Functional Parallel Programming with Revised Bulk Synchronous Parallel ML

Wadoud Bousdira*, Frédéric Gava†, Louis Gesbert‡, Frédéric Loulergue§, and Guillaume Petiot¶
*LIFO, Université d'Orléans, France, Wadoud.Bousdira@univ-orleans.fr
†LACL, Université Paris-Est Créteil, France, Frederic.Gava@univ-paris-est.fr
‡MLstate, Paris, France, Louis.Gesbert@mlstate.com
§LIFO, Université d'Orléans, France, Frederic.Loulergue@univ-orleans.fr
¶LIFO, Université d'Orléans, France, Guillaume.Petiot@etu.univ-orleans.fr

***Abstract*— Bulk Synchronous Parallel ML or BSML is a high-level language for programming parallel algorithms. Built upon the Objective Caml language, it provides a safe setting for implementing Bulk Synchronous Parallel (BSP) algorithms. It avoids concurrency related problems: deadlocks and non-determinism. BSML is based on a very small core of parallel primitives that extended functional sequential programming to functional BSP programming with a parallel data structure and operations to manipulate it. However, in practice the primitives for writing the parallel non-communicating parts of the program are not so easy to use. Thus we designed a new syntax that makes programs easier to write and read. Revised BSML is presented and its expressiveness and performance are illustrated through an application example.**

I. Introduction

In the context of "Think Parallel or Perish", parallel code would be the norm. But many programmers are not able to manipulate low-level routines [**?**] without introducing bugs as deadlocks and non-determinism. Furthermore, low-level programming forbids optimisations that could be done with a more structured parallelism. Collective operations and skeletons [**?**] offer a global view of the application and exhibit a more structured parallelism. However such high-level programming is still rare. One of the reasons is that they often do not provide a sufficiently wide set of patterns for a practical and efficient programming. That makes the design of new, robust and general parallel programming languages an important area of research. Creating such a language involves a tradeoff between expressiveness, by offering the programmer the freedom to write all the parallel details of algorithms, and structure necessary to build more easily correct programs with predictable performances.

Bulk Synchronous Parallel ML or BSML [**?**], is an extension of Objective Caml [**?**] to code Bulk Synchronous Parallel (BSP) algorithms [**?**], [**?**]. It combines the high degree of abstraction of ML and good performance with the scalable and predictable performances of BSP. In the BSP model, programs are written as a sequence of steps, called *super-steps*, each alternates a phase of computation, a phase of communication and a finishes with a global barrier. Communications are bulk and collective. That simplify parallel programs because programmer is not responsible for managing low-level communication details (how data are packaged, routed and received by other processors). Within a super-step, the work is done in parallel but the global structure of the algorithm is sequential. This simple structure has proven to be worth in practice for many parallel applications ([**?**] contains many references).

BSML is based on a structured model of parallelism, but is universal for this model: Any BSP algorithm could be written using BSML. Other structured parallelism approaches such as algorithmic skeletons first define a set of parallel patterns, the skeletons, and the model of parallelism is then derived from this set. When the set is not fit to a particular algorithm, it may be extended: This requires of course of lot of work for the library implementor and does not help the user of the library who has to choose among a bigger set of skeletons.

On the contrary, BSML offers a very small set of parallel primitives over a parallel data structure, called parallel vector: four functions to create and manipulate this structure as well as four constants to access the BSP parameters of the underlying architecture. By comparison, the standard BSPlib [**?**] library for BSP programming in C offers about fifteen primitives, and MPI more than a hundred.

However one of the primitive used to describe communications in BSML is not very simple to use with the current non-communicating primitives. Moreover the non-nesting requirement (explained below) of parallel vectors introduces additional constraints that makes a lot of construction/deconstruction of parallel vectors necessary. Novice BSML programmers offer find these two aspects difficult to deal with.

We have thus make the choice to design a revised syntax for BSML. Even if this syntax is just syntax and do not modify the principles of BSML programming, it makes BSML programs simpler to read and write. In practice, we believe simple syntax may be as important as simple semantics.

This paper describes our new syntax and illustrate it with application examples. First, we describe the classic and revised BSML syntax and informal semantics in section **??**. Section **??** is devoted to the implementation of bigger examples together with some timings. Section **??** presents related work. Future work and conclusion are discussed in Section **??**.

Some basic knowledge of any ML programming language is assumed. We refer to [**?**, Manual, chapter 1] for an Objective Caml tutorial.

II. Revised Bulk Synchronous Parallel ML

BSML is currently implemented as a library for the Objective Caml language [**?**]. The version used for this paper can be downloaded [**?**]. Before presenting the classic and revised syntax and informal semantics of BSML, lets us present the bulk synchronous parallel model on which BSML is based.

The bulk synchronous parallel model [**?**], [**?**] offers an abstract model of parallel architecture, a model of parallel program execution together with a performance model. A BSP computer is a homogeneous distributed memory machine with global synchronisation unit. Any general purpose parallel architecture can be seen as a BSP computer. A BSP program is executed as a sequence of *super-steps*. A super-step is composed of three successive and logically disjointed phases. In the local computation phase, each processor used its local data to perform sequential computations. Each processor may only *requests* data transfers to or/and from other processors. In the communication phase the network delivers the requested data transfers. A synchronisation barrier involving all the processors of the BSP computer is the third phase and ends the super-step. It is only at the end of this third phase that the transferred data becomes available for the local computation phase of the next super-step.

The performance of a BSP computer is characterised by 3 parameters. The parameter p is the number of processor-memory pairs. The communication and synchronisation performances are characterised by the parameter L that is the time required for a global synchronisation, and the parameter g that is the time for collectively delivering a 1-relation (communication phase where every processor receives or/and sends at most one word). The network can deliver a h-relation in time $g \times h$ for any natural h. In practice the BSP parameters can be determined using benchmarks (usually a fourth parameter r, the computing power of the processors, is first determined).

For a super-step, if at processor i, w_i is the local sequential work performed during the computation phase, h_i^+ is the size of data sent from i to other processors, and h_i^- the size of the received data by processor i from other processors, then the execution time (or *cost*) of the super-step is:

$$\max_{0 \leq i < \text{bsp\_p}} w_i + \max_{0 \leq i < \text{bsp\_p}} \max(h_i^+, h_i^-) \times g + L$$

The cost of a BSP program is the sum of the costs of its super-steps.

BSML is based on a data-type called parallel vector which, among all Objective Caml types, enables parallelism. A parallel vector has type 'a par and embeds p values of any type 'a at each of the p different processors in the parallel machine. The nesting of parallel vectors is not allowed. As BSML is currently implemented as a library, nested evaluation of an expression of type (t par) par for a given type t would lead to an unspecified behaviour. A type system could ensure that no such expression exist in the program [**?**]. However the current implementation of BSML as a library does not offer this specific type system. Only an implementation of BSML as a full language could offer such a feature. It is thus currently the responsibility of the programmer to avoid nesting of parallel vectors.

The number p of processors is fixed throughout the execution of the program. It can be accessed in BSML using the integer constant **bsp_p**. The other BSP parameters are also accessible as float values through constants **bsp_g**, **bsp_l** and **bsp_r**.

BSML comes, as Objective Caml, with three modes of compilation/evaluation:

- a byte-code compiler (a set of scripts calling the Objective Caml byte-code compiler with the appropriate BSML modules, in a similar way `mpicc` is not a full compiler; there are several scripts as there are several available implementations of BSML depending on the underlying low-level communication used: TCP, MPI or sequential),
- a native code compiler (also a set of scripts),
- an interactive loop also called top-level.

This interactive loop is a sequential simulator for BSML. Thanks to BSML semantical properties it is ensured that the parallel and the sequential implementations behave the same [**?**].

On starting, the BSML top-level is as follows:

```
Bulk Synchronous Parallel ML version 0.5
  The BSP Machine has 4 processors
  o BSP parameters g = 20.3 flops/word
  o BSP parameters L = 4571. flops
  o BSP parameters r = 498952227. flops/s
#
```

The values of the BSP parameters are measured values of a quad-core i7 machine. The # symbol is the prompt that invites the user to enter an expression to be evaluated. The top-level then gives an answer of the form: Name of the defined value (possibly none, written "-"), type and pretty-printing of the value. In case the value cannot be pretty-printed (for example functions), an abstract representation is given (for example <**fun**>).

The p processors are labelled with integers from 0 to $p-1$ which we call *pid* (Processor Identifier) of the processors. We distinguish this structure from an usual sequential vector or array of size p because the different values, that will be called *local*, are blind from each other: It is only possible to access the local value x_i in two cases: Locally, on processor i (by the use of a specific primitive), or after some communications.

These restrictions are inherent to distributed memory parallelism. This makes parallelism explicit and programs more readable. Since a BSML program deals with a whole parallel machine and individual processors at the same time, a distinction between the levels of execution that take place will be needed:

- **Replicated** execution is the default. Code that does not involve BSML primitives (nor, as a consequence, parallel vectors) is run by the parallel machine as it would be by a single processor. Replicated code is executed at the same time by every processor, and leads to the same result everywhere.

- **Local** execution is what happens inside parallel vectors, on each of their components: The processor uses its local data to do computation that may be different from the other's.
- **Global** execution concerns the set of all processors together, but as a whole and not as a single processor. Typical example is the use of communication primitives.

The implementation of the classic BSML library is based on the primitives of Figure **??** where the following denotes a parallel vector: $\langle x_0, x_1, \ldots, x_{p-1} \rangle$: 'a par. This vector holds the value x_i at processor i.

mkpar builds a parallel vector. The components of the obtained parallel vector are the results of the application of its argument function to the pid of every processor. Examples evaluated in the top-level follow (still using the quad-code machine):

```
# open Bsml;;
# let this = mkpar(fun pid → pid);;
val this : int Bsml.par = <0, 1, 2, 3>
# let replicate x = mkpar (fun _ → x);;
val replicate : 'a → 'a Bsml.par = <fun>
# let vec1 = replicate "PDAA";;
− : string Bsml.par = <"PDAA", "PDAA", "PDAA", "PDAA">
```

It is first necessary to open the Bsml module which contains the BSML primitives or write Bsml.**mkpar**.

apply is another primitive for local computation. It applies a parallel vector of functions to a parallel vector of values: At each processor, it applies a local function to a local argument. For example, if one wants to apply a same function at each component of a parallel vector:

```
# let parfun f v = apply (replicate f) v;;
val parfun : ('a → 'b) → 'a Bsml.par → 'b Bsml.par = <fun>
# parfun String.length vec1;;
− : int Bsml.par = <4, 4, 4, 4>
```

proj is the dual of **mkpar**, and the only way to extract a non-parallel value from a parallel vector. Given a parallel vector, it returns a function such that, applied to the pid of a processor, it returns the value of the vector at this processor. **proj** is often used at the end of a parallel computation to gather the computed results. For example, if we want to convert a parallel vector into a list, we write:

```
# let list_of_par vec = List.map (proj vec) procs;;
val list_of_par : 'a Bsml.par → 'a list = <fun>
# list_of_par this;;
− : int list = [0; 1; 2; 3]
```

where procs is the list of pids [0; 1; ... ; **bsp_p**−1]. **proj** should not be evaluated in the context of a **mkpar**. For example, if vec' as type t par for a given type t, the following code is incorrect: **let** vec = **mkpar**(**fun** i → **proj** vec' 0) but should be written: **let** vec = **let** x = **proj** vec' 0 **in mkpar**(**fun** i → x).

This kind of errors can also be detected by our type system but currently the programmer is responsible of avoiding them. The evaluation of an application of **proj** requires communication and synchronisation.

put is the comprehensive communication primitive: It allows any local value to be transferred to any other processor. It is synchronous, and ends the current super-step. Canonical use of **put** is **put** (**mkpar** (**fun** sender sendto → e)) where expression e computes (or usually, selects) the data that should be sent depending on sender to sendto. The return value of **put** is another vector of functions: At a processor j the function, when applied to i, yields the value *received from* processor i by processor j. Some values, as the empty list or the first constructor without parameters in a sum type, are considered to mean "no message". A total exchange function could be written as follows:

```
# let total_exchange vec =
    parfun
      (fun f→ List.map f procs)
      (put(apply (replicate(fun v dst → v)) vec));;
val total_exchange : 'a Bsml.par → 'a list Bsml.par = <fun>
# total_exchange this;;
− : int list Bsml.par = <[0;1;2;3], [0;1;2;3], [0;1;2;3], [0;1;2;3]>
```

A simpler version of this function is given below in the new syntax and commented.

Having a very small core of parallel operations is a great strength for the formalisation of the language. It makes the definitions clear and short. However, the program, even if high-level, still has to deal with replicated values and parallel vectors, and the use of the primitives can sometimes become awkward. Indeed, every operation inside of parallel vectors has to call a primitive and define an "ad hoc" function. This gets worse when working with multiple vectors, with nested calls to **apply**. Simply transforming a pair of vectors into a vector of pairs is written:

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

This could be made simpler with the definition of

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

We get then:

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

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

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

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

Instead of a point of view based on primitives, we can consider the execution levels such that one can declare code that will be executed globally as in standard Objective Caml and code that will be executed locally, inside a parallel vector. Then, to access to local data in a local section, we need no more to define additional functions because opening vectors now can be done locally. A local section is represented by $\ll$ e$\gg$, as a parallel vector. Replicated information is available inside the vector, as with the **mkpar** above. To access local information, we add the syntax $\$x\$$ to open the vector x and get the local value it contains; $ $ can obviously be used only within local sections. It is now possible to write combine_vectors as follows:

Primitive	Type	Description
mkpar	(int → 'a) → 'a par	$f \mapsto \langle f\,0\ ,\ \ldots\ ,\ f\,(p-1)\rangle$
apply	('a → 'b) par → 'a par → 'b par	$\langle f_0,\ldots,f_{p-1}\rangle \mapsto \langle v_0,\ldots,v_{p-1}\rangle \mapsto \langle f_0\,v_0,\ldots,f_{p-1}\,v_{p-1}\rangle$
proj	'a par → int → 'a	$\langle v_0,\ldots,v_{p-1}\rangle \mapsto$ (**fun** $i \to v_i$)
put	(int → 'a) par → (int → 'a) par	$\langle f_0,\ldots,f_{p-1}\rangle \mapsto \langle$ **fun** $i \to f_i 0,\ldots,$ **fun** $i \to f_i(p-1)\rangle$

Fig. 1. Classic BSML Parallel Primitives

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

Fig. 2. Summary of Revised BSML Syntax

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

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

The total_exchange example could be rewritten:

```
let total_exchange vec =
  let msg = put ≪ fun dst → $vec$≫ in
  ≪ List.map $msg$ procs ≫
```

It is much clearer now that *at each processor* the function given as argument to **put** is a function which returns the *local* value of parallel vector vec for every destination processor meaning that this local value will be sent to all processors. Using the received values is also clearer: At each processor the local function which encodes the messages received by the processor will be applied to every processor identifier thus yielding the list of all received messages.

Figure **??** gives a summary of the revised BSML syntax.

III. Applications and Experiments

A. Heat Equation

We now illustrate how BSML could be used to implement a scientific application. The heat equation describes the variation in temperature over time in a given material. It can be used to simulate the evolution of the distribution of heat in a material. We will consider here the one dimensional heat equation:

$$\frac{\delta u}{\delta t} - \gamma \frac{\delta^2 u}{\delta^2 x} = 0$$

where $u(x,t)$ gives the temperature in position x at time t and γ is the thermal diffusivity of the material.

Using a discretization method on time and space using steps dx and dt we obtain the following equation:

$$u(x,t+dt) = \frac{\gamma dt}{dx^2}\big(u(x+dx,t)+u(x-dx,t)-2u(x,t)\big)+u(x,t)$$

There exist parallel implementations of 1D heat equation using algorithmic skeletons, for example using the SkeTo library for C++ [**?**].

In sequential, using only functions List.map and List.map2 and with the following functions:

```
# let rec remove_last l = match l with
    | [x] → [] | x::xs → x::(remove_last xs);;
val remove_last : 'a list → 'a list = <fun>
# let shift_left a l = (List.tl l) @ [a];;
val shift_left : 'a → 'a list → 'a list = <fun>
# let rec shift_right a l = a::remove_last l;;
val shift_right : 'a → 'a list → 'a list = <fun>
```

the function computing a step of the heat diffusion simulation from a given state u (represented here as a list of floats) could be written:

```
# let heat gamma dx dt l_bound r_bound u =
  let u_plus_dx = L.shift_right l_bound u
  and u_minus_dx = L.shift_left r_bound u in
  let u1 = List.map2 ( +. ) u_plus_dx u_minus_dx in
  let u2 = List.map2 (fun v1 v2→ v1 −. 2.∗.v2) u1 u in
  let u3 = List.map (fun v → gamma∗.dt/.(dx∗.dx)∗.v) u2 in
    List.map2 ( +. ) u3 u
val heat : float → float → float → float → float →
           float list → float list = <fun>
```

l_bound and r_bound are the bounding conditions, i.e. the temperature at both ends of the material (outside the material). These are constant float values. This program is quite similar to SkeTo heat equation example. Skeleton libraries could not easily be extended by the users to add new skeletons. On the contrary it is easy in Objective Caml to define the map3 function and to rewrite the heat function which becomes both easier to read and a bit more efficient:

```
# let rec map3 f l1 l2 l3 = match (l1,l2,l3) with
  | [], [], [] → []
  | x1::xs1, x2::xs2, x3::xs3→
      (f x1 x2 x3)::(map3 f xs1 xs2 xs3);;
val map3 : ('a → 'b → 'c → 'd) →
           'a list → 'b list → 'c list → 'd list = <fun>
# let heat2 gamma dx dt l_bound r_bound u =
  let u_plus_dx = shift_right l_bound u
  and u_minus_dx = shift_left r_bound u in
   map3 (fun updx umdx u →
           gamma∗.dt/.(dx∗.dx)∗.(updx+.umdx−.2.∗.u) +. u)
     u_plus_dx u_minus_dx u;;
val heat2: float → float → float → float → float →
           float list → float list = <fun>
```

B. Parallel Heat Equation

Now if we want to develop a parallel version of this code, we can distribute u which would have type float list par. In

this case, at each processor and at each step of the simulation we would apply the sequential version of the heat function on the local part of u. Of course to be able to do so for the first and last elements of the local list u, one should have the values of the last element of the local part of u held by the left neighbour processor and the first element of the local part of u held by the right neighbour processor. With a last function defined as:

```
# let last l = List.hd(List.rev l);;
val last : 'a list → 'a = <fun>
```

we could implement the function that get the last values of the left neighbours as:

```
# let get_l_bounds l_bound u =
  let msg = put ≪ fun dst→
   if dst=($this$+1) && $this$<>(bsp_p−1)
   then [last $u$] else [] ≫ in
  ≪ if $this$=0
     then l_bound
     else List.hd ($msg$ ($this$−1)) ≫ ;;
val get_l_bounds: 'a → 'a list Bsml.par → 'a Bsml.par = <fun>
```

For the vector of function needed to the **put** primitive, the case of the last processor is specific as it should not send the last value of the local part of u it holds to its right neighbour. The case of the first processor is also specific when we retrieve the sent values: No left processor sent it a value thus it should use the l_bound boundary value. An example of use follows where Tools.from_to n1 n2 builds an integer list from n1 to n2:

```
# let vec = ≪ Tools.from_to (2∗($this$)) (2∗($this$)+1)≫ ;;
val vec : int list Bsml.par = <[0; 1], [2; 3], [4; 5], [6; 7]>
# get_l_bounds 10 vec;;
− : int Bsml.par = <10, 1, 3, 5>
```

The get_r_bounds could be written in a similar way (and has the same type) using a first function equals to List.hd. With these functions it is now possible to write a parallel version of heat:

```
# let par_heat gamma dx dt l_bound r_bound u =
  let l_bounds = get_l_bounds l_bound u
  and r_bounds = get_r_bounds r_bound u in
   ≪ heat2 gamma dx dt $l_bounds$ $r_bounds$ $u$ ≫ ;;
val par_heat : float → float → float → float → float →
               float list Bsml.par → float list Bsml.par = <fun>
```

get_l_bounds and get_r_bounds both need one super-step to be evaluated. It is possible to mix them together to use only one super-step and rewrite a parallel version of heat accordingly (same type as heat):

```
let par_heat2 gamma dx dt l_bound r_bound u =
 let l_bounds, r_bounds = get_bounds l_bound r_bound u in
  ≪ heat2 gamma dx dt $l_bounds$ $r_bounds$ $u$ ≫
```

C. Parallel Heat Equation with Arrays

If we use the following functions:

```
# let shift_right l_bound u = let len = Array.length u in
  Array.init len (fun i→ if i=0 then l_bound else u.(i−1));;
val shift_right : 'a → 'a array → 'a array = <fun>
# let shift_left r_bound u = let len = Array.length u in
  Array.init len (fun i→ if i=len−1 then r_bound else u.(i+1))
val shift_left : 'a → 'a array → 'a array = <fun>
# let map3 f a1 a2 a3 = let len = Array.length a1 in
  Array.init len (fun i→ f (a1.(i))(a2.(i))(a3.(i)));;
val map3 : ('a → 'b → 'c → 'd) →
           'a array → 'b array → 'c array → 'd array = <fun>
```

instead of the previous ones, the code of heat2 renamed heat3 would operate on arrays instead of lists. With the following definitions of last and first:

```
# let first a = a.(0);;
val first : 'a array → 'a = <fun>
# let last a = a.((Array.length a)−1);;
val last : 'a array → 'a = <fun>
```

the code of get_bounds would operate on a parallel vector of arrays instead of a parallel vector of lists. Then the following function operates on a parallel vector of arrays:

```
# let par_heat3 gamma dx dt l_bound r_bound u =
  let l_bounds, r_bounds = get_bounds l_bound r_bound u in
   ≪ heat3 gamma dx dt $l_bounds$ $r_bounds$ $u$ ≫ ;;
val par_heat3 : float → float → float → float → float →
                float array Bsml.par→ float array Bsml.par=<fun>
```

From the definition of map3 we see that we could write a more efficient version of heat3 that does not requires the creation of intermediate arrays u_plus_dx and u_minus_dx:

```
# let heat4 gamma dx dt l_bound r_bound u =
  let len = Array.length u in
   Array.init len (fun i→
    let updx = if i=(len−1) then r_bound else u.(i+1)
    and umdx = if i=0 then l_bound else u.(i−1) in
     gamma∗.dt/.(dx∗.dx)∗.(updx+.umdx−.2.∗.u.(i)) +. u.(i));;
val heat4 : float → float → float → float → float →
            float array → float array = <fun>
```

Using this new heat4 leads to a new par_heat4. It is even possible to change the pure functional style of arrays to an imperative style. For this we need two arrays:

```
# let heat5 gamma dx dt l_bound r_bound u u' =
  let len = Array.length u in
   for i=0 to len−1 do
    u'.(i) ←
     let updx=if i=(len−1) then r_bound else u.(i+1)
     and umdx=if i=0 then l_bound else u.(i−1) in
      gamma∗.dt/.(dx∗.dx)∗.(updx+.umdx−.2.∗.u.(i)) +. u.(i);
   done;;
val heat5 : float → float → float → float → float →
            float array → float array → unit = <fun>
```

and the parallel version becomes:

```
# let par_heat5 gamma dx dt l_bound r_bound u u' =
  let l_bounds, r_bounds = get_bounds l_bound r_bound u in
   ≪ heat5 gamma dx dt $l_bounds$ $r_bounds$ $u$ $u'$ ≫
val par_heat5 : float → float → float → float → float →
                float array Bsml.par → float array Bsml.par →
                Bsml.par unit = <fun>
```

When we iterate, we switch the two arrays at each step.

Experiments were performed on a cluster of 8 nodes connected by a giga-bit Ethernet network. Each node contains two processors (Quad-Core AMD Opteron Processor 2376, 2.29 GHz, 16Gb of RAM) and runs Linux Ubuntu with kernel

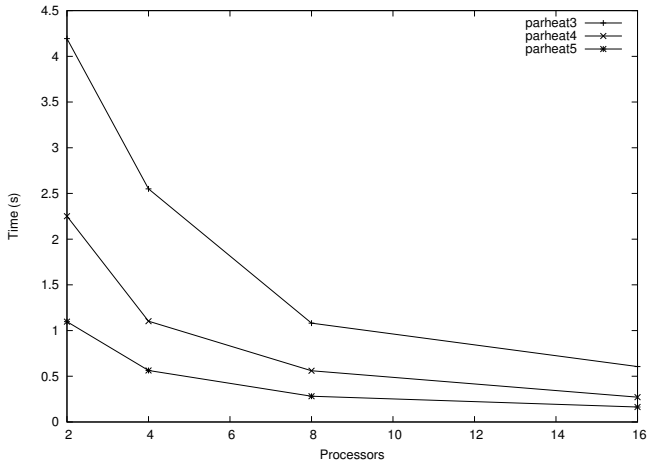

Fig. 3. Parallel Heat Diffusion Simulation (array versions)

2.6.24-24. The BSML programs were compiled in native code with the MPI version of the communication module (OpenMPI 1.4.2 with gcc 4.3.2) and Objective Caml 3.11.1.

Arrays versions are much more efficient. The version par_heat2 with lists, on 16 processors for a list of size 10^6 and 10 iterations, is about 17 times slower than the par_heat3 version. Figure **??** compares the versions with arrays on a global array of size 10^7 with 100 iterations.

IV. Related Work

As certain readers could point out, the notation $\ll$ e$\gg$ introduced in this paper and which built a parallel vector, is close to constructions like "e1 par e2" in parallel Haskell [**?**] (if it is used $p - 1$ times) or like creation of processes in Eden [**?**]. However the meaning of these constructions differ. Fine-grained parallelism introduced by GPH's par takes two arguments that are to be evaluated in parallel. The expression "e1 par e2" has the same value as "e2". Its dynamic behaviour is to indicate that "e1" could be evaluated by a new parallel thread, with the parent thread continuing evaluation of "e2". Threads are then distributed on the processors at run-time. Communications are implicit by the share of variables.

In our case, parallelism is explicit (as well as the communications and the distribution of data) and especially it is prohibited to nested parallelism to optimise performances of the implementation [**?**]. BSML is clearly a lower level programming language compared to algorithmic skeletons for example but it comes with a realistic cost model and is well adapt to the writing coarse-grain algorithms. Moreover, it can be used to implement higher-order parallel functions that could be used as algorithmic skeletons.

V. Conclusion and Future Work

Parallel architectures are taking the lead in computer hardware. There are many research in advanced programming paradigms to find the best ways to accommodate parallelism. We present in this paper a new syntax for our high-level BSP language: BSML.

This new syntax reduces size of the code. Code is also simpler to be read which ease debugging and reasoning about performances. Our new syntax could be simulated by our past BSML primitives making past codes compatible and allowing the use of existing proofs developments about BSML in Coq [**?**].

Future work includes the development of applications with BSML, the integration of an exception handling mechanism (that already exists [**?**]) in the public release. In a longer term, we plan to work on an implementation of BSML as a full language rather than a library. We are also working on proving the correctness of the implementation of the Revised BSML syntax.