DOI 10.12694/scpe.v18i3.1306
Scalable Computing: Practice and Experience ISSN 1895-1767
Volume 18, Number 3, pp. 261–274. http://www.scpe.org
© 2017 SCPE

A BSPLIB-STYLE API FOR BULK SYNCHRONOUS PARALLEL ML

FRÉDÉRIC LOULERGUE*

Abstract. Bulk synchronous parallelism (BSP) offers an abstract and simple model of parallelism yet allows to take realistically into account the communication costs of parallel algorithms. BSP has been used in many application domains. BSPlib and its variants are programming libraries for the C language that support the BSP style.

Bulk Synchronous Parallel ML (BSML) is a library for BSP programming with the functional language OCaml. It offers parallel operations on a data structure named parallel vector. BSML provides a global view of programs, i.e. BSML programs can be seen as sequential programs working on a parallel data structure (seq of par) while a BSPlib program is written in the SPMD style and understood as a parallel composition of communicating sequential programs (par of seq). The communication styles of BSML and BSPlib are also quite different.

The contribution of this paper is a BSPlib-style communication API implemented on top of BSML. It has been designed without extending BSML, but only using the imperative features of the underlying functional language OCaml. Programs implemented using this API are syntactically very close to programs implemented using a BSPlib library for the C language. It therefore shows that BSML is universal for the BSP model.

Key words: bulk synchronous parallelism, parallel programming, functional programming

AMS subject classifications. 68N18, 68M19, 68W10

1. Introduction. Introduced by Valiant [28] and McColl [20], bulk synchronous parallelism (BSP) is a model of parallelism that offers a high level of abstraction yet takes realistically into account communications and synchronization. Works on BSP algorithms are numerous and BSP has been used successfully for a broad variety of applications: scientific computation [2], artificial intelligence [5, 6, 25], parallel databases, *etc.* It has also inspired more recent frameworks such as Pregel [19], and open source frameworks such as Giraph[1].

The BSP model considers a BSP computer to be a distributed memory machine: p processor-memory pairs connected though a network allowing point-to-point communications, and a global synchronization unit. However this architecture should a thought as an abstraction one: The BSP model targets all general purpose parallel architectures. Such architectures can always be seen as BSP computers. For example shared memory machines as well as clusters of PC can be seen as BSP computers even if some of their components are implemented as software instead of being implemented as hardware.

A BSP program is a *sequence* of *super-steps*. The parallelism therefore resides in each super-step. A super-step is composed of three phases: A computation phase where each processor computes using only the data it holds in its local memory; a communication phase where processors request data from other processors; and finally a synchronization phase which ends the super-step. The data requests are guaranteed to have been fulfilled when the synchronization phase ends. This structured form of parallel programs is the basis for the BSP *cost model*. In this context, cost means the estimate of parallel execution time. For the sake of conciseness, we omit the description of the cost model.

BSP algorithms can of course be implemented using a general message passing programming library such as MPI. However it is possible to optimize communications by taking into account the specific structure of BSP programs. Therefore, soon after the BSP model was introduced, two main libraries to support BSP programming were developed: the Green BSP library [9], and the Oxford BSP tool set [22]. To promote the adoption of the BSP model, the developers of these libraries produced a standard proposal: The BSPlib library [11]. Several implementations of this standard exist, some proposing extensions to the standard [1, 3, 27, 29, 30]. All these libraries are for the C imperative programming language (or more recently C++), and follow an imperative programming style.

Bulk Synchronous Parallel ML or BSML [4, 16] is also a programming library that supports the BSP model, but for the functional programming language OCaml[2]. The style of this library is purely functional and there

*School of Informatics, Computing and Cyber Systems, Northern Arizona University, Flagstaff, USA, frederic.loulergue@nau.edu

[1] http://giraph.apache.org

[2] http://www.ocaml.org

even exists a formal semantics as an extension of the λ-calculus [17].

It has been often claimed that BSML is universal for BSP programming, meaning that any BSP algorithm can be implemented using BSML. However most of the published BSP algorithms are written in an imperative style and with imperative data structures. Even if OCaml supports the imperative, functional and object-oriented programming styles, communications and synchronizations styles are very different between BSPlib and BSML. It is thus interesting to explore that claim and check whether imperative BSP algorithms could be implemented using BSPlib-style programming in BSML.

The contribution of this paper demonstrates that indeed any imperative BSP algorithm can be implemented with BSML using only the BSML pure functional primitives and some imperative features of its host language OCaml.

The paper is organized as follows. In Section 2, we present BSP programming using BSPlib and using BSML and we emphasize the differences between these two styles of programming. Section 3 is devoted to a BSPlib-like programming API for BSML, from the point of view of the user of the API. Some examples are implemented with this API and compared with their BSPlib counterparts. We present the API from the point of view of the implementer in Section 4. This API is compared with a BSPlib implementation from a performance point of view in Section 5. Related work is discussed in Section 6. We conclude and give future research directions in Section 7.

This is an extended version of [15]: the current paper offers improved presentations of programming with BSPlib and BSML, of the API, as well as additional examples. Sections 4, 5, and 6 are completely new.

2. Bulk Synchronous Parallel Programming.

2.1. Programming with BSPlib. BSPlib is a standard proposal for C libraries supporting the BSP model. Some of the implementations of BSPlib provide extensions to the standard. We focus on the standard, and more precisely on the Direct Remote Memory Access (DRMA) communication primitives. The Message Passing sub-part of the standard could be treated in a similar way than the DRMA part. Moreover, they are more BSP algorithms implemented using the DRMA sub-part of the standard.

The BSPlib standard features initialization and finalization phases of a BSPlib program:

```
void bsp_begin(int maxprocs);
void bsp_end();
```

However these phases are implicit in BSML, so we do not describe them more here.

A BSPlib program follows the Single Program Multiple Data (SPMD) paradigm. BSPlib programmers write one program parametrized by the process identifier, and the parallel program is the (implicit) parallel composition of several instantiations of this parametrized communicating sequential program.

To access the processor identifier and the total number of processors, BSPlib provides respectively the following two functions:

```
int bsp_pid();
int bsp_nprocs();
```

Before being able to read/write from the memory of another processor, processors should indicate that some memory locations will be able to be accessed by distant processors. This is done using the following functions:

```
void bsp_push_reg(const void *ident, int size);
void bsp_pop_reg (const void *ident);
```

When a processor calls one of these two functions, all the processors should perform a call to the same function. The registration and un-registration are active only *after* the super-step where the call is done. It is invalid to try to un-register an unregistered address.

The current super-step ends when all the processors call the function:

```
void bsp_sync();
```

There are two main functions to read from/write to remote memory:

```
  void bsp_put(int pid, const void *src,
               void *dst, int offset, int nbytes);
  void bsp_get(int pid, const void *src,
               int offset, void *dst, int nbytes);
```

`bsp_put(pid, src, dst, offset, nbytes)` requests the writing of `nbytes` bytes of data from memory address `src`, of the process executing the function, to memory address `dst` (plus `offset` bytes displacement from `dst`) of processor `pid`. The destination address should have been previously registered.

`bsp_get(pid, src, offset, dst, nbytes)` requests the reading of `nbytes` bytes of data from address `src` (plus `offset`) of processor `pid` and writing into the address `dst` of the processor executing the call. The source address should have been previously registered.

The data is guaranteed to have been exchanged at the end of the super-step, i.e. after a call to `bsp_sync`.

```
void shift(int * value)
{
  int dst = (bsp_pid()+1)%bsp_nprocs();
  bsp_push_reg(value, sizeof(int));
  bsp_sync();
  bsp_put(dst , value, value, 0, sizeof(int));
  bsp_pop_reg(value);
  bsp_sync();
}

void scatter(int root, int * a, int * v)
{
  bsp_push_reg(v, sizeof(int));
  bsp_sync();
  if(bsp_pid()==root){
    for(int dst=0;dst<bsp_nprocs();dst++)
      bsp_put(dst,&a[dst],v,0,sizeof(int));
  }
  bsp_pop_reg(v);
  bsp_sync();
}

void gather(int root, int * value, int * array)
{
  bsp_push_reg(array,bsp_nprocs()*sizeof(int));
  bsp_sync();
  bsp_put(root, value, array, bsp_pid()*sizeof(int), sizeof(int));
  bsp_pop_reg(array);
  bsp_sync();
}
```

FIG. 2.1. *BSPlib for C Examples*

Figure 2.1 presents several examples of functions written using BSPlib. These are collective communication functions: all the processors are supposed to call the same function with coherent parameters. For example the `root` argument should be the same for all processor identifiers.

`shift` shifts "to the right" the content of an `int` variable on each processor. In a non SPMD view, `value` could be seen as a parallel array (of size `bsp_nprocs()`) of integers. `shifts` shifts circularly this array to the right.

`scatter` scatters the content of an array `a` of size `bsp_nprocs()` at processor `root` to an `int` variable `v` on each processor. This function must be called with the same value of `root` on all the processors and at processor `root` array `a` should contain at least `bsp_nprocs() int` values.

`gather` is the dual of `scatter`: It gathers all of each `int` value on each processor in an array in the memory of the `root` processor. `root` should have the same value on all the processors, while `array` only needs to be allocated and have size at least `bsp_nprocs()` on processor `root` only.

2.2. Programming with BSML. The remaining of the paper assumes some familiarity with a statically typed higher-order functional programming language such as Haskell, Standard ML or OCaml. A concise introduction to OCaml is [23].

Bulk Synchronous Parallel ML or BSML is an explicit parallel functional language, extension of the ML family of functional languages. Currently there exists an implementation as a library for the OCaml language, implemented on top of MPI. Compared to the implementation of a full BSML *language*, the current implementation does not provide a type checker specific to BSML code (even if such a type checker has been designed [8]). In essence the BSML programming style is purely functional. It is nevertheless possible to use the imperative features of OCaml in combination to BSML. However in this case, some care is required: some unsafe programs could be written (an implementation of the type checker previously mentioned would reject such programs).

BSML offers four constants to access the parameters of the underlying BSP architecture: **bsp_p** returns the number of processors in the parallel machine. The other parameters are omitted in this paper as they are related to the BSP cost model.

BSML parallelism is based on a parallel data structure named *parallel vector*. In a parallel vector, each processor contains one value of the vector. Using this data structure, BSML offers a global view of parallel programs, i.e. a program looks like a sequential program but operates on parallel data structures. It is very different from the SPMD paradigm: The global parallel structure of SPMD programs is much harder to understand than programs that offer a global view. Parallel vectors are the only way to obtain different values at different processors. All code outside parallel vectors is *replicated* among the processors and assured (if the program is purely functional) to be consistent everywhere. This view of replication is however the view of the BSML implementer: for the developer of BSML programs, code outside parallel vectors is just usual sequential code.

Parallel vectors have a polymorphic type: 'a par, meaning a parallel vector where each processor contains a value of type 'a. OCaml is a higher-order functional language: 'a could be replaced by any type, including a type of function. There is one restriction: Nesting is forbidden so 'a could not be a parallel type. The type system [8] also rejects programs with such nesting.

It is important to note that, like other high-level abstractions such as Powerlists [24], there is no index notation for parallel vectors. In other words, there is no direct access to individual values in parallel vectors: Vectors are handled globally through four BSML primitive functions.

The function **mkpar**: (int→ 'a) → 'a par builds a parallel vector from a function f. At processor i the vector will have value (f i). For example with 8 processors[3]:

```
# let r = mkpar (fun i->2*i);;
val r : int par = <0, 2, 4, 6, 8, 10, 12, 14>

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

where # is the prompt of the interactive loop, and the answer has the form *name : type = value* and, in the interactive loop, parallel vectors are written $\langle\, a_0 \,, \ldots ,\, a_{p-1} \,\rangle$. %% is a modulo operator that always returns a positive number.

[3] We show here the evaluation of BSML expressions inside the BSML interactive loop.

OCaml is a higher-order language, functions are first class citizens. It is therefore possible to build a parallel vector of functions:

```
# let vf = mkpar (fun i -> fun x -> x + i);;
val vf : (int->int) par = < <fun>, ..., <fun> >
```

vf is a parallel vector where each processor i contains the function **fun** x → x + i. In the toplevel, the value representing a function is symbolized by an abstract value `<fun>`.

A parallel vector of functions is not a function: It cannot be applied to another parallel vector. A BSML primitive is needed to apply point-wisely a parallel vector of functions to a parallel vector of values. For example:

```
# let vv1 = apply vf r;;
val vv1 : int Bsml.par = <0, 3, 6, 9, 12, 15, 18, 21>
```

The type of **apply** is ('a→ 'b) par → 'a par → 'b par.

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

The function **proj**: 'a par → (int→ 'a) is the almost the dual of **mkpar**: from a parallel vector it creates a (replicated) function. However for a function f, **proj**(**mkpar** f) is different from f: f may be defined for a negative input, or for an input greater or equal to **bsp_p**, but **proj**(**mkpar** f) is only defined on the interval $[0, \mathbf{bsp_p} - 1]$.

```
# let fr = proj r;;
val fr: int -> int = <fun>

# let four = fr 2;;
val four : int = 4
```

proj requires communications and a synchronization barrier to be evaluated. For more involved communication patterns, one needs to use the **put** function. It allows any local value to be transferred to any other processor. As **proj**, it ends the current super-step by a synchronization barrier.

The type if put is (int→ 'a)par→ (int→ 'a)par. A canonical use of **put** is **put** (**mkpar** (**fun** src dst → e)) where expression e computes (or usually, selects) the data that should be sent (depending on the processor identifier src) to processor dst.

Both the input and the result of the evaluation of a call to **put** are vectors of functions. In the input, each function encodes the messages to be sent by one processor to all the other processors, while in the output each function encodes the messages received by one processor from all the other processors. At a processor j the function obtained by a call to **put**, when applied to i, yields the value *received from* processor i by processor j.

For example, shifting the values of a parallel vector to the right could be written:

```
# let shift vv =
   let msg src v dst = if dst=(src+1) mod bsp_p then [v] else [] in
   let msgs = apply (mkpar msg) vv in
   let srcs = mkpar (fun i->(i-1)%%bsp_p) in
   parfun List.hd (apply (put msgs) srcs)
val shift : 'a par -> 'a par = <fun>

# let vv2 = shift (mkpar string_of_int);;
val vv2 : string par = <"7", "0", "1", "2", "3", "4", "5", "6">
```

where **let** parfun f v = **apply** (**mkpar**(**fun** i→ f)) v and List.hd returns the head of a non empty list.

In this example, the empty list [] is used as a way to mean "no message". The empty list is not sent to other processors. For any sum type the first constant constructor is considered to mean "no message". Even if **put** looks like a total exchange, it is not due to these special values.

The shift function proceeds as follows. msgs is a parallel vector of functions. At each processor src, it contains a function that returns the local value of vector vv (encapsulated in a singleton list) if applied to a process identifier equals to (src+1) **mod bsp_p** (the processor "to the right" of src, the processor to the right of

```
let shift (value: int ref) =
  begin
    bsp_push_reg value (ref int);
    bsp_sync();
    let dst = (bsp_pid()+1) mod bsp_nprocs() in
    bsp_put dst !value value int;
    bsp_pop_reg value (ref int);
    bsp_sync();
  end

let gather (root: int) (value: int) (a:int array) =
  begin
    bsp_push_reg a (array int);
    bsp_sync();
    bsp_put_sa root value a (bsp_pid()) int;
    bsp_pop_reg a (array int);
    bsp_sync()
  end

let scatter (root: int) (a: int array) (value: int ref) =
  begin
    bsp_push_reg value (ref int);
    bsp_sync();
    if (bsp_pid() = root) then
      begin
        for dst=0 to bsp_nprocs() − 1 do
          bsp_put dst a.(dst) value int;
        done
      end;
    bsp_pop_reg value (ref int);
    bsp_sync()
  end
```

Fig. 3.1. *BSPlib for BSML Examples*

the last processor being the first processor) and returns the empty list otherwise. Thus each of the functions of msgs encodes the messages to be sent by one processor to other processors, and some of the values mean "no message". (**put** msgs) is also a vector of functions. Now each of these functions encodes the messages received from other processors. We are only interested in messages coming from the processor immediately "to the left". The vector srcs contains at each processor the processor identifier of the processor to its left. Applying the vector of functions (**put** msg) to srcs therefore yields at each processor the message received from its left neighbour. As the value has been encapsulated into a singleton list, we need to apply, at each processor, the function List.hd.

3. An API for Imperative BSP Programming in BSML. In this section we present how to program with the proposed BSPlib API for OCaml implemented on top of BSML. The examples of BSPlib for C presented in Figure 2.1 are translated to this new API for OCaml in Figure 3.1. The imperative BSP programming styles for both C and OCaml are very close. With BSML it is therefore possible to use both styles: the BSPlib-like imperative style and the original BSML pure functional style.

One main difference between OCaml and C is that OCaml is strongly typed. Therefore it is not possible to directly translate in OCaml the BSPlib for C primitives where all data is considered as arrays of bytes.

Moreover, by default all variables are immutable in OCaml, but variables which types are *references* or *arrays* (and some others omitted here).

Furthermore programming in BSML does not follow the SPMD paradigm but the global view paradigm. However the proposed API does follow the SPMD paradigm. We explain in Section 4 how we dealt with this problem.

Finally, the implementation of a BSPlib-like API in OCaml requires to store references in a table. This is not easy as a reference to an int has a different type than a reference to a float and that data structures in OCaml are usually parametric polymorphic, meaning a structure should contain values of the same type.

The solution to this problem is to use generalized abstract data types (GADT) [7] to have values representing types so that a value v and a value v_τ representing the type τ of the value v can be packed together into a value v_{packed} whose type does not depend on the type τ. We will discuss this technical point further in Section 4.

Memory visibility. The impact on the API is that instead of having the size of an address we want to register, we have an expression that represents the type of the value we want to register (or un-register):

```
bsp_push_reg: 'a → 'a Type.ty → unit
bsp_pop_reg: 'a → 'a Type.ty → unit
```

These two functions are polymorphic, but if they are called on values that are not mutable, an error occurs (unit is the type of the value () usually returned by functions that only produce side effects).

For example, **let** x = **ref** 0 **in** bsp_push_reg x int will register the reference x (to an int) for DRMA communications. **ref** in OCaml is applied to a value v and creates a reference to a value of the type of v, with initial value v. In the example, the content referenced by x is therefore mutable. It is mandatory to specify the type also for bsp_pop_reg while the C version only needs the address of the memory location to un-register.

Synchronization barrier. As in BSPlib for C, a call to bsp_sync() ends the super-step by a synchronization barrier. A call to bsp_pid() returns the processor identifier, and a call to bsp_nprocs() return the number of processors of the BSP machine.

DRMA communications. The type of function bsp_put is:

```
bsp_put: int → 'a → 'a ref → 'a Type.ty → unit
```

bsp_put pid v r ty writes the value v to (registered) reference r of processor pid. The type of value v should be represented by ty. With bsp_put, values can only be written as a whole, even if they are arrays. This is therefore less flexible that the corresponding BSPlib for C function where arrays can be considered partly using the offset and nbytes arguments.

To provide such a flexibility, and due to typing constraints, we propose two variants of bsp_put:

```
bsp_put_sa : int → 'a → 'a array → int → 'a Type.ty → unit
bsp_put_aa : int → 'a array → 'a array → int → int → 'a Type.ty → unit
```

bsp_put_sa is used to put a scalar value into a remote array: bsp_put_sa pid v a k ty writes value v whose type is represented by value ty at index k of the remote array a at processor pid. The type of a should be represented by the value array ty.

bsp_put_aa is used to deal with arrays both at the source and at the destination. bsp_put_aa pid a1 a2 offset size ty writes size values of array a1 whose *elements'* type is represented by value ty at offset offset of the remote array a2 at processor pid.

The imperative API also offers functions to read from remote memory:

```
bsp_get : int → 'a ref → 'a ref → 'a Type.ty → unit
```

and the associated _sa and _aa variants.

4. Implementation of the Imperative API. In the development of the imperative API, two difficulties had to be overcome: typing issues and offering a SPMD view for the API while BSML offers a global view. Before explaining how we dealt with these issues, let us describe briefly the data structures used to implement the API.

Data Structures. **bsp_put** and **bsp_get** can only target memory locations that have been previously registered. The first data structure is a table of registered memory locations, implemented as a mutable list of such memory locations (or references). As in OCaml, references to a value of type int has a different type than a reference to a value of type float, this list should contain values of the type Dyn.t described below.

In our API the message requests are actually exchanged when **bsp_sync** is called. Therefore we need a data structure to contain the message requests added to this structure by calls to **bsp_put** and **bsp_get**. This data structure is a mutable list of value of type request. This type has six constructors that correspond to the three versions of **bsp_put** and the three versions of **bsp_get** (The arguments to these six constructors are the arguments to these six functions but the destination reference which is replaced by the *position* of the reference in the registered memory location table).

Typing Issues. We use a recently introduced feature of OCaml: generalised algebraic data-types or GADT. GADT are extensions of sum types (or variant types). A sum type in OCaml is similar to a union type in C or a record type with variant parts in Ada. However there is no discriminant field. Instead each case is associated with a symbol, called constructor. Constructors are used to discriminate between values of the sum type. For example abstract syntax trees for a small language of integers and boolean expressions could be implemented as the following type:

```
type expr =
| Int of int
| Bool of bool
| Add of expr * expr
| And of expr * expr
```

It is of course possible to write meaningless (non well typed) expressions such as And(Int 0, Bool **true**). To reject such expressions it would be necessary to write a type checking function, for example:

```
type ty = | TBool | TInt
let rec type_of : expr → ty option = function
  | Int _ → Some TInt
  | Bool _ → Some TBool
  | Add(e1, e2) →
     if (type_of e1) = Some TInt && (type_of e2) = Some TInt
     then Some TInt else None
  | And(e1, e2) →
     if (type_of e1) = Some TBool && (type_of e2) = Some TBool
     then Some TInt else None
```

where the type 'a option is a sum type with too constructor: Some that takes as argument a value of type 'a and None.

type_of can therefore return a value of type ty embedded in a Some constructor if the input expression is well typed, and returns None if the expression is not well typed.

GADT introduce two novelties with respect to sum types: The possibility to have more constrained type parameters depending on the constructor, and the possibility to introduce existential type variables, i.e. using a type variable in a constructor case that is not one of the type parameters.

Using GADT it is possible to rewrite the previous example in such a way that it is not possible to write meaningless expressions, using the type system of OCaml:

```
type _ expr =
| Int: int → int expr
| Bool: bool → bool expr
| Add: int expr * int expr → int expr
| And: bool expr * bool expr → bool expr
```

Now the type expr has a type parameter (here written _ because we don't need it to have an explicit name) that allows to indicate the type of the expression. This allows to specify the type of expressions the constructors Add

and And take as argument. Trying to use And(Int 0, Bool **true**) would raise a compile time type error indicating that Int 0 has type int expr but that And expects a value of type bool expr.

It is possible to write a GADT such that values of this type *represent* the type of the GADT type parameter:

```
type _ ty =
| Int : int ty
(* ... *)
| Ref : 'a ty → 'a ref ty
| Array : 'a ty → 'a array ty
```

For example the value Array Int has type int array ty and represents the type int array. In order to ease the use of this GADT, we defined constants and functions to build values of type ty that are very close syntactically to the type they represents:

```
let int = Int
(* ... *)
let array a = Array a
```

For example the type int list array of arrays containing a list of integers in each cell is represented by the value array (list int). Note that function **ref** that builds a reference from a value is masked by our definition of **ref**. The former is now written P.**ref**.

To be able to store different communication requests into the same data structure, we need a way to store data in a type such that the type does not depend on the type of the stored data, but still allowing some type checking by OCaml, to avoid to write or read data of a wrong type to/from remote memory. To do so we use the second feature of GADT: existential types.

The module Dyn provides an abstract type t implemented as

```
type t = D: 'a ty * 'a → t
```

A value of type Dyn.t contains a value v of some type τ plus a value v_τ representing the type τ i.e. a value of type τ ty. Dyn also provides a function make: 'a ty → 'a → t to build values of type Dyn.t and update functions: update_ref, update_array and update_array_a. These functions take two values of type Dyn.t (and possibly an offset and a size) and update the first argument if it is a mutable value, with the value of the second argument, if their types are equivalent.

When all processors call bsp_push_reg r ty, the API stores make r ty of type Dyn.t in the table of registered memory locations. As all processors are supposed to call this function, there is a consistent indexing of visible memory locations.

When a processor calls bsp_put pid v r ty, the API stores make v ty of type Dyn.t in a table of communication requests together with the index of r in the table of registered memory locations. At a call to bsp_sync the requests are sent to the destination processors. As the data part of write/read requests are communicated as values of type Dyn.t, at destination the memory is updated using the update functions of module Dyn. When all the updates are done, the next super-step can begin.

Global view vs. SPMD. As 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 is needed. *Replicated* execution is the default. Code that does not involve BSML primitives 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 other processors. *Global* execution concerns the set of all processors together, but as a whole and not as a single processor, for example the use of communication primitives.

Consider the following program:

```
let x = ref 0 in
  let vv3 = mkpar(fun pid → x := pid) in
  if !x <> 0 then shift vv1 else vv1
```

The mutable variable x is created at the replicated level. Therefore its value would be the same on all the processors. However it is mutated differently on each processor at the local level inside **mkpar**(**fun** pid → x:=pid). The content of x is no longer replicated. This could cause a problem if the value of x is used again at the replicated or global level. In the example all the processors will call shift but processor 0. As synchronization barriers should be called by all the processors in BSP, this program fails.

Such an example is rejected by the type system [8]. However BSML current implementation is a library only: it does not come with a type checker. In the implementation of the API we use a replicate reference to store the process identifier and modify it locally so that the content of this reference is the processor identifier. That allows to implement bsp_pid and write programs in the SPMD style. Registered memory locations as well as the communication requests are also in replicated mutable variables but don't remain replicated due to the use of bsp_pid in the control flow. Note that the problem exposed above is still present, and it is actually quite easy to write incorrect BSPlib C programs from the point of view of synchronization.

However instead of using the type checker [8] it may be possible to relax the type system to allow the implementation of the imperative API to type check and add an adaptation of the static analysis proposed by Jakobsson et al. [12] to avoid synchronization errors.

5. Experiments. The current implementation of the proposed API is a proof of concept, and we favor simplicity and conciseness over performances. However, we experimented the performances of the BSML BSPlib API with respect to a BSPlib implementation in C, and a functional style BSML implementation. The experiments were conducted on a shared memory machine running Ubuntu Linux 16.04, with two Intel Xeon E5-2683 v4 processors (16 cores each) at 2.10 GHz and 256Gb of memory. The following libraries and compilers were used:

- BSPonMPI version 0.4.2,
- BSML version 0.5.4,
- OpenMPI version 1.5.4,
- OCaml version 4.04.0,
- GCC version 5.4.0 (with optimization flag -O3).

The test application is an inner product. The main function of the BSPlib version in C is shown in Figure 5.1 (taken from BSPEdupack [2]). p is the total number of processors, s is the local processor identifier, and n is the size of the vectors. The corresponding version using the proposed API for BSML is depicted in Figure 5.2. Basically only the local variable declarations are different. We also tested variant where bsp_sync() was replaced by Put.bsp_sync() a slightly optimized version that avoids to do a second "empty" synchronization barrier when no communication request is a bsp_get. Finally we tested a version mostly in the functional BSML style (Figure 5.3).

The experiments were done using respectively 2, 4, 8, 16 and 32 cores, and for two global size of vectors: 10^3 and 10^9. The results are presented in Figures 5.4 and 5.5: each figure indicates the median value of 100 measures as well as the standard deviation.

For size 10^3 the communication and synchronization costs are dominant and therefore the running time increases with the number of cores. BSML communication and synchronization phases are more expensive than BSPlib for C communication and synchronisation phases. The first reason is that the communication functions **proj** and **put** are polymorphic: serialization of the data to be exchanged is performed, and the resulting raw data is bigger than its non serialized counterpart. The second reason is that for **put**, the encapsulation of messages into functions makes necessary the application of each of these functions to all the process identifiers. OCaml sequential computations on arrays are also slightly less efficient than C sequential computations on arrays.

The BSPlib API for BSML is even less efficient: Indeed a call to **bsp_sync** in this case requires two underlying calls to **put**, but for the BSPlib+BSML Opt. version: its performance is closer to the functional BSML version. For the general bsp_sync() is because to implement **bsp_get** messages, a first call to put is needed to send a request for the data, and a second call to put is needed for the requested processors to send back the requested data. When a super-step contains only **bsp_put** messages, we optimized the implementation of the API to perform only one call to **put**. For size 10^9, all the versions based on BSML have very close performances.

The difference of performance for size 10^9 between the C version and the BSML versions is mostly due to the difference in sequential performance, and the serialization for data that both takes time and makes messages

```
double bspip(int p, int s, int n, double *x, double *y){
    Inprod= vecallocd(p); bsp_push_reg(Inprod,p*SZDBL);
    bsp_sync();

    inprod= 0.0;
    for (i=0; i<nloc(p,s,n); i++) inprod += x[i]*y[i];
    for (t=0; t<p; t++) bsp_put(t,&inprod,Inprod,s*SZDBL,SZDBL);
    bsp_sync();

    alpha= 0.0;
    for (t=0; t<p; t++) alpha += Inprod[t];
    bsp_pop_reg(Inprod); vecfreed(Inprod);

    return alpha;
}
```

FIG. 5.1. *Inner Product: BSPlib C Version*

```
let bspip (p:int) (s:int) (n:int) (x:float array) (y:float array) : float =
  let inprod_array = Array.make (bsp_nprocs()) 0.0 in
  let inprod = P.ref 0.0 and alpha = P.ref 0.0 in
  begin
    bsp_push_reg inprod_array (array float);
    bsp_sync ();

    for i=0 to (nloc p s n)−1 do inprod := !inprod +. x.(i) *. y.(i) done;
    for t=0 to p−1 do bsp_put_sa t !inprod inprod_array s float done;
    bsp_sync();

    for t=0 to p−1 do alpha := !alpha +. inprod_array.(t) done;
    bsp_pop_reg inprod_array (array float);

    !alpha
  end
```

FIG. 5.2. *Inner Product: BSPlib API for BSML Version*

bigger. The performance difference between the C version and the BSML functional version could be reduced in general, if BSML was extended with more specialized versions of **put** (and **proj**) so that the messages are not encapsulated into functions, and that for basic types, values are not serialized.

6. Related Work. Recent implementations of BSPlib for C include BSPonMPI [27], MultiCore BSP [30], and Zefiros BSP [29]. BSPonMPI targets distributed memory machines while MultiCore BSP and Zefiros BSP target shared memory machines. They are very close to the standard. There exists a BSPlib implementation for Java [10]. This implementation is closer to the C style libraries than a new design focusing on object orientation. The proposed API is therefore very close to all these libraries. The structured parallelism of BSP also allowed to design a "mock" BSPlib library for testing and debugging imperative BSP programs [26].

The Paderborn University BSPlib (PUB) [3] implements the standard but also adds subset synchronization to BSPlib (and the BSP model). Basically all the BSP primitives are given an additional first argument of type `tbsp`. This "BSP object" is very similar to an MPI communicator. It allows to create subgroups of processors, to ensure modularity, and to support different threads on the same processor running different BSP computations.

```
let ip (x: float array)(y:float array) =
  let inprod = P.ref 0.0 in
  for i=0 to (Array.length x)−1 do inprod := !inprod +. x.(i) ∗. y.(i) done;
  !inprod

let bspip (p:int) (s:int) (n:int) (x:float array par) (y:float array par) : float =
  begin
    let partial_ip = parfun2 ip x y in
    List.fold_left (+.) 0. (List.map (proj partial_ip) procs)
  end
```

FIG. 5.3. *Inner Product: BSML Version*

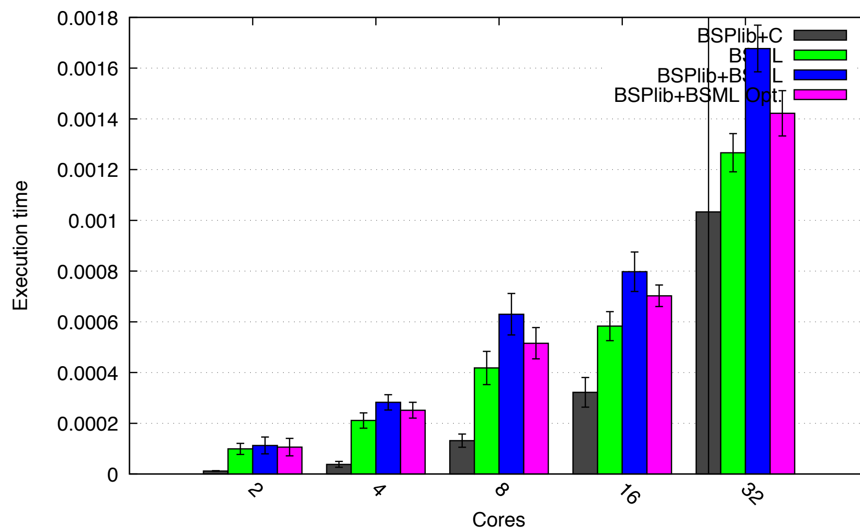

FIG. 5.4. *Experimental Results for Size* 10^3

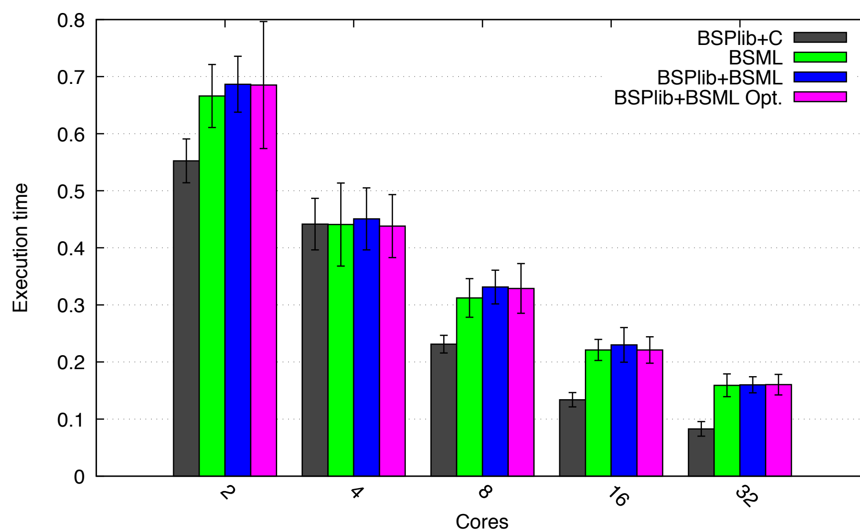

FIG. 5.5. *Experimental Results for Size* 10^9

Extensions of BSML feature two kinds of parallel compositions, juxtaposition [13] and superposition [14]. Juxtaposition supports subgroups of processors while still follows the pure BSP model: there is no subset synchronization. Superposition can be understood as a structured way to have several threads running different BSP computations. As it is structured, it is not as flexible as the PUB way of having multiple BSP threads. However this structure allows for safely *sharing* synchronization barriers between the threads rather than just ensuring that synchronization barriers do not interfere with each other. Therefore it would be possible to extend our API to handle some of the PUB extensions, subgroups and modularity, but with a different cost model.

There also exists a BSP programming library for the functional programming language Haskell [21]. The evaluation strategy of Haskell makes often more difficult to write BSP programs than BSML in OCaml. As Haskell is a pure functional language, a BSPlib-like API on top of [21] would be very difficult to design and not close to BSPlib C programming.

GADT are useful in the context of parallelism. We also used them for implementing recursive parallel data structures, namely Misra's powerlists [24], and associated operations in BSML while ensuring that no nesting of parallel vector occurs [18].

7. Conclusion and Future Work. In this paper we show that it is possible to implement a BSPlib-like imperative API using only Bulk Synchronous Parallel ML *purely functional* primitives together with the *sequential* imperative features of its host language OCaml. Bulk Synchronous Parallel ML is therefore universal for BSP programming.

This implementation relies on a recently introduced feature of OCaml: Generalized Algebraic Data Types (GADT). Experiments show reasonable performances for this proof-of-concept implementation of the API[4].

For full usability, the API should support also BSPlib message passing primitives. The type representation module we currently provide should also be extended in two ways: firstly it should support all the OCaml basic types, secondly it should be extensible in particular able to handle user-defined record types and sum types.

REFERENCES

[1] T. Bannink, A. Wits, and J.-W. Buurlage, *Epiphany BSP version 1.0.* http://github.com/coduin/epiphany-bsp, Jan. 2017.

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

[3] O. Bonorden, B. Judoiink, I. von Otte, and O. Rieping, *The Paderborn University BSP (PUB) library*, Parallel Computing, 29 (2003), pp. 187–207.

[4] W. Bousdira, F. Gava, L. Gesbert, F. Loulergue, and G. Petiot, *Functional Parallel Programming with Revised Bulk Synchronous Parallel ML*, in First International Conference on Networking and Computing (ICNC 2010), 2nd International Workshop on Parallel and Distributed Algorithms and Applications (PDAA), K. Nakano, ed., IEEE Computer Society, 2010, pp. 191–196.

[5] 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.

[6] 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.

[7] J. Garrigue and D. Rémy, *Ambivalent types for principal type inference with GADTs*, in Programming Languages and Systems, vol. 8301 of LNCS, Springer, 2013, pp. 257–272.

[8] F. Gava, L. Gesbert, and F. Loulergue, *Type System for a Safe Execution of Parallel Programs in BSML*, in 5th ACM SIGPLAN workshop on High-Level Parallel Programming and Applications, ACM, 2011, pp. 27–34.

[9] M. Goudreau, K. Lang, S. Rao, T. Suel, and T. Tsantilas, *Towards Efficiency and Portability: Programming with the BSP Model*, in Eighth Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA), New York, NY, USA, 1996, ACM, pp. 1–12.

[10] Y. Gu, B.-S. Lee, and W. Cai, *JBSP: A BSP programming library in Java*, Journal of Parallel and Distributed Computing, 61 (2001), pp. 1126–1142.

[11] J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling, *BSPlib: The BSP Programming Library*, Parallel Computing, 24 (1998), pp. 1947–1980.

[12] A. Jakobsson, F. Dabrowski, W. Bousdira, F. Loulergue, and G. Hains, *Replicated Synchronization for Imperative BSP Programs*, in International Conference on Computational Science (ICCS), Procedia Computer Science, Zurich, Switzerland, 2017, Elsevier, pp. 535–544.

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

[4]Available at `http://traclifo.univ-orleans.fr/BSML`

[14] ———, *Parallel Superposition for Bulk Synchronous Parallel ML*, in International Conference on Computational Science (ICCS), no. 2659 in LNCS, Springer Verlag, 2003, pp. 223–232.
[15] F. Loulergue, *Imperative BSPlib-style Communications in Bulk Synchronous Parallel ML*, in International Conference on Computational Science (ICCS), Procedia Computer Science, Zurich, Switzerland, 2017, Elsevier, pp. 2368–2372.
[16] F. Loulergue, F. Gava, and D. Billiet, *Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction*, in International Conference on Computational Science (ICCS), vol. 3515 of LNCS, Springer, 2005, pp. 1046–1054.
[17] F. Loulergue, G. Hains, and C. Foisy, *A Calculus of Functional BSP Programs*, Sci Comput Program, 37 (2000), pp. 253–277.
[18] F. Loulergue, V. Niculescu, and J. Tesson, *Implementing powerlists with Bulk Synchronous Parallel ML*, in Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC), Timisoara, Romania, 2014, IEEE, pp. 325–332.
[19] G. Malewicz, M. H. Austern, A. J. Bik, J. C. Dehnert, I. Horn, N. Leiser, and G. Czajkowski, *Pregel: a system for large-scale graph processing*, in SIGMOD, ACM, 2010, pp. 135–146.
[20] W. F. McColl, *Scalability, portability and predictability: The BSP approach to parallel programming*, Future Generation Computer Systems, 12 (1996), pp. 265–272.
[21] Q. Miller, *BSP in a Lazy Functional Context*, in Trends in Functional Programming, vol. 3, Intellect Books, may 2002.
[22] R. Miller and J. Reed, *The Oxford BSP Library Users' Guide*, Oxford Parallel, Oxford University Computing Laboratory, Wolfson Building, Parks Road, Oxford, OX1 3QD, 1.0 ed., May 1994.
[23] Y. Minsky, *OCaml for the masses*, Commun. ACM, 54 (2011), pp. 53–58.
[24] J. Misra, *Powerlist: A structure for parallel recursion*, ACM Trans Program Lang Syst, 16 (1994), pp. 1737–1767.
[25] R. O. Rogers and D. B. Skillicorn, *Using the BSP cost model to optimise parallel neural network training*, Future Generation Computer Systems, 14 (1998), pp. 409–424.
[26] W. Suijlen, *Mock BSPlib for Testing and Debugging Bulk Synchronous Parallel Software*, Parallel Processing Letters, 27 (2017), pp. 1–18.
[27] W. Suijlen and P. Krusche, *BSPonMPI*. https://github.com/pkrusche/bsponmpi, Apr. 2013. version 0.4.2.
[28] L. G. Valiant, *A bridging model for parallel computation*, Commun. ACM, 33 (1990), p. 103.
[29] M. van Duijn, K. Visscher, and P. Visscher, *BSPLib: a fast, and easy to use C++ implementation of the Bulk Synchronous Parallel (BSP) threading model*. http://bsplib.eu/, Sept. 2016. version 1.1.4.
[30] A. Yzelman, R. Bisseling, D. Roose, and K. Meerbergen, *MulticoreBSP for C: A High-Performance Library for Shared-Memory Parallel Programming*, International Journal of Parallel Programming, (2013), pp. 1–24.

Edited by: Dana Petcu
Received: Jul 7, 2017
Accepted: Sep 5, 2017