A MODULAR IMPLEMENTATION OF DATA STRUCTURES IN BULK-SYNCHRONOUS PARALLEL ML

FRÉDÉRIC GAVA*
Laboratory of Algorithms, Complexity and Logic, P2
University of Paris 12; 61, avenue du Général de Gaulle
94010 Créteil cedex, France, gava@univ-paris12.fr

ABSTRACT

A functional data-parallel language called BSML has been designed for programming Bulk-Synchronous Parallel algorithms. Many sequential algorithms do not have parallel counterparts and many non-computer science researchers do not want to deal with parallel programming. In sequential programming environments, common data structures are often provided through reusable libraries to simplify the development of applications. A parallel representation of such data structures is thus a solution for writing parallel programs without suffering from disadvantages of all the features of a parallel language. In this paper we describe a modular implementation in BSML of some data structures and show how those data types can address the needs of many potential users of parallel machines who have so far been deterred by the complexity of parallelizing code.

Keywords: BSP; Parallel Functional language; Parallel Data Structures.

1. Introduction

Bulk-Synchronous Parallel ML or BSML is an extension of ML to code *Bulk Synchronous Parallel* algorithms [18] as functional programs. Such algorithms offer scalable performances and BSML expresses them with a small set of primitives implemented as a parallel library (`http://bsmllib.free.fr/`) for the functional programming language Objective Caml (`OCaml`).

The complexity of parallel computing hardware calls for software that eases the development of programming models for researchers who do not have a deep knowledge of parallel computation and who do not have the time to write efficient code themselves. It has been known for a long time that well designed data structures are as important as algorithms. Most scientific and engineering applications are physical simulations in which at least one of the keys is the representation of the physical domain using data structures. Many of these computational applications involve solving problems with large data sets where using parallelism can reduce the computation time and increase the available memory size.

In order to have parallel programs, researchers often use automatic parallelizing compilers such as High Performance Fortran. But these languages do not provide powerful features such as polymorphism or algebraic data types needed for symbolic applications. Furthermore cost prediction is impossible. Researchers can also use the paradigm of algorithmic skeleton languages [5] in order to write algorithms as combinations of skeletons. But the set of needed skeletons often depends on the domain of application and it is often hard to increase this set. Moreover the execution cost cannot easily be determined from the source code and some combinations imply too many communications that decrease the performance.

*This work is supported by the ACI Young Researcher program under the ProPaC project.

Each application contains a small number of data structures that could be replaced when developing a parallel version. The main problem is to replace the primary data structures with their parallel versions and load balance these data structures without destroying locality properties (a too aggressive load balancing can lead to poor locality) or violating dependencies among the data that lead to incorrect semantics or unnecessary work.

This paper reports on several case studies of parallel implementations of data structures in BSML that make such scientific applications easier to develop using the high-level features of a functional language. These implementations employ the module system of `OCaml` so that one can freely specify a new parallel data structure from the sequential ones. The complexity of parallelism is hidden behind an interface that tries to stay as close as possible to the sequential one, thus making the code easy to learn. The parallel data structures are provided in source code form as independent pieces of software, resulting in comparatively straightforward maintenance. Another important part is formed by the computation of a histogram to perform load-balancing if needed — or at least, users specify when load-balancing is needed. In this paper, we also report on our experience with a scientific parallel application on a distributed memory multiprocessor machines.

This paper describes our first step in this direction. First, we briefly present in Section 2 the BSML language and some useful functions in Section 3. In Section 4 we describe parallel implementations of some classical data structures. Section 5 is devoted to some benchmarks. Related work is discussed in Section 6.

2. Functional Bulk-Synchronous Parallel ML

BSML does not rely on SPMD programming. Programs are identical to ordinary sequential `OCaml` programs but work on parallel data structures. Some of the advantages are simpler semantics and a better readability.

2.1. Asynchronous Primitives

The `BSMLlib` library is based on the elements given in Figure 1. It gives access to the BSP parameters of the underlying architecture. For example, **bsp_p**() is p, the *static* number of processes. There is an abstract polymorphic type α **par** which represents the type of p-wide parallel vectors of objects of type α one per processor. Those parallel vectors are created by **mkpar** so that (**mkpar** f) stores (f i) on process i for i between 0 and $p-1$: **mkpar** f = | (f 0) | (f 1) | $\cdots$ | (f i) | $\cdots$ | (f (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*, *i.e.* an ordinary ML expression. The expression (**mkpar** f) is a parallel object and it is said to be *global*. An ordinary ML expression that is not within a parallel vector is called *replicate*, *i.e.* identical on each processor. Asynchronous phases are programmed with **mkpar** and with **apply** such that (**apply** (**mkpar** f) (**mkpar** e)) stores ((f i)(e i)) on process i:

apply | $\cdots$ | f_i | $\cdots$ | | $\cdots$ | v_i | $\cdots$ | = | $\cdots$ | (f_i v_i) | $\cdots$ |

2.2. Synchronous Primitives

The **put** primitive expresses communication and synchronization phases. Con-

bsp_p: unit→int **bsp_l**: unit→float **bsp_g**: unit→float
mkpar: (int→α)→α **par** **apply**: (α →β)**par**→α **par**→β **par**
put: (int→α option)**par**→(int→α option)**par** **proj**: α option **par**→int→α option
type α option = None | Some **of** α **super**: (unit →α) →(unit →β) →$\alpha * \beta$

Figure 1: The core BSMLlib library

sider the expression: **put**(**mkpar**(**fun** i→$\mathsf{fs_i}$)) (∗). To send a value v (resp. no value) from process j to process i, the function $\mathsf{fs_j}$ at process j must be such that ($\mathsf{fs_j}$ i) evaluates to Some v (resp. None). The expression (∗) evaluates to a parallel vector containing functions $\mathsf{fd_i}$ of delivered messages. At process i, ($\mathsf{fd_i}$ j) evaluates to Some v (resp. None) if process j sent the value v (resp. no value) to the process i.

The **proj** primitive also expresses communication and synchronization phases. The expression (**proj** vec) returns a function f such that (f n) returns the nth value of the parallel vector vec. If this value is the empty value None then process n sends no message to the other processes. Otherwise this value is broadcast. Without this primitive, the global control cannot take into account data computed locally.

The nesting of **par** types is prohibited and the projection should not be evaluated inside the scope of a **mkpar**. Our type system enforces these restrictions [9].

2.3. Divide-and-conquer Primitive

The divide-and-conquer paradigm fits naturally into the BSP model, without any need for subset synchronization. The method is based on sequentially interleaved threads of BSP computations, called *super-threads* [20]: communication and synchronization phases are merged. The primitive **super** effects *parallel superposition*, which allows the evaluation of two BSML expressions in this way.

From the programmer's point of view, the functional semantics of the expression **super** $E_1\, E_2$ is the same as pairing but of course the evaluation is different from the evaluation of (E_1, E_2) [10]. The first phases of asynchronous computation of E_1 and E_2 are run ; then the communication phase of E_1 is merged with that of of E_2 (the messages are obtained by concatenation of the messages) ; only one barrier occurs and *bis repetita.* If the evaluation of E_1 needs more supersteps than that of E_2 then the evaluation of E_1 continues as if there were no superposition (and *vice versa*).

3. Examples of Useful Functions

The primitives described in the previous section constitute the core of the BSMLlib. In this section, we define some useful functions for BSP computing.

3.1. Often Used Asynchronous Functions

Asynchronous function replicate creates a parallel vector which contains the same value everywhere. The primitive **apply** can be used only for a parallel vector of functions that take one argument. To deal with functions that take two arguments we need to define the apply2 function.

```
(* replicate: α →α par and apply2: (α →β →γ )→α par→β par→γ par *)
let replicate x = mkpar(fun pid→x) and apply2 vf v1 v2 = apply (apply vf v1) v2
```

It is also common to apply the same sequential function at each process. We use the parfun functions where only the number of arguments to apply differs:

```
(* parfun: (α →β )→α par→β par and parfun2: (α →β →γ )→α par→β par→γ  par *)
let parfun f v = apply (replicate f) v and parfun2 f v1 v2 = apply (parfun f v1) v2
```

It is also common to apply a different function at a specific process. applyat n f1 f2 v applies function f1 at process n and function f2 at other processes:

```
(* applyat: int→(α →β )→(α →β )→α par→β par *)
let applyat n f1 f2 v = apply (mkpar (fun i→if i=n then f1 else f2)) v
```

3.2. Often Used Communication Functions

Our example is a replicated total exchange. Each processor contains a value (represented as a parallel vector of values) and the result of (rpl_total $\boxed{v_0 \mid \cdots \mid v_{p-1}}$) is $[v_0, \ldots, v_{p-1}]$ — a replicated list of these values on each processor:

```
(* rpl_total: α par→α list *)
let rpl_total vec =
 let rpl_totex vec = compose noSome (proj (parfun (fun v →Some v) vec)) in
  List.map (rpl_totex vec) (procs())
```

$$\text{where} \left\{ \begin{array}{rcl} \mathsf{compose\ f\ g\ x} & = & \mathsf{(f\ (g\ x))} \\ \mathsf{noSome\ (Some\ v)} & = & \mathsf{v} \\ \mathsf{List.map\ f}\ [v_0; \ldots; v_n] & = & [(\mathsf{f}\ v_0); \ldots; (\mathsf{f}\ v_n)] \\ \mathsf{procs()} & = & [0; 1; \ldots; \mathbf{bsp_p}()\text{-}1]. \end{array} \right.$$

We can then define some useful functions, such as parfun_total which allows to apply a function to all the components of a vector, then totally exchanges these values and, finally, applies another function to the previous result, e.g.:

parfun_total f1 f2 $\boxed{v_0 \mid \cdots \mid v_{p-1}} \Rightarrow$ (f2 $[v'_0; \ldots; v'_{p-1}]$) where (f1 v_i) $= v'_i$.

4. Parallel Data Structure

The standard library of `OCaml` provides many data structures. Each of them is defined by a module with a clear semantics of the operators and for some of them the complexity of the operations. Modules of the parallel data structures may have the same signatures as `OCaml` data structures together with some specific parallel operators to provide a better use of these parallel data structures. In this way, programmers may use these representations without suffering from disadvantages of a parallel language while still taking advantage of a parallel machine. Implementations of our parallel data structures need thus to have the same interfaces and the same high-level semantics as the sequential ones but with different costs. We give the BSP cost of these operators. Using a modulare implementatation, these costs are given independently of the complexity of the sequential data structures used. We write $\mathcal{C}_\mathbf{t}$ to denote the complexity of a function and $\mathcal{S}$ for the size of a datum.

Currently BSML implements five such modules: Set, Map, Hashtable, Queue and Stack [10]. Let us consider the parallel implementation of Map (also known as association tables or dictionaries) and Set in order to illustrate this section.

4.1. Parallel Dictionaries

The module of the parallel maps is a functor Make which builds a polymorphic parallel dictionary given : a module with the type for the keys and a total ordering function over them ; a hashing function ; a module that defines the load-balanced strategy (Section 4.3) ; and another functor to build local (sequential) maps:

```
module Make (Ord : OrderedType)(Bal:BALANCE)
             (MakeLocMap:functor(Ord:OrderedType) →Map.S with type key=Ord.t)
    : S with type key = Ord.t and type α seq_t = α MakeLocMap(Ord).t
```

with the following input signature of the functor:

```
module type OrderedType =
 sig
  type t
  val compare:t→t→int
  val hash:t→int
 end
```

```
module type S = sig
 type key
 type α t (* polymorphic parallel dictionary *)
 type α seq_t (* polymorphic sequential dictionary *)
 (* classical operators *)
 (* parallel operators *)
end
```

We define a parallel map as a pair comprising a parallel vector of sequential maps (one local map per processor) and a pair of a boolean and an integer. The boolean tells whether the parallel maps have been rebalanced or not. This integer will be used for load-balancing described in Section 4.3. We ignore it in this section and refer to Section 4.3 for more details about functions and mechanisms of load balancing. In the following we write m_i to denote the local map of the processor i.

A parallel dictionary thus puts its elements into one of its local (sequential) maps. The parallel data structure uses a generic sequential data structure (given as a module parameter) and are thus independent of the sequential implementation[†]

```
module Make (Ord : OrderedType)(Bal:BALANCE)
             (MakeLocMap:functor(Ord:OrderedType) →Map.S with type key=Ord.t) =
 struct
  (* LocMap is the sequential map data structure *)
  module LocMap = MakeLocMap(Ord)
  type key = Ord.t
  type α t = ((α  LocMap.t) Bsmllib.par) * (int * bool)
  type α seq_t = α LocMap.t
  (* operators on α t, the parallel dictionary *)
 end
```

Now we describe some simple and classical operators over the map structure. All operations over parallel maps and sets are purely applicative (no side-effects). At first, we need to create an empty parallel map:

```
(* empty: unit→t *)
let empty () = (replicate LocMap.empty,(0,true))
```

which fits the following sketch: $\boxed{\{\}}\boxed{\cdots}\boxed{\{\}}$.

The first important operator to add is a binding. Operation add k e m returns a map containing the same bindings as m, plus a binding of k to e. If k was already bound in m, its previous binding disappears. We choose a processor in which k is sequentially added (denoted $\mathsf{m}\oplus\{\mathsf{k}\rightarrow\mathsf{e}\}$). For this, we use the hashing function hash which is assumed to be deterministic, i.e. first of all it terminates, then returns the same pid when applied to the same value. If the parallel map has been rebalanced and thus elements are not in their original processors (defined by the hash function), we remove k from the other processors in order not to duplicate it:

[†] The OCaml sequential implementation uses balanced binary trees which is reasonably efficient. But we could also use a certified implementation as described in [7].

```
(* add: key→α →α t→α t *)
let add k e pmap = let (vect_map',(step',nbal')) = rebalance_if_needed pmap in
 ((applyat (hash k) (LocMap.add k e) (if nbal' then id
                                     else LocMap.remove k) vect_pmap'),(step'+1,nbal'))
```

which corresponds to the following sketch:

add k e | m_0 | ⋯ | m_i | ⋯ | m_{p-1} | ⇒

$m_0 \ominus \{k\}$	⋯	$m_i \oplus \{k \to e\}$	⋯	$m_{p-1} \ominus \{k\}$	If rebalanced map
m_0	⋯	$m_i \oplus \{k \to e\}$	⋯	m_{p-1}	Otherwise

The second operator is to search for the value bound to a key. The expression find k m returns the current binding of k in m, or raises an exception if no such binding exists. Searching a binding value in a parallel map is done asynchronously and the result is broadcast to every processor. The local searching (denoted $k \in_? m_i$) is done by only one processor if the map has not been rebalanced and by all the processors otherwise — in this case, only one processor really stored the binding:

```
(* find: key→α t→α  *)
let find k pmap = let (pmap',_,nbal') = rebalance_if_needed pmap in
 if nbal' then let root = (hash k) in
  let loc_find= (applyat root (fun t →try Some (LocMap.find k t) with Not_found →None)
                               (fun _ →None) pmap') in
  let res = rpl_bcast root loc_find in match res with None →raise Not_found | Some x →x
 else parfun_total(fun t→try Some(LocMap.find k t) with Not_found→None)choose_l pmap'
```

where choose_l:α option list→α selects the first (Some x) from a list or raises Not_found. This code corresponds to the following sketch:

find k | m_0 | ⋯ | m_{p-1} | ⇒

$k \in_? m_0$	⋯	$k \in_? m_{p-1}$	$\Rightarrow b_?$	If rebalanced map
⋯	$k \in_? m_i$	⋯	$\Rightarrow b_i$	Otherwise

where $\forall j\ b_j = k \in_? m_j$. The cost of this function is thus the maximum cost of the local search and of a direct broadcast of the result: $\max_{i=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(k \in_? m_i)) + (\mathbf{p}-1) \times \mathcal{S}(v_k) \times \mathbf{g} + \mathbf{l}$ where v_k is the size of the binding value.

Now we describe a new important iteration operator over parallel maps. This operator has been added for the following reason: the high-level semantics of the fold operator specifies that the elements of the map are computed in increasing order with respect to the ordering over the type of the keys. In this way, no parallel code could be used and the parallel map is first reduced to a sequential one which incurs a high communication cost. For example, one could compute the cardinality of a map (a function not present in the standard library) with ParMap.fold (**fun** _ _ i→i+1) 0 pmap. This code first reduces the parallel map into a sequential one to preserve the order and then computes the fold. We have thus added an asynchronous iterator operator which reduces all the elements of the parallel map in an unspecified order:

```
(* async_fold: (key→α →β →β )→α t→β →β par *)
let async_fold f pmap e = let pmap'= rebalance_if_needed pmap in
 parfun (fun s →LocMap.fold f s e) (fst pmap')
```

```
(* from_sets_to_parset: LocSet.t par→LocSet.t par *)
let from_sets_to_parset vect_sets =
 let sets_to_send = parfun (fun set→LocSet.fold (fun elt map→
       let pid=(hash elt) in
       let s_add=(try MapSet.find pid map with Not_found→LocSet.empty) in
        MapSet.add pid (LocSet.add elt s_add) map) set MapSet.empty) vect_sets in
 let to_send = parfun (fun map pid→
               try Some (MapSet.find pid map) with Not_found→None) sets_to_send in
 let delivery=put to_send in
   parfun (fun f→let list_sets = map_some_split f (procs ())
                      in List.fold_left LocSet.union LocSet.empty list_sets) delivery
```

Figure 2: Reorganization of a vector of sequential sets

The BSP cost of the above function (assuming op has a constant cost c_{op}) is $c_{\mathsf{op}} \times \max_{i=0}^{p-1} |\mathsf{m}_i|$. In this way, the function of cardinality could be coded as:

```
let cardinal pmap=List.fold_left (+) 0 (rpl_total (ParMap.async_fold (fun _ _ i→i+1) pmap 0))
```

which corresponds to the following sketch:

$$\mathsf{cardinal}\ \boxed{\mathsf{m}_0 \mid \cdots \mid \mathsf{m}_{p-1}} \Rightarrow \boxed{|\mathsf{m}_0| \mid \cdots \mid |\mathsf{m}_{p-1}|} \Rightarrow \sum_{i=0}^{p-1} |\mathsf{m}_i|$$

The BSP cost of this operation is $\max_{i=0}^{\mathbf{p}-1} |\mathsf{m}_i| + (\mathbf{p}-1) \times \mathbf{g} + \mathbf{l} + \mathbf{p}$.

4.2. Parallel Sets

The BSML implementation of parallel sets uses the same modular technique as parallel maps, *i.e.* each processor contains a subset of the original set as a sequential set. Operators such as add, remove, mem, is_empty, async_fold *etc.* are thus implemented as in Map. However, the set data structure provides other kinds of binary operators ($\cup, \cap, \setminus$). We take for example in this section, the implementation of the diff ($\setminus$) operator — description of the other operators can be found in [10].

To begin, we need a function that reduces a parallel vector of sets into a parallel set by eliminating duplicate elements from the sequential sets. A simple algorithm would broadcast every local set so as to detect non-local duplicates. A better algorithm minimizes communications by reorganizing the vector of sets by sending every value to a processor indicated by the hashing function. Duplicates are then eliminated at the destination by local unions of the sent sets. Figure 2 gives the code of such a function where MapSet is a map of sets of elements where pids are the keys and where map_some_split:($\alpha \rightarrow \beta$ option)→α list→β list applies a function f to all elements of a list l and if (f x)=(Some a) then a is added to a list result. Using the parallel superposition we could reorganize two parallel vectors in one superstep. Note that twice the number of supersteps is otherwise needed:

```
(* distributed_two_sets: LocSet.t par→LocSet.t par→LocSet.t par * LocSet.t par *)
let distributed_two_sets vect_sets1 vect_sets2 =
 super(fun ()→from_sets_to_parset vect_sets1) (fun ()→from_sets_to_parset vect_sets2)
```

To compute the difference of two parallel sets, we have four cases: none, one (first or second) or both of the parallel sets have been rebalanced. Therefore, elements may not be in their original processors indicated by the hashing function. In the first case, we just compute the differences of the local sets and the BSP cost is just the maximum time to compute the local differences. The second and third cases

```
(* diff: t→t→t *)
let diff (psets1,(step1,nbal1)) (psets2,(step2,nbal2)) = match (nbal1, nbal2) with
   (true,true) (* two no re−balanced sets *) →(parfun2 LocSet.diff psets1 psets2,(step1+step2,true))
 | (true,false) (* the second set was re−balanced *) →
    let diff1 = parfun2 LocSet.diff psets1 psets2
    and diff2 = parfun2 LocSet.diff psets2 psets1 in
    let diff' = from_sets_to_parset diff2 in (parfun2 LocSet.diff diff1 diff2, (step1+step2,true))
 | (false,true) (* the first set was re−balanced *) →
    let diff1 = parfun2 LocSet.diff psets1 psets2 in
    let distr = from_sets_to_parsets diff1 in (parfun2 LocSet.diff distr psets2, (step1+step2,true))
 | (false,false) (* two re−balanced sets *) →
    let diff1 = parfun2 LocSet.diff psets1 psets2
    and diff2 = parfun2 LocSet.diff psets2 psets1 in
    let distr = distributed_two_sets diff1 diff2 in
    let new_set = parfun2 LocSet.diff (fst distr) (snd distr) in (new_set, (step1+step2,true))
```

Figure 3: Difference of two parallel sets

are described in [10].

For the fourth case, we first compute the local differences of the local sets. In this way we compute the possible duplicate elements, *i.e.* each element that is in two different processors but needs to be eliminated by the difference. Those two parallel vectors of sets are then reorganized to eliminate the duplicate elements:

$$\mathsf{diff}\ \boxed{\mathsf{s}^1_0 \mid \cdots \mid \mathsf{s}^1_{p-1}}\ \boxed{\mathsf{s}^2_0 \mid \cdots \mid \mathsf{s}^2_{p-1}} \Rightarrow \begin{cases} \boxed{\mathsf{s}^1_0\setminus\mathsf{s}^2_0 \mid \cdots \mid \mathsf{s}^1_{p-1}\setminus\mathsf{s}^2_{p-1}} \\ \boxed{\mathsf{s}^2_0\setminus\mathsf{s}^1_0 \mid \cdots \mid \mathsf{s}^2_{p-1}\setminus\mathsf{s}^1_{p-1}} \end{cases} \Rightarrow$$

$$\begin{cases} \boxed{\mathsf{s}'_0 \mid \cdots \mid \mathsf{s}'_{p-1}} \\ \boxed{\mathsf{s}''_0 \mid \cdots \mid \mathsf{s}''_{p-1}} \end{cases} \Rightarrow \boxed{(\bigcup_{i=0}^{p-1} \mathsf{s}'^{i}_{0})\setminus(\bigcup_{i=0}^{p-1} \mathsf{s}''^{i}_{0}) \mid \cdots \mid (\bigcup_{i=0}^{p-1} \mathsf{s}'^{i}_{p-1})\setminus(\bigcup_{i=0}^{p-1} \mathsf{s}''^{i}_{p-1})}$$

where $\forall j\ \mathsf{s}'_j = \mathsf{s}^1_j\setminus\mathsf{s}^2_j$ and $\mathsf{s}''_j = \mathsf{s}^2_j\setminus\mathsf{s}^1_j$ and s'^{i}_{j} (respectively, s''^{i}_{j}) is the set of the elements sent by processor i to processor j from the set s'_j (resp. s''_j). The brace corresponds to the computations of both super-threads. The BSP cost is:

$$\max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(\mathsf{s}^1_j\setminus\mathsf{s}^2_j) + \mathcal{C}_{\mathbf{t}}(\mathsf{s}^2_j\setminus\mathsf{s}^1_j)) + \max_{j=0}^{\mathbf{p}-1}(\mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1}\mathsf{s}'^{i}_{j}) + \mathcal{C}_{\mathbf{t}}(\bigcup_{i=0}^{\mathbf{p}-1}\mathsf{s}''^{i}_{j})) + (\max_{j=0}^{\mathbf{p}-1}(\mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1}\mathsf{s}'^{i}_{j}) + \mathcal{S}(\bigcup_{i=0}^{\mathbf{p}-1}\mathsf{s}''^{i}_{j}))) \times \mathbf{g} + \mathbf{l}$$

The code of Figure 3 corresponds to the above sketches. Note that after one of the binary operations the resulting parallel set has all its elements stored in their original processors and thus could be considered as not rebalanced. Proofs of the methods can be found in [10].

4.3. Load Balancing

As shown in the above sections, the layout of the elements over the processors could be unpredictable after some unlucky sequence of operations. This phenomenon can degrade performance by slowing a parallel program to the speed of the most loaded processor. Owing to useless communications and synchronizations a parallel program may thus execute more slowly than its sequential equivalent. In [2] a general strategy has been introduced. It trades communications against better

load-balancing with only two phases of communication. The first one is a total exchange of the local sizes. Every processor then holds complete information about the load balancing of the input parallel data structure. Once the size of all data transfer has been decided, every processor selects the relevant information for its part in the exchange. Some values are then removed from the local data structures to produce new data structures and messages containing values to be sent. Finally these messages are forwarded and are incorporated into the local data structures.

The generic code of this load-balancing algorithm can be found in [10] — taken from [13] where the code works only with sets. The user could give to its data structures some parameters with a module of the following signature:

```
module type BALANCE=sig val need_balance: unit→bool
  val unbalanced: unit→int val max_op_without_balance: unit→int end
```

where max_op_without_balance defines the maximum number of operations that could modify the data structure before rebalancing it. It is used in a comparison with the integer mentioned in the above sections. This integer is used to define the number of operations that have modified the data structure. The function unbalance defines the maximum size difference among the local data structures. In this way, parallel data structures could be automatically rebalanced (currently Maps, Sets and Hashtables) and/or the user could rebalance itself in its code using the rebalance function of the module's signature. To rebalance, we use the following polymorphic function:

```
val rebalance_if_needed: (α →β par)→(int→β →γ list ∗ β )→(β →β →β )→(γ list→β )
                         (β par→α )→(α →int par)→(α →bool)→int→α →α
```

where the parameters are (1) a function that transforms the parallel data structure into a parallel vector of local data structures; (2) a function that takes n elements from a local data structure and returns this data structure without those elements, paired with messages to send; (3) a function that is able to unite two local data structures; (4) a function that transforms messages to a local data structure; (5) a function that reduces a parallel vector of local data structures into a parallel data structure; (6) a function that computes histograms of a parallel data structure; (7) a function that tests if the rebalancing is needed; (8) the maximum size difference between the local data structures; (9) the parallel data structure to rebalance.

5. Example of a Scientific Application

Our example is the computation of finding the nth-nearest neighbors of the ith vertex in an infinitely-repeated graph representing the topology of atoms in a molecule. The edges represent chemical bonds between pairs of atoms[‡].

Given an undirected graph, the task is to find the first, second, third $\cdots$ nth neighbors of the vertex i in the graph, for given n and i. The 1st neighbors of a vertex i refers to the set of vertices connected to i by an edge. The 2nd neighbors are the neighbors of the neighbors of i (we thus used the union of these sets) excluding the first neighbors (computed with a set difference). This can be generalized into a recurrence relation to compute the nth neighbors in terms of set operations. To make things more interesting, the graph is infinitely repeated. This rules out some

[‡] Web page at `http://www.ffconsultancy.com/products/ocaml_for_scientists/`

```
module AtomKey = struct type t = int * int list let compare = compare end
module AtomSet = Set.Make(AtomKey)

(* nth_nn : int →AtomSet.elt →AtomSet.t *)
let rec nth_nn = let memory=Hashtbl.create 1 in
 fun n (i, io)→ try Hashtbl.find memory (n,i) with Not_found→match n with
   0 →AtomSet.singleton (i,io)
 | 1 →let nn=bonds.(i−1) in if io = zero then nn else
        let aux (j,jo) s=AtomSet.add (j,add_i io jo) s in
         AtomSet.fold aux nn AtomSet.empty
 | n →let pprev=nth_nn (n−2) (i,io) and prev=nth_nn (n−1) (i,io) in
        let aux j t=AtomSet.union (nth_nn 1 j) t in
        let t=AtomSet.fold aux prev AtomSet.empty in
        let t'=AtomSet.diff (AtomSet.diff t prev) pprev in
          Hashtbl.add memory (n, i) t'; t'
```

Figure 4: Sequential Code

of the more common breadth-first search algorithms. Also, the number of vertices is very large (*e.g.* 10^5) so we shall not have enough space to store the neighbor matrix.

Figure 4 gives the sequential code where expression `bonds:AtomSet.t array` is the initial set of atoms, `add_i: int list→int list→int list` computes a new neighbor from two coordinates and `zero:int list` is the initial atom. Figure 5 gives the parallel code of this algorithm with straightforward modifications from the sequential code (replace `fold` by `async_fold`) where `from_list_of_parlist:elt list par list→t` gives a parallel set from a list of vectors of lists of elements and `list_of_set:AtomSet.t→AtomSet.elt list` returns a list of all the elements of a parallel set. The complexity of the sequential algorithm (where m is the number of atoms) is:

$$\frac{n \times (n-1)}{2} \times (2 \times m \log(m) + m^2 \log(m))$$

i.e. time to compute the difference of two sets (standard library's implementation assumed a logarithmic time to compute the difference of two sets), time to compute the union of the neighbors of the selected atom and this by the number of inductive steps — we have a logarithmic time to access to the previous neighbors. It is thus a quadratic algorithm.

For simplicity, we make the hypothesis that the hashing function is "perfect" (perfect distribution of the data) and that we do not need to use load balancing. We have thus $n/\mathbf{p}$ elements on each processor. In the parallel code, there is one communication phase (total exchange of the local neighbors elements) per computation of n. Therefore, we have the following BSP cost:

$$\frac{n \times (n-1)}{2} \times (2 \times \frac{m}{\mathbf{p}} \log(\frac{m}{\mathbf{p}}) + (\frac{m}{\mathbf{p}})^2 \log(\frac{m}{\mathbf{p}})) + (nm) \times \mathbf{g} + (\frac{n \times (n-1)}{2}) \times \mathbf{l}$$

i.e. time to compute the local neighbors, time to globally exchange them and a synchronization for each inductive step.

Some experiments have been done on a cluster with 5 or 10 Pentium IV nodes (with 1GB of main memory per node) interconnected with a Gigabit Ethernet network to show a performance comparison between the theoretical predictions, the sequential and parallel codes. Using a BSML implementation of the method described in [3], we have computed the following BSP parameters for the 5 nodes

```
module Balance = struct
 let need_balance ()=true and unbalanced ()=1000 and max_op_without_balance ()=10 end
module AtomSet = ParSet.Make(AtomKey)(Balance)

(* async_nth: int*int list →(int*int list) list par *)
let async_nth (i,io) = let nn = bonds.(i − 1) in
 if io = zero then (set_to_parlist nn) else
  let aux (j, jo) l = (j, add_i io jo)::l in (AtomSet.async_fold aux nn [])

let rec nth_nn = let memory = Hashtbl.create 1 in
 fun n (i, io) →try Hashtbl.find memory (n, i) with Not_found →match n with
   0 →AtomSet.singleton (i, io)
 | 1 →let nn = bonds.(i − 1) in if io = zero then nn else
          let aux (j, jo) l = (j, add_i io jo)::l in
           from_list_of_parlist [(AtomSet.async_fold aux nn [])]
 | n →let pprev = nth_nn (n−2) (i, io) and prev = nth_nn (n−1) (i, io) in
         let lt = List.map (fun j →(async_nth j)) (list_of_set prev) in
         let t = from_list_of_parlist lt in
         let t' = AtomSet.diff (AtomSet.diff t prev) pprev in
          Hashtbl.add memory (n, i) t'; t'
```

Figure 5: Parallel Code

cluster: $r = 469$, $g = 28$ and $l = 227512$. The testing graph actually represents a 100,000-atom model of an amorphous silicon. The MPI, PUB, TCP[§] implementations of the `BSMLlib` are used. These programs ran 100 times consecutively. Diagrams show the average of the results of 5 executions.

Figures 6 (a) and (b) summarize the results. For small n, owing to the overhead of synchronization barriers, the parallel version is equivalent to the sequential one. However parallel program clearly outperforms the sequential ones when n increases. Note that there is a step between the predictions of the performances and the true performances. This is due to the safe garbage collector of the `OCaml` language.[¶]

Figures 6 (c) and (d) summarize a performance comparison between parallel computations with load balancing of the sets and those without. We use the parameters of load balancing defined in Figure 5. Thanks to the parallel superposition, reorganizing two sets costs only one superstep and thus parallel set differences are efficient: performances are scalable and close to the predicted ones allowing a better distribution of the data without to much synchronization and communication.

Note that the parallel program is faster with 5 processors owing to the large number of barriers: they are less costly with 5 processors than with 10 processors. We recall that this program does not use a true parallel algorithm but uses a sequential one that heavily uses a parallel data structure.

6. Related Work

There are many data structure libraries in many programming languages such as Leda [17] for C++. Unfortunately, all these libraries are fully-sequential. Paper [21] presents a library of different parallel implementations of classical data-structures such as sets and priority queues applied to some symbolic computations. Neverthe-

[§]primitives implemented with the MPI or PUB libraries or only with the TCP features of `OCaml`.
[¶]unlike in C where the programmer has to allocate and de-allocate the data of the memory.

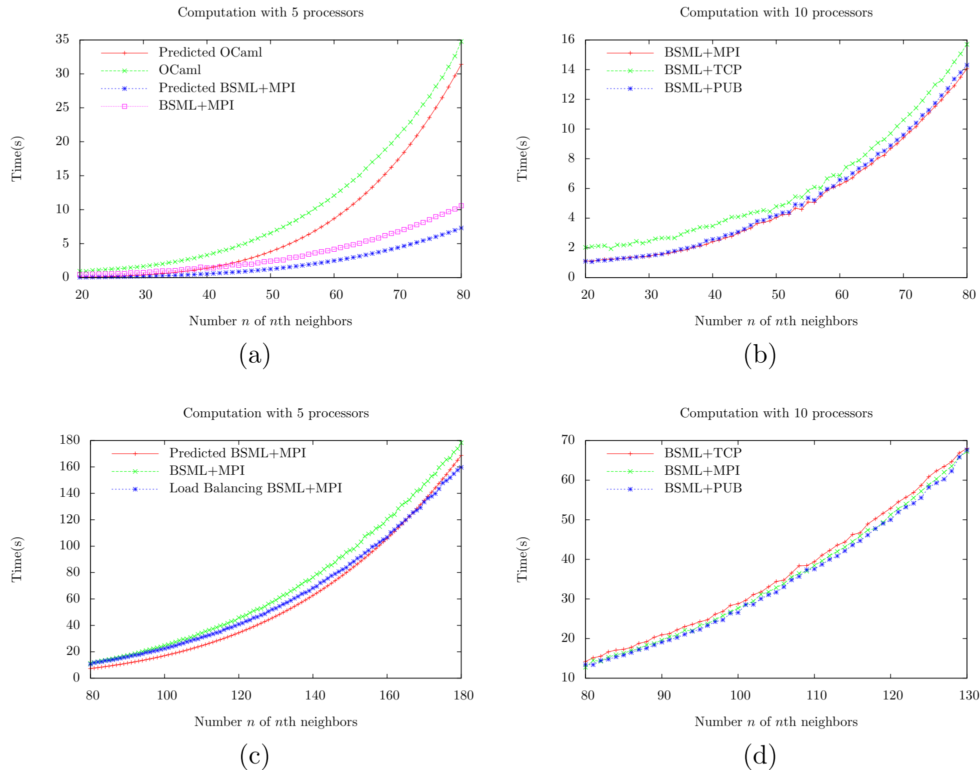

Figure 6: Benchmarks of nth nearest neighbors

less no cost model has been used and costs of the operations are thus unpredictable. In the same manner, paper [15] describes the implementation in C++ with MPI of two collections of data structures that are designed to support the modeling of particle systems in Lagrangian formulation and partial differential equations.

For high-level languages, there are many libraries based on the algorithmic skeletons framework and used for computations and scientific problems. OcamlP3L is one of them and has been used for programming numerical applications [4]. However this library does not provide parallel data structures. Without an explicit distribution of data and the parallel parts of programs, it seems to be hard to implement parallel data structures and to extend the set of skeletons in order to have "data structure skeletons". Gph and Eden [14] are parallel languages based on the functional language Haskell. Gph uses explicit parallelism with two special constructors "Par" and "Seq" while Eden uses processes exchanging data on communication channels which are visible to the programmer as lazy lists. This way, it seems possible to implement parallel data structures. Nevertheless the system has to put the "parallel threads" on the processors and thus performance predictions are difficult.

Since the advent of parallel computing, considerable work has been done towards solving the problem of realizing distributed data structures and several results have been obtained for particular network architectures [1]. PRAM algorithms for the dictionary data structure has been presented in [16]. In [11], authors present dif-

ferent efficient BSP algorithms for different designs of data structures as priority queues or dictionaries. Some benchmarks have been done for the classical operations but no scientific applications have been coded and benchmarked. The parallel data structures have been coded in C+BSPlib resulting in a difficult maintenance. In [6], authors present a CGM algorithm for the distance-bound smoothing (problem which is similar to the nth neighbors) for particular molecular configurations.

Paper [13] describes a first implementation of parallel sets in BSML. Unfortunately, the authors did not use the module feature of `OCaml`; they used their own set implementation (unbalanced trees) and thus could not test their work with a scientific application. Furthermore, most of the binary operations over the sets transform a parallel set into a sequential one by a total exchange of the elements that make these operations slower than sequential ones.

7. Conclusions and Future Works

The BSML language allows direct mode BSP programming. This paper shows how to obtain a modular implementation of generic parallel data structures and how it is possible to use them in order to get an efficient scientific application (which makes extensive use of data structures) without suffering from disadvantages of the complexity of a parallel algorithm. The cost of these data structures is defined using the BSP cost model and some benchmarks are done.

Several directions can be followed for future work. The first one is the design and implementation of other parallel data structures such as parallel lists with optimized algorithms as in [12] — for parallel priority queues. Ocamlgraph[‖] is a generic graph library for `OCaml`. At the moment, Ocamlgraph uses Maps or Hashtables of Sets for the representation of graphs. Sets are used for the edges of the graphs whereas Maps or Hashtables are used for vertices. In this way, it seems possible to use the BSML's parallel data structures instead of the `OCaml` ones and thus get a parallel representation of the graphs.

Another direction is a semantic investigation is another direction. We have certified BSML programs [8] with the Coq proof assistant. In paper [7], the authors have certified the `OCaml`'s set and map implementation in Coq (and other implementations such as red-black trees). An interesting result would be the certification of the above parallel implementation which massively and independently used the sequential ones. Yet another direction is performance comparisons with low level languages such as C/Fortran with parallel implementations of data structures [19] is another direction.[**]

References

[1] M. J. Atallah, F. Dehne, R. Miller, A. Rau-Chaplin, and J.-J. Tsay. Multi-search Techniques: Parallel Data Structures on Mesh-connected Computers. *Journal of Parallel and Distributed Computing*, 20:1–13, 1994.

[‖] Web pages at `http://www.lri.fr/~filliatr/ocamlgraph`

[**] This comparison is a hard work since, even for the sequential code, there is no tools for this purpose: unlike C or Fortran, `OCaml` is a functional language with a garbage collector but with a polymorphic type system for safety.

[2] M. Bamha and G. Hains. Frequency-adaptive Join for Shared Nothing Machines. *Parallel and Distributed Computing Practices*, 2(3):333–345, 1999.

[3] R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.

[4] F. Clement, A. Vodicka, R. Di Cosmo, and P. Weis. Parallel Programming with the OCamlp3l System, Application to Numerical Code Coupling. Technical Report RR-5131, INRIA, 2004.

[5] M. Cole. Bringing Skeletons out of the Closet: A Pragmatic Manifesto for Skeletal Parallel Programming. *Parallel Computing*, 30(3):389–406, 2004.

[6] N. Deo and P. Micikevicius. Coarse-grained Parallelization of Distance-bound Smoothing for the Molecular Conformation Problem. In S. K. Das and S. Bhattacharya, editors, *4th International Workshop Distributed Computing, Mobile and Wireless Computing (IWDC)*, volume 2571 of *LNCS*, pages 55–66. Springer, 2002.

[7] J.-C. Filliatre and P. Letouzey. Functors for Proofs and Programs. In *Proceedings of The European Symp. on Programming*, volume 2986 of *LNCS*, pages 370–384, 2004.

[8] F. Gava. Formal Proofs of Functional BSP Programs. *PPL*, 13(3):365–376, 2003.

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

[10] F. Gava. *Functional Approaches of Parallel Computing and Meta-computers: Semantics, Implementations and Certification.* PhD thesis, University of Paris XII Val-de-Marne, December 2005.

[11] A. V. Gerbessiotis and C. J. Siniolakis. Architecture Independent Parallel Selection with Applications to Parallel Priority Queues. *Theoretical Computer Science*, 301(1-3):119–142, 2003.

[12] A. V. Gerbessiotis, C. J. Siniolakis, and A. Tiskin. Parallel Priority Queue and List Contraction: The BSP Approach. *Computing and Informatics*, 21:59–90, 2002.

[13] 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.

[14] K. Hammond, P.W. Trinder, et al. Comparing Parallel Functional Languages: Programming and Performance. *Higher-order and Symbolic Computation*, 15(3), 2003.

[15] K. Hofler, M. Muller, and S. Schwarzer. Design and Application of Object Oriented Parallel Data Structures in Particle and Continuous Systems. In E. Krause and W. Jager, editors, *High Performance Computing in Science and Engineering*, LNCS, 1999.

[16] M. Medidi and N. Deo. Parallel Dictionaries using AVL Trees. *Journal of Parallel and Distributed Computing*, 49(1):146–155, 1998.

[17] K. Mehlhorn and S. Nher. LEDA: A Platform for Combinatorial and Geometric Computing. *Communication of the ACM*, 38(1):96–102, 1995.

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

[19] N. Thomas, G. Tanase, O. Tkachyshyn, J. Perdue, N. M. Amato, and L. Rauchwerger. Framework for Adaptive Algorithm Selection in STAPL. In *ACM SIGPLAN Symp. Prin. Prac. Par. Prog. (PPOPP)*, 2005.

[20] A. Tiskin. A New Way to Divide and Conquer. *PPL*, 11(4):409–422, 2001.

[21] K. A. Yelick, S. Chakrabarti, E. Deprit, J. Jones, A. Krishnamurthy, and C. Wen. Parallel Data Structures for Symbolic Computation. In *Workshop on Parallel Symbolic Languages and Systems*, 1995.