The `BSMLlib` version 0.25 Reference Manual

Frédéric Loulergue
with[1] Olivier Ballereau, Frédéric Gava, Gaétan Hains
`http://bsmllib.free.fr`

March 8, 2004

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

Contents

1 Introduction **3**
- 1.1 The BSP Model . . . 4
- 1.2 Explicit processes and the SPMD programming style . . . 5
- 1.3 Overview of the core `BSMLlib` library . . . 6
  - 1.3.1 Examples . . . 7
  - 1.3.2 Remark on nesting . . . 8

2 The `BSMLlib` tools **9**
- 2.1 Batch compilation (`bsmllibc` et `bsmllibc.seq`) . . . 9
- 2.2 Native-code compilation . . . 9
- 2.3 The toplevel system (`bsmllib`) . . . 9

3 The `BSMLlib` library **10**
- 3.1 Module `Bsmllib` : The core of the BSMLlib library . . . 10
- 3.2 Module `Bsmlutils` : Useful sequential functions . . . 11
- 3.3 Module `Bsmlbase` : Very often used functions . . . 12
- 3.4 Module `Bsmlcomm` : Parallel functions with communications . . . 13
- 3.5 Module `Bsmlsort` : Sorting . . . 16
- 3.6 Module `Bsmlbckcomp` : For backward compatibility . . . 16

Bibliography **16**

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 [29] or non-functional [2, 10] — 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 [7, 30], 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 tradeoff between:

- the possibility of expressing parallel features necessary for predictable efficiency, but which make 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 thoroughly 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 [27] 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 [16] 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 [25], and to a library of bulk synchronous primitives for the Objective Caml [21] language which is sufficiently expressive and allows the prediction of execution times [15].

This framework is a good tradeoff 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 parametrized by the BSP parameters of the target architecture.

The version 0.1 of our `BSMLlib` library implements the BSλ-calculus primitives using Objective Caml [21] and BSPlib [17] 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 [28] which has been proved correct w.r.t. the confluent BSλ_p-calculus [22] using an intermediate semantics [23]. A polymorphic type system [12] has been designed, for which type inference is possible. The small number of basic operations allows `BSMLlib` to be taught to BSc. students.

The BSPlib library is no longer supported nor updated. Moreover `BSMLlib` 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 prescribe by the BSP model and global synchronisation barriers are too costly. Thus we will need encapsulated communications between differents architectures and subset synchronization [32]. The new version 0.2 of the `BSMLlib` library is hence base on MPI [34]. It also has a smaller number of primitives which are closer to the BSλ-calculus than the primitives of the 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.

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 `BSMLlib` library.

1.1 The BSP Model

The Bulk Synchronous Parallel (BSP) model [37, 26, 33] 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 $\mathbf{p}$; the time $\mathbf{l}$ required for a global synchronization ; the time $\mathbf{g}$ 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 [17].

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.

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:

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

where $w_i^{(s)}$ = local processing time on processor i during super-step s and $h_i^{(s)} = \max\{h_{i+}^{(s)}, h_{i-}^{(s)}\}$ 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 $\sum_s \text{Time}(s)$ of a BSP program composed of S super-steps is, therefore, a sum of 3 terms:

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

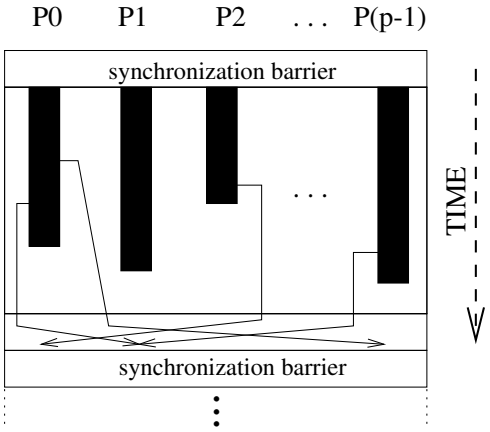

Figure 1.1: A BSP super-step

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 [5, 19], genetic algorithms [6] and genetic programming [8], neural networks [31], parallel databases [4], constraint solvers [13], *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" [9].

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 $E@i = E[\texttt{pid} \leftarrow i]$ and $[\![E]\!]_{CSP}$ refers to concurrent semantics defined by the communication library, for example the meaning of a CSP process [18]. 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 [20] 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 `BSMLlib` library, which is described below.

1.3 Overview of the core `BSMLlib` 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 `BSMLlib` library is based on the following elements.

It gives access to the BSP parameters of the underling architecture. In particular, it offers the function `bsp_p:unit->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 [24]).

There is also an abstract polymorphic type `'a par` which represents the type of p-wide parallel vectors of objects of type `'a`, one per process. The nesting of `par` types is prohibited. Our type system enforces this restriction [11]. This improves on the earlier design DPML/Caml Flight [14, 10] 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 [34]). 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:

```
mkpar:  (int -> 'a) -> 'a 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*.

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:

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

`apply (mkpar f) (mkpar e)` stores `(f i) (e i)` on process i. Neither the implementation of `BSMLlib`, nor its semantics [23] prescribe a synchronization barrier between two successive uses of `apply`.

Readers familiar with BSPlib [33, 17] 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:

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

Consider the expression: `put(mkpar(fun i->fs`$_i$`))` (*)

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 `Some v`. To send no value from process `j` to process `i`, (`fs`$_j$ `i`) must evaluate to `None`.

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 `None` if process `j` sent no message to process `i` or evaluates to `Some v` if process `j` sent the value `v` to the process `i`.

The full language would also contain a synchronous conditional operation:

```
ifat:  (bool par) * int * 'a * 'a -> 'a
```

such that `ifat (v,i,v1,v2)` will evaluate to `v1` or `v2` depending on the value of `v` at process `i`. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core `BSMLlib` contains the function: `at:bool par -> int -> bool` to be used only in the construction: `if (at vec pid) then...  else...` where `(vec:bool par)` and `(pid:int)`. `if at` expresses communication and synchronization phases. Global conditional is necessary of express algorithms like :

Repeat Parallel Iteration **Until** Max of local errors < **epsilon**

Without it, the global control cannot take into account data computed locally.

1.3.1 Examples

For example, one can define `get_one` such that :

$$\texttt{get\_one} < x_0, \ldots, x_{p-1} > < i_0, \ldots, i_{p-1} > = < x_{i_0}, \ldots, x_{i_{p-1}} >$$

```
(* val replicate : 'a -> 'a par *)
let replicate x =
  mkpar (fun pid -> x)

(* val apply2 : ('a -> 'b -> 'c) par -> 'a par -> 'b par -> 'c par
let apply2 f x y =
  apply (apply f x) y

(* val get_one; 'a 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 parts of the module `Bsmlbase`.

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, is 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 [12, 11]. This type system will be included in a full implementation of the BSMLlight language.

Chapter 2

The BSMLlib tools

2.1 Batch compilation (bsmllibc et bsmllibc.seq)

bsmllibc.mpi and bsmllibc.seq are scripts that calls the Objective Caml batch compiler ocamlc, which compiles Caml source files to bytecode object files and link these object files to produce standalone bytecode executable files with arguments to use the BSMLlib library.

bsmllibc.mpi produces a parallel version of the program (it also uses the mpicc command ; the ocamlc command is called with the -custom flag). To execute such a compiled program use the mpirun command.

bsmllibc.seq produces a sequential version of the program. It may be useful if you want to test a program on a sequential machine, particularly for programs that must be compiled with the -custom flag.

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

```
number_of_procs,g_parameter,l_parameter
```

g_parameter and l_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 correspond to the number of processor available on your machine.

See also the ocamlc command and the mprun command.

2.2 Native-code compilation

bsmllibopt.mpi and bsmllibopt.seq produce respectively parallel and sequential native code if the ocamlopt compiler is present on your machine.

See also the ocamlopt command.

2.3 The toplevel system (bsmllib)

bsmllib permits interactive use of the Objective Caml system with the BSMLlib 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 BSMLlib library

This chapter describes the functions provided by the BSMLlib library modules. These modules are automatically linked with the user's object code files by the bsmllibc and bsmllibopt commands.

3.1 Module Bsmllib : The core of the BSMLlib library

`type 'a par`

Abstract type for parallel vector of size p. In the following comments we will note <v_0,...,v_(p-1)> the parallel vector whose value at processor 0 is v_O, value at processor 1 is v_1, *etc.*

`val bsp_p : unit -> int`

`bsp_p()` returns the number p of processes of the parallel machine.

`val bsp_g : unit -> float`

`bsp_g()` returns the BSP parameter g of the parallel machine.

`val bsp_l : unit -> float`

`bsp_l()` returns the BSP parameter l of the parallel machine.

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

`mkpar f` evaluates to the parallel vector $<v_0,...,v_{p-1}>$ where v_i=`(f i)`.

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

`apply vf vv` where `vf` is a parallel vector of functions $<f_0,...,f_{p-1}>$ and `vv` a parallel vector of values $<v_0,...,v_{p-1}>$. evaluates to $<w_0,...,w_{p-1}>$ where w_i is the result of the evaluation of `(f{_i} v{_i})`. This operation is called pointwise parallel application.

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

Consider the expression `put(mkpar(fun i->fs`$_i$`))` (1). 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 `Some` v. To send no value from process j to process i, $(fs_j\, i)$ must evaluate to `None`. Expression (1) evaluates to a parallel vector containing a function fd_i of delivered messages on every process. At process i, $(fd_i\, j)$ evaluates to `None` if process j sent no message to process i or evaluates to `Some` v if process j sent the value v to the process i.

`exception At_failure of string`

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

NO MORE TRUE: `at` must be only used in a `if then else` expression. The expressions `if at` <b_0,...,b_{p-1}> `then` E_1 `else` E_2 will be evaluated to `E_1` if b_{pid} is `true`, to `E_2` if b_{pid} is `false`. E_1 and E_2 must be expressions of global type. `At_failure` is raised if one tries to access a pid not in the interval 0,...,p-1 where p is the number of processors of the machine.

`val unsafe_proj : 'a par -> 'a`
`safe_proj` <v,...,v> = v, otherwise unpredictable and unspeficied behaviour if the argument has not the same value at each process

`val initialize : unit -> unit`
`initialize()` initiates a parallel computation. When used with the sequential version of the module Bsmllib, it reads the first line of `$HOME/.bsmllibrc` in order to obtain the values for `bsp_p()`, `bsp_g()` and `bsp_l()`. When used with the parallel version of the module Bsmllib, it automatically determines the number of processors `bsp_p ()`, then reads the appropriated line of `$HOME/.bsmllibrc` in order to obtain the values for `bsp_g` and `bsp_l`. A program must have one and only one call to `initialize()`.

`val abort : int -> string -> unit`
`(abort error msg)` prints the message `msg` and aborts the computation with exit code `error`.

`exception Timer_failure of string`
`val start_timing : unit -> unit`
`val stop_timing : unit -> unit`
`val get_cost : unit -> float par`
`start_timing()` starts the timing. `stop_timing()` stops it. `get_cost()` returns a parallel vector which contains for each processor the time elapsed between the call of `start_timing` and `stop_timing`. The exception `Timer_failure` is raised if the call to one of those functions is meaningless (for eg a call to `stop_timing` if `start_timing` have not been called before.

3.2 Module `Bsmlutils` : 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 is_Some : 'a option -> bool`
`isSome v` is `true` if `v=Some v'`, `false` otherwise

`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 none : 'a -> 'b option`
Constant function which always returns `None`

`val noSome : 'a option -> 'a`
`val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b`
`val id : 'a -> 'a`
`val mklist : 'a -> 'a list`
`mklist v=[v]`

3.3 Module `Bsmlbase` : Very often used functions

```
val replicate : 'a -> 'a Bsmllib.par
```
`replicate x` gives a parallel vector with the value `x` on each process.

```
val parfun : ('a -> 'b) -> 'a Bsmllib.par -> 'b Bsmllib.par
```
`parfun f` $<x_0,\ldots,x_{(p-1)}> = <f\ x_0,\ldots,f\ x_{(p-1)}>$

```
val parfun2 :
  ('a -> 'b -> 'c) -> 'a Bsmllib.par -> 'b Bsmllib.par -> 'c Bsmllib.par
val parfun3 :
  ('a -> 'b -> 'c -> 'd) ->
'a Bsmllib.par -> 'b Bsmllib.par -> 'c Bsmllib.par -> 'd Bsmllib.par
val parfun4 :
  ('a -> 'b -> 'c -> 'd -> 'e) ->
'a Bsmllib.par ->
'b Bsmllib.par -> 'c Bsmllib.par -> 'd Bsmllib.par -> 'e Bsmllib.par
```
Same thing as `parfun` but with a function of arity 2, 3 or 4.

```
val apply2 :
  ('a -> 'b -> 'c) Bsmllib.par ->
'a Bsmllib.par -> 'b Bsmllib.par -> 'c Bsmllib.par
val apply3 :
  ('a -> 'b -> 'c -> 'd) Bsmllib.par ->
'a Bsmllib.par -> 'b Bsmllib.par -> 'c Bsmllib.par -> 'd Bsmllib.par
val apply4 :
  ('a -> 'b -> 'c -> 'd -> 'e) Bsmllib.par ->
'a Bsmllib.par ->
'b Bsmllib.par -> 'c Bsmllib.par -> 'd Bsmllib.par -> 'e Bsmllib.par
```
Same thing as `apply` but with aa function of arity 2, 3 or 4.

```
val mask :
  (int -> bool) -> 'a Bsmllib.par -> 'a Bsmllib.par -> 'a Bsmllib.par
val applyat :
  int -> ('a -> 'b) -> ('a -> 'b) -> 'a Bsmllib.par -> 'b Bsmllib.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 Bsmllib.par -> 'b Bsmllib.par
val procs : unit -> int list
```
`procs()` returns the list of the process numbers

```
val this : unit -> int Bsmllib.par
```
`this()` returns the parallel vector such as each process hold its number

```
val last : unit -> int
```
`last()`=*p-1*

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

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

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

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

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

3.4 Module `Bsmlcomm` : Parallel functions with communications

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

`val shift_right : 'a Bsmllib.par -> 'a Bsmllib.par`

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

`val total_exchange : 'a Bsmllib.par -> 'a list Bsmllib.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.

```
exception Scatter
val scatter :
  ('a -> int -> 'b option) -> int -> 'a Bsmllib.par -> 'b Bsmllib.par
```

`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_list : int -> 'a list Bsmllib.par -> 'a list Bsmllib.par
val scatter_array : int -> 'a array Bsmllib.par -> 'a array Bsmllib.par
val scatter_string : int -> string Bsmllib.par -> string Bsmllib.par
```

Specialized version for lists, arrays and strings respectively.

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

`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};...;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.

```
exception Bcast
val bcast_direct : int -> 'a Bsmllib.par -> 'a Bsmllib.par
```

`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_totex_gen :
  ('a -> int -> 'b option) ->
((int -> 'b) -> 'c) -> int -> 'a Bsmllib.par -> 'c Bsmllib.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 Bsmllib.par -> 'a list Bsmllib.par
val bcast_totex_array : int -> 'a array Bsmllib.par -> 'a array Bsmllib.par
val bcast_totex_string : int -> string Bsmllib.par -> string Bsmllib.par
val bcast_totex : int -> 'a Bsmllib.par -> 'a Bsmllib.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 scan_direct : ('a -> 'a -> 'a) -> 'a Bsmllib.par -> 'a Bsmllib.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 Bsmllib.par -> 'a Bsmllib.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 Bsmllib.par -> 'a Bsmllib.par) ->
(('a -> 'a -> 'a) -> 'b -> 'b) ->
('b -> 'a) ->
(('a -> 'a) -> 'b -> 'b) ->
('a -> 'a -> 'a) -> 'b Bsmllib.par -> 'b Bsmllib.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 Bsmllib.par -> 'b Bsmllib.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 Bsmllib.par -> 'b Bsmllib.par
```

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

```
val scan_list_direct :
  ('a -> 'a -> 'a) -> 'a list Bsmllib.par -> 'a list Bsmllib.par
val scan_list_logp :
  ('a -> 'a -> 'a) -> 'a list Bsmllib.par -> 'a list Bsmllib.par
val scan_array_direct :
  ('a -> 'a -> 'a) -> 'a array Bsmllib.par -> 'a array Bsmllib.par
val scan_array_logp :
  ('a -> 'a -> 'a) -> 'a array Bsmllib.par -> 'a array Bsmllib.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 Bsmllib.par -> 'a Bsmllib.par
val fold_wide :
  (('a -> 'a -> 'a) -> 'a Bsmllib.par -> 'a Bsmllib.par) ->
(('a -> 'a -> 'a) -> 'b -> 'a) ->
('a -> 'a -> 'a) -> 'b Bsmllib.par -> 'a Bsmllib.par
val fold_logp : ('a -> 'a -> 'a) -> 'a Bsmllib.par -> 'b Bsmllib.par
val fold_list_direct :
  ('a -> 'a -> 'a) -> 'a list Bsmllib.par -> 'a Bsmllib.par
val fold_list_logp :
  ('a -> 'a -> 'a) -> 'a list Bsmllib.par -> 'b Bsmllib.par
val fold_array_direct :
  ('a -> 'a -> 'a) -> 'a array Bsmllib.par -> 'b Bsmllib.par
```

```
val fold_array_logp :
  ('a -> 'a -> 'a) -> 'a array Bsmllib.par -> 'b Bsmllib.par
exception Unsafe_proj
val safe_proj : 'a Bsmllib.par -> 'a
```

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

3.5 Module `Bsmlsort` : Sorting

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

`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 [36, 35]. 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.

3.6 Module `Bsmlbckcomp` : 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 Bsmllib.par -> int list Bsmllib.par -> (int, 'a) Hashtbl.t Bsmllib.par
exception Put_failure of string
val put : (int * 'a) list Bsmllib.par -> (int, 'a) Hashtbl.t Bsmllib.par
val bsml_abort_string : string -> unit
val scatter :
  ('a -> (int * 'b) list) -> int -> 'a Bsmllib.par -> 'b Bsmllib.par
```

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 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.

[6] 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.

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

[8] 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.

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

[10] 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.

[11] F. Gava and F. Loulergue. Synthèse de types pour Bulk Synchronous Parallel ML. In *Journées Francophones des Langages Applicatifs (JFLA 2003)*, january 2003.

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

[13] 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.

[14] 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.

[15] 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.

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

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

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

[19] 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.

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

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

[22] 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.

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

[24] F. Loulergue. Parallel Composition and Bulk Synchronous Parallel Functional Programming. In S. Gilmore, editor, *Trends in Functional Programming, Volume 2*, pages 77–88. Intellect Books, 2001.

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

[26] 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.

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

[28] 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.

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

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

[31] 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.

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

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

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

[35] 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.

[36] 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.

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