Bulk Synchronous Parallel ML 0.5
Reference Manual

Frédéric Loulergue
with[1]
Wadoud Bousdira, Frédéric Gava, Louis Gesbert,
Gaétan Hains, Guillaume Petiot, Julien Tesson
`http://traclifo.univ-orleans.fr/BSML`

October 29, 2010

[1] and some code from Xavier Leroy's OcamlMPI

Contents

Chapter 1

Introduction

Some problems require performance that only massively parallel computers offer whose programming is still difficult. Works on functional programming and parallelism can be divided in two categories: explicit parallel extensions of functional languages — where languages are either non-deterministic [32] or non-functional [2, 12] — and parallel implementations with functional semantics [1] — where resulting languages do not express parallel algorithms directly and do not allow the prediction of execution times. Algorithmic skeleton languages [9, 33], in which only a finite set of operations (the skeletons) are parallel, constitute an intermediate approach. Their functional semantics is explicit but their parallel operational semantics is implicit. The set of algorithmic skeletons has to be as complete as possible but it is often dependent on the domain of application.

The design of parallel programming languages is therefore a trade-off between:

- the possibility of expressing parallel features necessary for predictable efficiency, but which makes programs more difficult to write, to prove and to port

- the abstraction of such features that are necessary to make parallel programming easier, but which must not hinder efficiency and performance prediction.

We are exploring the intermediate position of the paradigm of algorithmic skeletons in order to obtain universal parallel languages where execution cost can be easily determined from the source code (in this context, cost means the estimate of parallel execution time). This last requirement forces the use of explicit processes corresponding to the parallel machine's processors. *Bulk Synchronous Parallel* (BSP) computing [30] is a parallel programming model which uses explicit processes, offers a high degree of abstraction and yet allows portable and predictable performance on a wide variety of architectures.

A denotational approach led us to study the expressiveness of functional parallel languages with explicit processes [18] but this is not easily applicable to BSP algorithms. An operational approach has led to a BSP λ-calculus that is confluent and universal for BSP algorithms [28], and to a library of bulk synchronous primitives for the Objective Caml [23] language which is sufficiently expressive and allows the prediction of execution times [17].

This framework is a good trade-off for parallel programming because:

- we defined a *confluent calculus* so

  - we can design purely functional parallel languages from it. Without side-effects, programs are easier to prove, and to re-use (the semantics is compositional)
  - we can choose any evaluation strategy for the language. An eager language allows good performances.

- this calculus is based on BSP operations, so programs are easy to port, their costs can be predicted and are also portable because they are parameterized by the BSP parameters of the target architecture.

The version 0.1 of our BSML library implements the BSλ-calculus primitives using Objective Caml [23] and BSPlib [19] and its performance follows curves predicted by the BSP cost model [3]. This environment is a safe one. Our language is deterministic, is based on a parallel abstract machine [31] which has been proved correct w.r.t. the confluent BSλ_p-calculus [24] using an intermediate semantics [25]. A polymorphic type system [14] has been designed, for which type inference is possible. The small number of basic operations allows BSML to be taught to BSc. students.

The BSPlib library is no longer supported nor updated. Moreover BSML is used as the basis for the Caraml project which aims to use Objective Caml for Grid computing with, for example, applications to parallel databases and molecular simulation. In such a context, the parallel machine is no longer a homogeneous machine as prescribed by the BSP model and global synchronization barriers are too costly. Thus we will need encapsulated communications between differents architectures and subset synchronization [35]. Version 0.2 of the BSML library is hence based on MPI [37]. It also has a smaller number of primitives which are closer to the BSλ-calculus than the primitives of version 0.1. In version 0.1, communication primitives manipulate parallel vectors of lists and parallel vectors of hash tables and are less easy to be taught.

Version 0.4 [27] adds the following features:

- The primitive **at** is replaced by the more general **proj** primitive.
- The type of the **put** primitive has been generalized. This new version of **put** is backward compatible.
- The whole BSML Library has been modularized. A new low-communication module has been added: A module for TCP/IP communications.
- Pretty-printing of BSML parallel vectors is now possible in the toplevel.

Version 0.5 [7] adds :

- a revised syntax (1.3.3) which improves the ease of writing and reading of BSML programs,
- new modules in the standard library.

The section 1.1 presents the BSP model, section 1.2 explains why processes should be explicit in parallel programming languages and compares our approach with the SPMD paradigm. Section 1.3 gives an overview of the core BSML library.

1.1 The BSP Model

The Bulk Synchronous Parallel (BSP) model [40, 29, 36] describes: an abstract parallel computer, a model of execution and a cost model. A BSP computer has three components: a homogeneous set of processor-memory pairs, a communication network allowing inter processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. A wide range of actual architectures can be seen as BSP computers.

The performance of the BSP computer is characterized by three parameters (expressed as multiples the local processing speed): the number of processor-memory pairs **p** ; the time **l** required for a global synchronization ; the time **g** for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an h-relation (communication phase where every processor receives/sends at most h words) in time $g \times h$. Those parameters can easily be obtained using benchmarks [19].

A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjointed phases (Fig. 1.1):

1. each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes;
2. the network delivers the requested data transfers;
3. a global synchronization barrier occurs, making the transferred data available for the next super-step.

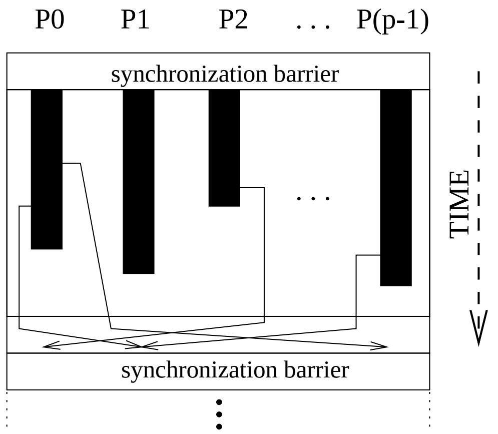

Figure 1.1: A BSP super-step

The execution time of a super-step s is, thus, the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

$$\mathrm{Time}(s) = \max_{i:processor} w_i^{(s)} + \max_{i:processor} h_i^{(s)} \times g + l$$

where $\begin{cases} w_i^{(s)} = \text{ local processing time on processor } i \text{ during super-step } s, \\ h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\} \end{cases}$
where $h_{i+}^{(s)}$ (resp. $h_{i-}^{(s)}$) is the number of words transmitted (resp. received) by processor i during super-step s.

The execution time $\sum_s \mathrm{Time}(s)$ of a BSP program composed of S super-steps is, therefore, a sum of 3 terms:

$$W + H \times g + S \times l \text{ where } \begin{cases} W & = & \sum_s \max_i w_i^{(s)} \\ H & = & \sum_s \max_i h_i^{(s)}. \end{cases}$$

In general, W, H and S are functions of p and of the size of data n, or of more complex parameters like data skew. To minimize execution time, the BSP algorithm design must jointly minimize the number S of super-steps, the total volume h with imbalance of communication and the total volume W with imbalance of local computation.

Bulk Synchronous Parallelism (and the Coarse-Grained Multicomputer, CGM, which can be seen as a special case of the BSP model) is used for a large variety of applications: scientific computing [6, 21], genetic algorithms [8] and genetic programming [10], neural networks [34], parallel databases [4], constraint solvers [15], *etc.* It is to notice that "A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures, between the late eighties and the time from the mid nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain" [11].

1.2 Explicit processes and the SPMD programming style

Among researchers interested in declarative parallel programming, there is a growing interest in execution cost models taking into account global hardware parameters like the number of processors and bandwidth. With similar motivations we are designing an extension of ML called BSML for which the BSP cost model facilitates performance prediction. Its main advantage in this respect is the use of *explicit processes*: the map from processors to data is programmed explicitly and does not have to be recovered by inverting the semantics of layout directives.

In BMSL, a parallel value is built from an ML function from processor numbers to local data. A computation superstep results from the pointwise application of a parallel functional value to a parallel value. A communication and synchronization superstep is the application of a communication template (a parallel value of processor numbers) to a parallel value. A crucial restriction on the language's constructors is that parallel values are not nested. Such nesting would imply either dynamic process creation or some non-constant dynamic costs for mapping parallel values to the network of processors, both of which would contradict our goal of direct-mode BSP programming.

The popular style of SPMD programming in a sequential language augmented with a communication library has some advantages due to its explicit processes and explicit messages. In it, the programmer can write BSP algorithms and control the parameters that define execution time in the cost model. However, programs written in this style are far from being pure-functional: they are imperative and even non-deterministic. There is also an irregular use of the `pid` (Processor ID i.e. processor number) variable which is bound *outside* the source program. Consider for example p static processes (we refer to processes as *processors* without distinction) given an SPMD program E to execute. The meaning of E is then

$$[\![E]\!]_{SPMD} = [\![E@0 \mid\mid \ldots \mid\mid E@(p-1)]\!]_{CSP}$$

where $E@i = E[\mathtt{pid} \leftarrow i]$ and $[\![E]\!]_{CSP}$ refers to concurrent semantics defined by the communication library, for example the meaning of a CSP process [20]. This scheme has two major disadvantages. First, it uses concurrent semantics to express parallel algorithms, whose purpose is to execute predictably fast and are deterministic. Secondly, the `pid` variable is used without explicit binding. As a result there is no syntactic support for escaping from a particular processor's context to make global decisions about the algorithm. The global parts of the SPMD program are those which *do not* depend on any conditional using the pid variable. This dynamic property is thus given the role of defining the most elementary aspect of a parallel algorithm, namely its local vs global parts.

We propose to eliminate both of these problems by using a minimal set of algorithmic operations having a BSP interpretation. Our parallel control structure is analogous to the PAR of Occam [22] but without possibility of nesting. The `pid` variable is replaced by a normal argument to a function within a parallel constructor. The property of being a local expression is then visible in the syntax and types. The current implementation of BSML is the BSML library, which is described below.

1.3 Overview of the core BSML library

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation: a library for Objective Caml. The so-called BSML library is based on the following elements.

It gives access to the BSP parameters of the underling architecture. In particular, it offers the constant **bsp_p**: int such that the value of **bsp_p** is p, the static number of processes of the parallel machine. The value of this variable does not change during execution (for "flat" programming, this is not true if a parallel juxtaposition is added to the language [26]).

There is also an abstract polymorphic type α **par** which represents the type of p-wide parallel vectors of objects of type α, one per process. The nesting of **par** types is prohibited. Our type system enforces this restriction [13]. This improves on the earlier design DPML/Caml Flight [16, 12] in which the global parallel control structure sync had to be prevented *dynamically* from nesting.

This is very different from SPMD programming (Single Program Multiple Data) where the programmer must use a sequential language and a communication library (like MPI [37]). A parallel program is then the multiple copies of a sequential program, which exchange messages using the communication library. In this case, messages and processes are explicit, but programs may be non deterministic or may contain deadlocks.

Another drawback of SPMD programming is the use of a variable containing the processor name (usually called "pid" for Process Identifier) which is bound outside the source program. A SPMD program is written using this variable. When it is executed, if the parallel machine contains p processors, p copies of the program are executed on each processor with the pid variable bound to the number of the processor on which it is run. Thus parts of the program that are specific to each processor are those which depend on the pid variable. On the contrary, parts of the program which make global decision about the algorithms are those which do not depend on the pid variable. This dynamic and undecidable property is given the role of defining the most elementary aspect of a parallel program, namely, its local vs global parts.

The parallel constructs of BSML operate on parallel vectors. Those parallel vectors are created by:

$$\textbf{mkpar}: (\text{int} \to \alpha) \to \alpha\ \textbf{par}$$

so that (**mkpar** f) stores (f i) on process i for i between 0 and $(p-1)$. We usually write f as **fun** pid→ e to show that the expression e may be different on each processor. This expression e is said to be *local*. The expression (`mkpar f`) is a parallel object and it is said to be *global*.

The dual operation of **mkpar** is:

$$\textbf{proj}: \alpha\ \textbf{par} \to (\text{int} \to \alpha)$$

This primitive requires a full super-step to be evaluated. **It should not be evaluated in the context of a mkpar.**

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a super-step) and phases of global communication (second phase of a super-step) with global synchronization (third phase of a super-step). Asynchronous phases are programmed with **mkpar** and with:

$$\textbf{apply}: (\alpha \to \beta)\ \textbf{par} \to \alpha\ \textbf{par} \to \beta\ \textbf{par}$$

apply (**mkpar** f) (**mkpar** e) stores (f i) (e i) on process i. Neither the implementation of BSML, nor its semantics [25] prescribe a synchronization barrier between two successive uses of **apply**.

Readers familiar with BSPlib [36, 19, 5] will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by:

$$\textbf{put}:(\text{int} \to \alpha)\ \textbf{par} \to (\text{int} \to \alpha)\ \textbf{par}$$

Consider the expression: **put**(**mkpar**(**fun** i→ fs_i)) (*)

To send no value from process j to process i, (fs_j i) must evaluate to a value v such as Tools.is_empty v is **true**. Such values include the empty list, the None value of type α option, the value of type `unit` and any first constant constructor in a sum type. To send a value v from process j to process i, the function fs_j at process j must be such as (fs_j i) evaluates to v and v is such as Tools.is_empty v is **false**.

Expression (*) evaluates to a parallel vector containing a function fd_i of delivered messages on every process. At process i, (fd_i j) evaluates to the empty value of the type if it exists and if process j sent no message to process i or evaluates to v if process j sent the value v to the process i.

1.3.1 Examples

For example, one can define get_one such that :

$$\text{get\_one}\ \langle\, x_0, \ldots, x_{p-1} \,\rangle \langle\, i_0, \ldots, i_{p-1} \,\rangle = \langle\, x_{i_0}, \ldots, x_{i_{p-1}} \,\rangle$$

```
(* val replicate: α → α par *)
let replicate x =
  mkpar (fun pid → x)

(* val apply2: ( α → β → γ ) par → α par → β par → γ par *)
let apply2 f x y =
  apply (apply f x) y

(* val get_one: α par → int par → ' a par *)
let get_one datas srcs =
  let pids = parfun (fun i→ natmod i (bsp_p())) srcs in
  let ask = put(parfun (fun i dst→ if dst=i then Some() else None) pids)
  and replace_by_data =
    parfun2 (fun f d dst→ match(f dst)with Some() → Some d|_→ None) in
  let reply = put(replace_by_data ask datas) in
  parfun (fun(Some x)→ x) (apply reply pids)
```

replicate, apply2 and get_one are part of the module Stdlib.Base and Stdlib.Comm.

1.3.2 Remark on nesting

As explained in the introduction, parallel vectors must not be nested. The programmer is responsible for this absence of nesting. A program containing e.g. a type int **par par** will have a unpredictable behaviour. This kind of nesting is easy to detect. But nesting can be more difficult to detect, e.g :

```
let vec1 = mkpar (fun pid → pid)
and vec2 =
      get_one
        (replicate 1)
        (mkpar (fun pid→ if pid=0
                          then last()
                          else pid−1)) in
let couple1 = (vec1,1)
and couple2 = (vec2,1) in
mkpar (fun pid → if pid<(bsp_p())/2
                  then snd (couple1)
                  else snd (couple2))
```

Objective Caml being a strict language, the evaluation of the last expression would imply the evaluation of vec1 on the first half of the network and vec2 on the second half of the network. But a get implies a synchronization barrier and a **mkpar** implies no synchronization barrier. So this will lead to mismatched barriers and the bevahiour of the program will be unpredictable.

In order to avoid such problems, it is sufficient that every subexpression of a sequential expression (i.e. with no **par** type) is also sequential. The only exception is **at** whose type is bool **par** → int → bool. But **at** must only be used in a **if then else** expression and the two branches of the conditional must be non-sequential expressions.

We have now a polymorphic type system which ensures the absence of such nesting [13]. There is no implementation which supports the whole Objective Caml language.

1.3.3 A new syntax for BSML

Having a very small core of parallel operations is a great strength for the formalization of the language. It makes the definitions clear and the proofs shorter. Being able to embed these primitives in higher-order functions is precious and allows complex parallel operations in little code. However, the program, even if high-level, still has to deal with replicated values and parallel vectors, and the use of the primitives can sometimes become awkward. Indeed, every operation inside of parallel vectors has to call a primitive and define an "ad hoc" function. This gets worse when working with multiple vectors, with nested calls to **apply**. Simply transforming a pair of vectors into a vector of pairs is written

```
let combine_vectors(v, w) = apply(parfun(fun v w → v, w) v) w
```

This could be made simpler with the definition of

```
let parfun2 f x y = apply (parfun f x) y
```

We get then:

```
let combine_vectors(v, w) = parfun2(fun v w → v, w) v w
```

which is easier to read, but still unsatisfactory because we have to define, each time, a specific function. This implies creating named parameters although our function will only be applied to our vectors, and can be confusing:

```
let combine_vectors(v, w) = parfun2 (fun w v → w, v) v w
```

which is exactly the same as above but can lead the programmer to errors.

Instead of a point of view based on primitives, we can consider the execution levels such that one can declare code that will be executed globally as in standard Ocaml and code that will be executed locally, from a parallel vector. Then, to access to local data in a local section, we need no more to define additional functions because opening vectors now can be done locally. A local section is represented by ≪≫. This is a new syntax for parallel vectors: ≪ x≫ . Replicated information is available inside

Primitive	Type	Description
$\ll$ e $\gg$	t **par** if e : t	$\langle e, \ldots, e\rangle$
\$this\$ (within a local section)	int	i on processor i
\$v\$ (within a local section)	t (if v : t **par**	v_i on processor i (if $v = \langle v_0, \ldots, v_{p-1}\rangle$)

Figure 1.2: Summary of Revised BSML Syntax

the vector, as with the **mkpar** above. To access local information, we add the syntax \$ x \$ to open the vector x and get the local value it contains ; \$ \$ can obviously be used only within local sections. It is now possible to write combine_vectors as follows:

```
let combine_vectors (v, w) = ≪ $v$, $w$ ≫
```

which is shorter, clearer and thus less error-prone. Additionally, the local pid can be accessed with \$this\$, to replace calls to **mkpar**. Synchronous primitives (**proj** and **put**) do not need a special syntax, but their use is already made more simple.

Now what about the expressiveness of BSML primitives? The answer is, our new syntax is as expressive as the initial one. Indeed, ≪≫ combined with \$this\$ is strictly equivalent to **mkpar** with the definition:

```
let mkpar f = ≪ f $this$ ≫
```

and the use of any other vector in \$\$ is equivalent to a call to **apply**. For example,

```
let p = put(apply (mkpar (fun sendfrom x sendto → e (sendfrom, sendto, x))) x)
```

which computes values to be communicated depending on source, destination and a vector x can now be written:

```
let p = put ≪ fun sendto → e($this$, sendto, $x$)≫
```

Figure 1.2 gives a summary of the revised BSML syntax.

Chapter 2

The BSML Scripts

2.1 Compilation (bsmlc and bsmlopt)

`bsmlc.mpi`, `bsmlc.seq`, and `bsml.tcp` are scripts that call the Objective Caml batch compiler `ocamlc`, which compiles Caml source files to bytecode object files and links these object files to produce standalone bytecode executable files with arguments to use the BSML library :

- `bsmlc.mpi` produces a MPI parallel version of the program.
- `bsmlc.tcp` produces a parallel version of the program, communications in the program are performed as TCP/IP communications.
- `bsmlc.seq` produces a sequential version of the program. It may be useful if you want to test a program on a sequential machine.

`bsmlc` is a more general script. For exameple `bsmlc -seq` is equivalent to `bsmlc.seq`. Other flags are `-mpi` and `-tcp`.

Parallel versions of the programs should be run with: `bsmlrun.mpi`, `bsmlrun.tcp`, or the general script `bsmlrun`.

The `bsmlrun.mpi` script has the same options as the `mpirun` program you are using. The `bsmlrun.tcp` script looks for a `.bsmlnodes` file in your home directory. This file should contain the list of the names (or IP addresses) of the machines in your cluster. Another file could be given using the `-nodes` option.

When you run a program which uses the BSML library, the machine's BSP parameters are read from the file `$HOME/.bsmlrc`. Entries in this file are of the form

```
number_of_procs,g_parameter,l_parameter,r_parameter
```

`g_parameter`, `l_parameter` and `r_parameter` must be written as caml `float` ie, `1` is written `1.` or `1.0`. The sequential version of the library reads the first line of the file, the parallel version reads the line which corresponds to the number of processor available on your machine.

See also the `ocamlc` command and the `mprun` command.

`bsmlopt.mpi`, `bsmlopt.tcp` and `bsmlopt.seq` produce respectively parallel and sequential native code if the `ocamlopt` compiler is present on your machine. There also exists a more general `bsmlopt` script.

See also the `ocamlopt` command.

2.2 The toplevel system (bsml)

`bsml` permits interactive use of the Objective Caml system with the BSML library through a read-eval-print loop. In this mode, the system repeatedly reads Caml phrases from the input, then typechecks, compiles and evaluates them, then prints the inferred type and result value, if any. The system prints a # (sharp) prompt before reading each phrase. The evaluation is done sequentially. If you use the pure functional subset of Ocaml the result will be exactly the same as in the parallel case (Even if you use imperative features the result may be the same).

See also the `ocaml` command.

Chapter 3

The BSML Library

This chapter describes the functions provided by the BSML library modules. These modules are automatically linked with the user's object code files by the `bsmlc` and `bsmlopt` scripts.

Most of the described modules are functors. To ease the use of the BSML Library, we provide the following modules :

- Bsml whose signature is Bsmlsig.BSML ;
- Stdlib which is defined as :

  module Base = Bsmlbase.Make(Bsml)
  module Comm = Bsmlcomm.Make(Bsml)
  module Sort = Bsmlsort.Make(Bsml)
  module Back = Bsmlbckcomp.Make(Bsml)
  module Array = Bsmlskeleton.MakeArray(Bsml)
  module List = Bsmlskeleton.MakeList(Bsml)

Thus, for example to use the **mkpar** primitive of BSML, you should write Bsml.**mkpar** or you need to open the module Bsml before calling the function **mkpar**. To use the function replicate, one can write Stdlib.Base.replicate or one can first open the module Stdlib and its submodule Base.

There are in fact three different implementations of Bsml, one based on MPI, one on TCP/IP, and one sequential implementation). There are also three different implementations since there are three different Bsml modules. The different scripts (`bsmlc` or `bsmlopt` with suffix `.mpi`, `.tcp` or `.seq`) handle these different versions and you do not need to worry about them when you write BSML programs.

3.1 Module `Bsmlsig` : Interface of all the main components of BSML.

Author(s): Louis Gesbert, Frédéric Gava, Frédéric Loulergue

3.1.1 BSML Primitives

Interface of the modules implementing the BSML primitives

```
module type BSML =
  sig
```

Types

```
type 'a par
```

Abstract type for parallel vector of size p.

In the following we will note $<v_0, \ldots, v_{p-1}>$ the parallel vector with value v_i at processor i

Machine parameters accessors

```
val argv : string array
```

Returns the arguments from command line with implementation-specific arguments removed.

```
val bsp_p : int
```

Number p of processes in the parallel machine.

```
val within_bounds : int -> bool
```

`within_bounds n` is `true` is `n` is between 0 and *p-1*, `false` otherwise.

```
val bsp_g : float
```

BSP parameter g of the parallel machine.

```
val bsp_l : float
```

BSP parameter l of the parallel machine.

```
val bsp_r : float
```

BSP parameter r of the parallel machine.

Exceptions

```
exception Invalid_processor of int
```

Raised when asked for a processor id that is not between `0` and `bsp_p - 1`. In particular, this exception can be raised by the functions that `proj` and `put` return.

```
exception Timer_failure of string
```

Parallel operators

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

Parallel vector creation. **Parameters :**

- `f` function to evaluate in parallel

Returns the parallel vector with `f` applied to each pid: <`f 0`, ..., `f (p-1)`>

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

Pointwise parallel application. **Parameters :**

- `vf` a parallel vector of functions $<f_0, \ldots, f_{p-1}>$
- `vv` a parallel vector of values $<v_0, \ldots, v_{p-1}>$

Returns the parallel vector $<f_0\ v_0, \ldots, f_{p-1}\ v_{p-1}>$

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

Global communication. **Parameters :**

- `f` $= <f_0, \ldots, f_{p-1}>$, f_i j is the value that processor `i` should send to processor `j`.

Returns a parallel vector $g = <g_0, \ldots, g_{p-1}>$ where $g_j\ i = f_i\ j$ is the value received by processor `j` from processor `i`.

```
val proj : 'a par -> int -> 'a
```

projection (dual of `mkpar`). Makes all the elements of a parallel vector global. **Parameters :**

- `v` a parallel vector <v_0, . . ., v_{p-1}>

Returns a function f such that f i = v_i

```
val abort : int -> string -> 'a
```

Aborts computation and quits. **Parameters :**

- `err` error code to return
- `msg` message to print on standard error

```
val start_timing : unit -> unit
val stop_timing : unit -> unit
val get_cost : unit -> float par
```

returns a parallel vector which contains, at each processor, the time elapsed between the calls to `start_timing` and `stop_timing`.
Raises `Timer_failure` if the call to one of those functions was meaningless (*e.g.* `stop_timing` called before `start_timing`).

```
end
```

3.1.2 Interface for modules providing BSP parameters

Access to the machine parameters from a configuration file.

```
module type MACHINE_PARAMETERS =
  sig

    type bsp = {
      p : int ;
      g : float ;
      l : float ;
      r : float ;
    }
```

Describes the BSP parameters of the machine.

```
    val read : int -> unit
```

Reads the parameters from the configuration file. **Parameters :**

- `bsp_p` The current number of processors to choose among the possible configurations

```
    val get : unit -> bsp
```

Get the current parameters.
Returns the value of the parameters as initialised by `read ()`

```
  end
```

3.1.3 Interface for low-level communication modules

Module providing the implementation of the communication functions

```
module type COMM =
  sig

    val initialize : unit -> unit
```

Performs implementation-dependent initialization. Should be called only once in the course of a program

`val finalize : unit -> unit`

Performs implementation-dependent finalization. This will be called at the end of the program.

`val pid : unit -> int`

Returns the processor ID of the host processor

`val nprocs : unit -> int`

Returns the number of processors in the parallel machine

`val argv : unit -> string array`

Returns the array of command-line arguments

`val send : 'a array -> 'a array`

`val wtime : unit -> float`

Returns the clock

`val abort : int -> unit`

Aborts the computation

`end`

3.2 Module `Bsmlbase` : Very often used functions

```
module Make :
  functor (Bsml :  Bsmlsig.BSML) ->   sig
```

Very often used functions

`val replicate : 'a -> 'a Bsml.par`

`replicate x` gives a parallel vector with the value `x` on each process.

`val parfun : ('a -> 'b) -> 'a Bsml.par -> 'b Bsml.par`

`parfun f` $<x_0,\ldots,x_{(p-1)}> = <f\ x_0,\ldots,f\ x_{(p-1)}>$

Same thing as `parfun` but with a function of arity 2, 3 or 4.

```
val parfun2 : ('a -> 'b -> 'c) -> 'a Bsml.par -> 'b Bsml.par -> 'c Bsml.par
val parfun3 :
  ('a -> 'b -> 'c -> 'd) ->
  'a Bsml.par -> 'b Bsml.par -> 'c Bsml.par -> 'd Bsml.par
val parfun4 :
  ('a -> 'b -> 'c -> 'd -> 'e) ->
  'a Bsml.par -> 'b Bsml.par -> 'c Bsml.par -> 'd Bsml.par -> 'e Bsml.par
```

Same thing as `apply` but with a function of arity 2, 3 or 4.

```
val apply2 :
```

```
  ('a -> 'b -> 'c) Bsml.par -> 'a Bsml.par -> 'b Bsml.par -> 'c Bsml.par
val apply3 :
  ('a -> 'b -> 'c -> 'd) Bsml.par ->
  'a Bsml.par -> 'b Bsml.par -> 'c Bsml.par -> 'd Bsml.par
val apply4 :
  ('a -> 'b -> 'c -> 'd -> 'e) Bsml.par ->
  'a Bsml.par -> 'b Bsml.par -> 'c Bsml.par -> 'd Bsml.par -> 'e Bsml.par
val mask : (int -> bool) -> 'a Bsml.par -> 'a Bsml.par -> 'a Bsml.par
val applyat : int -> ('a -> 'b) -> ('a -> 'b) -> 'a Bsml.par -> 'b Bsml.par
```

`applyat` n f_1 f_2 v applies function f_1 at process n and f_2 otherwise

```
val applyif :
  (int -> bool) -> ('a -> 'b) -> ('a -> 'b) -> 'a Bsml.par -> 'b Bsml.par
```

`applyif` b f_1 f_2 v applies function f_1 at process n if (b n) is true and f_2 otherwise

```
val procs : int list
```

`procs` is the list of the process numbers

```
val this : int Bsml.par
```

`this` is the parallel vector such as each process hold its number

```
val bsml_print : ('a -> unit) -> int -> 'a Bsml.par -> unit Bsml.par
```

`bsml_print print_element pid element` prints the value of `element` at process `pid` using the printer `print_element`

```
val parprint : ('a -> unit) -> 'a Bsml.par -> unit Bsml.par
```

`parprint print v` print the parallel vector `v` using the printer `print`, one line per process, each line beginning with the number of the process. For example, `(parprint print_int (this()))` will give the following for 4 processes:

```
0: 0
1: 0
2: 0
3: 0
```

```
val get_one : 'a Bsml.par -> int Bsml.par -> 'a Bsml.par
```

`get_one` $<x_0,\ldots,x_{p-1}>$ $<i_0,\ldots,i_{p-1}>$ evaluates to $<x_{i\_0},\ldots,x_{i\_p-1}>$. The process numbers are considered module p

```
val get_list : 'a Bsml.par -> int list Bsml.par -> 'a list Bsml.par
```

The order of the elements of the result list is the same as the order of the process numbers in the argument list.

```
val put_one : (int * 'a) Bsml.par -> 'a list Bsml.par
```

Each process holds a pair `(dst,v)` where `dst` is the number of the process of destination and `v` the value to send. If `dst` is not a valid process number, it is ignored. The result list is ordered by source process.

```
val put_list : (int * 'a) list Bsml.par -> 'a list Bsml.par
```

Each process holds an association liste of pairs (`dst,v`) where `dst` is the number of the process of destination and `v` the value to send. If `dst` is not a valid process number, it is ignored. If there are two pairs with the same key, only the first is considered.

```
val proj_list_pids : 'a Bsml.par -> (int * 'a) list
```

`proj_list_pids` returns a (int * 'a) list in which each first couple element is the number of the proc holding the 'a value

```
end
```

3.3 Module `Bsmlcomm` : Parallel functions with communications

```
module Make :
  functor (Bsml :  Bsmlsig.BSML) ->   sig
```

Parallel functions with communications

```
val shift : int -> 'a Bsml.par -> 'a Bsml.par
```

Shifts the values from processes to processes. The parallel cost is *n*p+l* where *n* is the average size of the values.

```
val shift_right : 'a Bsml.par -> 'a Bsml.par
val shift_left : 'a Bsml.par -> 'a Bsml.par
val totex : 'a Bsml.par -> (int -> 'a) Bsml.par
```

`totex` $<v_0,\ldots,v_{p-1}>$ evaluates to $<f_0,\ldots,f_{p-1}>$ such as $(f_i\ j)=v_j$. `total_exchhange` $<v_0,\ldots,v_{p-1}>$ evaluates to $<l_0,\ldots,l_{p-1}>$ such as the j^{th} element of l_i is v_j.

```
val total_exchange : 'a Bsml.par -> 'a list Bsml.par
exception Scatter
```

`scatter partition from` $<v_0,\ldots,v_{p-1}>$, scatters the value v_{from} which is partioned by the function partition. `partition v pid` indicates the part of `v` which will be send to process `pid` (it is possible to send nothing by using the value `None`). `from` must be a valid process number, otherwise `Scatter` is raised.

```
val scatter : ('a -> int -> 'b option) -> int -> 'a Bsml.par -> 'b Bsml.par
val scatter_list : int -> 'a list Bsml.par -> 'a list Bsml.par
```

Specialized version for lists, arrays and strings respectively.

```
val scatter_array : int -> 'a array Bsml.par -> 'a array Bsml.par
val scatter_string : int -> string Bsml.par -> string Bsml.par
exception Gather
```

`gather dst` $<v_0,\ldots,v_{p-1}>$ gathers the values $v_0,\ldots,v_{p-1}$ to process `dst`. With `gather` the result is a function `f` such as `(f i)` gives v_i with i being a valid process number. With `gather_list` the result is the list $[v_0;\ldots;v_{p-1}]$. `gather_list` corresponds to the function `gather` of BSMLlib 0.1. If `dst` is not a valid process, then `Gather` is raised.

```
val gather : int -> 'a Bsml.par -> (int -> 'a option) Bsml.par
val gather_list : int -> 'a Bsml.par -> 'a list Bsml.par
exception Bcast
```

`bcast_direct root` $v_0,\ldots,v_{p-1}$=$v_n,\ldots,v_n$ if `root` is a valid process number, otherwise `Bcast` is raised. The parallel cost is *size*(p-1)*g+l*, where *size* is the size of the value v_{root}.

```
val bcast_direct : int -> 'a Bsml.par -> 'a Bsml.par
```

```
val bcast_totex_gen :
  ('a -> int -> 'b option) ->
  ((int -> 'b) -> 'c) -> int -> 'a Bsml.par -> 'c Bsml.par
```

`bcast_totex_gen partition paste root v` broadcasts the value at process `root` of parallel vector `v`. The algorithm is the so called total exchange broadcast. It proceeds in two super-steps: First the value at process `root` is scattered using `partition`. Then those parts are totally exchanged and pasted. For large values this algorithms is faster than `bcast_direct`.

```
val bcast_totex_list : int -> 'a list Bsml.par -> 'a list Bsml.par
```

Specialized versions for lists, arrays, strings and values of any type (but this general version implies the marshalling of values and then the use of `bcast_totex_string`.

```
val bcast_totex_array : int -> 'a array Bsml.par -> 'a array Bsml.par
```

```
val bcast_totex_string : int -> string Bsml.par -> string Bsml.par
```

```
val bcast_totex : int -> 'a Bsml.par -> 'a Bsml.par
```

```
val scan_direct : ('a -> 'a -> 'a) -> 'a Bsml.par -> 'a Bsml.par
```

If `op` is an associative operation, `scan_direct op` $<v_0,\ldots,v_{p-1}> = <s_0,\ldots,s_{p-1}>$ where $s_i = op_{0<=k<=i} v_k$. Communication cost: *(p-1)*n*g+l* where n is the average size of values v_i.

```
val scan_logp : ('a -> 'a -> 'a) -> 'a Bsml.par -> 'a Bsml.par
```

Computes the same result than `scan_direct` but with communication cost: $_{i(logp)*2*n*g+l}$.

```
val scan_wide :
  (('a -> 'a -> 'a) -> 'a Bsml.par -> 'a Bsml.par) ->
  (('a -> 'a -> 'a) -> 'b -> 'b) ->
  ('b -> 'a) ->
  (('a -> 'a) -> 'b -> 'b) -> ('a -> 'a -> 'a) -> 'b Bsml.par -> 'b Bsml.par
```

`scan_wide par_scan seq_scan last_element map op vv` is used to compute a parallel scan over a parallel vector of collections of values. `par_scan` is the parallel scan used. `seq_scan` is the sequential scan used. `last_element` is a function which return the last element of a collection. `map` is a map function over the collection, `op` is the operation used for the reduction and `vv` is the parallel vector of collections.

```
val scan_wide_direct :
  (('a -> 'a -> 'a) -> 'b -> 'b) ->
  ('b -> 'a) ->
  (('a -> 'a) -> 'b -> 'b) -> ('a -> 'a -> 'a) -> 'b Bsml.par -> 'b Bsml.par
```

Specialized version of `scan_wide` using `scan_direct` as parallel scan.

```
val scan_wide_logp :
  (('a -> 'a -> 'a) -> 'b -> 'b) ->
  ('b -> 'a) ->
  (('a -> 'a) -> 'b -> 'b) -> ('a -> 'a -> 'a) -> 'b Bsml.par -> 'b Bsml.par
```

Specialized version of `scan_wide` using `scan_logp` as parallel scan.

```
    val scan_list_direct :
      ('a -> 'a -> 'a) -> 'a list Bsml.par -> 'a list Bsml.par
    val scan_list_logp : ('a -> 'a -> 'a) -> 'a list Bsml.par -> 'a list Bsml.par
    val scan_array_direct :
      ('a -> 'a -> 'a) -> 'a array Bsml.par -> 'a array Bsml.par
    val scan_array_logp :
      ('a -> 'a -> 'a) -> 'a array Bsml.par -> 'a array Bsml.par
```

Folds. Similar to scans except that the produced vector contains the same value everywhere. This value is the value at the last process if a scan was computed (non wide case) or the value of the last element of the collection at the last processor if a wide scan was computed

```
    val fold_direct : ('a -> 'a -> 'a) -> 'a Bsml.par -> 'a Bsml.par
    val fold_wide :
      (('a -> 'a -> 'a) -> 'a Bsml.par -> 'a Bsml.par) ->
      (('a -> 'a -> 'a) -> 'b -> 'a) ->
      ('a -> 'a -> 'a) -> 'b Bsml.par -> 'a Bsml.par
    val fold_logp : ('a -> 'a -> 'a) -> 'a Bsml.par -> 'b Bsml.par
    val fold_list_direct : ('a -> 'a -> 'a) -> 'a list Bsml.par -> 'a Bsml.par
    val fold_list_logp : ('a -> 'a -> 'a) -> 'a list Bsml.par -> 'b Bsml.par
    val fold_array_direct : ('a -> 'a -> 'a) -> 'a array Bsml.par -> 'b Bsml.par
    val fold_array_logp : ('a -> 'a -> 'a) -> 'a array Bsml.par -> 'b Bsml.par
  end
```

3.4 Module `Bsmlsort` : Sorting

```
module Make :
   functor (Bsml :  Bsmlsig.BSML) ->   sig
```

Sorting

`regular_sampling_sort cmp list` sorts the list (or array) with respect to the order given by `cmp`. The regular sampling BSP algorithm is described in [39, 38]. This sort requires that the total number of elements be greater than p^2. Otherwise `Regular_sampling_sort` is raised. The regular sampling sort insures that at the end of the sort each processor will contains at most 2*n/p elements, where n is the total number of elements.

```
    exception Regular_sampling_sort
    val regular_sampling_sort_list :
      ('a -> 'a -> bool) -> 'a list Bsml.par -> 'a list Bsml.par
    val regular_sampling_sort_array :
      ('a -> 'a -> int) -> 'a array Bsml.par -> 'a array Bsml.par
  end
```

3.5 Module `Tools` : Useful sequential functions

```
val natmod : int -> int -> int
```

Modulo

```
val from_to : int -> int -> int list
```

from_to n1 n2 = [n1;n1+1;...;n2]

`val filtermap : ('a -> bool) -> ('a -> 'b) -> 'a list -> 'b list`

filtermap p f l applies f to each element of l which satifies the predicate p

`val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`

Function composition.

`val id : 'a -> 'a`

Identity function

`val is_empty : 'a -> bool`

Tests whether a value is considered as an empty message.

3.6 Module Bsmlbckcomp : For backward compatibility

```
module Make :
  functor (Bsml :  Bsmlsig.BSML) ->   sig
```

For backward compatibility

See the documentation of version 0.1. Those functions must be avoided from now on.

```
val bsml_begin : unit -> unit
val bsml_end : unit -> unit
exception Get_failure of string
val get : 'a Bsml.par -> int list Bsml.par -> (int, 'a) Hashtbl.t Bsml.par
exception Put_failure of string
val put : (int * 'a) list Bsml.par -> (int, 'a) Hashtbl.t Bsml.par
val bsml_abort_string : string -> unit
val scatter : ('a -> (int * 'b) list) -> int -> 'a Bsml.par -> 'b Bsml.par
```

See the documentation of version 0.2. This function should be avoided from now on.

```
exception Unsafe_proj
val safe_proj : 'a Bsml.par -> 'a
```

safe_proj <v,...,v> = v, raises the exception Unsafe_proj otherwise

```
end
```

3.7 Module Bsmlskeleton : Distributed structures

```
module type TYPE =
  sig
    type 'a t
    val init : int -> (int -> 'a) -> 'a t
    val length : 'a t -> int
    val empty : 'a t
    val map : ('a -> 'b) -> 'a t -> 'b t
```

```
    val zip : ('a -> 'b -> 'c) ->
      'a t -> 'b t -> 'c t
    val zip2 : ('a -> 'b -> 'c -> 'd) ->
      'a t ->
      'b t -> 'c t -> 'd t
    val to_array : 'a t -> 'a array
    val to_list : 'a t -> 'a list
    val shift_left : 'a -> 'a t -> 'a t
    val shift_right : 'a -> 'a t -> 'a t
    val concat : 'a t list -> 'a t
    val get_last : 'a t -> 'a
    val get_first : 'a t -> 'a
    val sub : 'a t -> int -> int -> 'a t
  end
```

```
module Make :
  functor (T : TYPE) -> functor (Bsml :  Bsmlsig.BSML) ->   sig
```

```
val make : (int -> 'a) -> int -> 'a T.t Bsml.par
```

`make f n` gives a parallel vector holding a distributed structure of size n, with f applied onto each index of the structure (written `[...]`):
$<$ `[ f 0 ; f 1 ; ...  ; f (n-1) ]` , `[ f n ; f (n+1) ; ...  ]` , ... $>$.
The parallel cost is *max* (w_i) where w_i is the execution time of *f i*.

```
val length : 'a T.t Bsml.par -> int
```

`length` gives the size of the distributed structure. The parallel cost is 2 *max* (w_i) where w_i is the exploring time of the structure at process i.

```
val extract : 'a T.t Bsml.par -> 'a T.t
```

`extract` gathers the data from the distributed structure into an array. The parallel cost is 2 *max* (w_i) where w_i is the exploring time of the structure at process i.

```
val par : 'a T.t -> 'a T.t Bsml.par
```

`par` is the dual of extract, it distributes a sequential structure over many processors.

```
val map : ('a -> 'b) -> 'a T.t Bsml.par -> 'b T.t Bsml.par
```

`map` applies a function of arity 1 on each element of the distributed structure. The parallel cost is 2 *max* (w_i) where w_i is the execution time of *f i*.

```
val zip :
  ('a -> 'b -> 'c) -> 'a T.t Bsml.par -> 'b T.t Bsml.par -> 'c T.t Bsml.par
```

`zip` applies a function of arity 2 on each pair (element of the first distributed structure, element of the second distributed structure). The parallel cost is 3 *max* (w_i) where w_i is the execution time of *f i*.

```
val map_index : (int -> 'a -> 'b) -> 'a T.t Bsml.par -> 'b T.t Bsml.par
```

`map_index` applies a function of arity 2 (first argument is the index within the distributed structure) on each element of the distributed structure. The parallel cost is 5 *max* (w_i) where w_i is the execution time of *f i*.

```
val zip_index :
  (int -> 'a -> 'b -> 'c) ->
  'a T.t Bsml.par -> 'b T.t Bsml.par -> 'c T.t Bsml.par
```

`zip_index` applies a function of arity 3 (first argument is the index within the distributed structure on each pair (element of the first distributed structure, element of the second distributed structure). The parallel cost is *6 max* (w_i) where w_i is the execution time of *f i*.

```
val to_array : 'a T.t Bsml.par -> 'a array Bsml.par
```

`to_array` converts a distributed structure into a distributed array. The parallel cost is *2 max* (w_i) where w_i is the exploring time of the structure at process i.

```
val to_list : 'a T.t Bsml.par -> 'a list Bsml.par
```

`to_list` converts a distributed structure into a distributed list. The parallel cost is *2 max* (w_i) where w_i is the exploring time of the structure at process i.

```
val shift_left : 'a -> 'a T.t Bsml.par -> 'a T.t Bsml.par
```

`shift_left v a` shifts the distributed structure a to the left and inserts v to its end. The parallel cost is *16 max* $(w_i) + g\ h + L$ where w_i is the exploring time of the structure at process i and h = 1.

```
val shift_right : 'a -> 'a T.t Bsml.par -> 'a T.t Bsml.par
```

`shift_right v a` shifts the distributed structure to the right and inserts v to its beginning. The parallel cost is *16 max* $(w_i) + g\ h + L$ where w_i is the exploring time of the structure at process i and h = 1.

```
end

module MakeArray :
  functor (Bsml : Bsmlsig.BSML) -> sig

    val make : (int -> 'a) -> int -> 'a array Bsml.par
```

`make f n` gives a parallel vector holding a distributed array of size n, with f applied onto each index of the array. < [|f 0 ; f 1 ; . . . ; f (n-1)|] , [|f n ; f (n+1) ; . . .|] , . . . >. The parallel cost is *max* (w_i) where w_i is the execution time of *f i*.

```
val length : 'a array Bsml.par -> int
```

`length` gives the size of the distributed array. The parallel cost is 2 *max* (w_i) where w_i is the exploring time of the array at process i.

```
val extract : 'a array Bsml.par -> 'a array
```

`extract` gathers the data from the distributed array into an array. The parallel cost is 2 *max* (w_i) where w_i is the exploring time of the array at process i.

```
val par : 'a array -> 'a array Bsml.par
```

`par` is the dual of extract, it distributes a sequential array over many processors.

```
val map : ('a -> 'b) -> 'a array Bsml.par -> 'b array Bsml.par
```

`map` applies a function of arity 1 on each element of the distributed array. The parallel cost is 2 *max* (w_i) where w_i is the execution time of *f i*.

```
val zip :
  ('a -> 'b -> 'c) ->
  'a array Bsml.par -> 'b array Bsml.par -> 'c array Bsml.par
```

`zip` applies a function of arity 2 on each pair (element of the first distributed array, element of the second distributed array). The parallel cost is *3 max* (w_i) where w_i is the execution time of *f i*.

```
val map_index : (int -> 'a -> 'b) -> 'a array Bsml.par -> 'b array Bsml.par
```

`map_index` applies a function of arity 2 (first argument is the index within the distributed array) on each element of the distributed array. The parallel cost is *5 max* (w_i) where w_i is the execution time of *f i*.

```
val zip_index :
  (int -> 'a -> 'b -> 'c) ->
  'a array Bsml.par -> 'b array Bsml.par -> 'c array Bsml.par
```

`zip_index` applies a function of arity 3 (first argument is the index within the distributed array on each pair (element of the first distributed array, element of the second distributed array). The parallel cost is *6 max* (w_i) where w_i is the execution time of *f i*.

```
val to_list : 'a array Bsml.par -> 'a list Bsml.par
```

`to_list` converts a distributed array into a distributed list. The parallel cost is *2 max* (w_i) where w_i is the exploring time of the array at process i.

```
val shift_left : 'a -> 'a array Bsml.par -> 'a array Bsml.par
```

`shift_left v a` shifts the distributed array a to the left and inserts v to its end. The parallel cost is *16 max* $(w_i) + g\ h + L$ where w_i is the exploring time of the array at process i and h = 1.

```
val shift_right : 'a -> 'a array Bsml.par -> 'a array Bsml.par
```

`shift_right v a` shifts the distributed array to the right and inserts v to its beginning. The parallel cost is *16 max* $(w_i) + g\ h + L$ where w_i is the exploring time of the array at process i and h = 1.

```
end
```

```
module MakeList :
  functor (Bsml : Bsmlsig.BSML) -> sig
```

```
val make : (int -> 'a) -> int -> 'a list Bsml.par
```

`make f n` gives a parallel vector holding a distributed list of size n, with f applied onto each index of the list. < [f 0 ; f 1 ; . . . ; f (n-1)] , [f n ; f (n+1) ; . . .] , . . . >. The parallel cost is *max* (w_i) where w_i is the execution time of *f i*.

```
val length : 'a list Bsml.par -> int
```

`length` gives the size of the distributed list. The parallel cost is *2 max* (w_i) where w_i is the exploring time of the list at process i.

```
val extract : 'a list Bsml.par -> 'a list
```

`extract` gathers the data from the distributed list into a list. The parallel cost is *2 max* (w_i) where w_i is the exploring time of the list at process i.

```
val par : 'a list -> 'a list Bsml.par
```

par is the dual of extract, it distributes a sequential list over many processors.

```
val map : ('a -> 'b) -> 'a list Bsml.par -> 'b list Bsml.par
```

map applies a function of arity 1 on each element of the distributed list. The parallel cost is *2 max* (w_i) where w_i is the execution time of *f i*.

```
val zip :
  ('a -> 'b -> 'c) -> 'a list Bsml.par -> 'b list Bsml.par -> 'c list Bsml.par
```

zip applies a function of arity 2 on each pair (element of the first distributed list, element of the second distributed list). The parallel cost is *3 max* (w_i) where w_i is the execution time of *f i*.

```
val map_index : (int -> 'a -> 'b) -> 'a list Bsml.par -> 'b list Bsml.par
```

map_index applies a function of arity 2 (first argument is the index within the distributed list) on each element of the distributed list. The parallel cost is *5 max* (w_i) where w_i is the execution time of *f i*.

```
val zip_index :
  (int -> 'a -> 'b -> 'c) ->
  'a list Bsml.par -> 'b list Bsml.par -> 'c list Bsml.par
```

zip_index applies a function of arity 3 (first argument is the index within the distributed list on each pair (element of the first distributed list, element of the second distributed list). The parallel cost is *6 max* (w_i) where w_i is the execution time of *f i*.

```
val to_array : 'a list Bsml.par -> 'a array Bsml.par
```

to_array converts a distributed list into a distributed array. The parallel cost is *2 max* (w_i) where w_i is the exploring time of the list at process i.

```
val shift_left : 'a -> 'a list Bsml.par -> 'a list Bsml.par
```

shift_left v a shifts the distributed list a to the left and inserts v to its end. The parallel cost is *16 max* $(w_i) + g\ h + L$ where w_i is the exploring time of the list at process i and h = 1.

```
val shift_right : 'a -> 'a list Bsml.par -> 'a list Bsml.par
```

shift_right v a shifts the distributed list to the right and inserts v to its beginning. The parallel cost is *16 max* $(w_i) + g\ h + L$ where w_i is the exploring time of the list at process i and h = 1.

```
end
```

Bibliography

[1] G. Akerholt, K. Hammond, S. Peyton-Jones, and P. Trinder. Processing transactions on GRIP, a parallel graph reducer. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93, Parallel Architectures and Languages Europe*, number 694 in Lecture Notes in Computer Science, Munich, June 1993. Springer.

[2] Arvind and R. Nikhil. I-structures: Data structures for parallel computing. *ACM Transactions on Programming Languages and Systems*, 11(4), 1989.

[3] O. Ballereau, F. Loulergue, and G. Hains. High-level BSP Programming: BSML and BSλ. In G. Michaelson and Ph. Trinder, editors, *Trends in Functional Programming*, pages 29–38. Intellect Books, 2000.

[4] M. Bamha, F. Bentayeb, and G. Hains. An efficient scalable parallel view maintenance algorithm for shared nothing multi-processor machines. In T. Bench-Capon, G. Soda, and A. Min Tjoa, editors, *10th International Conference on Database and Expert Systems Applications, DEXA'99*, number 1677 in LNCS, pages 616–625. Springer-Verlag, August 30 – September 3 1999.

[5] R. H. Bisseling. *Parallel Scientific Computation*. Oxford University Press, 2004.

[6] R. H. Bisseling and W. F. McColl. Scientific computing on bulk synchronous parallel architectures. In B. Pehrson and I. Simon, editors, *Technology and Foundations: Information Processing '94, Vol. I*, volume 51 of *IFIP Transactions A*, pages 509–514. Elsevier Science Publishers, Amsterdam, 1994.

[7] W. Bousdira, F. Gava, L. Gesbert, F. Loulergue, and G. Petiot. Functional Parallel Programming with Revised Bulk Synchronous Parallel ML. In Koji Nakano, editor, *2nd International Workshop on Parallel and Distributed Algorithms and Applications (PDAA)*. IEEE Computer Society, 2010.

[8] A. Braud and C. Vrain. A parallel genetic algorithm based on the BSP model. In *Evolutionary Computation and Parallel Processing GECCO & AAAI Workshop*, Orlando (Florida), USA, 1999.

[9] M. Cole. *Algorithmic Skeletons: Structured Management of Parallel Computation*. MIT Press, 1989.

[10] D. C. Dracopoulos and S. Kent. Speeding up genetic programming: A parallel BSP implementation. In *First Annual Conference on Genetic Programming*. MIT Press, July 1996.

[11] F. Dehne (Guest Editor). Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999.

[12] C. Foisy and E. Chailloux. Caml Flight: a portable SPMD extension of ML for distributed memory multiprocessors. In A. W. Böhm and J. T. Feo, editors, *Workshop on High Performance Functionnal Computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory, USA.

[13] F. Gava and F. Loulergue. A Static Analysis for Bulk Synchronous Parallel ML to Avoid Parallel Nesting. *Future Generation Computer Systems*, 21(5):665–671, 2005.

[14] Frédéric Gava. A Polymorphic Type System for BSML. Technical Report 2002-12, University of Paris Val-de-Marne, LACL, 2002.

[15] L. Granvilliers, G. Hains, Q. Miller, and N. Romero. A system for the high-level parallelization and cooperation of constraint solvers. In Y. Pan, S. G. Akl, and K. Li, editors, *Proceedings of International Conference on Parallel and Distributed Computing and Systems (PDCS)*, pages 596–601, Las Vegas, USA, 1998. IASTED/ACTA Press.

[16] G. Hains and C. Foisy. The Data-Parallel Categorical Abstract Machine. In A. Bode, M. Reeve, and G. Wolf, editors, *PARLE'93*, number 694 in LNCS, pages 56–67. Springer, 1993.

[17] G. Hains and F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch, editor, *Second International Workshop on Constructive Methods for Parallel Programming (CMPP'2000)*, Research Report MIP-2000-07, June 2000.

[18] G. Hains, F. Loulergue, and J. Mullins. Concrete data structures and functional parallel programming. *Theoretical Computer Science*, 258(1-2):233–267, 2001.

[19] J.M.D. Hill, W.F. McColl, and al. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.

[20] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, 1985.

[21] Guy Horvitz and Rob H. Bisseling. Designing a BSP version of ScaLAPACK. In Bruce Hendrickson et al., editor, *Proceedings Ninth SIAM Conference on Parallel Processing for Scientific Computing*. SIAM, Philadelphia, PA, 1999.

[22] G. Jones. *Programming in Occam*. Prentice-Hall, 1987.

[23] Xavier Leroy. The Objective Caml System 3.04, 2001. Web pages at `http://www.caml.org`.

[24] F. Loulergue. BSλ_p: Functional BSP Programs on Enumerated Vectors. In J. Kazuki, editor, *International Symposium on High Performance Computing*, number 1940 in Lecture Notes in Computer Science, pages 355–363. Springer, October 2000.

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

[26] F. Loulergue. Parallel Juxtaposition for Bulk Synchronous Parallel ML. In H. Kosch, L. Boszorményi, and H. Hellwagner, editors, *Euro-Par 2003*, number 2790 in LNCS, pages 781–788. Springer Verlag, 2003.

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

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

[29] W. F. McColl. Universal computing. In L. Bouge and al., editors, *Proc. Euro-Par '96*, volume 1123 of *LNCS*, pages 25–36. Springer-Verlag, 1996.

[30] W.F. McColl. Scalable parallel programming. Course given in the Oxford IGDP (Integrated Graduate Development Programme in Software Engineering), 1996.

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

[32] P. Panangaden and J. Reppy. The essence of concurrent ML. In F. Nielson, editor, *ML with Concurrency*, Monographs in Computer Science. Springer, 1996.

[33] S. Pelagatti. *Structured Development of Parallel Programs*. Taylor & Francis, 1998.

[34] R. O. Rogers and D. B. Skillicorn. Using the BSP cost model to optimise parallel neural network training. *Future Generation Computer Systems*, 14(5-6):409–424, 1998.

[35] D. B. Skillicorn. Multiprogramming BSP programs. Department of Computing and Information Science, Queen's University, Kingston, Canada, October 1996.

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

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

[38] K. R. Sujithan and J. M. D. Hill. Collection types for database programming in the BSP model. In *Fifth Euromicro Workshop on Parallel and Distributed Processing*. IEEE CS Press, January 1997.

[39] K. Ronald Sujithan. Towards a scalable, parallel object database - the bulk synchronous parallel approach,. Technical Report PRG-TR-17-96, Programming Research Group, Oxford University Computing Laboratory, August 1996.

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