BSML) [12] is such a language [13]. BSML functions follow the BSP model requirements, and so its advantages are introduced in OCaml parallel code. The design and implementation of a set of such output patterns (or skeletons) is the contribution of this paper.

The paper is organised as follows. We give first a general description of the powerlists in section II, and on BSML (section III) before discussing how powerlists programs could be implemented in BSML (section IV). Section V presents some applications and the experiments related to them. Related work (section VI) and also our goals for the future work are

presented before giving the conclusions (section VII).

The paper assumes some familiarity with statically typed higher-order functional programming language such as Haskell, SML or OCaml. A concise introduction to OCaml is [14].

II. POWERLISTS

Powerlist data structures were introduced by J. Misra [5], and they allow working at a high level of abstraction, especially because the index notations are not used. To assure methods that verify the correctness of parallel programs, an algebra and structural induction principles are defined on these data structures. The functions and the operators, which represent the parallel programs, are defined on these structures based on corresponding structural induction principles.

A powerlist is a linear data structure whose elements are all of the same type. The length of a powerlist data structure is a power of two. The type constructor for powerlist is:

$$powerlist : Type \times \mathbb{N} \to Type$$

and so, a powerlist l with 2^n elements of type X is specified by $powerlist.X.n$ (where $n = log\ (length\ l)$, and the real length of l is 2^n). A powerlist with a single element a is called a *singleton*, and is denoted by $[a]$. If two powerlist structures have the same length and elements of the same type, they are called *similar*.

Two *similar* powerlists can be combined into a powerlist data structure with double length, in two different ways: using the operator *tie* $p \mid q$; the result contains elements from p followed by elements from q; using the operator *zip* $p \natural q$; the result contains elements from p and q, alternatively taken.

Powerlist algebra is defined by operators and axioms, and the existence of unique decomposition of a powerlist, using one of *tie* or *zip* operators, is assured. A structural induction principle is defined on powerlist data structures, which consider a base case, and two possible variants for the inductive step: one based on operator *tie*, and the other based on *zip*.

For example, the high order function map, which applies a scalar function to each element of a powerlist is defined as follows:

$$\begin{array}{l} map : (X \to Z) \times powerlist.X.n \to powerlist.Z.n \\ map\ f\ [a] = [f\ a] \\ map\ f\ (p \mid q) = map\ f\ p \mid map\ f\ q \end{array}$$

Function inv permutes the input list p such that the element with index b in p will be on the position given by the reversal of bit string b in p:

$$\begin{array}{l} inv :\ powerlist.X.n \to powerlist.X.n \\ inv\ [a] = [a] \\ inv\ (p \mid q) = inv\ p \natural inv\ q \end{array}$$

The parallelism of the functions is implicit: each application of a deconstruction operator (*zip* or *tie*) means that we may achieve two processes (programs) that could run in parallel. So, we obtain a tree decomposition, which is specific to divide&conquer programs.

Having two decomposition operators eases the definition of different programs (as can be noticed from inv definition), but in the same time induces some problems when these high-level programs have to be implemented on concrete parallel machines.

In [8] Achatz and Schulte present transformation rules to parallelize divide-and-conquer (DC) algorithms over powerlists. Their goal was to derive programs for the *massively data parallel model*. The rules convert the parallel multiple control structure of DC into a single control flow structure, thereby making the implicit massive data parallelism in a DC scheme explicit. The transformations use some predefined functions and operators.

The *apply-to-all* $*$ operator represents parallel application of a scalar function (that takes one or several arguments) to each element of one or several powerlist. When there is only one argument, and one powerlist, $*$ is indeed map. The function $join$ is used as a specialisation of a *parallel conditional*, and functions that exhibit communication patterns: $corr$, $distL/distR$, and inv are used too. The operator $\#$ is used to return the length of a powerlist.

Function $join$ transform a pair of powerlists p, q, having equal lengths, into a new powerlist, which consists of alternate slices of p and q each of length $n = 2^i$, $0 \le i < log_2(\#p)$. Formally, it is defined by:

$$\begin{array}{ll} join\ n\ (p \mid q)(r \mid s) = p \mid s & \text{if } n = \#p \\ join\ n\ (p \mid q)(r \mid s) = join\ n\ p\ r \mid join\ n\ q\ s, & \text{if } n < \#p \end{array}$$

The function $corr$ expresses butterfly-like communication pattern, and $distL/distR$ express directed broadcast. Their definitions are:

$$\begin{array}{ll} corr\ n\ (p \mid q) = q \mid p & \text{if } n = \#p \\ corr\ n\ (p \mid q) = corr\ n\ p \mid corr\ n\ q & \text{if } n < \#p \\ distL\ n\ p = copy\ n\ (last\ p) & \text{if } n = \#p \\ distL\ n\ (p \mid q) = distL\ n\ p \mid distL\ n\ q & \text{if } n < \#p \end{array}$$

where the function *copy* returns a powerlist of identical elements. The function $distR$ is similar to $distL$ but operates from left to right. The function $first$ applied to a powerlist returns the first element of the powerlist, and the function $last$ returns the last one.

The function $mapn$ is defined by:

$$\begin{array}{lll} mapn\ n\ f\ p & = f\ p & \text{if } n = \#p \\ mapn\ n\ f\ (p|q) & = mapn\ n\ f\ p \mid mapn\ n\ f\ q & \text{if } n \le \#p \end{array}$$

Note that for $mapn$, contrary to map, f is a function on powerlists, not on scalars. In the same way $*$ generalises map, the operator $*_n$ generalises $mapn$ and may take m powerlists as arguments if the function f takes m arguments. $*_n$ is called the *apply-to-slices* operator.

The idea presented in [8] is to transform the DC scheme expressed as a powerlist function f into a data parallel computation, based on tail-recursive computation.

The transformation considers the computation as a top-down, or as a bottom-up computation. $F\downarrow$ exhibits top-down, and $F\uparrow$ bottom-up computation, and they represents input patterns.

Output patterns are represented by $F \Downarrow$ and $F \Uparrow$ that describes tail-recursive top-down, resp. bottom-up computation with pre-adjustment, resp. post-adjustment. $F\Uparrow$ is defined by

$$\begin{array}{rcl} F\Uparrow\ \delta\ l\ r\ s\ p & = & f\Uparrow\ \#p\ 1\ s\ p \text{ where} \\ f\Uparrow\ 1\ n\ s\ p & = & p \\ f\Uparrow\ (2m)\ n\ s\ p & = & f\Uparrow\ m\,(2n)\,(\delta s)(join\ n \\ & & \qquad ((l\ s) *_n\ p\ q)((r\ s) *_n\ q\ p)) \end{array}$$

where $q = corr\ n\ p$, l and r are postadjusting functions and δ is a function that updates the scalar s at each step.

The correspondence between $F\downarrow$ resp. $F\uparrow$, and $F\Downarrow$ resp. $F\Uparrow$ is established and proved by induction. For example, for top-down computation we have $F\downarrow\,|\,| \equiv F\Downarrow$ and expresses the equivalence between cascading recursion of $F\downarrow\,|\,|$ to the tail-recursive computation of $F\Downarrow$. (The correspondence between the general patterns $F\uparrow$ and $F\Uparrow$ is more complicated, but could be simplified for concrete cases.) because the operator zip is unnatural when used as a basis for parallelism, the input patterns are transformed using the inv function, such that operator zip is replaced by the operator tie.

The general strategy to optimise a function f on powerslits, is based on the following steps ($\updownarrow$ and $\Updownarrow$ indicate either a top-down or bottom-up computation):

1) modify f to match a predefined input named $f\updownarrow$;
2) rewrite $f\updownarrow$ to replace $\natural$ by $|$;
3) parallelise $f\updownarrow$ to $f\Updownarrow$ based on a parallel application (the operator $*_n$ (*apply-to-slices*) is used);
4) specialise $f\Updownarrow$ to eliminate $*_n$;
5) optimise $f\Updownarrow$ to increase parallelism.

The strategy presented by Achatz and Schulte could be very useful in the process of automatising the implementation of the powerlists programs. This leads to a framework formed by high level functions that could be used in the implementation of each specific powerlist program. More details could be found in [8].

Example: Prefix sum: One of the most useful building blocks for parallel programs is the *prefix* function which takes a binary, associative operator $\oplus$ and a list $[p_0, \ldots, p_{n-1}]$, and returns a list of all prefix "sums" $[p_0, p_0 \oplus p_1, \ldots, p_0 \oplus \cdots \oplus p_{n-1}]$. Since the computation is done only in the bottom-up phase, it could be expressed as:

$$\begin{array}{rcl} ps\uparrow \oplus [x] & = & [x] \\ ps\uparrow \oplus (p \mid q) & = & v \mid ((last\ v)\oplus)\ *\ w \\ & & \text{where}(v, w) = (ps\uparrow \oplus p, ps\uparrow \oplus q) \end{array}$$

For transformation, the first two steps are trivially passed, and finally it is obtained:

$$\begin{array}{rcl} ps\uparrow \oplus p & \equiv & ps\Uparrow\ \#p\ 1\ p \text{ where} \\ ps\Uparrow\ 1\ n\ p & = & p \\ ps\Uparrow\ 2m\ n\ p & = & ps\Uparrow\ m\,(2n)\,(join\,n\,p\,(\oplus *\,(distL\,n\,q)\,p)) \end{array}$$

where $q = corr\ n\ p$

III. Bulk Synchronous Parallel ML

Bulk Synchronous Parallel ML or BSML [12], [15], [16] is an explicit parallel functional language, extension of the ML family of functional languages. It follows the bulk synchronous parallel model (BSP) that considers a general purpose computer seen as a distributed memory computer, where the processor/memory pairs could communicate point-to-point and be globally synchronised, and where a program is a sequence of super-steps. A super-step has three phases: (1) computation with data held in local memories, (2) data exchange that is ensured only after a (3) global synchronisation.

There is currently no full implementation of BSML as a language (i.e. with a type system) but there exists an implementation as a library for the OCaml language, and it is implemented on top of MPI. BSML follows the flat direct style BSP programming [17] but is purely functional. It has been extended with two parallel compositions : juxtaposition [18] that is impure and offers a way to divide-and-conquer using subset of processors, and superposition that is pure but has a different style of divide-and-conquer behaviour. We omit the latter in this paper.

BSML offers a function (and three constants omitted here) to access the BSP parameters of the underlying architecture: **bsp_p** returns the number of processors in the parallel machine, it could be less than the actual number of processors if it is called within a juxtaposition.

BSML operates on a parallel data structure named *parallel vector*. Each processor contains one value in the vector. Using this data structure, BSML offers a global view of parallel programs, i.e. a program looks like a sequential program but operates on parallel data structures. It is very different from the SPMD paradigm were programs are implicit parallel compositions of sequential communication programs. The global parallel structure of SPMD programs is much harder to understand than programs that offer a global view.

There is a polymorphic type for parallel vectors: par!. Each processor thus has to contain a value that has the same type than the other values in the parallel vectors. Nesting is forbidden: 'a could not be a parallel type. The type system [15] rejects programs with such nesting, but this type checking is not provided in the library implementation. There is no direct access to individual values in parallel vectors. Manipulation is done through four functions.

The function **mkpar**: (int→ 'a) → 'a par builds a parallel vector from a function f: at processor i the vector will have value $(f\ i)$. For example with 8 processors[1]:

```
# let r = mkpar (fun i->i+1);;
val r : int par = <1, 2, 3, 4, 5, 6, 7, 8>
# let l =
    let f i = (i-1+bsp_p()) mod (bsp_p()) in
    mkpar f;;
val l : int par = <7, 0, 1, 2, 3, 4, 5, 6>
```

where # is the prompt of the toplevel, and the answer has the form *name : type = value* and in this sequential simulator parallel vectors are written $\langle\ a_0\ , \ldots,\ a_{p-1}\ \rangle$.

OCaml is a higher-order language, functions are first class citizens. It it therefore possible to define parallel vector of functions. But then, a parallel vector of functions is not a function. Therefore we need a BSML primitive to apply

[1]We show here the evaluation of the BSML expression inside the BSML toplevel or interactive loop

pointwise a parallel vector of functions to a parallel vector of values. For example:

```
# let vf = mkpar (fun i -> (+) i);;
val vf : (int->int) par = < <fun>, ..., <fun> >
# apply vf r;;
- : int par = <1, 3, 5, 7, 9, 11, 13, 15>
```

mkpar and **apply** only operate in the computation phase of a BSP super-step. Communications and implicit global synchronisations are performed using **proj** and **put**.

The function **proj**: 'a par → (int→ 'a) is the dual of **mkpar**. It creates a function back from a parallel vector. It incurs communications: it performs an optimised all-to-all communication. The optimisation comes from the fact that for an inductive type, the first constructor is considered as the empty message. For example the empty list is not communicated as it is considered to represent the empty message. It is not allowed to evaluate **proj** inside the scope of the other BSML primitives: it would be a kind of parallel nesting. The type system [15] rejects programs with such nesting, but this type checking is not provided in the current BSML implementation. **proj** could be used for example to write a reduce skeleton:

```
# let reduce op vv =
   let rec seq =
     function [x]->x | x::t->op x (seq t) in
   let f = proj vv in
   seq (List.map f processors);;
val reduce: ('a->'a->'a)->'a par->'a = <fun>

# let sum = reduce (+) r;;
val sum : int = 36
```

where processors is a list of processors, seq is the sequential reduction recursively defined by case on lists, and List.map is a map function on lists from the OCaml standard library.

For more involved communication patterns, one needs to use the **put**:(int→ 'a)par→ (int→ 'a)par function. It allows any local value to be transferred to any other processor. As **proj**, it ends the current super-step. Canonical use of **put** is **put** (**mkpar** (**fun** src dst → e)) where expression e computes (or usually, selects) the data that should be sent (depending on src) to dst. The return value of **put** is another vector of functions. At a processor j the function, when applied to i, yields the value *received from* processor i by processor j. For example, shifting the values of a parallel vector to the right could be written:

```
# let shift vv =
   let msg src v dst =
    if dst=(src+1) mod (bsp_p())
    then [v] else [] in
   let msgs = apply(mkpar msg) vv in
   parfun List.hd (apply (put msgs) l);;
shift: 'a par -> 'a par = <fun>
# shift (mkpar string_of_int);;
- : string par = <"7", "0", "1", "2",
                  "3", "4", "5", "6">
```

where: **let** parfun f v = **apply** (**mkpar**(**fun** _→ f)) v.

juxta: int→ (unit→ 'a par)→ (unit→ 'a par) → 'a par is used to divide the available processors in two parts and evaluate the given two expressions on each part. Here the expressions to evaluate are functions as OCaml is a strict language, but the parameters are just used to delay the evaluation: the only value of the type unit is (). As we will see in the following sections, **juxta** could be used to write divide-and-conquer algorithms. We just give here a very small example to illustrate its semantics:

```
# juxta (bsp_p()/2) (fun _->l) (fun _->r);;
- : int par = < 7; 0; 1; 2; 5; 6; 7; 8 >
```

On the BSP point of view, there is no subset synchronisation, but still full global synchronisation, shared by the two expressions, if they contain some.

IV. Powerlists and skeletons in BSML

A. The Powerlist Data-structure in BSML

In the remaining of the paper we assume **bsp_p**() is a power of two. The length of a powerlist is also a power of two, but it could be smaller or bigger that the number of processors. Therefore we could define the type of powerlists as:

type 'a powerlist = | S **of** 'a array | P **of** 'a array par

where 'a array is the pre-defined type of generic arrays in OCaml. 'a is a type variable meaning an array could contain values of any type as long as all the values in an array have the same type. The enumerative array value [| "Hello"; "World"|] is an array of two elements, and the type of this value is string array.

A type definition like powerlist in OCaml is similar to a union type in C or a record type with variant parts in Ada. However there is no discriminant field. The symbols S and P, called constructors, are used to discriminate between values of the type powerlist. For example S [|0;1|] is a sequential powerlist of integers, while P (**mkpar**(**fun** i→ [|i|])) is a parallel powerlist of integers, of size **bsp_p**. Both have type int powerlist.

The function map on powerlists could then be defined as:

let map f = **function**
  | S a → S (Array.map f a)
  | P a → P (parfun (Array.map f) a)

But then, it would be more convenient to have recursive definition of powerlist with a recursive definition of map:

type 'a powerlist = | S **of** 'a array | P **of** 'a powerlist par
let rec map f = **function** | S a → S (Array.map f a)
                                | P a → P (parfun (map f) a)

In the case of map, the advantage is not so big, but if we imagine the sequential case is not a function that already exists in the module Array of OCaml standard library, the benefits are much bigger in term of concision and readability.

However, with this definition of powerlist, we need to be very careful when writing functions as it is forbidden in BSML to nest parallel vectors. In this context it means that in the case of constructor P the parallel vector should contain only powerlist values built with constructor S.

Actually we can modify the definition of the powerlist to use a newly introduced feature of OCaml: generalised algebraic datatypes or GADT. They introduce two novelties with respect to sum types: the possibility to have more constrained type parameters depending on the constructor, and the possibility to introduce existential type variables (i.e. using a type variable in a constructor case that is not one of the type parameters).

There are several possibilities to define the type powerlist in order to ensure that the constructor P is only applied to parallel vectors of powerlists built with constructor S. Johann and Ghani showed that the essence of GADTs [19] is the following type and function:

```
type ('a,'b) eq = Eq: ('a,'a) eq
let cast: type a b. (a,b) eq→ a→ b = fun Eq x → x
```

Here the type eq has two type parameters: 'a and 'b. It has only one constructor: Eq. This constructor does not have any argument. In the previous definitions of powerlist we only give the type of the arguments of the constructors (after the keyword **of**) but the type of the result is implicitly 'a powerlist. With GADTs, one must also provide the type of the result of an application of the constructor: this type should be an instantiation of the polymorphic type being defined. In the case of the construction Eq, the return type is ('a,'a) eq, meaning the second parameter of the type eq is instantiated with 'a. This type actually adds a type constraint: if a value Eq exists it means that the parameters of type eq are equal. If we think in term of logic, anoter way to see the value Eq is to see it as a witness of the equality of two types (the arguments of eq). The function cast uses such a witness to perform type coercion.

In practice we found using this GADT is the easiest way to deal with our problem by defining the type powerlist as:

```
type seq and dist
type ('a,'kind) powerlist =
| S of (seq,'kind) eq ∗ 'a Array.t
| P of (dist,'kind) eq ∗ ('a,seq) powerlist par
```

The types seq and dist are just tags: we cannot build values of these types because they have no constructor. The type powerlist has now two type parameters: 'a is the type of the values the powerlist contains, and 'kind indicates how the powerlist contains the scalar values: in a sequential data structure, or in a distributed data structure. In the following we call the nature of the data structure used, the kind of the powerlist and use k or kind and variants as type variable names. In the case of the distributed data structure, the parallel vector contains only sequential powerlist values. Note also that the type for arrays in now Array.t: in OCaml standard library it is a synonym of array, but in this case we partially re-implemented the Array module so that there is sharing rather than copying when an array is cut in half then the two halves are appended later. The map function could be defined in a very similar way as before:

```
let rec map: type k.('a→ 'b)→ ('a,k) powerlist→ ('b,k) powerlist =
fun f → function
| S (eq,l) → S (eq, Array.map f l)
| P (eq,v) → P (eq, parfun (map f) v)
```

Note that while being moderately less convenient to write than the previous definition, it is much more informative: we know that the map function preserves the kind of the powerlist it works on.

However in Hindler-Milner system, types of polymorphic terms contains type variables and implicit universal quantification, but the quantifiers are restricted to appear only in the front of the type and quantify only over monomorphic types: it is rank-1 polymorphism. In practice it means that if we define a function f that takes as argument a kind preserving function g, then in the body of f, g could only be applied to the same kind of powerlist. This is too restrictive for our goal. Fortunately, OCaml relaxes the rank restriction for records and methods. We therefore define a new type and map as follows:

```
type ('a,'b) preserving =
  { body: 'k. ('a,'k)powerlist→ ('b,'k)powerlist }
let rec map : ('a→ 'b)→  ('a,'b) preserving = fun f → {
  body = function
    | S (eq,l) → S (eq, Array.map f l)
    | P (eq,v) → P (eq,parfun ((map f).body) v)
  }
```

B. Basic Functions

In addition to map and other basic functions, we defined functions used to build powerlists, in a way similar to **mkpar**. We followed the OCaml naming convention for arrays: init is a function that builds a powerlist from a length and a function of signature int→ 'a. There is a specificity: if the length is smaller that the number of processors, then we build a sequential powerlist, otherwise we build a distributed powerlist. However if we define this function recursively, we want that the recursive call builds a sequential powerlist, even if the length is greater than **bsp_p**. Therefore we need an additional argument stating the kind of powerlist we want to produce. Moreover such an argument is needed to produce the kind in the result type:

```
type _ kind = Seq: seq kind | Par: dist kind

let rec init:type k. k kind→ int→ (int→ 'a)→ ('a,k)powerlist =
  fun kind size f → assert(is_power_of_2 size);
    match kind with
    | Par when size >= bsp_p() →
        let lsize = size / (bsp_p()) in
        P(Eq,mkpar(fun i→ init Seq lsize (fun j→ f(j+i∗lsize))))
    | Seq → S(Eq,Array.init size f)
    | _ → failwith("init:␣cannot␣create␣a␣parallel␣powerlist␣"^
                   "whose␣size␣if␣smaller␣than␣bsp_p")
```

We implemented all the functions mentioned in section II, we just present here one of them, mapn:

```
let rec mapn: 'k. int → ('a,'b) preserving →
 ('a,'k)powerlist → ('b,'k)powerlist =
 fun n f pl →
  if length pl = n then f.body pl
  else match pl with
   | S(eq,_) →
      let t1,t2 = untie (castk (sym eq) pl) in
      castk eq (tie (mapn n f t1) (mapn n f t2))
   | P(eq, a) →
      if bsp_p() = 1 then P(eq,parfun (mapn n f) a)
      else let e() = par_of(mapn n f ((castk (sym eq)) pl)) in
       P(eq, juxta (bsp_p()/2) e e)
```

where tie and untie have the usual semantics of the tie operation on powerlists, but in this case only defined for *sequential* powerlists. In the case of distributed powerlists, we use BSML juxtaposition operation. However as juxtaposition only deals with expressions of type par we need to extract the parallel vector from the distributed powerlist:

```
let par_of : ('a,dist)powerlist→ ('a,seq)powerlist par =
 function | P(_, a) → a | _ → assert false
```

We also need to help OCaml type inference using explicit casts. These expressions are in fact kind of proof terms to show the equality between type expressions. This is better understood through the Curry-Howard correspondence, where a type corresponds to the statement of a property, and a program or an expression of this type corresponds to a proof of this property. In our case type expressions of the form (t1,t2)eq represent the statement of the fact that types t1 and t2 are equivalent. Let us explain one of this "proof terms" used to cast. In the sequential case (branch S(eq,_) of the pattern matching), the term castk (sym eq) relies on the following signatures and definitions:

```
let sym : type a b. (a,b) eq → (b,a) eq = (* ... *)
let subst : type a b k k'. (a,b) eq → (k,k') eq →
  ((a,k)powerlist, (b,k')powerlist) eq = (* ... *)
let castk : type k k'.(k,k')eq→ ('a,k)powerlist→ ('b,k')powerlist=
  fun eq → cast (subst Eq eq)
```

It means that as eq is a proof that the type seq is equal to the type variable 'k, then 'k is equal to seq (by symmetry) and then ('a,'k)powerlist is equal to ('a,seq)powerlist (by substitution). This last equality is used to cast pl that has type ('a,'k)powerlist into the same value but of type ('a,seq)powerlist that is the required type input for function untie.

C. Divide-and-Conquer Patterns

As described in section II, the divide and conquer functions over powerlists are transformed into one of the two output pattern: a top-down recursion denoted $F\Downarrow$ or a bottom-up computation $F\Uparrow$. Once a function is expressed in such a way, it can easily be computed in parallel by calling the corresponding parallel skeleton. We describe here the implementations of the two parallel skeletons. For the sake of conciseness, we present explicitly only the bottom-up skeleton; the top-down computation is very similar. The main difference is that in top-down recursion, adjust functions are applied before splitting the list, the recursion ends by computation on singletons, whereas the bottom-up skeleton first computes on singletons then computes adjustment of merged sub-lists until the full size is reached (remember that these skeletons compute a list of the same the size as the input list).

For each skeleton, we implemented first a naive implementation very close to the definition given by Achatz and Shulte. In figure 1 we present the bottom up pattern where sigma is a function used to update a counter initially set to s, l and r are the "adjusting functions" and p is a powerlist.

These definitions use join in a very inefficient way: at each position of the list, they computes two values, then selects the value of interest using join. These inefficient implementations were used as specifications against which we tested efficient

```
let bottom_up_spec : 'k. ('a → 'a) →
  ('a → ('b, 'b, 'b) preserving2) → ('a →
('b, 'b, 'b) preserving2) →
  'a → ('b,'k) powerlist → ('b,'k) powerlist =
  fun sigma l r s p →
   let rec f_up : type k. int→ int→ 'a→
     ('b,k) powerlist → ('b,k) powerlist =
     fun n len s p →
       if n=1 then p
       else
         let n' = n/2 in
         let q = corr len p in
         let joined = join len (mapn2 len (l s) p q)
                                (mapn2 len (r s) q p) in
         f_up n' (2*len) (sigma s) joined in
   f_up (length p) 1 s p
```

Fig. 1. Naive Bottom-Up Pattern

versions of the skeletons named bottom_up and top_down where we only compute the needed value. The implementations are not shown here for the sake of conciseness but can be found online[2].

For distributed lists, we replaced join by a juxtaposition where the result of the function l (resp.r) is computed only on the first (resp. second) half of the processors. During the recursion, functions are applied to sequential lists, in this case the list is split and once again only the relevant values are computed on the sub-parts.

The butterfly communication skeleton corr can be costly to use: if used with a parameter n greater than the number n of elements per processor, it performs a communication of n values per processors and a synchronisation barrier. As some derivations follow the output patterns but do not use the communicated values, we added an optional argument to the skeletons, use_corr, which allows to specify that corr has not to be computed. It is set by default to true, so that the normal behaviour of the implementation is to perform the communications; it is the programmer responsibility to call the skeleton with ~use_corr:**false** as argument to avoid unnecessary communications.

V. Applications and Experiments

A. Bitonic sort.

A bitonic sequence of values is the concatenation of two monotonic (i.e. increasing or decreasing) sequences. The bitonic merge is an operation that produces a sorted list by merging two bitonic lists. In this context the bm function takes only one list as parameter and processes it by sorting the two sub-parts of this list. It is implemented using the top_down skeleton as follows:

```
let bm pl1 =
  top_down id id' (fun s → map2 min) (fun s → map2 max) 0 pl1
```

The bitonic sort can then be implemented by a bottom-up recursion performing successive bitonic merges and reverse bitonic merges:

[2]The full code is available at http://traclifo.univ-orleans.fr/PaPDAS/

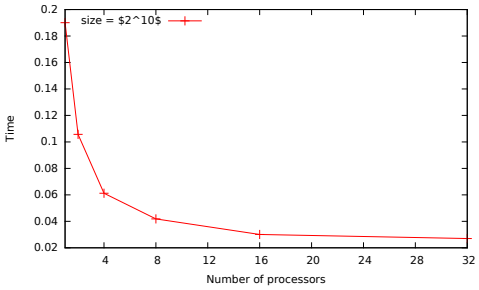

Fig. 2. Prefix Sum on a Shared-Memory Computer – Execution Time

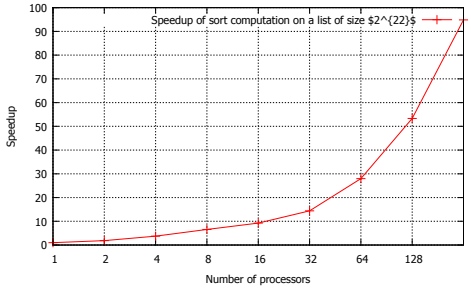

Fig. 3. Bitonic Sort on a Distributed-Memory Machine – Speedup

```
let bs pl1 = bm ( bottom_up ~use_corr:false id
                  (fun s → {body=fun p q→ bm p})
                  (fun s → {body=fun p q→ bm_rev q})
                  0 pl1 )
```

This definition does not use the communications, thus the optional parameter use_corr is set to false.

B. Prefix Sum.

Following the prefix sum problem derivation shown in section II, the definition leads to this instantiation of the bottom_up skeletons:

```
let psum op pl = bottom_up id (fun s → {body = fun p q → p})
    (fun s → {body = fun p q →
                  let lst = last p in (map (op lst)).body q})
    0 pl
```

The counter is not used, so we took the identity function for sigma and 0 as starting value. The left adjustment function returns the left sub-list unmodified; the right adjustment function adds the last element of the left sub-list to each element.

C. Experiments

We measured the execution times of some of the applications we developed on two parallel machines: SPEED on a shared-memory computer containing 4 AMD Opteron 6174 processors with 12 computing cores each, for a total of 48 cores. As it is required that the number of processing elements are a power of two, we used only up to 32 cores, and ARTEMIS a cluster of 32 nodes of Intel Xeon E5-2630 processors, with Ethernet and Intel Truescale networks. We used up to 256 cores.

For the prefix sum application, we generated powerlists of random 20×20 matrices of 64 bits floating point numbers, and we used a naive $\mathcal{O}(n^3)$ multiplication as associative operator. Figure 2 shows the results for the computation of prefix sum on a powerlist of size 2^{18} with matrix elements on the SPEED machine.

The experiments for sort computation were done on powerlists of size 2^{22} on the ARTEMIS machine. The timings are the average value of a series of measures. The speedup obtained are shown in Figure 3.

These experiments shows good performances in the case of prefix sum and good scalability for bitonic sort. However the speed of the bitonic sort could be improved: for the moment the sharing of powerlists for very fast sequential tie/untie is not done all the time, as we preserve a functional style. Therefore some copies of array are performed and decrease the overall performance. Nevertheless we plan to improve that by providing also patterns with imperative style in some parts of the computation.

VI. RELATED AND FUTURE WORK

BSP model [2] was developed around the following idea: structured parallel programs ought to be conceived as two separate and complementary entities – computation, which expresses the calculations in a procedural manner, and – coordination, which abstracts the interaction and communication. Many other models imported this idea directly or indirectly.

Algorithmic Skeletons [1] abstract commonly used patterns of parallel computation, communication, and interaction, and provides high abstraction, portability across different architectures, and high performance. In the functional programming setting, this approach proved to be a very successful one, since functional programming concepts allow simple representation of the skeletons [20], [21], [22].

Homomorphisms which represent important skeletons (in particular on join lists [4], [23]) are special kind of functions that are very efficient for simple representation of parallel programs that follow the divide and conquer structure. Powerlists data structures are in a way similar to join lists, and as we have presented, they can be successfully used in defining simple, provably correct, functional parallel programs, which are divide and conquer in nature.

The possibility of using powerlists to prove the correctness of several algorithms has encouraged some researchers to pursue automated proofs of theorems about powerlists. Kapur and Subramaniam [24] have implemented the powerlist notation for the purpose of automatic theorem proving. They have proved many of the algorithms described by Misra using an inductive theorem prover, called Rewrite Rule Laboratory. In [25] adder circuits specified using powerlists are proved correct with respect to addition on the natural numbers. The attempt done in [26] shown how ACL2 can be used to verify theorems about powerlists. Still, the considered powerlists are not the regular structures defined by Misra, but structures corresponding to binary trees, which are not necessarily balanced.

We have presented in [27] a formalisation of powerlists in the Coq [9] proof assistant. Our methodology was to obtain a small axiomatisation of this data structure, as close as possible

to the pen-and-paper version, and then to build on it. As BSML is also formalised in Coq [13], it is possible to verify the correctness of pure functional parallel versions of the powerlist functions presented in this paper. We intend to join the results presented in [27] with the work presented here, in order to be used in a complete, more general framework; this will allow the development of correct and verifiable parallel programs with predictable performances using theories and tools that facilitate the development of efficient applications by implementing simple programs satisfying conditions easily, or ideally automatically, proved. The framework will use the axiomatisation of lower level parallel programming primitives and their use to implement the high-level primitives in order to extract [28] actual *parallel* code from the developments made within proof assistants.

VII. Conclusion

In this paper we have presented how parallel programs defined on powerlists could be transformed to real code in the functional language OCaml plus calls to the parallel functional programming library Bulk Synchronous Parallel.

In order to transform the abstract specifications of the powerlist programs to BSML concrete implementations, we have used the methodology presented in [8] that transform the divide&conquer functions into tail-recursive computations. Then we have adapted these single-control flow computations to BSML by giving an efficient definition of powerlists in OCaml based on GADTs, and by giving efficient predefined implementations for top-down and bottom-up patterns of computations defined in [8]. These implementations have been improved by replacing the basic function $join$ with juxtaposition, and also by allowing the user to bypass the costly butterfly communication when it is not necessary.

Examples for prefix sum, bitonic sort have been presented and the experiments done for them show that the framework is practical and allows simple development of efficient parallel programs.

The existence of Coq formalisation for BSML [13] and powerlists [27] will allow us to include the implementations methods discussed in this paper into the more formal and general practical framework.

Acknowledgements

This work is partly supported by ANR (France) and JST (Japan) (project PaPDAS ANR-2010-INTB-0205-02 and JST 10102704).

References

[1] M. Cole, *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989, available at http://homepages.inf.ed.ac.uk/mic/Pubs.

[2] L. G. Valiant, "A bridging model for parallel computation," *Commun. ACM*, vol. 33, no. 8, p. 103, 1990.

[3] R. Bisseling, *Parallel Scientific Computation. A structured approach using BSP and MPI*. Oxford University Press, 2004.

[4] M. Cole, "Parallel Programming with List Homomorphisms," *Parallel Processing Letters*, vol. 5, no. 2, pp. 191–203, 1995.

[5] J. Misra, "Powerlist: A structure for parallel recursion," *ACM Trans. Program. Lang. Syst.*, vol. 16, no. 6, pp. 1737–1767, November 1994.

[6] J. Kornerup, "Data structures for parallel recursion," Ph.D. dissertation, University of Texas, 1997.

[7] V. Niculescu, "PARES – A Model for Parallel Recursive Programs," *Romanian Journal of Information Science and Technology (ROMJIST)*, vol. 14, no. 2, pp. 159–182, 2011.

[8] K. Achatz and W. Schulte, "Architecture independent massive parallelization of divide-and-conquer algorithms," Fakultaet fuer Informatik, Universitaet Ulm, 1995.

[9] The Coq Development Team, "The Coq Proof Assistant," http://coq.inria.fr.

[10] X. Leroy, D. Doligez, A. Frisch, J. Garrigue, D. Rémy, and J. Vouillon, "The OCaml System release 4.00.0," http://caml.inria.fr, 2012.

[11] G. Cousineau and M. Mauny, *The Functional Approach to Programming*. Cambridge University Press, 1998.

[12] F. Loulergue, F. Gava, and D. Billiet, "Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction," in *International Conference on Computational Science (ICCS)*, ser. LNCS, vol. 3515. Springer, 2005, pp. 1046–1054.

[13] J. Tesson and F. Loulergue, "A Verified Bulk Synchronous Parallel ML Heat Diffusion Simulation," in *International Conference on Computational Science (ICCS)*, ser. Procedia Computer Science. Elsevier, 2011, pp. 36–45.

[14] Y. Minsky, "OCaml for the masses," *Commun. ACM*, vol. 54, no. 11, pp. 53–58, 2011.

[15] F. Gava and F. Loulergue, "A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting," *Future Generation Computer Systems*, vol. 21, no. 5, pp. 665–671, 2005.

[16] L. Gesbert, F. Gava, F. Loulergue, and F. Dabrowski, "Bulk Synchronous Parallel ML with Exceptions," *Future Generation Computer Systems*, vol. 26, pp. 486–490, 2010.

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

[18] F. Loulergue, "Parallel Juxtaposition for Bulk Synchronous Parallel ML," in *Euro-Par 2003*, ser. LNCS, H. Kosch, L. Boszormányi, and H. Hellwagner, Eds., no. 2790. Springer Verlag, 2003, pp. 781–788.

[19] P. Johann and N. Ghani, "Foundations for structured programming with GADTs," in *POPL*. ACM, 2008, pp. 297–308.

[20] R. Loogen, Y. Ortega-Mallen, and R. Pena-Mari, "Parallel Functional Programming in Eden," *Journal of Functional Programming*, vol. 3, no. 15, pp. 431–475, 2005.

[21] R. D. Cosmo, Z. Li, S. Pelagatti, and P. Weis, "Skeletal Parallel Programming with OcamlP3l 2.0," *Parallel Processing Letters*, vol. 18, no. 1, pp. 149–164, 2008.

[22] N. Scaife, S. Horiguchi, G. Michaelson, and P. Bristow, "A parallel SML compiler based on algorithmic skeletons," *Journal of Functional Programming*, vol. 15, no. 4, pp. 615–650, 2005.

[23] Z. Hu, H. Iwasaki, and M. Takechi, "Formal derivation of efficient parallel programs by construction of list homomorphisms," *ACM Trans. Program. Lang. Syst.*, vol. 19, no. 3, pp. 444–461, 1997.

[24] D. Kapur and M. Subramaniam, "Automated reasoning about parallel algorithms using powerlists," State University of New York at Alban, Tech. Rep. TR-95-14, 1995.

[25] ——, "Mechanical verification of adder circuits using rewrite rule laboratory," *Formal Methods in System Design*, vol. 13, pp. 127–158, 1998.

[26] R. A. Gamboa, "A formalization of powerlist algebra in ACL2," *J. Autom. Reason.*, vol. 43, no. 2, pp. 139–172, 2009.

[27] F. Loulergue, V. Niculescu, and S. Robillard, "Powerlists in Coq: Programming and Reasoning," in *First International Symposium on Computing and Networking (CANDAR)*. IEEE Computer Society, 2013, pp. 57–65.

[28] P. Letouzey, "Coq Extraction, an Overview," in *Logic and Theory of Algorithms, Fourth Conference on Computability in Europe, CiE 2008*, ser. LNCS 5028, A. Beckmann, C. Dimitracopoulos, and B. Löwe, Eds. Springer, 2008.