MATHEMATICAL ENGINEERING
TECHNICAL REPORTS

A Tutorial Implementation of the Diffusion Algorithmic Skeleton with the BSMLlib Library

Frederic LOULERGUE and Zhenjiang HU and
Kazuhiko KAKEHI

METR 2004–06 February 2004

DEPARTMENT OF MATHEMATICAL INFORMATICS
GRADUATE SCHOOL OF INFORMATION SCIENCE AND TECHNOLOGY
THE UNIVERSITY OF TOKYO
BUNKYO-KU, TOKYO 113-8656, JAPAN

WWW page: http://www.i.u-tokyo.ac.jp/mi/mi-e.htm

A Tutorial Implementation of the Diffusion Algorithmic Skeleton with the BSMLLIB Library

Frederic Loulergue [*] Zhenjiang Hu[†] Kazuhiko Kakehi[†]

February 5, 2004

Abstract

Skeleton programming enables programmers to build parallel programs easier by providing efficient ready-made parallel algorithms. The diffusion skeleton was proposed (associated with a method for program derivation) to abstract a good combination of primitive skeletons, such as map, parallel reduction and parallel prefix sum (scan).

The BSMLLIB library whose design is based on formal semantics is a library for the Objective Caml language to support Bulk Synchronous Parallelism. It offers a small set of primitives which permits to write any deterministic BSP algorithm.

In this paper we study the implementation of the diffusion parallel skeleton using the BSMLLIB library in a tutorial way.

1 Introduction

With the increasing popularity of parallel programming environments such as PC cluster, more and more people, including those who have little knowledge of parallel architecture and parallel programming, are hoping to write parallel programs. This situation eagerly calls for models and methodologies which can assist programming parallel computers effectively and correctly.

The design of parallel programming languages is a trade-off 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.

[*]Laboratory of Algorithms, Complexity and Logic, University Paris XII, France, loulergue@univ-paris12.fr

[†]Department of Mathematical Informatics The University of Tokyo, {hu,kaz}@mist.i.u-tokyo.ac.jp

The data parallel model turns out to be one of the most successful ones for programming massively parallel computers. To support parallel programming, this model basically consists of two parts:

- a parallel data structure to model a uniform collection of data which can be organized in a way that each element can be manipulated in parallel; and
- a fixed set of parallel skeletons on the parallel data structure to abstract parallel structures of interest, which can be used as building blocks to write parallel programs.

Typically, these skeletons include element-wise arithmetic and logic operations, reductions, prescans, and data broadcasting. This model not only provides programmers an easily understandable view of a single execution stream of a parallel program, but also makes the parallelizing process easier because of explicit parallelism of the skeletons.

Parallel skeletons are often too primitive to describe programs solving a bit complicated problems. In order to make programs efficients, programmers are required to choose appropriate primitive skeletons and combine them in a suitable way. It is not an easy task, since programming is apt to become a process with much a trial and error.

To overcome this problem, we proposed a parallel skeleton, namely *diffusion skeleton* **diff** [2]. This skeleton is derived from the *Diffusion Theorem* [19] and is defined in terms of primitive skeletons **map**, **reduce** and **scan**. It abstracts a 'good' combination of parallel primitives, and thanks to the underlying theorem, recursive functions defined naturally in some specific from over recursive data structure can be, under some conditions, turned into the form using the **diff** skeleton.

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), we use explicit processes corresponding to the processors of the parallel machine. *Bulk Synchronous Parallel* (BSP) computing [28, 33] 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.

An operational approach has led to a BSP λ-calculus that is confluent and universal for BSP algorithms [27, 21, 24], and to a library of bulk synchronous primitives for the Objective Caml [20, 7, 30] language which is sufficiently expressive and allows the prediction of execution times [16, 23].

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

- the defined calculus is a *confluent calculus* so:
  - one can design purely functional parallel languages from it. Without side-effects, programs are easier to prove [10, 11], 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.

Bulk Synchronous Parallel ML or BSML is our extension of ML for programming direct-mode parallel BSP algorithms as functional programs. A BSP algorithm is said to be in *direct mode* [14] when its physical process structure is made explicit. Such algorithms offer predictable and scalable performance and BSML expresses them with a small set of primitives taken from the *confluent* BSλ-calculus: a parallel constructor, asynchronous parallel function application, synchronous global communications and a synchronous global conditional.

There is currently no full implementation of BSML but there is a partial implementation as a library. The BSMLLIB library [1] implements the BSML primitives using Objective Caml [20] and MPI [34]. BSMLLIB can be taught to BSc. students due to the small number of basic operations[1]. There are additional modules which provide several usual parallel algorithms. They constitute what is called the BSMLLIB standard library.

This paper is tutorial implementation of the diffusion skeleton using the BSMLLIB library. We first describe the bulk synchronous parallel model and the core BSMLLIB library with some basic examples (section 2). Then we give some elements of the Bird-Merteens formalism (BMF) and describe the diffusion algorithmic skeleton (section 3). Section 4 explains the parallel implementation of the diffusion skeletons as an higher-order function written in Objective Caml using the BSMLLIB library. It also presents an example of use of this skeleton. Appendix B explains the installation and basic use of the BSMLLIB library.

2 Functional Bulk Synchronous Parallelism

2.1 The Bulk Synchronous Parallel Model

The Bulk Synchronous Parallel (BSP) model [35, 29, 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 **p**

[1]It is actually taught at the universities of Orl 饌 ns and Paris Val de Marne

- 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 [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. 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.

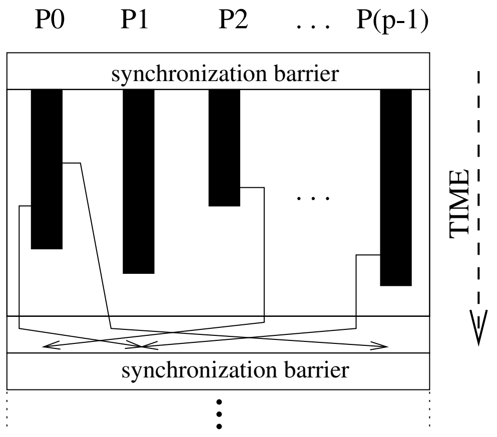

Figure 1: A BSP super-step

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

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

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, 18], genetic algorithms [6] and genetic programming [9], neural networks [31], parallel databases [3], 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" [8].

2.2 The Bulk Synchronous Parallel ML 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 [25]).

It also offers the `bsp_g` and `bsp_l` functions which both have type `unit->float`. In BSMLLIB, these parameters are read from the `~/.bsmllibrc` file (which should contain lines of the form: p, g, l for example $4, 1., 11.5$).

The abstract polymorphic type `'a par` 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 [13, 12].

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*.

Example 1 *The expression* `mkpar(fun pid->pid)` *will create the following parallel vector:*

0	1	$\cdots$	$(p-1)$

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 [22] prescribe a synchronization barrier between two successive uses of `apply`.

Example 2 *Let consider the following expression:*

```
let vf = mkpar(fun i->(+) i)
and vv = mkpar(fun i->2*i+1) in
apply vf vv
```

The two parallel vectors are respectively equivalent to:

`fun x->x+0`	`fun x->x+1`	$\cdots$	`fun x->x+(p-1)`

and

0	3	$\cdots$	$2\times(p-1)+1$

The expression `apply vf vv` *is then evaluated to:*

0	4	$\cdots$	$2\times(p-1)+2$

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`.

Example 3 *Consider a parallel machine with 4 processors and functions* f_i *whose types are* **int** $\rightarrow$ α**option par** *such as* $(f_i\ (i+1)) =$ **Some** v_i *for* $i = 0,1,2$ *and* $(f_i\ j) =$ **None** *otherwise.*

The expression **mkpar**(**fun** $i \rightarrow f_i$) *would be evaluated as follows:*

1. *First at each process the function is applied to all process identifiers to produce p values, the messages to be sent by the processes. In the following figure, a column represents the values produced at one process and the lines are ordered by destination (first line represents the messages to be sent to process 0, etc.):*

None	*None*	*None*	*None*
Some v_0	*None*	*None*	*None*
None	*Some* v_1	*None*	*None*
None	*None*	*Some* v_2	*None*

2. *Then the exchange of messages is actually performed. If we think of the previous table as a matrix, the resulting matrix is obtained by transposition:*

None	*Some* v_0	*None*	*None*
None	*None*	*Some* v_1	*None*
None	*None*	*None*	*Some* v_2
None	*None*	*None*	*None*

3. *Finally the parallel vector of functions is produced. Each process i holds an array* a_i *of size p (a column of the previous matrix) and the function is* **fun** $x \rightarrow a_i.(x)$. *In our example at process* 3, $(f_3\ 0) =$ **None** *which means that process* 3 *received no message from process* 0 *and* $(f_3\ 2) =$ **Some** v_2 *which means that process* 2 *sent the value* v_2 *to process* 3.

The full language would also contain a synchronous conditional operation:

if e **at** n **then** e_1 **else** e_2

It evaluates to `v1` (the value obtained by evaluating e_1) or `v2` (the value obtained by evaluating e_2) depending on the value of the parallel vector of booleans `e` at process given by the integer `n`. 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: 'a par -> int -> 'a
```

to be used in the constructions:

- `if (at vec pid) then ... else ...` where `(vec:bool par)` and `(pid:int)`
- `match at e pid with ...` where `pid:int`

`at` expresses communication and synchronization phases. Global conditional is necessary to express algorithms like:

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

Without it, the global control cannot take into account data computed locally. It is possible to use the `at` functions in other situations but one should avoid the (hidden) nesting of parallel vectors. For example the following expression (types are given for subexpressions to ease the understanding):

```
(* com: (int->true option) par *)
let com = put(mkpar(fun i d->if i=d then Some true else None))
and this = mkpar(fun i->i) in
mkpar(fun i->if i<(bsp_p()/2) then at (apply com this) 0 else None)
```

is not a correct program (you can write it and compile it with the BSMLLIB library but the execution will fail and the type checking by our type system [12] fails) because the parallel expression `(apply com this)` would be evaluated *inside* a `mkpar`. It breaks the BSP model because one part of the parallel machine will evaluate an expression with communications and *synchronization* and another half will evaluate an expression without communication and synchronization: *global* synchronizations are required in the BSP model (all the processes must be involved). For a detailed discussion about these problems, see [12]. The following program can be safely executed because `e1` is already evaluated when it is used inside the `mkpar` function.

```
let com = put(mkpar(fun i d->if i=d then Some true else None))
and this = mkpar(fun i->i) in
let e1 = at (apply com this) 0 in
mkpar(fun i->if i<(bsp_p()/2) then e1 else None)
```

2.3 Examples

This small set of primitives is enough to write any deterministic BSP algorithm. There are only few additional primitives for initialization (to be called at the beginning of the program):

```
initialize: unit->unit
```

and for timing:

```
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. The core library is contained in the module `Bsmllib`.

All these primitives are used for "flat" programming. There also exist three other primitives for parallel composition [25, 26]. These primitives are not yet available in the current distribution.

When one start to program using the BSMLLIB library, it appears that some other functions ease the programming. These functions are given in the BSMLLIB standard library. It is to notice that one can program without them and that they are all written using the primitives only.

`replicate` create a parallel vector with the same value everywhere:

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

It is often convenient to apply the same sequential function to all the elements of a parallel vector. It can be done using the `parfun` functions. `parfun` is for functions with one argument, `parfun2` is for functions with two arguments, etc.:

```
(* val parfun: ('a -> 'b) -> 'a par -> 'b par *)
let parfun f = apply (replicate f)

(* val parfun2: ('a -> 'b -> 'c) -> 'a par -> 'b par -> 'c par *)
let parfun2 f v1 = apply (parfun f v1)
```

In the following we will also use two additional functions which are not (yet) part of the standard library.

`applyat n` $\mathtt{f}_1$ $\mathtt{f}_2$ `vv` applies function $\mathtt{f}_1$ at process *n* and function $\mathtt{f}_2$ at other processes.

```
(* val applyat: int -> ('a -> 'b) -> ('a -> 'b) -> 'a par -> 'b par *)
let applyat n f g =
  let pf = mkpar(fun pid ->if pid=n then f else g) in
  apply pf
```

parpair_of_pairpar transforms a parallel vector of pairs in a pair of parallel vectors:

```
(* val parpair_of_pairpar: ('a * 'b) par -> 'a par * 'b par *)
let parpair_of_pairpar vv = (parfun fst vv,parfun snd vv)
```

The semantics of the total exchange function is given by:

$$\texttt{totex}\,\langle\, v_0\,, \ldots,\, v_{p-1}\,\rangle \;=\; \langle\, f\,, \ldots,\, f\,, \ldots,\, f\,\rangle$$

where $\forall i.(0 \leq i < p) \Rightarrow (fi) = v_i$. The code is as follows where noSome just removes the Some constructor (the function is given in appendix A) and compose is usual function composition:

```
(* val totex: 'a par -> (int -> 'a) par *)
let totex vv =
  parfun (compose noSome) (put(parfun (fun v dst->Some v) vv))
```

3 The Diffusion Parallel Skeleton and the Diffusion Theorem

We will use the BMF data parallel programming model [4, 32] to describe the diffusion skeleton. We choose BMF because it can provide us a concise way to describe both programs and transformation of programs. In this section we present the BMF notation and the diffusion skeleton

3.1 Basic BMF Notation

Functions. Function application is denoted by a space and the argument which may be written without brackets. Thus $f\ a$ means $f(a)$.

Functions are curried, and application associates to the left. Thus $f\ a\ b$ means $(f\ a)\ b$. Function application binds stronger than any other operator, so $f\ a \oplus b$ means $(f\ a) \oplus b$, not $f(a \oplus b)$. Function composition is denoted by a centralized circle $\circ$. By definition, we have $(f\ \circ\ g)a = f(g\ a)$. Function composition is an associative operator. Infix binary operators will often be denoted by $\oplus$, $\otimes$ $\odot$ and can be *sectioned*; an infix binary operator like $\oplus$ can be turned into unary or binary functions by $a \oplus b = (a\oplus)b = (\oplus b)a = (\oplus)\ a\ b$.

Parallel Data Structure: Join Lists. Join lists (or append lists) are finite sequences of values of the same type. A list is either the empty, a singleton, or the concatenation of two other lists. We write $[]$ for the empty list, $[a]$ for the singleton list with element a and $x +\!\!+ y$ for the concatenation (join) of two lists x and y. Concatenation is associative, and $[]$ is its unit. For example, $[1] +\!\!+ [2] +\!\!+ [3]$ denotes a list with three elements, often abbreviated to $[1;2;3]$. We also write $a : x$ for $[a] +\!\!+ x$. If a list is constructed only by the constructor of $[]$ and $:$, we call it cons list.

Parallel Skeletons: map, reduce, scan, zip. It has been shown [32] that BMF is a nice architecture-independent parallel computation model, consisting of a small fixed set of specific higher order functions which can be regarded as parallel skeletons suitable for parallel implementation. Four important higher order functions are **map**, **reduce**, **scan** and **zip**.

map is the skeleton which applies a function to every element in a list. **reduce** is the skeleton which collapses a list into a single value by repeated application of some associative binary operator. **scan** is the skeleton which computes the prefix sums using some associative binary operator. **zip** combines two lists into a list of pairs.

Their informal definitions are:

$$\begin{aligned}
\mathbf{map}\ f\ [x_1,x_2,\ldots,x_n] &= [f\,x_1, f\,x_2,\ldots, f\,x_n] \\
\mathbf{reduce}\ (\oplus)\ [x_1,x_2,\ldots,x_n] &= x_1 \oplus x_2 \oplus \ldots \oplus x_n \\
\mathbf{scan}\ (\oplus)\ [x_1,x_2,\ldots,x_n] &= [\iota_\oplus; x_1; x_1 \oplus x_2; \ldots; x_1 \oplus x_2 \oplus \ldots \oplus x_n] \\
\mathbf{zip}\ [x_1,x_2,\ldots,x_n]\ [y_1,y_2,\ldots,y_n] &= [(x_1,y_1),(x_2,y_2,\ldots,(x_n,y_n))]
\end{aligned}$$

The length of the result of **scan** is $n+1$ for an input list of length n. **scan'** and **prescan** skeletons are scan-like skeletons whose result has length n. Their informal definitions are:

$$\begin{aligned}
\mathbf{scan}'\ (\oplus)\ [x_1,x_2,\ldots,x_n] &= [x_1; x_1 \oplus x_2; \ldots; x_1 \oplus x_2 \oplus \ldots \oplus x_n] \\
\mathbf{prescan}\ (\oplus)\ [x_1,x_2,\ldots,x_n] &= [\iota_\oplus; x_1; x_1 \oplus x_2; \ldots; x_1 \oplus x_2 \oplus \ldots \oplus x_{n-1}]
\end{aligned}$$

3.2 Diffusion

Diffusion Theorem [19] describes a transformation rule from a recursive definition into a composition of **map**, **reduce**, **scan**.

Theorem 1 (Diffusion) *Given a function h defined in the recursive following form:*

$$\begin{aligned}
h\ []\ c &= g_1\ c \\
h\ (x : xs)\ c &= k(x,c)\ \oplus\ h\ xs\ (c \otimes g_2\, x)
\end{aligned}$$

If $\oplus$ and $\otimes$ are associative and have units, then h can be diffused into the following form.

$$h\ xs\ c\ =\ \mathbf{reduce}\,(\oplus)\,(\mathbf{map}\ f\ as) \oplus g_1\ b$$

where

$$\begin{aligned} bs \mathbin{+\!\!+} [b] &= \mathbf{map}\ (c\otimes)\ (\otimes)\ (\mathbf{map}\ g_2\ xs)) \\ as &= \mathbf{zip}\ xs\ bs \end{aligned}$$

The **diff** skeleton [2] is defined, based upon this theorem.

Definition 1 (Diffusion Skeleton)

$$\mathbf{diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ xs\ c \ = \ \mathbf{reduce}\ (\oplus)\ (\mathbf{map\ k\ as}\ \oplus\ g_1\ b$$

where

$$\begin{aligned} bs \mathbin{+\!\!+} [b] &= \mathbf{map}\ (c\otimes)\ (\otimes)\ (\mathbf{map}\ g_2\ xs)) \\ as &= \mathbf{zip}\ xs\ bs \end{aligned}$$

and $\oplus$ *and* $\otimes$ *are associative operations with units.*

3.3 Example

To see how the diffusion theorem works in practice, consider a simple problem of eliminating smaller elements. An element is said to be smaller if it is less than some element be- fore itself in the list. For example, for the list $[1;5;3;4;5;7]$, 3 and 4 are smaller elements, and thus the resultant list is $[1;5;5;7]$. This problem can be solved directly; scan the list from left to right and eliminate every element which is less than the maximum of the scanned elements. That is,

$$\begin{aligned} \mathbf{se}\ [\,]\ c &= [\,] \\ \mathbf{se}\ (x:xs)\ c &= \mathbf{if}\ x<c\ \mathbf{then}\ \mathbf{se}\ xs\ c\ \mathbf{else}\ [x]\ \mathbin{+\!\!+}\ se\ xs\ x \end{aligned}$$

The second equation is not in the form where the theorem can be applied. A simple transformation of merging two recursive calls into a single one soon gives the appropriate form:

$$\mathbf{se}\ (x:xs)\ c \quad = \quad (\mathbf{if}\ x<c\ \mathbf{then}\ [\,]\ \mathbf{else}\ [x]\ ) \mathbin{+\!\!+} \mathbf{se}\ xs\ (\mathbf{if}\ x<c\ \mathbf{then}\ c\ \mathbf{else}\ x)$$

Now matching the recursive definition of **se** with that in the diffusion theorem yields:

$$\mathbf{se}\ xs\ c \ = \ \mathbf{reduce}\ (\oplus)\ (\mathbf{map}\ k\ as)\ \oplus\ g_1\ b$$

where

$$\begin{aligned} bs \mathbin{+\!\!+} [b] &= \mathbf{map}\ (c\otimes)\ (\mathbf{scan}\ (\otimes)\ (\mathbf{map}\ g_2\ xs)) \\ as &= \mathbf{zip}\ xs\ bs \\ p\oplus q &= p \mathbin{+\!\!+} q \\ c\otimes a &= \mathbf{if}\ a<c\ \mathbf{then}\ c\ \mathbf{else}\ a \\ k(x,c) &= \mathbf{if}\ x<c\ \mathbf{then}\ [\,]\ \mathbf{else}\ [x] \\ g_1\ c &= [\,] \\ g_2\ x &= x \end{aligned}$$

Consequently, we have come to an efficient parallel algorithm for this problem.

We can easily code it in terms of **diff** as follows:

$$\mathbf{se}\ xs\ c \ = \ \mathbf{diff}\ (\oplus)\ (\otimes)\ k\ g_1\ g_2\ xs\ c$$

where

$$\begin{aligned} p \oplus q &= p +\!\!+ q \\ c \otimes a &= \mathbf{if}\ a < c\ \mathbf{then}\ c\ \mathbf{else}\ a \\ k(x,c) &= \mathbf{if}\ x < c\ \mathbf{then}\ []\ \mathbf{else}\ [x] \\ g_1\ c &= [] \\ g_2\ x &= x \end{aligned}$$

4 Implementation of the Diffusion Skeleton Using the BSMLLIB

This section shows how to implement the diffusion skeleton using the BSMLLIB library. We begin to explain the `diff` function and then describe the functions called inside this `diff` function.

4.1 The `diff` function

The definition of the `diff` function follows exactly the definition given in section 3.2. The first argument is a binary (associative) operator and the second argument is its neutral. The third argument is also a binary operator. The fourth argument is what corresponds to the k binary function, functions g_1 and g_2 are the two next arguments. Then `diff` takes a parallel vector of lists which should be seen as one wide list. One can define a function which will take a list as input and distribute it among all processes. The last argument of `diff` is the accumulator c:

```
(* val diff:
  ('a->'a->'a) -> 'a ->                        binary operator, neutral
  ('b->'b->'b) ->                              binary operator
  ('c->'b->'a) -> ('b->'a) -> ('c->'b) ->      k, g1 and g2
  'c list par -> 'b -> 'a par                  list and accumulator *)

let diff op1 op1neutral op2 k g1 g2 xs c =
  let bs'=scanl op2 c (map g2 xs) in
  let nocut l = None,l in
  let b',bs=parpair_of_pairpar(applyat (bsp_p()-1) cutlast nocut bs') in
  let (Some b)=at b' (bsp_p()-1) in
  reducer op1 op1neutral (g1 b) (map2 k xs bs)
```

First `bs'` is calculated. We apply function `g2` to all the elements of the parallel vector of lists using the `map` function (defined section 4.2). Then we compute the parallel prefix sum `scanl` on the obtained parallel vector of lists using the operator `op2` and `c` as initial value. The `scanl` is the most complex function used in `diff`. Its implementation is explained in section 4.4.

If `xs` represents the list $[x_1; x_2; \ldots; x_n]$, it is the parallel vector (to ease the explanation we assume here that n can be divided by p but the program works for all cases):

$$\langle\ [x_1; x_2; \ldots; x_{n/p}]\ , \ldots,\ [x_{n+1-n/p}; x_{n+2-n/p}; \ldots; x_n]\ \rangle$$

then `bs'` is the parallel vector:

$$\langle\ [c; \mathtt{op2}\ c\ x_1; \ldots; \mathtt{op2}\ (\ldots)\ x_{n/p-1}]\ , \ldots,\ [\mathtt{op2}\ (\ldots)\ x_{n-n/p}; \ldots; \mathtt{op2}\ (\ldots)\ x_n]\ \rangle$$

At process $p-1$ we have $\frac{n}{p}+1$ values. We need to cut this last value and send it to all processes: this is done be the three next lines of the program. First at process $p-1$ we cut the last value using the `cutlast` sequential function which is given in appendix A. For non-empty lists it is defined by:

$$\mathtt{cutlast}\ [x_1; \ldots; x_n]\ =\ (\mathtt{Some}\ x_n,\ [x_1; \ldots; x_{n-1}])$$

At other processes the lists are unchanged. Then we transform the obtained parallel vector of pairs in pairs of parallel vectors using `parpairs_of_pairpar`. The first component is a parallel vector which contains `None` everywhere but at process $p-1$ contains `Some(`$\mathtt{op2}\ (\mathtt{op2}(\mathtt{op2}\ c\ x_1)\ldots)x_n$`)`. This last value is projected to all processes using the `at` function and the `Some` constructor is removed by pattern matching (fifth line of the program).

To end we apply the binary function `k` to all elements of the parallel vectors of lists `xs` and `bs` using the `map2` function defined in section 4.2. Then we reduce (sum) the obtained parallel vector of lists using the `reducer` function defined in section 4.3. The result is a parallel vector which contains the same value everywhere.

4.2 The `map` and `map2` functions

The Objective Caml functions `List.map` and `List.map2` are defined on lists with the following semantics:

$$\begin{array}{rcl} \mathtt{map}\ f\ [x_1; \ldots; x_n] & = & [(f\ x_1); \ldots; (f\ x_n)] \\ \mathtt{map2}\ f\ [x_1; \ldots; x_n]\ [y_1; \ldots; y_n] & = & [(f\ x_1\ y_1); \ldots; (f\ x_n\ y_n)] \end{array}$$

For `map2` it is mandatory that the two lists have the same number of elements.

To have such functions but for parallel vectors of lists instead of lists, only `parfun` functions are needed:

```
(* val map: ('a -> 'b) -> 'a list par -> 'b list par *)
let map f = parfun (List.map f)

(* val map2:
  ('a -> 'b -> 'c) -> 'a list par -> 'b list par -> 'c list par *)
let map2 f = parfun2 (List.map2 f)
```

4.3 The `reducer` function

To reduce a parallel vector of lists we can proceed in two super-steps:

- reduction of the lists held by each process
- total exchange of the computed values
- reduction of the p values (all the processes do the same work)

In this case the BSP cost (assuming the binary operation used has constant complexity) is:

$$(\frac{n}{p} + g \times (p-1) \times s + l) + p$$

where s is the size of the values exchanges (we assume that the p values have the same size).

This can of course may not be the most efficient BSP algorithm for reduction: it depends on the values of g,s and l. Thus it is possible to write different BSP algorithms with the same semantics but with different cost formula. The `diff` function could take the `reducer` algorithm as argument and the best reduction could be selected at runtime.

For example the reduction can be performed using $\log p$ super-steps using a binary communication schema where at each super-step values are exchanged only pairs of processes. In this case the cost would be:

$$\frac{n}{p} + (\log p) \times (g \times s + l + 1)$$

Such an algorithm could be interesting. For example if we have p=64 processors: we have to compare

$$63 \times g \times s + l + 64 \qquad < \qquad 6 \times g \times s + 6 \times l + 1$$

which is equivalent to:

$$g \times s < \frac{5 \times l - 62}{57}$$

For the a Cray T3D machine $g = 1.7$ and $l = 148$ (values found in the BSPlib pages, http://www.bsp-worldwide.org) which gives $s < 6.99$. Thus for the example given in section 4.5 the first algorithm is better since we work on integers which are smaller than 7 words. In the next section we will propose a generic scan algorithm which could be instantiated differently with respects to the parameters of the machine and of the problem.

Another point is that the total exchange function `totex` (presented in section 2.3) returns a parallel vector of functions. To avoid the creation of an intermediate data-structure (a list for example) we use a reduction `funreduce` where the ordered collection to reduce is given by a function:

```
let rec funreduce n1 n2 op e f =
  if n1>n2
  then e
  else op (f n1) (funreduce (n1+1) n2 op e f)
```

Using `funreduce`, the `reducer` function is written as follows:

```
(* val reducer: ('a -> 'a -> 'a) -> 'a -> 'a -> 'a list par -> 'a par *)
let reducer op neutral e vl =
  let locally_reduced = parfun (List.fold_left op neutral) vl in
  parfun (funreduce 0 (bsp_p()-1) op e) (totex locally_reduced)
```

4.4 The `scanl` function

In order to implement the `scanl` function we will instantiate a generic scan operation. This generic operation could compute the sum of prefixes of any parallel vectors of some collection of indexed elements. The collections could be lists, arrays or any indexed collection of elements (for example unary functions from integers to something else).

`generic_scan` operates on a parallel vector of collections of indexed elements. It depends on:

- the data structure used for the collection of elements
- the sequential scan on this collection of elements
- the parallel prescan on vectors of this kind of elements (not parallel vector of collections)
- how to map on this kind of collection
- how to take and remove the last element of this kind of collection

```
(* val generic_scan:
  (('a -> 'b) -> 'c -> 'd -> 'e list) ->
  (('a -> 'b) -> 'c -> 'e par -> 'a par) ->
  ('b -> 'e list -> 'f) ->
  ('e list -> 'e option * 'e list) ->
  ('a -> 'b) -> 'c -> 'd par -> 'f par *)
```

```
let generic_scan sscan pprescan map cutlast op neutral vv =
  let local_scan = parfun (sscan op neutral) vv in
  let last l = Some (List.hd(List.rev l)),l in
  let tmp = applyat (bsp_p()-1) last cutlast local_scan in
  let last_elements = parfun (compose noSome fst) tmp
  and new_lists = parfun snd tmp in
  let values_to_add = pprescan op neutral last_elements in
    parfun2 map (parfun op values_to_add) new_lists
```

If `vv` is (to ease the presentation we consider that the collections are lists):

$$\langle\, [\mathtt{x}_1;\mathtt{x}_2;\ldots;\mathtt{x}_{\mathtt{n/p}}]\,,\ldots,\, [\mathtt{x}_{\mathtt{n+1-n/p}};\mathtt{x}_{\mathtt{n+2-n/p}};\ldots;\mathtt{x}_{\mathtt{n}}]\,\rangle$$

then `local_scan` is the following vector:

$$\langle\, [\mathtt{c};\mathtt{op2}\ \mathtt{c}\ \mathtt{x}_1;\ldots;\mathtt{op2}\ (\ldots)\ \mathtt{x}_{\mathtt{n/p}}]\,,\ldots,\, [\mathtt{c};\mathtt{op2}\ \mathtt{c}\ \mathtt{x}_{\mathtt{n+1-n/p}};\ldots;\mathtt{op2}\ (\ldots)\ \mathtt{x}_{\mathtt{n/p}}]\,\rangle$$

Each process now holds $\frac{n}{p}+1$ values. For each process (except the last one) we need to remove the last value of the collection. The `tmp` vector is a parallel vector of pairs. The first component is (`Some` v_i) where v_i was the last element of the list and the second component is the list without this last element. At process $p-1$ the second component is the unchanged list. From this vector we obtain `last_elements` the parallel vector composed with the last elements and `new_lists` the parallel vector of lists without their last elements. `values_to_add` computes the partial reductions of the `last_elements` vector. At process i, only the first $i-1$ values will be reduced: this operation is called `pprescan` (for parallel prescan). To end the values obtained are added to the `new_lists` parallel vector of lists.

In order to have a `scanl` function we need now to provide a sequential scan, a parallel prescan, a sequential map and a function to remove the last element of the collection. We chose here to use lists. The sequential scan is given in appendix A. The parallel prescan follows. It proceeds in two super-steps. Like `reducer` other choices are possible and we do not need to rewriting the full `scanl` function, we could only provide another parallel prescan. The sequential map is the usual Objective Caml `List.map` function and the last required function is provided by the `cutlast` function which has already been discussed:

```
(* val pprescan_direct: ('a -> 'a -> 'a) -> 'a -> 'a par -> 'a par *)

let pprescan_direct op neutral vv =
  let tosend = mkpar(fun pid v dst->
    if pid<dst then Some v  else if dst=pid then Some neutral else None) in
  let sent = put(apply tosend vv) in
  let local_reduce = mkpar(fun pid->funreduce 0 pid op neutral) in
  apply local_reduce (parfun (compose noSome) sent)
```

```
(* val scanl: ('a -> 'a -> 'a) -> 'a -> 'a list par -> 'a list par *)
let scanl op e =
  generic_scan sscan pprescan_direct List.map cutlast op e
```

4.5 Example

The smaller elements example explained in section 3.3 can be simply implemented using the `diff` higher-order function. To transform the result which is a parallel vector into a usual Objective Caml value we can use the `unsafe_proj` function (not available in BSMLLIB v 0.2) which is defined by:

$$\texttt{unsafe\_proj}\,\langle\, v\,,\ldots,\, v\,,\ldots,\, v\,\rangle \;=\; v$$

No check is done: it means that if the parallel vector does not contain the same value everywhere then the behavior is unspecified. The `safe_proj` function checks if the same value is everywhere but it needs communications. In this case we are sure that there is no such problem. One may avoid the nesting of parallel vectors when using these projection functions.

```
(* va se: int list par -> int list *)
let se xs =
  let k x c = if x<c then [] else [x]
  and g1 x = []
  and g2 x = x in
  unsafe_proj(diff (@) [] max k g1 g2 xs min_int)
```

In this program, `(@)` is the concatenation of lists, `max` is the maximum function and `min_int` is the smallest integer value. All these expressions are predefined in Objective Caml.

We can new try the program. For the sake of simplicity, assume we have a 4 processors machine. The input for `se` can be defined as the following function:

```
let f = function
    0 -> [4;5;6;7]
  | 1 -> [3;10;8;11]
  | 2 -> [4;5;6;7]
  | 3 -> []
```

Of course for a real use of `se` we could for example read the data from a file on process 0 and then cut the list and distribute it among the processors.

The application of `se` will then give (in the BSMLLIB top-level):

```
let _ = se (mkpar f)
-: int list = [4; 5; 6; 7; 10; 11]
```

Acknowledgments

This works was done during a short visit by the first author at the Information Processing Laboratory, Department of Mathematical Informatics / Department of Mathematical Engineering and Information Physics, The University of Tokyo. It was supported by the PRESTO project, Japan Science and Technology Corporation. Frederic Loulergue wishes to thank Pr. Zhenjiang Hu for this invitation and the fruitful discussions with him and other members of the IPL, Kazuhiko Kakehi and Yasushi Hayashi.

References

[1] The BSMLLIB library version 0.2. http://bsmllib.free.fr.

[2] S. Adachi, H. Iwasaki, and Z. Hu. Diff: A Powerfull Parallel Skeleton. In *The 2000 International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, volume 4, pages 425–527. CSREA Press, 2000.

[3] 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.

[4] R.S. Bird. An introduction to the theory of lists. In M. Broy, editor, *Logic of Programming and Calculi of Discrete Design*, pages 3–42. Springer-Verlag, 1987.

[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] E. Chailloux, P. Manoury, and B. Pagano. *D 騅 eloppement d'applications avec Objective Caml*. O'Reilly France, 2000. freely available in english at http://caml.inria.fr/oreilly-book/index.html.

[8] F. Dehne. Special issue on coarse-grained parallel algorithms. *Algorithmica*, 14:173–421, 1999.

[9] 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.

[10] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.

[11] F. Gava. Une biblioth 鑷 ue certifi 馥 de programmes fonctionnels BSP. In M 駭 issier-Morain, V., editor, *Journ* **馥** *s Francophones des Langages Applicatif, JFLA*. INRIA, january 2004. to appear.

[12] F. Gava and F. Loulergue. A Polymorphic Type System for Bulk Synchronous Parallel ML. In V. Malyshkin, editor, *Seventh International Conference on Parallel Computing Technologies (PaCT 2003)*, number 2763 in LNCS, pages 215–229. Springer Verlag, 2003.

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

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

[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 F. Loulergue. Functional Bulk Synchronous Parallel Programming using the BSMLlib Library. In S. Gorlatch and C. Lengauer, editors, *Constructive Methods for Parallel Programming*, Advances in Computation: Theory and Practice, pages 165–178. Nova Science Publishers, august 2002.

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

[18] 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.

[19] Z. Hu, M. Takeichi, and H. Iwasaki. Diffusion: Calculating Efficient Parallel Programs. In *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation (PEPM'99)*, pages 85–94. ACM Press, January 22-23 1999.

[20] Xavier Leroy. The Objective Caml System 3.07, 2003. web pages at www.ocaml.org.

[21] 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.

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

[23] F. Loulergue. Implementation of a Functional Bulk Synchronous Parallel Programming Library. In 14*th IASTED International Conference on Parallel and Distributed Computing Systems*, pages 452–457. ACTA Press, 2002.

[24] F. Loulergue. A Calculus of Functional BSP Programs with Explicit Substitution. In G. Joubert, W. Nagel, F. Peters, and W. Walter, editors, *Parallel Computing: Software Technology, Algorithms, Architectures and Applications, Proceeding of the 10th ParCo Conference*, Dresden, 2003. North Holland/Elsevier. to appear.

[25] 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.

[26] F. Loulergue. Parallel Superposition for Bulk Synchronous Parallel ML. In Peter M. A. Sloot and al., editors, *International Conference on Computational Science (ICCS 2003), Part III*, number 2659 in LNCS. Springer Verlag, june 2003.

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

[28] W. F. McColl. Scalability, portability and predictability: The BSP approach to parallel programming. *Future Generation Computer Systems*, 12:265–272, 1996.

[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] D. Rémy. Using, Understanding, and Unravellling the OCaml Language. In G. Barthe, P. Dyjber, L. Pinto, and J. Saraiva, editors, *Applied Semantics*, number 2395 in LNCS, pages 413–536. Springer, 2002.

[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. *Foundations of Parallel Programming*. Number 6 in International Series on Parallel Computation. Cambridge University Press, 1994.

[33] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.

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

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

A Sequential Functions Used in the Programs

```
(* val noSome: 'a option -> 'a *)
let noSome (Some x) = x

(* val compose: ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b *)
let compose f g x = f(g x)

(* val cutlast: 'a list -> 'a option * 'a list *)
let rec cutlast = function
    [] -> None,[]
  | [h] -> Some h,[]
  | h::t -> let last,beginning = cutlast t in last,h::beginning

(* In our case we will use the sscan' which is undefined for
   empty lists *)
(* sscan' op [x_1;...;x_n] = [x_1;op x_1 x_2;...;op (...) x_n] *)
let sscan' op (h::t) =
  let rec sscan_aux op acc = function
      [] -> []
    | h::t-> let newacc = op acc h in newacc::(sscan_aux op newacc t) in
  h::(sscan_aux op h t)

(* Usual sequential scan with neutral *)
let sscan op neutral l = sscan' op (neutral::l)
```

B Installation of the BSMLLIB Library

First download the lastest version of the library at `http://bsmllib.free.fr`. Uncompress and untar it: `tar zvf bsmllib-0.2.tar.gz`. Then in the `bsmllib-0.2` directory, edit the file `Makefile`. The beginning of this file should look like:

```
# MODIFY THOSE VARIABLES TO MATCH YOUR INSTALLATION
export CC=mpicc
export CLIBS=
export MPIINCDIR=/usr/include
export MPILIBDIR=/usr/lib

# MODIFY THOSES VARIABLES TO CHANGE THE INSTALLATION DIRECTORIES
OCAML_LIB_INSTALL=$(HOME)/bsmllib
SCRIPT_INSTALL=$(HOME)/bin
```

You should change the values of the variables to fit your installation. `CC`, `CLIBS`, `MPIINCDIR` and `MPILIBDIR` are used to indicate where your MPI tools and libraries can be found. You can compile the sequential version (which offers the same possibilities that the parallel version) without any MPI library installed. You just need the Objective Caml language [20].

The BSMLLIB distribution contains of courses the libraries but also some scripts to ease the compilation of programs using the BSMLLIB library and a toplevel based on the sequential version of the library. The Objective Caml libraries are installed in the `OCAML_LIB_INSTALL` directory and the scripts and toplevel are installed in the `SCRIPT_INSTALL` directory.

Then the compilation and installation is done by `make install`.

To try the toplevel, be sure that the `SCRIPT_INSTALL` directory is in your path and create a ~/**.bsmllibrc** file which should contain lines of the form: p,g,l for example $4,1.,11.5$. It means that if you run the sequential version of the library it will behave as if your were running your programs on a 4 processors machine with $g = 1$ and $l = 11.5$.

Then call it with: `bsmllib`. The following lines will appear:

```
        Bsmllib version 0.2
        Objective Caml version 3.07

#
```

Now you can write your programs as in the Objective Caml toplevel. For example (do not write the "#" symbol, it is the toplevel prompt):

```
# open Bsmllib;;
# open Bsmlbase;;
# let my_first_parallel_vector = mkpar(fun i -> 2*i+1);;
val my_first_parallel_vector: int par = <abstr>
# parprint print_int my_first_parallel_vector;;
Process 0: 1
Process 1: 3
Process 2: 5
Process 3: 7
-: unit par = <abstr>
#
```

To compile programs, use the `bsmllibc` script (see the reference manual for more informations).