The Journal of Supercomputing (2020) 76:5079–5097
https://doi.org/10.1007/s11227-019-02822-9

Programming BSP and MULTI-BSP algorithms in ML

Victor Allombert[1] · Frédéric Gava[2]

Published online: 27 March 2019
© Springer Science+Business Media, LLC, part of Springer Nature 2019

Abstract
The BSML and MULTI-ML languages have been designed for programming, *à la* ML, algorithms of the respectively BSP and MULTI-BSP bridging models. MULTI-BSP is an extension of the well-know BSP model by taking into account the different levels of networks and memories of modern hierarchical architectures. This is a rather new model, as well as MULTI-ML is a new language, while BSP and BSML have been used for a long time in many different domains. Intuitively, designing and programming MULTI-BSP algorithms seems more complex than with BSP, and one can ask whether it is beneficial to rewrite BSP algorithms using the MULTI-BSP model. In this paper, we thus investigate the *pro and cons* of the aforementioned models and languages by experimenting with them on different typical applications. For this, we use a methodology to measure the level of difficulty of writing code and we also benchmark them in order to show whether writing MULTI-ML code is worth the effort.

Keywords BSP · MULTI-BSP · ML · Hierarchical · Performance · Algorithms

1 Introduction

Context Our previous work aimed at designing a parallel *functional* language based on the bulk synchronous parallelism (BSP) *bridging model* called BSML [14]. BSP is a model of parallelism which offers a high level of abstraction and takes into account real communication and synchronisation cost [23]. BSP has been used successfully for a broad variety of applications such as scientific computation [6], artificial intelligence, big-data and graph frameworks (PREGEL [19]). To be compliant to a *bridging model* eases the way of writing code and ensures *efficiency* and *portability* from one

✉ Victor Allombert
victor.allombert@lacl.fr; victor.allombert@univ-orleans.fr

Frédéric Gava
frederic.gava@univ-paris-est.fr

[1] LIFO, Université d'Orléans, Orléans, France

[2] LACL, Université Paris-Est Créteil, Créteil, France

architecture to another. Thanks to a cost model, it is also possible to reason on the algorithmic *costs*.

As modern high-performance computing (HPC) architectures are *hierarchical* and have multiple layers of parallelism, communication between distant nodes cannot be as fast as among the cores of a given processor. Because BSP was designed for *flat* architectures, we now consider the MULTI-BSP bridging model [26], an extension of BSP which is dedicated to hierarchical architectures. MULTI-ML uses a small set of ML-like primitives [3] for programming MULTI-BSP algorithms similarly to BSML for the BSP algorithms. As BSML, MULTI-ML [3] is implemented using the ML language OCAML (http://ocaml.org/).

Comparing languages Since we now have two programming languages dedicated each for two rather different bridging models, we ask ourselves whether they differ in terms of performances and written code. Both have structured models of execution. Both have an OCAML + MPI implementation. However, BSML has been designed for flat parallelism, whereas MULTI-ML is designed for controlled nested parallelism. To compare them, we have chosen three different cases that are in the field of symbolic, numerical, and big-data computations.

We want to *compare* the two languages for both *performance* and the *ease* of writing code.[1] For the performance, we have chosen *speedup* and/or timing for some data-sets. We also change the target architecture by modifying the number of cores, processors, and nodes of the machines. For the code, we have choose some traditional *metrics* which are the *Halstead difficulty* and the *McCabe cyclomatic complexity*. These metrics mainly count the number of operands and programming structures such as conditionals, loops, etc. We have adapted them to take into account the number of *parallel operators*. These metrics are not perfect but are easy to use. Finally, we have used one interesting ability of both BSML and MULTI-ML which is programming parallel algorithms in an *incremental* manner from sequential codes, which simplifies the development of parallel codes. As explained later, this is due to the fact that both BSML and MULTI-ML provide a global view of programs, i.e. their programs can be seen as sequential programs working on parallel data structures ("seq of par"), while in many HPC libraries such as MPI, programs are written in the SPMD style and are understood as a parallel composition of communicating sequential programs ("par of seq").

Outline The rest of this paper is structured as follows. First, Sect. 2 briefly presents the two aforementioned languages. Then, Sect. 3 defines our methodology of comparison of the languages and we apply it to different *use cases* (Sects. 3.2–3.4), hoping they are general enough to stand for a representative sample of HPC applications. For all of them, we give some *benchmarks* in terms of both performance and

[1] We currently make the hypothesis that the more complicated the codes are, the more complicated the algorithmic design is. We leave the problematic of algorithmic design for future work and we focus on code writing only.

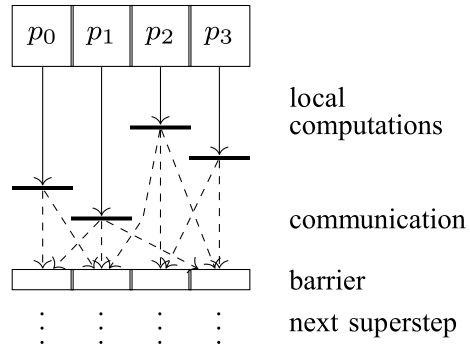

Fig. 1 A BSP superstep

difficulty of writing the code. Finally, in Sect. 4, we discuss related work and Sect. 5 concludes the paper by giving a brief outlook of future work.

2 BSML and multi-ML: similar but different languages

2.1 Programming BSP algorithms in ML

2.1.1 The BSP bridging model

In the BSP *bridging model*, a computer is a set of **p** *uniform* pairs of processor–memory components with a communication network [6, 25]. A BSP program is executed as a *sequence* of *supersteps* (Fig. 1), each one divided into three successive disjointed phases: (1) each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; (2) the network delivers the requested data; (3) a *global synchronisation barrier* occurs, making the transferred data available for the next superstep.

The *performance* of the BSP computer is characterised by 4 *parameters* (that we do not detail in this article[2]). To reliably *estimate* the execution time of a BSP program, these parameters could be easily benchmarked [6]. The execution time (cost) of a superstep is the maximal of the sum of the local processing time, the data delivery and the global synchronisation times. The total cost of a BSP program is the sum of its supersteps's costs.

2.1.2 The BSML language

BSML [14] uses a *small set of primitives* and is currently implemented as a library (http://traclifo.univ-orleans.fr/bsml/) for the ML programming language OCAML.

[2] We do not show the BSP parameters nor the MULTI-BSP ones because we do not use them at all in this work. In fact, cost prediction is rather impossible in some of our application cases, e.g. the state-space construction where the number of computing states is unknown and no upper bound is possible in practice [9].

Primitive	Type	Informal semantics
≪e≫	'a par (if e:'a)	$\langle e, \ldots, e \rangle$, a vector of size $\mathbf{p}$ the number of processors
pid	int par	A predefined vector: i on processor i
\$v\$	'a (if v: 'a par)	v_i on processor i, assumes $v \equiv \langle v_0, \ldots, v_{\mathbf{p}-1} \rangle$
proj	'a par→ (int→ 'a)	$\langle x_0, \ldots, x_{\mathbf{p}-1} \rangle \mapsto (\textbf{fun } i \to x_i)$
put	(int→ 'a)par→ (int→ 'a)par	$\langle f_0, \ldots, f_{\mathbf{p}-1} \rangle \mapsto \langle (\textbf{fun } i \to f_i\ 0), \ldots, (\textbf{fun } i \to f_i\ (\mathbf{p}-1)) \rangle$

Fig. 2 Summary of the BSML primitives

An important feature of BSML is its *confluent* semantics: whatever the order of execution of the processors is, the final value will be the same. Confluence is convenient for *debugging* since it allows to get an *interactive loop*, the toplevel. It also simplifies programming since the parallelisation can be done *incrementally* from an OCAML program.

A BSML program is built as a ML one but using a specific data structure called *parallel vector*. Its ML type is `'a par`. A vector expresses that each of the **p** processors *embeds* a value of any type `'a`. Figure 2 summarises the BSML primitives. Informally, they work as follows: **`let`** ≪e≫ be the vector holding `e` everywhere (on each processor), the ≪ ≫ indicates that we enter into the scope of a (parallel) vector. Within a vector, the syntax `$x$` can be used to read the vector `x` and get the local value it contains. The *ids* can be accessed with the predefined vector **`pid`**.

The **`proj`** primitive is the only way to *extract* local values from a vector. Given a vector, it returns a function such that, applied to the *pid* of a processor, returns the value of the vector at this processor. **`proj`** performs communication to make local results available globally and ends the current superstep.

The **`put`** primitive is another communication primitive. It allows any local value to be *transferred* to any other processor. It is also synchronous, and ends the current superstep. The parameter of **`put`** is a vector that, at each processor, holds a function returning the data to be sent to processor *j* when applied to *j*. The result 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*.

To illustrate the previous primitives with a small piece of code, we use the BSML toplevel and a *simulation* of a BSP machine with three processors. Now if we want to convert a vector into a replicated list (that is to say an identical list on each processor [14]), we write:

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

(where `procs` is a list of the processors's ids) and we applied it to a vector which contains the processors's ids.

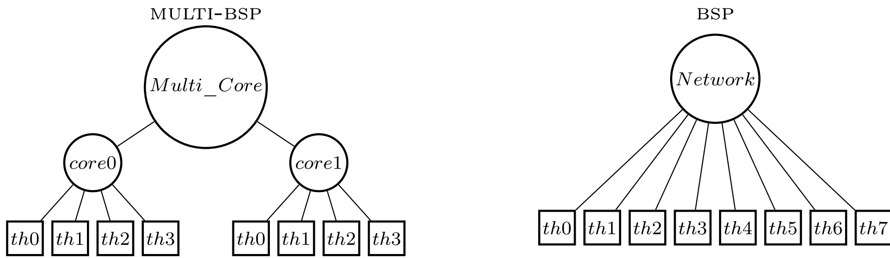

Fig. 3 The difference between the MULTI-BSP and BSP models for a multi-core architecture

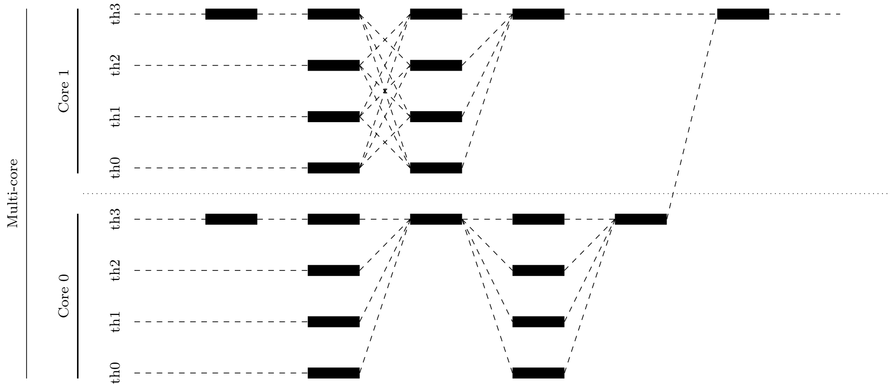

Fig. 4 Example of the MULTI-BSP execution model

2.2 Programming Multi-BSP algorithms in ML

2.2.1 The multi-BSP bridging model

MULTI-BSP is a *bridging model* [26] which is adapted to hierarchical architectures, mainly *clusters* of *multi-cores*. There exist other hierarchical models [17], but MULTI-BSP describes them in a simpler way. The structure and abstraction brought by MULTI-BSP allows to have portable programs with *scalable* performance predictions, without dealing with low-level details of the architectures. This model brings a *tree*-based view of nested components (*sub-machines* or *siblings*) of hierarchical architectures where the lowest stages (*leaves* symbolised by squares) are processors and every other stage (*node* symbolised by circles) contains memory (or a network). Figure 3 illustrates the difference between the BSP and MULTI-BSP models for a multi-core.

Every component can execute code, but they have to synchronise in favour of data exchange. Thus, MULTI-BSP does not allow sub-group synchronisation of any group of processors: at a stage i there is only a synchronisation of the sub-components, a synchronisation of each of the computational units that manage the stage

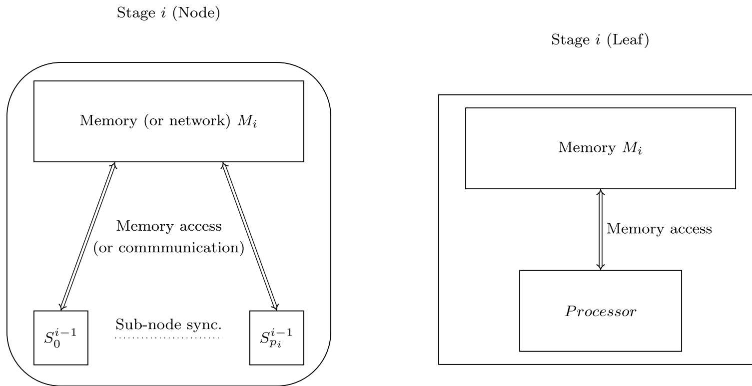

Fig. 5 The MULTI-BSP components

$i-1$. So, a node executes some code on its nested components (*aka* "*children*"), then waits for results, does the communication and synchronises the sub-machine. A MULTI-BSP algorithm is thus composed of several supersteps, each step is synchronised for each sub-machine. Figure 4 illustrates such an execution model where black rectangles scheme computations and dash lines between rectangles stand for communications.

Mainly, an instance of MULTI-BSP is defined by **d**, the fixed depth of the balanced and homogeneous tree architecture, and by the 4 BSP performance parameters (plus the memory size) for each *stage i* of the tree. Figure 5 illustrates nodes and leaves. Thus, for each $i \in \{0, \ldots, d-1\}$, p_i is the number of sub-components inside the $i-1$ stage.

The cost of a MULTI-BSP algorithm is the sum of the costs of the supersteps of the root node, where the cost of each of these supersteps is the maximal cost of the supersteps of the sub-components (plus communication and synchronisation); and so on.

2.2.2 The multi-ML language

MULTI-ML [2, 3] (https://git.lacl.fr/vallombert/Multi-ML) is based on the idea of executing BSML-like codes on every stage of a MULTI-BSP architecture. This approach facilitates *incremental* development from BSML codes to MULTI-ML ones. MULTI-ML follows the MULTI-BSP approach where the hierarchical architecture is composed of *nodes* and *leaves*. On nodes, it is possible to build parallel vectors, as in BSML. This parallel data structure aims to manage values that are stored on the sub-nodes: at stage i, the code `let v=≪e≫` evaluates the expression `e` on each $i-1$ stages. Inside a parallel vector, we note `#x#` to copy the value `x` stored at stage i to the memory $i-1$.

We also introduce the concept of *multi-function* to recursively go through a MULTI-BSP architecture. A *multi-function* is a particular recursive function, defined by the keyword `let multi`, which is composed of two codes: the node and the

Fig. 6 A multi-function code

```
1 let multi mf [args]=
2   where node =
3     (* BSML code*)
4     ...
5     ≪mf [args]≫
6     ... in v
7   where leaf =
8     (* OCaml code *)
9     ... in v
```

Fig. 7 A multi-function

```
1 let multi mf () =
2  where node =
3   let rec_call = ≪mf()≫
4    in "hello␣node"
5  where leaf = "hello␣leaf"
```

leaf codes. The *recursion* is initiated by calling the multi-function (recursively) inside the scope of a parallel vector, that is to say, on the sub-nodes. The evaluation of a multi-function starts (and ends) on the root node. The code of Fig. 6 shows how a multi-function is defined. After the definition of the multi-function `mf` on line 1 where `[args]` symbolises a set of arguments, we define the node code (from line 2 to 6). The recursive call of the multi-function is done on line 5, within the scope of a parallel vector. The node code ends with a value `v`, which is available as a result of the recursive call from the upper node. The leaf code, from lines 7 to 9 consists of sequential computations. When a multi-function is applied to a value at the MULTI-BSP level, the evaluation is initiated on the root node of the architecture. In the example (Fig. 7), the multi-function `mf` is called inside a parallel vector (line 3) to initiate the recursion on the sub-nodes. As the evaluation of a multi-function starts from the root node, the recursion will spread from the root towards the leaves.

As expected, the synchronous communication primitives of BSML are also available to communicate values from/to parallel vectors. We also propose another parallel data structure called *tree*. A tree is a distributed structure where a value is stored in every node and leave memories. A tree can be built using a multi-tree-function, using the **`let multi-tree`** keyword. We propose three primitives to handle such a parallel data structure: (1) To easily construct a tree with a simple expression, **`mktree`** `e` can be used; it aims to execute the expression `e` on each component of the architecture, resulting in a tree; (2) the function **`at`** can be used to access the value of a tree within a component; (3) the *global identifier* **`gid`** is shaped as a tree of identifiers, and is useful, for example, to distribute data depending on the position in the architecture.

3 Application cases

We now compare BSML and MULTI-ML. To do so, we first describe our methodology for the comparison in Sect. 3.1 and then we have selected three typical cases of various domains and apply the methodology to each case: (1) state-space calculation (basis of model-checking) in Sect. 3.2; (2) implementation of the FFT using algorithmic skeletons in Sect. 3.3; (3) finally, in Sect. 3.4, a classical big-data problem that is computing the similarity of millions of pairs.

It is to notice that in this paper we present non optimal implementations that cannot be compared to cutting-edge implementations: we neither use specific data structures nor domain specific tricks. For example, in the state-space algorithms, our sets of states do not use shared possibilities of the states as modern model-checkers do [9]. We use the standard OCAML 's sets and a naive representation of the states. Here, our objective is to compare the languages and their performances. Programming the most optimised implementations is not the purpose of this work.

3.1 Methodology

BSML and MULTI-ML have been designed to program parallel algorithms incrementally: from a sequential OCAML code to a BSP code using BSML and finally to a MULTI-BSP code using MULTI-ML. We can thus expect better performance, but we presume that the programs will be, unfortunately, more and more complex.

To measure this difficulty we have used: (a) the Halstead effort (HE) which depends on both the length and complexity of the code; the Halstead difficulty (HD) considers the amount of different operators used; (b) the McCabe cyclomatic complexity (CC) that is the number of linearly independent paths through the source; (c) the maintainability index (MI) which depends on HE and CC. To do so we have adapted the OCAML metrics tool (http://forge.ocamlcore.org/projects/ocaml-metrics/) to BSML and MULTI-ML. We now count, as an operand, each parallel primitive; and, as a new path, each multi-function and vector. Benchmarks were performed on two architectures:

1. (MIREV2) 8 nodes with 2 quad-cores (AMD 2376 at 2.3 Ghz) with 16 GB of memory per node and a 1 Gbit/s network;
2. (MIREV3) 4 nodes with 2 octa-cores with 2 hyper-threads (INTEL XEON $E5-2650$ at $2.6Ghz$) with $64GB$ of memory per node and a $10Gbit/s$ network.

To measure the code performance, we use the version 4.02.1 of OCAML and MPICH 3.1. We measure the *speedup* for different sizes of data and different configurations of our architectures with a variation of the number of nodes $\times$ processors $\times$ cores $\times$ threads used. We execute the codes on MIREV2 and MIREV3 by using different configurations. For example, $2 \times 2 \times 8 \times 2$ means that the code is executed on an architecture made of 2 nodes with 2 multi-cores using 8 cores with 2 threads; thus, we use 64 computing units. The configurations were chosen

Fig. 8 The OCAML code for MC

```
1  let algo_seq succ s0 =
2    let known=HashSet.empty
3    and todo=HashSet.empty in
4    HashSet.add s0 todo;
5    while not(todo ≠ ∅) do
6      let t=RandChoose todo in
7      HashSet.remove t todo;
8      HashSet.add known t;
9      update todo known (succ t);
10   done
```

Fig. 9 The BSML code for MC

```
1  let algo_bsp succ s0 =
2    let finish=ref false in
3    let known=≪HashSet.empty≫ in
4    let todo=≪HashSet.empty≫ in
5    ≪$todo$:=HashSet.add s0 $todo$≫ ;
6    while not(!finish) do
7      let tosend=≪local_successors succ
8                  $this$  $known$  $todo$≫
9      in exchange finish todo known tosend
10   done
```

arbitrarily, in order to compare performances with a growing number of components with both distributed and shared memories. All the sources and data are available in the MULTI-ML's GIT repository. We note ∞ when the program fails.

Thus, we aim to compare both the *code difficulty* and the difference, in terms of performances, between BSML and MULTI-ML codes.

3.2 First case: symbolic computation

Our first experiment is about model-checking (MC) which is a formal method often used to verify *safety-critical systems* [9]. Before verifying a *logical formula*, one must first compute the *state-space* of the systems. The parallelisation of this construction is a frequently used method in the industry [12].

The finite *state-space construction* problem consists of exploring all the states *accessible via* a *successor* function succ (returning a set of states) from an *initial state* s_0. This problem is computing and data intensive because realistic systems have a tremendous amount of scenarios. Usually, during this operation, all the explored states must be kept in memory in order to avoid multiple explorations of a same state. Figure 8 shows the usual sequential algorithm in ML where a set called `known` contains all the states that have been processed and would finally contain the state-space. It also involves a set `todo` that is used to hold all the states whose successors have not yet been constructed; each state `t` from `todo` is processed in turn (lines 5 to 10) and added to `known` (line 8), while its successors are added to `todo` unless they are already known—line 9.

```
1  let algo_multi succ s0 =
2    let known=mktree HashSet.empty
3    and todo =mktree HashSet.empty in
4    let multi space toPerform =
5      where node =
6        let sendBrothers=[| Map.empty ... Map.empty |] in
7        let sendUp=[| Map.empty ... Map.empty |] in
8        if (isEmpty toPerform) then (sendUp,sendBrothers)
9        else
10         let finish = ref false in
11         let scatt = (scatterDown toPerform mygid nbChildren) in
12         while not(!finish) do
13           let up = exchange finish <<space #scatt#>> scatt in
14           gatherUp up sendUp sendBrothers mygid nbChildren;
15         done;
16         (sendUp,sendBrothers)
17     where leaf =
18       local_successors succ (at gid) nbSiblings (at known) toPerform.(at gid)
19   in space [| ...hash(HashSet.single(s0)) ... |]
```

Fig. 10 The MULTI-ML code for MC

The standard flat parallelisation of this problem is based on the idea that each process only computes the successors for its own states. The MC code written with BSML is given in Fig. 9. To do this incremental parallelisation, a *partition* function (hashing) returns, for each state, a processor id; i.e. hash(s) returns the owner of s. Sets `known` and `todo` are still used but become local to each processor and thus provide only a partial view of the ongoing computations (lines 3–4). Initially, only state s_0 is known and only its owner puts it in its `todo` set (line 5). Once again, processors enlarge their own local sets of states by applying the successor function on the received states; recursively to their descendants until no new states are computed (line 7). Then, a synchronous communication primitive computes and performs, for each processor, the set of received states that are not yet locally known (line 9). To ensure termination, we use the additional variable `finish` in which we test whether some states have been exchanged or not by the processors. If not, there is no need to continue the computation (line 6).

The main difference between the BSP and MULTI-BSP codes is that the MULTI-BSP algorithm uses a hashing function to distribute the states on the sub-trees. Figure 10 summarises the MULTI-ML code. Each sub-tree of the MULTI-BSP architecture is in charge of keeping the states it owns. Moreover, on two different sub-trees, there could be different numbers of supersteps depending of the verifying system: the synchronous communication primitive is performed on different sub-trees leading to implicit sub-group synchronisations. Due to a random strategy of walk (hashing) in the set of states, the load-balancing is mainly preserved—with specific industrial systems, different kinds of load-balancing strategies would be necessary for an industrial development. There is not only communication between the *siblings* but also between parents and children. Indeed, some states might not be in their right sub-trees. Thus, like in the BSP algorithm (line 18), each leaf only computes its own

Languages \ Measures	HE	HD	CC	MI
OCAML	2k	25	6	208
BSML	17k	56	15	420
MULTI-ML	110k	346	78	845

OCAML { MIREV2 →22m49s; MIREV3 →12m14s }	**BSML**	**MULTI-ML**
MIREV2 $2 \times 2 \times 2$	3.2	3
MIREV2 $4 \times 2 \times 1$	4.3	5.3
MIREV2 $8 \times 2 \times 1$	5.8	12.5
MIREV2 $8 \times 2 \times 4$	7	14.7
MIREV3 $2 \times 2 \times 2 \times 1$	4	4.9
MIREV3 $2 \times 2 \times 4 \times 1$	5.4	8
MIREV3 $2 \times 2 \times 8 \times 1$	1.2	6.5
MIREV3 $2 \times 2 \times 8 \times 2$	0.9	6.7

Fig. 11 Benchmarks (measures and speedup) of the MC of a security protocol

states. Each node manages the sub-trees of its children by performing exchanges between siblings as in the BSP algorithm (line 13) but also gathers the states that are not in the right sub-trees (line 14); it also distributes the states between the sub-trees of its children (line 11). To perform the communications, the code uses two arrays of sets, each of the size of the number of siblings of each level of the MULTI-BSP architecture. The reader can notice that the MULTI-ML code is again an incremental update of the BSML one, using the hierarchical ability of the MULTI-BSP model: "same" main loop and local computations.

For our experiments, we compute the state-space of the well-known cryptographic Needham–schroeder public-key protocol with a standard universal Dolev–Yao intruder residing in the network [11]. Note that during the computation, most of the scenarios can be detected as faulty at their very beginning using a specialised MC, but this is not the subject of this article.

Figure 11 summarises the benchmarks. As intended, the MULTI-ML code is the more complex: by a factor of 2 compared to BSML, which is also 2 times more complex than the OCAML code.

In this example, using the hierarchical capacities is not beneficial for small architectures. But when the number of cores increases too much on nodes, for both MIREV2 and MIREV3, MULTI-ML exceeds BSML. This is not surprising since more communications append between cores without communicating through the network at every step of the algorithm. The BSP algorithm saturates the network with a large amount of communications and thus, this congestion drastically decreases the performances. On the contrary, the MULTI-ML program focuses on communications through local memories and communicates through the network only when necessary. Thus, the network is less saturated and the performance is better. On a configuration with many cores and physical threads, but for a small number of machines, the performance is disappointing. This

is due to too much caches-misses and RAM congestion. Indeed, our current algorithms take into account the different network capacities but *not* the memory sizes.

We have notice a strange and disappointing behaviour when using OCAML +MPI. Indeed, the OCAML 's runtime slows down by a factor of $\simeq 2$ when it massively allocates memory. We suspect an overhead (or incompatibility) between the OCAML garbage collector and the MPI 's memory allocation system. Unfortunately, we do not know how to go beyond this problem, which is just technical.

3.3 Second case: algorithmic skeletons and a numerical application

We can observe that many parallel algorithms can be characterised and classified by their adherence to a small number of generic *patterns* of computation. Skeletal programming proposes that such patterns can be *abstracted* and provided as a programmer's toolkit with specifications [10]. Thus, they can transcend architectural variations with implementations which enhance performance.

A well-known disadvantage of skeleton languages is that the only admitted parallelism is, usually, the skeleton one, while many applications cannot be easily expressed as instances of known skeletons. Skeleton languages must be constructed to allow the integration of skeletal and ad hoc parallelism [10]. In this way, having skeletons in a more general language would combine the expression power of collective communication patterns with the clarity of the skeleton approach.

In this work, we consider the implementation of well-known data-parallel skeletons as they are simpler to use than task-parallel ones and because they encode many scientific computation problems and scale naturally. Even if this implementation is surely less efficient compared to a dedicated skeleton language, the programmer can compose skeletons when it is natural for him and uses a BSML or MULTI-ML programming style when new patterns are needed.

The functional semantics of the considered set of data-parallel skeletons is described in [4]. It can also be seen as a naive sequential implementation using lists. The skeletons work as follows: skeleton **repl** creates a new list containing n times element x. The **map** and **mapidx** skeletons are equivalent to the classical single-program–multiple-data (SPMD) style of parallel programming, where a single program f is applied to different data, in parallel. The **scan** skeleton, like the collective operation `MPI_Scan`, computes the partial (prefix) sums for all list elements. A more complex data-parallel skeleton, the distributable homomorphism (**dh**) presented in [4], is used to express divide-and-conquer algorithms, i.e. $(dh \oplus \otimes l)$ transforms a list $l = [x_1, \dots, x_n]$ of size $n = 2^m$ into a result list $r = [y_1, \dots, y_n]$ of the same size, whose elements are recursively computed as follows:

$$y_i = \begin{cases} u_i \oplus v_i & \text{if } i \leq \frac{n}{2} \\ u_{i-\frac{n}{2}} \otimes v_{i-\frac{n}{2}} & \text{otherwise} \end{cases}$$

where $u = \mathbf{dh} \oplus \otimes [x_1, \dots, x_{\frac{n}{2}}]$, i.e. **dh** applied to the left half of the input list l and $v = \mathbf{dh} \oplus \times [x_{\frac{n}{2}+1}, \dots, x_n]$, i.e. **dh** applied to the right half of l. The **dh** skeleton pro-

```
1 let dh op1 op2 arr =
2   let multi dh_call arr =
3   where node =
4     let v=mkpar (fun i →
5           split arr nbChildren i) in
6     let res=gather ≪dh_call $ v $≫
7     in local_dh op1 op2 res; res
8   where leaf=local_dh op1 op2 arr; arr
9 in dh_call arr
```

Fig. 12 The MULTI-ML code for **dh**

vides the well-known butterfly pattern of computation which can be used to implement many computations with the appropriate $\oplus$ and $\otimes$ operators [4].

In this work, we choose the fast Fourier transform (FFT) where a list $x = [x_0, \ldots, x_{n-1}]$ of length $n = 2^m$ yields a list where the ith element is defined as: $(FFT\,x)_i = \sum_{k=0}^{n-1} x_k \omega_n^{ki}$ where ω_n denotes the nth complex root of unity $e^{2\pi\sqrt{-1}/n}$. The skeletal code is:

$$
\begin{aligned}
(FFT\ l) \equiv & \text{let}\ \Omega = \text{scan} + 1(\text{repl}\,(\omega\, n)\ \frac{n}{2}) \\
& \text{in map}\ \pi_1\ (\text{dh}\ \oplus_\Omega\ \otimes_\Omega\ (\text{mapidx}\ \textit{triple}\ l))
\end{aligned}
$$

The code for asynchronous skeletons such as **map** is trivial. Using BSML:

```
1 let map f fl = ≪(Lisp.map f) fl≫
```

Each processor owns a sub-part of the list. The scan code for both BSML and MULTI-ML can be found in [3]: we use a logarithmic parallel reducing algorithm for BSML and a divide-and-conquer one for MULTI-ML.

The BSML code for **dh** looks like a reducing and can be found in [2]. The one for MULTI-ML is in Fig. 12 and works as follows. We recursively split the list (lines 4–5) from the root node to the leaf where `local_dh` computes, locally, the **dh** skeleton. Then we gather the temporary results and perform a `local_dh` on the data (line 7). Note that these skeletons do not change the size of the "lists" so they can be implemented using vector of arrays, that is one array per processor. It is still a divide-and-conquer strategy and, in this case, the codes for BSML and MULTI-ML really differ. This is mainly due to the fact that there is no sub-group synchronisation using BSML whereas it is natural using MULTI-ML.

We have tested our two implementations of the **dh** skeleton (Fig. 13). To measure the difficulty, we use the implementation of the skeletons only. We test the FFT for two values of m, 19 and 21, leading to 2^m elements as input. We can notice an overhead with MULTI-ML on small architecture with a small input; and thus a speedup in favour of BSML. This is mainly due to the fact that the MULTI-ML **dh** implementation needs to transfer the data between each memory level of the architecture. This issue was an expected drawback of the MULTI-ML algorithm. In this algorithm, the sub-synchronisation mechanism on MULTI-ML is under-exploited. As the architecture grows in terms of both machines and cores, MULTI-ML takes a small advantage on

Languages \ Measures	HE	HD	CC	MI
OCAML	46	62	11	304
BSML	118k	158	29	149
MULTI-ML	81k	139	17	607

	BSML		MULTI-ML	
	19	21	19	21
MIREV2 $2 \times 2 \times 2$	4.3	3.7	2.5	2.7
MIREV2 $4 \times 2 \times 1$	3.6	4.2	3.1	3.2
MIREV2 $8 \times 2 \times 1$	4.2	5	4	3.8
MIREV2 $8 \times 2 \times 4$	3.8	3.9	2.6	2.8
MIREV3 $2 \times 2 \times 2 \times 1$	3.5	4.5	3.4	3.3
MIREV3 $2 \times 2 \times 4 \times 1$	3.6	4.3	3.7	3.8
MIREV3 $2 \times 2 \times 8 \times 1$	3.5	4.5	2.6	2.5
MIREV3 $4 \times 2 \times 8 \times 1$	2.6	3.9	3.8	4.5

For OCAML:

m=	19	21
MIREV2	16s	74s
MIREV3	10s	41s

Fig. 13 Benchmarks (measures and speedup) of FFT using skeletons

MIREV3 as BSML floods the network. However, the complexity of the code is in favour of MULTI-ML. So there is ultimately no problem using it. The overall performance of both implementations are disappointing, but it is the best we can hope for such a *toy* implementation of the FFT.

3.4 Third case: big-data application

Given a collection of objects, the all pairs similarity search problem (APSS) involves discovering all the pairs of objects whose similarity is above a given threshold. It may be used to detect redundant documents, similar users in social networks, etc. Due the huge number of objects present in real-life systems and its quadratic complexity, similarity scores are usually computed off-line.

Assuming a set of n documents of terms $D = \{d_1, d_2, \dots, d_n\}$. Each document d is represented as a sparse vector containing at most m terms. $d[i]$ denotes the number of occurrences of the ith term in the document d. The problem is to find all pairs (x, y) of documents and their exact value of similarity $sim(x, y) = \sum_i^m x[i] * y[i]$ if the similarity is greater than a certain threshold σ.

Different parallel algorithms have been proposed to APSS [1] and some of them deal with approximation techniques. Our work focuses on exact solutions only and we use inverted indexes as it is now the most common technique [5] (our OCAML code is an implementation of this kind of algorithm). For BSP-like computing, two

Fig. 14 The BSML code for APPS

```
1 let algoMapReduce vectors =
2   (* indexing *)
3   let map1=≪merge $vectors$≫ in
4   shuffle map1;
5   (* Similarity *)
6   let map2=≪allPairs $map1$≫ in
7   shuffle map2;
8   ≪reduce $map2$≫
```

Fig. 15 The MULTI-ML code for APPS

```
1 let multi apps docs =
2   where node =
3     while not(newDoc) do
4       let down=proj ≪apps (select docs)≫ in
5       systolic ≪apps (#down# (at gid))≫
6     done
7   where leaf = update (at indexList) docs
```

algorithms are mainly used [1]. The first algorithm is based on a systolic-like loop. We assume that each processor i holds a subset of documents D_i. Initially, each processor i computes the similarity $sim(D_i, D_i)$ of D_i's documents with each other documents of D_i. Then each subset is passed around from processor to processor in a sequence of $\mathbf{p}/2$ supersteps (exploiting the symmetric similarity of two subsets): each processor receives a subset D_j and calculates $sim(D_i, D_j)$ and then it sends D_j to its right-hand neighbour, while at the same time receiving the documents from its left-hand neighbour. If $\mathbf{p}$ is odd, half of the processors perform an additional exchange.

The second algorithm is based on two simple MAPREDUCE [21] phases as illustrated in Fig. 14: (1) Indexing; for each term in the document, each processor merges the term as key, and a pair $(d, d[t])$ consisting of document id d and weight of the term as the value (line 3). Then the algorithm handles the grouping by key of these pairs (the shuffle, line 4); (2) Similarity; each processor emits pairs of document ids that are in the same group G as keys (line 6). There will be $m \times (m-1)/2$ exchange pairs where $m = |G|$ for the shuffle (line 7); then they associate with each pair the product of the corresponding term weights. Finally they reduce the sums of all the partial similarity scores for a pair to generate the final similarity scores (line 8).

The MULTI-BSP algorithm is again an incremental improvement of the two above BSP algorithms. It is based on the following idea (Fig. 15): Initially, on leaves, we perform a systolic-like loop to initiating the index lists and the similarities. Then, each node selects some documents (on the sub-nodes, line 4) that already have a similarity: these documents are thus a greater opportunity to be similar with other documents. These documents are passed around from sibling to sibling (line 5) and are passed down to leaves as pairs as in the MAPREDUCE method. Now all the leaves update their similarity scores (line 7). And so on until no more documents are sending around siblings.

As before, to measure the performances and the difficulty of writing the programs, we take into account the algorithm part only and not the reading of the data.

Languages \ Measures	HE	HD	CC	MI
OCAML	47k	38	10	203
BSML(MAPREDUCE-like)	148k	104	26	863
BSML(systolic loop)	128k	68	19	424
MULTI-ML (hybrid)	250k	224	33	1100

In seconds	BSML		MULTI-ML	
	10M	17M	10M	17M
MIREV2 $2 \times 2 \times 2$	∞	∞	∞	∞
MIREV2 $4 \times 2 \times 1$	201	∞	190	∞
MIREV2 $8 \times 2 \times 1$	91	155	87	143
MIREV2 $8 \times 2 \times 4$	38	269	31	253
MIREV3 $2 \times 2 \times 2 \times 1$	∞	∞	∞	∞
MIREV3 $2 \times 2 \times 4 \times 1$	∞	∞	785	∞
MIREV3 $2 \times 2 \times 8 \times 1$	64	96	57	90
MIREV3 $2 \times 2 \times 8 \times 2$	41	87	39	72

Fig. 16 Benchmarks of the all pairs similarity search problem (APPS)

We use the Twitter's follower graph (July 2009, 24GB file with approximately 1.5 billion of followings) available at http://an.kaist.ac.kr/traces/WWW2010.html as data-set. For our experiments (Fig. 16), we take sub-parts of the original file. A pair corresponds to the same kind of following. We do not use any disc to store temporary results. There are 600M (resp. 1.5G) of pairs for 10M (resp. 17M) followings. The sequential code fails on these too large data-sets (not enough memory), so we give the execution times only. We do not use the common pruning of documents [5]: that reduces the overall computing time by reducing the number of documents to be compared and communicated but that is "independent" of using parallel algorithms. Our threshold is very low (even if it's not realistic), thus many pairs are computed. The number of pairs quadratically increases to the size of the data-set. The fails (∞) corresponds to "out of memory" or MPI fails when too much data are exchanged during a superstep (i.e. when less nodes take part in the computation or there are too many cores in use on a single node).

As already seen in [1], for BSP computing, the systolic method is faster than the MAPREDUCE one by an important order of magnitude. This is mainly due to a quadratic number of sending pairs. Thus, we do not give these timings.

The performances of the programs are not impressive because we use the generic data structures of OCAML which are not optimised for pairs and thus consumes too much memory. The losses are mainly due to a large use of the RAM. As intended, the MULTI-BSP code is more complex but gives better performance. The gain is not spectacular: using a BSP systolic algorithm, only one core sends data to a another core of a machine. So there are few data exchanged in the network and there is thus not a congestion as in the MC example. This also explains why the performances are better using MIREV2 than MIREV3: there are too many memory accesses in the RAM. However when using the MULTI-BSP algorithm, even if the computations and the memory uses

are of the same order of magnitude as in the BSP algorithm, there is a massive use of synchronisation of sub-machines which allow a better load-balancing. Even if there is more (local) supersteps, each performs less computation leading to less congestion when accessing to the RAM.

4 Related work

Hierarchical programming and multi-BSP libraries There are many papers about the gains of mixing shared and distributed memories, e.g. MPI and OPEN-MP [7]. As intended, the programmer must manage the distribution of the data for these two different models. For example, with the MC case, the algorithm of [20] handles a specific data structure (with locks) shared by the threads on cores and distributes the states across the nodes using the hash technique. We can also cite the work of [15] in which a BSP extension of C++ runs the same code on both a cluster and on multi-cores. But it is the responsibility of the programmer to avoid harmful nested parallelism. This is thus not a dedicated language working for hierarchical architectures. We can also highlight the work of NESTSTEP [16] which is a C/JAVA library for BSP computing, which authorises nested computations in case of a cluster of multi-cores—but without any safety.

Distributed functional languages Except in [18], there is a lack of comparisons between parallel functional languages. It is difficult to compare them since many parameters have to be taken into account: used libraries, used frameworks or architectures and what we want to measure that are efficiency, scalability, expressiveness, etc.

A data-parallel extension of HASKELL call NEPAL has been done in [8], an abstract machine is responsible for the distribution of the data over the available processors. MULTI-MLTON [22] is a multi-core aware runtime for standard ML, which is an extension of the MLTON compiler. It manages composable and asynchronous events using, in particular, *safe-futures*. A description of other bridging models for hierarchical architectures and other parallel languages can be found in [2]. Currently, we are not aware of any safe and efficient functional parallel language dedicated to hierarchical architectures.

5 Conclusion and future work

5.1 Conclusion

We have benchmarked different distributed applications using a flat bridging model (BSP) and its hierarchical extension (MULTI-BSP). We used two ML-like languages for both. We tried to compare both speedup and difficulty to write codes on rather different and typical HPC applications. Currently, we are not aware of similar works in the literature. Regarding the proposed case study, in general, the hierarchical programs are more efficient, but they are more difficult to write as a counterpart. As expected, there are also some cases where designing and programming a hierarchical algorithm does not

yield much. Intuitively, to get a performance gain, you have to maximise the locality (in the lowest memories of your machine) of calculations as well as the synchronisations/communications. That is to say that we can conclude, without surprise, that to have efficient MULTI-BSP algorithms, we need to massively exchanges data between the fastest memories. Indeed, on standard INTEL or AMD architectures, memories close to the physical threads (L1, L2 and L3 memories) are very fast. As a counter part, they are so small that it is a challenge to maximise their usage. As expected, we must concentrate on maximising data exchanges between computation units of the same memory locality.

Thanks to our approach, the BSP programs have been written incrementally from the sequential ones, as well as the MULTI-BSP programs extend the BSP ones. This seems to be an interesting point for software HPC development engineering: in a project, it is possible to work by successive additions of codes and it is not necessary to rewrite the code from scratch. However, it is still less flexible than the skeleton approach where only the patterns need to be efficiently implemented. Nevertheless, regarding an efficient MULTI-BSP algorithm, it is simpler to implement it using MULTI-ML code rather than in a skeleton framework.

5.2 Future work

The next phase will be to work on the optimisation of the previous programs. For example, how the states are kept in the memories is not optimised at all and induces many cache-misses. Using the last parameter of the MULTI-BSP model, that is the size of the memories, should leads to better algorithms. That should also reduce the execution time for exact APSS by using a cache-conscious data layout [24]. Our methodology also suffers from the fact that we make the hypothesis that the algorithms are known. However, designing an efficient BSP algorithm is harder than a sequential one. The effort is even harder for MULTI-BSP even though we perform an incremental development.

In the continuity of this work, we see two interesting points:

1. Doing programming experiments of our languages with students or users; this will allow to test if coding MULTI-BSP algorithms using MULTI-ML is really more difficult than coding BSP algorithms with BSML and/or sequential algorithms with OCAML; we think that designing the algorithms themselves is clearly the most difficult part;
2. Comparing the experimental timings with the expected cost formulae. The second author has already done this work in the context of BSP and BSML [13]. The conclusion is that the main difficulty is finding these cost formulae.

References

1. Alabduljalil MA, Tang X, Yang T (2013) Optimizing parallel algorithms for all pairs similarity search. In: Proceedings of the Sixth ACM International Conference on Web Search and Data Mining, WSDM '13, New York, NY, USA. ACM, pp 203–212
2. Allombert V (2017) Functional abstraction for programming multi-level architectures: formalisation and implementation
3. Allombert V, Gava F, Tesson J (2017) Multi-ML: programming multi-BSP algorithms in ML. Int J Parallel Program 45(2):20

4. Alt MH (2007) Using algorithmic skeletons for efficient grid computing with predictable performance. Ph.D. thesis, Münster University
5. Bayardo RJ, Ma Y, Srikant R (2017) Scaling up all pairs similarity search. In: Proceedings of the 16th International Conference on World Wide Web, WWW '07, New York, NY, USA. ACM, pp 131–140
6. Bisseling RH (2004) Parallel scientific computation: a structured approach using BSP and MPI. Oxford University Press, Oxford
7. Cappello F, Etiemble D (2000) MPI versus MPI+OpenMP on IBM SP for the NAS benchmarks. In: Proceedings of the 2000 ACM/IEEE Conference on Supercomputing, SC '00, Washington, DC, USA. IEEE Computer Society
8. Chakravarty MMT, Keller G, Lechtchinsky R, Pfannenstiel W (2001) Nepal-nested data parallelism in Haskell. In: Proceedings of the 7th International Euro-Par Conference Manchester on Parallel Processing, London, UK. Springer-Verlag, pp 524–534
9. Clarke EM, Henzinger TA, Veith H, Bloem R (2012) Handbook of model checking. Springer, Berlin
10. Cole M (2004) Bringing skeletons out of the closet: a pragmatic manifesto for skeletal parallel programming. Parallel Comput 30(3):389–406
11. Dolev D, Yao AC (1983) On the security of public key protocols. IEEE Trans Inf Theory 29(2):198–208
12. Garavel H, Mateescu R, Smarandache I (2001) Parallel state space construction for model-checking. Research report
13. Gava F (2008) BSP functional programming: examples of a cost based methodology. In: Bubak M, van Albada GD, Dongarra J, Sloot PMA (eds) Computational Science—ICCS 2008. Springer, Berlin Heidelberg, pp 375–385
14. Gesbert L, Gava F, Loulergue F, Dabrowski F (2010) Bulk synchronous parallel ML with exceptions. Future Gener Comput Syst 26(3):486–490
15. Hamidouche K, Falcou J, Etiemble D (2011) A framework for an automatic hybrid MPI+OpenMP code generation. In: Proceedings of the 19th High Performance Computing Symposia, San Diego, CA, USA. Society for Computer Simulation International, pp 48–55
16. Kessler CW (2000) NestStep: nested parallelism and virtual shared memory for the BSP model. J Supercomput 17(3):245–262
17. Li C, Hains G (2012) SGL: towards a bridging model for heterogeneous hierarchical platforms. Int J Parallel Program 7(2):139–151
18. Loidl H-W, Rubio F, Scaife N, Hammond K, Horiguchi S, Klusik U, Loogen R, Michaelson GJ, Peña R, Priebe S, Rebón ÁJ, Trinder PW (2003) Comparing parallel functional languages: programming and performance. High Order Symb Comput 16(3):203–251
19. Malewicz G, Austern MH, Bik AJ, Dehnert JC, Horn I, Leiser N, Czajkowski G (2010) Pregel: a system for large-scale graph processing. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, New York, NY, USA. ACM, pp 135–146
20. Saad RT, Dal Zilio S, Berthomieu B (2011) Mixed shared-distributed hash tables approaches for parallel state space construction. In: International Symposium on Parallel and Distributed Computing (ISPDC 2011), Cluj-Napoca, Romania
21. Seo S, Yoon E, Kim J, Jin S, Kim J-S, Maeng S (2010) HAMA: an efficient matrix computation with the MapReduce framework. In: 2010 IEEE Second International Conference on Cloud Computing Technology and Science (CloudCom), pp 721–726
22. Sivaramakrishnan KC, Ziarek L, Jagannathan S (2014) MultiMLton: a multicore-aware runtime for standard ML. J Funct Program 24(06):613–674
23. Skillicorn DB, Hill JMD, McColl WF (1997) Questions and answers about BSP. Sci Program 6(3):249–274
24. Tang X, Alabduljalil M, Jin X, Yang T (2017) Partitioned similarity search with cache-conscious data traversal. ACM Trans Knowl Discov Data 11(3):1–34
25. Valiant LG (1990) A bridging model for parallel computation. Commun ACM 33(8):103–111
26. Valiant LG (2011) A bridging model for multi-core computing. J Comput Syst Sci 77(1):154–166

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.