Journal of Logical and Algebraic Methods in Programming 109 (2019) 100479

Contents lists available at ScienceDirect

Journal of Logical and Algebraic Methods in Programming

www.elsevier.com/locate/jlamp

Axiomatization and characterization of BSP algorithms

Yoann Marquer[a], Frédéric Gava[b,*]

[a] INRIA – Bretagne Atlantique, Rennes, France
[b] Inria, Univ Renes, CNRS, IRISA, France

A R T I C L E   I N F O

Article history:
Received 1 October 2018
Received in revised form 29 May 2019
Accepted 23 July 2019
Available online 26 July 2019

Keywords:
Algorithmic completeness
ASM
Axiomatization
BSP
Semantics

A B S T R A C T

Bulk-synchronous Parallel (BSP) is a *bridging model* for HPC (High Performance computing) algorithm design. It provides a conceptual bridge between the physical implementation of the machine and the *abstraction* available to a programmer, while having *portable* and scalable performance predictions of BSP algorithms on most HPC systems. Two questions may come to mind. What are formally BSP algorithms? And how to ensure that the programmer can effectively *program* every BSP algorithm with a BSP language, especially with the *right cost*? Gurevich proved that three convincing *postulates* for the sequential algorithms are equivalent to what is called Abstract state Machines (ASMs), and thus that ASMs capture the sequential algorithms. Firstly, we extend these sequential postulates and ASMs in order to intuitively and realistically capture the BSP algorithms (and not more). Secondly, by using an *operational semantics* and an *algorithmic simulation*, we prove that ASM_{BSP} is equivalent to IMP_{BSP}, a minimal imperative BSP programming language. Therefore, BSP programming languages (extending at least IMP_{BSP}) are BSP algorithmically complete, involving the definition of a class model of the BSP algorithms.

1. Introduction

1.1. Context of the work and background

It is well established that everyone *informally* agrees to what constitutes a sequential algorithm even if usually the word "algorithm" is at most a vague concept, or a way to name programs written in *pseudo-codes*. And now, half a century later, there is a growing interest in defining *formally* the notion of algorithms, in other words, in defining a church thesis not only for computable functions but for algorithms [6,7,9]. Currently, there is no common agreement on this definition [5], and even less for parallel algorithms.

1.1.1. Axiomatization and characterization of sequential algorithms

An *axiomatic* presentation (largely machine independent) of the sequential algorithms (step-by-step and discrete time) has been done in [6]. The main idea is that there is not a standard language that truly represents all the sequential algorithms. In fact, every algorithmic book presents the algorithms in its own way and programming languages give too many details. This definition [6] of the algorithms has been mapped to the notion of Abstract state Machine (ASM[1]). Every algo-

* Corresponding author.
E-mail addresses: yoann.marquer@inria.fr (Y. Marquer), frederic.gava@univ-paris-est.fr (F. Gava).

[1] A kind of Turing machine with the appropriate level of abstraction; ASMs are also called *evolving algebras*.

https://doi.org/10.1016/j.jlamp.2019.100479
2352-2208/© 2019 Elsevier Inc. All rights reserved.

rithm can be captured by an ASM and every ASM is an algorithm [6,47] (the ASM thesis). This allows a common vocabulary about algorithms and that has been studied by the ASM community for several years.

Class models [48] allow to classify what can or cannot be effectively programmed. In fact, common sequential imperative languages such as C or JAVA are Turing-complete, which means that they can simulate the input-output (*i.e.* functional) relation of a Turing machine and thus compute (up to unbounded memory) every calculable functions.[2] However, *algorithmic completeness* is a stronger notion than Turing-completeness. It focuses not only on the input-output behavior of the computation, but on the *step-by-step behavior*. A model could compute all the desired functions, but some algorithms (ways to compute these functions) could be missing. For instance, one-tape and multi-tape Turing machines are Turing-equivalent, but the palindrome recognition can be done in $\Theta(n^2/\log(n))$ steps with a two-tapes machine and requires at least $\Theta(n^2)$ steps with one-tape [10]. That means that the best algorithm requires at least multi-tapes. Therefore, a multi-tape Turing machine is functionally equivalent to a one-tape Turing one, but they are *not* algorithmically equivalent. Another example is the computation of the greatest common divisor: an imperative language with only "for-loop" statements can compute all the primitive recursive functions (and not more), but such a language *cannot* compute the GCD with the *smallest complexity*, whereas it is possible by using a "while" statement [11].

Therefore, some algorithms may not be written in a programming language, even if the same result can be computed by using another algorithm. Is it possible to prove whether a model of computation is algorithmically complete, or not? In other words, *if this model can compute a set of functions, is it possible to compute them by using every possible algorithms?* This question rises interest because this could guarantee that the best possible algorithm for a desired function can effectively be written in this model. The conceptual difficulty of this question is again to clearly define what is meant by "*algorithm*" [5]. By using an operational semantics for an imperative core-language (IMP), and an *algorithmic simulation* between ASM and IMP, it has been proved in [12] that common imperative languages are not only Turing-complete, but algorithmically complete, which was only informally assumed so far. That means that mainstream languages (*e.g.* C or JAVA) with at least unbounded loops and conditionals are algorithmically complete (the *class* of sequential algorithms). In addition to that, we can obtain subclasses (*e.g.* primitive recursive execution times) if a restriction on loops is given [11,13].

1.1.2. Multi-processors algorithms and general solution

Parallel computers (multi-processors systems *a.k.a.* HPC machines, High Performance computing) are composed of more than one processor or unit of computation. Nowadays, HPC is the *norm* in many areas but it remains *more difficult* to have well defined paradigms and a common vocabulary as it is the case in the traditional sequential world. For instance, it is common to *informally classify* parallel computers (Flynn's taxonomy) by distinguishing them by the way they access the system memory (shared or distributed). But this is difficult to get a *formal taxonomy* of computer architectures and frameworks [1]: there is a zoo of definitions of systems, languages, paradigms and programming models. Indeed, in the HPC community, several words could be used to name the same thing, so that misunderstandings are easy. We can cite parallel patterns [3] versus algorithmic skeletons [4]; shared memory (PRAM) versus thread concurrency and Direct ReMote Access (DRMA[3]); asynchronous send/receive routines (MPI, http://mpi-forum.org/) versus communicating processes (π-calculus).

So, even if the three *postulates* for sequential algorithms [6] are mainly *consensual*, to our knowledge there is not such a work for HPC, in the sense of parallel/distributed/concurrent frameworks [19]. Firstly, due to the aforementioned zoo of definitions and secondly, due to a lack of realistic *cost models* of common HPC architectures to, at least, define the class of HPC algorithms that is a set of algorithms running on well established times. In HPC, typically, the *cost measurement* is (unfortunately) not based on the complexity of an algorithm (with the help of a cost model) but is rather on the execution time, measured using empirical *benchmarks*[4] (even if the trend is towards more rigorous analysis of complexity [1]). This is regrettable because the community is failing to obtain rigorous characterizations of subclasses for HPC algorithms. There is indeed the PRAM model (and its algorithms) to define the degree of parallelism of problems, but it is not realistic for concrete machines [20].

Without any class model [48], there is also a lack in formal study for *algorithmic completeness* of HPC languages. Such a result is nowadays impossible to obtain due to both aforementioned issues that are the *lack* of definitions of what HPC computing *formally* is and the *absence* of formal classes of HPC algorithms. For many years, concurrent ASMs [15] and parallel ones [17–19] tried to capture more and more general definitions of parallel, multi-agents, asynchronous or distributed computations but, finally, did not exhibit the needed class model. To do this, we promote a rather different *bottom-up* approach consisting on focusing on the model under consideration, in order to better *highlight the algorithmic execution time*, which is often too difficult to assess for general models. More generally we want to formalize a class of algorithms [48] at its natural level of abstraction instead of using a more general model then restrict it with "arbitrary" *ad-hoc* hypothesis. Because we think that taking into account all the features of all HPC paradigms is a daunting task that is unlikely to be achieved [1], a bottom up strategy (from the simplest models to the most complex ones) may be a solution that could serve as a basis for more general HPC subclass models.

[2] Using any parallel machines does not improve this born except in term of duration and, for distributed architectures, available memory.

[3] Also known as one-sided communications.

[4] Programmers are benchmarking load balancing, communication (size of data), *etc.* Using such techniques, it is difficult to explain why one code is faster than another one and which one is more suitable for one architecture or another.

Using a *bridging model* [20] is a *first step* to the desired formalization. It provides a *conceptual* bridge between the physical implementation of the machine and the *abstraction* available to a programmer of that machine. It also simplifies the task of the design of algorithms, their programming, and ensures a better *portability* from one system to another. More importantly, it allows to reason on the *costs* of the algorithms, and thus to of a class model (we recall that PRAM is more studying the degree of parallelism of problems rather than define a class model and having a link with physical machines). We conscientiously limit our work to the Bulk-Synchronous Parallel (BSP) bridging model because it has the advantage of being endowed with a simple *model of execution* [21,22]. Moreover, there are many different libraries and languages for programming BSP algorithms. The best known are the BSPLIB for C [23] or JAVA [24], ML-like language BSML,[5] PREGEL [26] for big-data, *etc. Informally*, it is known that all these languages and libraries allow the programming of every BSP algorithms, but we want to *formally guarantee it*.

1.2. Content of the work

As a basis to this work, we first give an *axiomatic definition of* BSP *algorithms* (ALGO_{BSP}, p. 6) by generalizing the Gurevich's axiomatization [6] to tuples of processors, and by adding a single and simple postulate. This new postulate (p. 9) is about the organization into *supersteps* of the BSP algorithms. Then, we extend the ASM model of computation for BSP (ASM_{BSP}, p. 15). We aim to prove that the axiomatic presentation ALGO_{BSP} and the operational presentation ASM_{BSP} define the same set of objects: the BSP algorithms. This can be summarized by (p. 18):

$$\textit{(Intended theorem)}\ \text{ALGO}_{\text{BSP}} = \text{ASM}_{\text{BSP}}.$$

But in "practice", it is more *convenient* to use a mainstream language for programming algorithms rather than ASMs.[6] The question (and main goal of this work) that arises is: *is a* BSP *programming language expressive enough (algorithmically equivalent) to program all the* BSP *algorithms*? We will prove that it is the case for a core imperative programming language[7] with BSP routines (p. 33). To do so, we will define the notion of *algorithmic simulation* (p. 30) and then prove that this core language is algorithmically equivalent to the class of BSP algorithms: every BSP algorithm could be programmed with this language (assuming the same elementary operations) and every program corresponds to an algorithm (even if is useless). That is (p. 37):

$$\textit{(Intended theorem)}\ \text{ASM}_{\text{BSP}} \simeq \text{IMP}_{\text{BSP}}$$

which means that the executions are the same, up to fresh variables and a temporal dilation (more details in the SubSection 3.1 p. 30). An interesting and novel point of this work is that the BSP *cost model is preserved*.

We finally answer previous criticisms by defining a convincing set of parallel algorithms, a *class model*, running in a predictable time (*cost model*) and by constructing a programming language (using common imperative statements) which is algorithmically complete for BSP algorithms: $\text{IMP}_{\text{BSP}} \simeq \text{ALGO}_{\text{BSP}}$.

We want to emphasize that our work is about a formal algorithmic model and not on architectures/machines or on formal verification of specific BSP algorithms/programs [34]. Our work aims to be an intermediary between programming languages and an algorithmic model, the class of BSP algorithms. Thus we extend previous works of [6,12] about sequential algorithms, where is has been proved that $\text{ALGO}_{\text{SEQ}} = \text{ASM}_{\text{SEQ}} \simeq \text{IMP}_{\text{SEQ}}$.

To be constructive, we also apply our work to a standard BSP library that is BSPLIB. We show how to build the communications in our formalism (p. 24) by having a function working *step-by-step* as required in the ASM framework and by constructing an *exploration witness* it.

1.3. Outline

The paper is structured as follows. Section 2 p. 4 is dedicated to the BSP algorithms. We first recall the BSP bridging model in SubSection 2.1 p. 4. Then we define the postulates of the BSP algorithms (the axiomatic point-of-view in SubSection 2.2 p. 6) with some general definitions used through out this paper, and the ASM_{BSP} (the operational point-of-view in SubSection 2.4 p. 14). We conclude this section by proving p. 18 that $\text{ALGO}_{\text{BSP}} = \text{ASM}_{\text{BSP}}$ and that the cost model is preserved (SubSection 2.6 p. 24), and by constructing a realization of a subpart of the BSPLIB's communications in SubSection 2.7 p. 24.

Section 3 p. 30 is dedicated to BSP programming. We define a notion of algorithmic simulation (SubSection 3.1 p. 30) and a core imperative language IMP_{BSP} (SubSection 3.2 p. 31), then prove (SubSection 3.3 p. 33) the bi-simulation with ASM_{BSP}.

[5] As stands in [25] about BSML: *"An operational approach has led to a* BSP *λ-calculus that is confluent and universal for* BSP *algorithms"*. The argument is that BSML's primitives can simulate any BSPLIB program, if the BSPLIB is algorithmically complete for BSP which was, before this work, formally unknown so far.

[6] Mainly because there is only one global loop in pure ASMs, so the code may be not structured enough for complexity analysis. We are aware that to formally study systems or to generate safe codes (functional correctness), ASMs are useful objects [27], but their implementation AsmL (pure ASMs with constructions allowing iterations and sequential composition) is not (strictly speaking) algorithmically complete [49].

[7] We think that such a language is more convenient than ASMs to determine classes in time or space, or simply to be compared to mainstream programming languages.

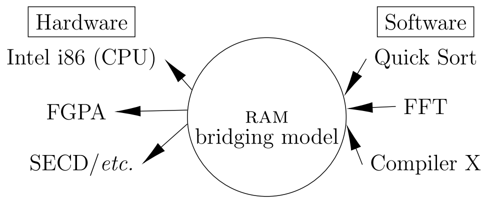

Fig. 1. The sequential case [20].

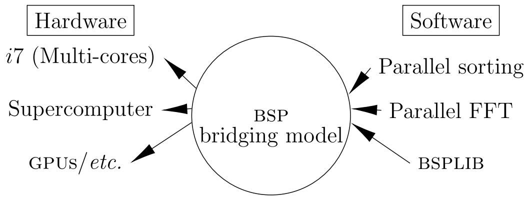

Fig. 2. The BSP bridging model [20].

We conclude this section by the second theorem p. 38 stating that $\text{ASM}_{\text{BSP}} \simeq \text{IMP}_{\text{BSP}}$, and by proving that the BSP cost model is preserved.

Section 4 p. 40 concludes and finishes with a brief outlook on future work.

Remarks: sketches of the proofs are given along the paper, more details are given in the technical reports [44,45], but the sketches of the proofs should provide enough details. A reader that have deep knowledge in BSP can jump directly to the SubSection 2.2 p. 6. On the other hand, a reader familiar with ASM can jump the common definitions (throughout used in SubSection 2.2) about first-order structures and their interpretation. We let them in the text in order to be self-contained.[8]

2. An axiomatization of BSP algorithms

If sequential algorithms has been successfully axiomatized in [6], it is not the case for BSP algorithms. The main objective of this section is to formally define them, before proving in the next section that a BSP imperative language is algorithmically BSP complete (which is the main goal of this work).

2.1. The BSP bridging model of computation

But before formally axiomatizing the BSP algorithms, we must describe in a fairly precise way what is the bridging BSP model. To do so, we present the notion of bridging model and then we detail the BSP model: machines, model of execution, cost model and thus what are informally BSP algorithms.

2.1.1. The notion of bridging model

The von Neumann model (RAM) has always served as the main model for designing sequential algorithm. It has also served as a reference model for hardware design. In the context of parallel algorithm design, no such ubiquitous model exists. The PRAM model (shared memory) allows a theoretical reasoning about algorithms by highlighting the *parallelism's degree* of problems. Nevertheless, it makes a number of assumptions that cannot be fulfilled in HPC applications and hardware, mainly because, in HPC, the number of processors is limited and the cost of communication is greater than the cost of computation.

The RAM model is thus the connecting bridge that enables programs from the diverse and chaotic world of sequential software to run efficiently on sequential machines from the diverse and chaotic world of hardware (Fig. 1 [20]). In other words, it is intended to provide a common level of understanding between hardware and software engineers. A *bridging model* provides software developers with an attractive escape route from the world of architecture-dependent parallel software. The term was introduced in [20], which argued that the strength of the RAM model was largely responsible for the success of sequential computing, so valiant [20] sought for a unifying model that could provide an effective bridge between parallel hardware and software.

The BSP model (assumed in direct mode in this paper, Fig. 2 [20]) allows algorithm design without any overspecification requiring the use of a large number of architectural parameters of HPC computers. A model of execution is also provided ensuring a *cost model* to be able to compare the efficiently and scalability of BSP algorithms. A bridging model has the

[8] This paper is an extended version of [46] which mainly corresponds to SubSection 2.4 and Section 3. We add the axiomatization, the realization and many details about the methodology.

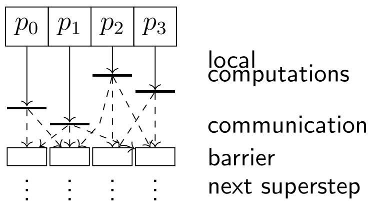

Fig. 3. A BSP superstep.

advantage that if an algorithm is correct and efficient, then this is the case for "all" physical architectures, present and future. This is known as *immortal* algorithms.

2.1.2. The BSP model

The BSP bridging model describes a BSP *computer*, an *execution model* for the algorithms, and a cost model which allows performance predictions of BSP algorithms on a given BSP computer. A BSP computer can be specified by three main components: (1) A set of **p** *homogeneous*[9] pairs of processors-memories (units of computation) where each unit is capable of performing one *elementary operation* or accessing a local memory in one time unit; (2) A communication network to exchange messages between the units; (3) A barrier mechanism is able to synchronize all the units. A wide range of current architectures can be seen as BSP computers. Clusters of PCs and multi-cores, GPUs, *etc.* can be thus considered as BSP computers.

A BSP algorithm, in direct mode, is organized as a *sequence* of *supersteps* (see Fig. 3) each of which is divided into three successive disjointed *phases*: (1) Each unit only uses its local data to perform *sequential* computations and to request *data transfers* to other units; (2) The network delivers the requested data; (3) A *barrier* occurs, making the transferred data available for the next superstep. This *structured* model enforces a strict *separation* of communication and computation: during the first phase of each superstep, no communication between the units is allowed; the information is accessible only after the *barrier*, and messages sent during the previous superstep are available at the destination at the start of the next superstep only.

BSP abstracts architectural parameters of parallel hardwares to provide a simple performance model for algorithms and programs to target. Units communicate by sending data to every other units. The time **g** (in flops, gap which reflects network bandwidth inefficiency) is for performing a 1-*relation* which is a collective exchange where every unit receives/sends at most one word. The network can deliver an h-*relation* in time $\mathbf{g} \times h$. Synchronize all the units costs **L** (in flops, "latency" which is the ability of the network to deliver messages under a continuous load). The parameter **L** was originally defined as the synchronization periodicity [20] and now is typically the minimal time between two successive supersteps (*e.g.* counting the initialization of the "network" and some memory buffers *etc.* and performing the barrier and guarantying that all messages were well transmitted). Such values, along with the processor's speed **r** (*e.g.* Mflops per second) can be empirically determined by executing benchmarks[10] [21] and allows the theoretical cost of algorithms to be compared and, finally, the duration of concrete BSP programs to be estimated.

The *execution time* (cost) of a superstep s is the sum of the maximal of the local processing, the data delivery and the barrier times. It is expressed by the following formula: $\text{Cost}(s) = w^s + h^s \times \mathbf{g} + \mathbf{L}$ where $w^s = \max_{0 \leqslant i < \mathbf{p}}(w_i^s)$ (where w_i^s is the local processing time on unit i during superstep s), and $h^s = \max_{0 \leqslant i < \mathbf{p}}(h_i^s)$ (where h_i^s is the maximal number of words transmitted or received by the unit i). The cost of a BSP algorithm is the sum of its superstep costs.[11]

Some papers rather use in the formula of the cost the sum of words for h_i^s but modern networks are capable of sending while receiving data. For some authors, **g** relates to a throughput cost that is sending at most and receiving at most one message per processor. And a message is supposed to have "lots of words" (depending of the size of the network's packets) and **L** is so not just the cost of a barrier but the cost of small message communication where the h-relation is too low: the overall superstep s costs is now $\max(\mathbf{L}, w^s + \mathbf{g} \times h^s)$. However, it is worth noticing that all the expressions of the BSP cost presented in the literature are equivalent within a small multiplicative constant, and the consistent use of any of them would lead to similar results. We still prefer the linear form of the costs because it is simpler for the analysis [21].

Definition 1. (Informal). A BSP algorithm is a computation, organized in a sequence of supersteps, that can be executed on any BSP machine.[12]

[9] To be homogeneous is an important constraint of the BSP model. This forbids a processor to run faster than the other ones. Load-balancing and distribution of data in general are also simplified since they depend on the size of the data only, assuming that the computations are a pro rata of data.

[10] **g** and **L** typically depend of **p** and of the underlying used BSP library (how the data are packaged and received in the memory, *i.e.* serialized and garbage collected or not when using a high-level language such as JAVA) and thus of the level of abstraction of the BSP algorithm.

[11] This cost is consistent with the fact that most machines are often unable to fully overlap computation and communication. Indeed, on many computers the transfer of data from the operating system to the application buffers typically requires processor participation. Furthermore, the parameter **L** is typically large enough to cover for the start-up cost of an h-relation.

[12] Assuming, if necessary, some properties on the BSP performance parameters such as **p** is even and/or $\mathbf{L} >> \mathbf{g}$, *etc.*

Notice that there exist other bridging models [33,41] but only BSP is currently widely used and accepted for parallel algorithm design. BSP has also been used with success in a wide variety of problems. A complete book of numerical algorithms is [21] and many other algorithms can be found in the literature. More discussions about BSP and algorithms could be find in Q1-Q3 p. 12

2.2. Axiomatization of BSP algorithms

We now give four postulates to axiomatize the BSP algorithms together with some general definitions that will be used throughout this work. The BSP algorithms are the objects verifying the four following postulates. We follow [6] for presenting only the necessary definitions before every postulate.

Postulate 1 *(Sequential time). A BSP algorithm A is given by:*

1. *A (potentially infinite) set of states $S(A)$;*
2. *A (potentially infinite) set of initial states $I(A) \subseteq S(A)$;*
3. *A transition function $\tau_A : S(A) \to S(A)$.*

Sets are potentially infinite because an algorithm A, such as an integer sorting, can have an infinite number of inputs, *e.g.* all possible integer arrays, or the possible distributions whatever the number of processors for a parallel sorting algorithm. These inputs are encoded in the initial states.

An *execution* of a BSP algorithm A is an infinite sequence of states $\overset{-\to}{S} = S_0, S_1, S_2, \dots$ such that S_0 is an initial state and for every $t \in \mathbb{N}$, $S_{t+1} = \tau_A(S_t)$.

The states occurring in an execution are said *accessible*. We do not assume that every state is accessible [6]: $S(A)$ might be easily defined (*e.g.* the natural numbers) but the set of the accessible states may not (*e.g.* the prime numbers if A is the sieve of Eratosthenes with infinite executions).

As for sequential algorithms [6], instead of defining a set of *final states* for the BSP algorithms, we will say that a state S_t of an execution is *final* if $\tau_A(S_t) = S_t$ and thus encoded the "output" of the algorithm. Indeed, in that case the execution $\overset{-\to}{S}$ is $S_0, S_1, \dots, S_{t-1}, S_t, S_t, \dots$. So, from an external point of view, the execution will seem to have stopped. We will say that an execution is *terminal* if it contains a final state. The *duration* is defined by:

$$\text{time}(A, S_0) \overset{\text{def}}{=} \begin{cases} \min \left\{ t \in \mathbb{N} \mid \tau_A^t(S_0) = \tau_A^{t+1}(S_0) \right\} & \text{if the execution is terminal} \\ \infty & \text{otherwise} \end{cases}$$

We follow [6] in which *states*, as *first-order structures*, are full instantaneous descriptions of an algorithm. Using structures has also the advantage to sufficiently abstract algorithms (*to be at the right level of abstraction* [6]); every algorithm can access *elementary operations* that could be executed. For instance, the Euclidean algorithm is different by using a native division or by simulating it with subtractions. In the same manner, a BSP algorithm could be different whether a broadcasting primitive is available or is simulated by lower-level communicating primitives such as point-to-point sending of data. These elementary operations are called *primitives* and depend only on the architecture.[13] Our work is thus independent of a given architecture, in the spirit of a bridging model. We now first define structures with the notion of interpretation [6] and then the execution of algorithms.

Definition 2 *(Structure [6]).* A (first-order) structure X is given by:

(1) A (potentially infinite) set $\mathcal{U}(X)$ called the universe of X;
(2) A finite set of function symbols $\mathcal{L}(X)$ called the signature of X;
(3) For every symbol $s \in \mathcal{L}(X)$ an interpretation $\overline{s}^X$ such that:
   (a) If c has arity 0 then $\overline{c}^X$ is an element of $\mathcal{U}(X)$;
   (b) If f has an arity $\alpha > 0$ then $\overline{f}^X$ is a function: $\mathcal{U}(X)^\alpha \to \mathcal{U}(X)$.

A BSP algorithm works on independent and uniform units. Therefore, a state S_t of the algorithm A must be a tuple $(X_t^1, \dots, X_t^p)$. To simplify, we annotate tuples from 1 to p and not from 0 to $p-1$. Notice that p is not fixed for the algorithm, so *A can have states using different size of "p-tuples"* (informally p is the number of units). In this work, we will simply consider that *this number is preserved during a particular execution*. In other words, the size of the p-tuples is fixed for an execution by the initial state of A for such an execution.[14] If $(X^1, \dots, X^p)$ is a state of the algorithm A, then the structures

[13] These functions are intrinsically oracular in order to be independent of the considered machine architectures; for instance, to perform an integer addition, the physical machine can use whatever implementation, only the functional result matters.

[14] The two others BSP parameters g and L can also be fixed by the initial states as static symbols in the structures; and two initial states can have different parameters, that models two executions on two different BSP machines.

$X^1, \dots, X^p$ will be called *processors* or *local memories*. The set of the homogeneous local memories (seen as independent) of A will be denoted by $M(A)$. The computation for every processor would be done in parallel, step-by-step and on terms:

Definition 3 *(Term [6]).* A term θ of $\mathcal{L}(X)$ is defined by induction: (1) If c has arity 0, then c is a term; (2) If f has an arity $\alpha > 0$ and $\theta_1, \dots, \theta_\alpha$ are terms, then $f(\theta_1, \dots, \theta_\alpha)$ is a term. The interpretation $\overline{\theta}^X$ of a term θ in a structure X is defined by induction:

(1) If $\theta = c$ is a constant symbol, then $\overline{\theta}^X \overset{\text{def}}{=} \overline{c}^X$;
(2) If $\theta = f(\theta_1, \dots, \theta_\alpha)$ where f is a symbol of the language $\mathcal{L}(X)$ with arity $\alpha > 0$ and $\theta_1, \dots, \theta_\alpha$ are terms, then $\overline{\theta}^X \overset{\text{def}}{=} \overline{f}^X(\overline{\theta_1}^X, \dots, \overline{\theta_\alpha}^X)$.

In order to have a uniform [6] (and homogeneous) presentation, we consider constant symbols in $\mathcal{L}(X)$ as 0-ary function symbols, and relation symbols R as their indicator function χ_R. Therefore, every symbol in $\mathcal{L}(X)$ denotes a function. Moreover, partial functions can be implemented with a special symbol **undef**, and we assume in this paper that every $\mathcal{L}(X)$ contains at least the boolean primitives (**true**, **false**, $\neg$ and $\wedge$) and the equality $=$. The symbols of the signature $\mathcal{L}(X)$ are distinguished between [6]:

(1) $\text{Dyn}(X)$ the set of *dynamic symbols*, whose interpretation can change during an execution, like a variable[15] x;
(2) $\text{Stat}(X)$ the set of *static symbols*, which have a fixed interpretation during an execution. They are also distinguished between [6]:
   (a) $\text{Init}(X)$, the set of *parameters*, whose interpretation depends only on the initial state, like an array in a sorting algorithm; the symbols depending on the initial state are the dynamic symbols and the parameters, so we call them the *inputs*;
   The other symbols have a uniform interpretation in every state (up to isomorphism, see the Definition 5 p. 8), and they are also distinguished between:
   (b) $\text{Cons}(X)$ the set of *constructors* (*e.g.* **true** and **false** for the booleans, 0 and **S** (successor) for the unary integers, *etc.*);
   (c) $\text{Oper}(X)$ the set of *operations* (*e.g.* $\neg$ and $\wedge$ for the booleans, $+$ and $\times$ for the integers, *etc.*).

We assume that every element $a \neq \overline{\textbf{undef}}^X$ of the universe $\mathcal{U}(X)$ is representable, which means that there exists a unique term θ_a formed only by constructors such that $\overline{\theta_a}^X = a$. This θ_a is called the *representation* of a. This can be proven for every usual data structure [12]. For instance, the binary integers use a different copy of $\mathbb{N}$ than the decimal integers. In other words $\overline{\underline{314}_{10}}^X \neq \overline{\underline{100111010}_2}^X$ but of course there is a bijection between the two copies. The *size* of an element is the length of its representation, in other words the number of constructors necessary to write it. For instance $|\overline{\underline{314}_{10}}^X| = 3$ and $|\overline{\underline{100111010}_2}^X| = 9$. These sizes will be used in SubSection 2.7.3 p. 25 to construct an example of a communication function and its exploration witness.

Example 1 *(Booleans $\mathbb{B}$).* The constructors are **true** and **false**, interpreted by two distinct values $\overline{\textbf{true}}^X$ and $\overline{\textbf{false}}^X$. For instance, the interpretation of the operation $\neg$ can be defined by $\overline{\neg}^X(\overline{\textbf{true}}^X) \overset{\text{def}}{=} \overline{\textbf{false}}^X$ and $\overline{\neg}^X(\overline{\textbf{false}}^X) \overset{\text{def}}{=} \overline{\textbf{true}}^X$. The interpretation of other logical connectives can be defined in the same way.

Example 2 *(Unary integers $\mathbb{N}_1$).* The constructors are the constant symbol $\underline{0}$ and the unary symbol S, interpreted respectively by 0 and $n \mapsto n+1$. Therefore, the terms have the form $S^n\underline{0}$, and are interpreted by $\overline{S^n\underline{0}}^X = n$ (thus which has size $n+1$). Operations can be added to the integer data structures (e.g. the addition $+$ or the multiplication $\times$), depending of the elementary operations considered for the algorithm.

Definition 4 *(Formula [6]).* A formula F is a term with a particular form: $F \overset{\text{def}}{=} \textbf{true} \mid \textbf{false} \mid R(\theta_1, \dots, \theta_\alpha) \mid \neg F \mid (F_1 \wedge F_2)$ where R is a relation symbol (a function with output $\overline{\textbf{true}}^X$ or $\overline{\textbf{false}}^X$), and $\theta_1, \dots, \theta_\alpha$ are terms. We say that a formula is true (resp. false) in X if $\overline{F}^X = \overline{\textbf{true}}^X$ (resp. $\overline{\textbf{false}}^X$).

We are interested in algorithms and not a particular implementation (*e.g.*, the variables' names). Therefore we will consider states up to *isomorphism*:

[15] We assume there is no logical variables and every term is closed, so in this paper a "variable" will be only a dynamical symbol with arity 0.

Definition 5 *(Isomorphism [6]).* Let X and Y be two structures with the same signature $\mathcal{L}$, and let $\zeta : \mathcal{U}(X) \to \mathcal{U}(Y)$ be a function. ζ is an isomorphism between X and Y if: (1) ζ is surjective; (2) For every symbol $c \in \mathcal{L}$ with arity 0, $\zeta(\overline{c}^X) = \overline{c}^Y$; (3) For every $f \in \mathcal{L}$ with arity $\alpha > 0$, and for every $a_1, \dots, a_\alpha \in \mathcal{U}(X)$, $\zeta(\overline{f}^X(a_1, \dots, a_\alpha)) = \overline{f}^Y(\zeta(a_1), \dots, \zeta(a_\alpha))$.

We say that two structures X and Y are isomorphic if there exists an isomorphism between them. Notice that if ζ is an isomorphism between X and Y, then $\zeta(\overline{\theta}^X) = \overline{\theta}^Y$. And because we assumed that every signature contains the equality symbol then every isomorphism is injective. Therefore, we have that $a = b$ in X if and only if $\zeta(a) = \zeta(b)$ in Y. In particular, a formula F is **true** (resp. **false**) in X if and only if F is **true** (resp. **false**) in Y. Moreover, because ζ is injective and by definition surjective, it is bijective. In particular we can introduce ζ^{-1}, which is also an isomorphism.

Definition 6 *(Replacement in a structure).* Let X be a structure, and let $U_1 \subseteq \mathcal{U}(X)$ and U_2 be two sets such that there exists a bijection φ from U_1 to U_2. The structure Y obtained by replacing in X the elements of U_1 by the elements of U_2 is defined by: (1) $\mathcal{L}(Y) = \mathcal{L}(X)$; (2) $\mathcal{U}(Y) = (\mathcal{U}(X) \backslash U_1) \cup U_2$; (3) If $c \in \mathcal{L}(X)$ is a 0-ary symbol then:

$$\overline{c}^Y \overset{\text{def}}{=} \begin{cases} \varphi(\overline{c}^X) & \text{if } \overline{c}^X \in U_1 \\ \overline{c}^X & \text{otherwise} \end{cases}$$

If $f \in \mathcal{L}(X)$ is a α-ary symbol with $\alpha > 0$, and $a_1, \dots, a_\alpha \in \mathcal{U}(Y)$, then:

$$\overline{f}^Y(a_1, \dots, a_\alpha) \overset{\text{def}}{=} \begin{cases} \varphi\left(\overline{f}^X(a'_1, \dots, a'_\alpha)\right) & \text{if } \overline{f}^X(a'_1, \dots, a'_\alpha) \in U_1 \\ \overline{f}^X(a'_1, \dots, a'_\alpha) & \text{otherwise} \end{cases}$$

$$\text{where } a' \overset{\text{def}}{=} \begin{cases} \varphi^{-1}(a) & \text{if } a \in U_2 \\ a & \text{otherwise} \end{cases}$$

A replacement in a structure is a structure. Moreover, it has been proved in [12] that the replacement is an isomorphism, which we will use in Lemma 1 p. 9 and Lemma 2 p. 17. Finally, because BSP algorithms manipulates states as p-tuples of structures (one per processor), we easily extend the notion of isomorphisms to tuples of structures:

Definition 7 *(Multi-Isomorphism).* $\overrightarrow{\zeta}$ is a multi-isomorphism between two states $(X^1, \dots, X^p)$ and $(Y^1, \dots, Y^q)$ if $p = q$ and $\overrightarrow{\zeta}$ is a p-tuple of functions $\zeta_1, \dots, \zeta_p$ such that for every $1 \leqslant i \leqslant p$, ζ_i is an isomorphism between X^i and Y^i.

Postulate 2 *(Abstract states). For every* BSP *algorithm A:*

(1) *The states of A are p-tuples of structures with the same finite signature $\mathcal{L}(A)$ (containing the booleans and the equality);*
(2) *$S(A)$ and $I(A)$ are closed by multi-isomorphism;*
(3) *The transition function τ_A preserves p, the universes and commutes with multi-isomorphisms.*

For a BSP algorithm A, let X be a local memory of A, $f \in \mathcal{L}(A)$ be a dynamic α-ary function symbol, and $a_1, \dots, a_\alpha, b$ be elements of the universe $\mathcal{U}(X)$. We say that $(f, a_1, \dots, a_\alpha)$ is a location of X, and that $(f, a_1, \dots, a_\alpha, b)$ is an *update* [6] on X at the location $(f, a_1, \dots, a_\alpha)$. For instance, if x is a variable then $(x, 42)$ is an update at the location x. But symbols with arity $\alpha > 0$ can be updated too. For instance, if f is a one-dimensional array, then $(f, 0, 42)$ is an update at the location $(f, 0)$. If u is an update then $X \oplus u$ is a new structure of signature $\mathcal{L}(A)$ and universe $\mathcal{U}(X)$ such that the interpretation of a function symbol $f \in \mathcal{L}(A)$ is:

$$\overline{f}^{X \oplus u}(\overrightarrow{a}) \overset{\text{def}}{=} \begin{cases} b & \text{if } u = (f, \overrightarrow{a}, b) \\ \overline{f}^X(\overrightarrow{a}) & \text{otherwise} \end{cases}$$

where we noted $\overrightarrow{a} = a_1, \dots, a_\alpha$. For instance, in $X \oplus (f, 0, 42)$, every symbol has the same interpretation as in X, except maybe for f because $\overline{f}^{X \oplus (f,0,42)}(0) = 42$ and $\overline{f}^{X \oplus (f,0,42)}(a) = \overline{f}^X(a)$ otherwise. We precised "maybe" because it may be possible that $\overline{f}^X(0)$ is already 42. If $\overline{f}^X(\overrightarrow{a}) = b$ then the update $(f, \overrightarrow{a}, b)$ is said *trivial* in X, because nothing has changed. Indeed, if $(f, \overrightarrow{a}, b)$ is trivial in X then $X \oplus (f, \overrightarrow{a}, b) = X$.

If Δ is a set of updates then Δ is *consistent*[6] if it does not contain two distinct updates with the same location. Notice that if Δ is inconsistent, then there exists $(f, \overrightarrow{a}, b), (f, \overrightarrow{a}, b') \in \Delta$ with $b \neq b'$ and, in that case, the entire set of updates clashes [6]:

$$\overline{f}^{X \oplus \Delta}(\overrightarrow{a}) \overset{\text{def}}{=} \begin{cases} b & \text{if } (f, \overrightarrow{a}, b) \in \Delta \text{ and } \Delta \text{ is consistent} \\ \overline{f}^X(\overrightarrow{a}) & \text{otherwise} \end{cases}$$

If X and Y are two local memories of the same algorithm A then there exists a unique consistent set $\Delta = \{(f, \overrightarrow{a}, b) \mid \overline{f}^Y(\overrightarrow{a}) = b \text{ and } \overline{f}^X(\overrightarrow{a}) \neq b\}$ of non trivial updates such that $Y = X \oplus \Delta$ [6]. This Δ is called the *difference* between the two structures (local memories), and is denoted by $Y \ominus X$.

Let $\overrightarrow{X} = (X^1, \dots, X^p)$ be a state of A. According to the transition function τ_A, the next state is $\tau_A(\overrightarrow{X})$, which will be denoted by $(\tau_A(\overrightarrow{X})^1, \dots, \tau_A(\overrightarrow{X})^p)$. We denote by $\Delta^i(A, \overrightarrow{X}) \overset{\text{def}}{=} \tau_A(\overrightarrow{X})^i \ominus X^i$ the set of updates done by the i-th processor of A on the state $\overrightarrow{X}$, and by $\overrightarrow{\Delta}(A, \overrightarrow{X}) \overset{\text{def}}{=} (\Delta^1(A, \overrightarrow{X}), \dots, \Delta^p(A, \overrightarrow{X}))$ the "multiset" of updates done by A on the state $\overrightarrow{X}$. In particular, if a state $\overrightarrow{X}$ is final, then $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$, so $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \overrightarrow{\emptyset}$. More generally, we can extend the notation $\ominus$ to tuples $(Y^1, \dots, Y^p) \ominus (X^1, \dots, X^p) \overset{\text{def}}{=} (Y^1 \ominus X^1, \dots, Y^p \ominus X^p)$. Therefore $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \tau_A(\overrightarrow{X}) \ominus \overrightarrow{X}$.

Let A be a BSP algorithm and T be a set of terms of $\mathcal{L}(A)$. We say that two states $(X^1, \dots, X^p)$ and $(Y^1, \dots, Y^q)$ of A *coincide over* T if $p = q$ and for every $1 \leqslant i \leqslant p$ and for every $\theta \in T$ we have $\overline{\theta}^{X^i} = \overline{\theta}^{Y^i}$.

Postulate 3 *(Bounded exploration for processors). For every BSP algorithm A there exists a finite set $T(A)$ of terms (closed by subterms) such that for every state $\overrightarrow{X}$ and $\overrightarrow{Y}$, if they coincide over $T(A)$ then $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \overrightarrow{\Delta}(A, \overrightarrow{Y})$, i.e. for every $1 \leqslant i \leqslant p$, we have $\Delta^i(A, \overrightarrow{X}) = \Delta^i(A, \overrightarrow{Y})$.*

$T(A)$ is called the *exploration witness* [6] of A. Some typical examples can be found in [6,47].[16] If a set of terms T is finite then its closure by subterms is finite too. We assume that $T(A)$ is closed by subterms and [6] contains the boolean symbol **true**. An example of $T(A)$ is given in Example 3 p. 10. The interpretations of the terms in $T(A)$ are called the *critical elements* and we can prove that every value in an update is a critical element:

Lemma 1 *(Critical elements). For every state $(X^1, \dots, X^p)$ of A, $\forall i$ $1 \leqslant i \leqslant p$, if $(f, \overrightarrow{a}, b) \in \Delta^i(A, \overrightarrow{X})$ then $\overrightarrow{a}, b$ are interpretations in X^i of terms in $T(A)$.*

Proof (Sketch of). The proof is similar to [6]. We assume by contradiction that one interpretation a_i is not an interpretation in X^i of a term in $T(A)$. By replacing it (see Definition 6 p. 8) by a fresh value v we obtain a state that coincide over $T(A)$. Thus, by the third postulate, v should appear in $\Delta^i(A, \overrightarrow{X})$, which contradicts that v is fresh. □

That implies that for every step of the computation, for a given processor, only a bounded (thus finite) number of terms are read or written, thus that a bounded *amount of work* is done at every step:

Lemma 2 *(Bounded set of updates). For every state $(X^1, \dots, X^p)$ of the algorithm A, for every $1 \leqslant i \leqslant p$, $card(\Delta^i(A, \overrightarrow{X}))$ (the cardinality) is bounded.*

Proof (Sketch of). According to the Lemma 1, if $(f, a_1, \dots, a_\alpha, b) \in \Delta^i(A, \overrightarrow{X})$ then $a_1, \dots, a_\alpha, b$ are critical elements. But, according to the third postulate, $T(A)$ is finite. So the number of possible $a_1, \dots, a_\alpha, b$ in $\Delta^i(A, \overrightarrow{X})$ is bounded. Moreover, because $\mathcal{L}(A)$ is finite there exists a bounded number of dynamic symbols f. Therefore $\Delta^i(A, \overrightarrow{X})$ has a bounded number of updates. □

We now organize the sequence of states into *supersteps*. The communication between local memories occurs only during a communication phase. In order to do so, a BSP algorithm A will use two functions $\mathbf{compu}_A$ and $\mathbf{comm}_A$ indicating if A runs computations or communications:

Postulate 4 *(Supersteps phases). For every BSP algorithm A there exists two functions $\mathbf{compu}_A : M(A) \to M(A)$ commuting with isomorphisms, and $\mathbf{comm}_A : S(A) \to S(A)$, such that for every state $(X^1, \dots, X^p)$:*

$$\tau_A\left(X^1, \dots, X^p\right) = \begin{cases} (\mathbf{compu}_A(X^1), \dots, \mathbf{compu}_A(X^p)) & \text{if } \exists 1 \leqslant i \leqslant p \text{ such that } \mathbf{compu}_A(X^i) \neq X^i \\ \mathbf{comm}_A\left(X^1, \dots, X^p\right) & \text{otherwise} \end{cases}$$

A BSP *algorithm* is an object verifying these four postulates, and we denote by ALGO$_{\text{BSP}}$ the set of the BSP algorithms. A state $(X^1, \dots, X^p)$ will be said in a *computation phase* if there exists $1 \leqslant i \leqslant p$ such that $\mathbf{compu}_A(X^i) \neq X^i$. Otherwise, the state will be said in a *communication phase*.

Because a state $\overrightarrow{X}$ is final if $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$, according to the fourth postulate, $\overrightarrow{X}$ must be in a communication phase which is like a final phase that would terminate the whole execution as found in MPI. More discussions about the postulates could be find in Q7-Q18 p. 14.

[16] We insist that such exploration witnesses are used to prove that the execution is done step-by-step, but not that the overall execution is bounded or finite (it is not a kind of variant or measurement as in the Hoare logic). Step-by-step means that at every step, only a bounded number of terms are manipulated, not a growing number of terms, making any machine/algorithm unrealistic (and thus opposed to the bridging model approach).

This still requires some remarks. Firstly, at every computation step, every processor which has not terminated performs its local computations. Secondly, we did not specify the function $\mathbf{comm}_A$ in order to be generic about which BSP library is used. We discuss in Section 2.7.4 p. 26 the difference between $\mathbf{comm}_A$ and the usual communication routines in the BSP community. Thirdly, during a computation phase, if a processor has finished its computations, the processor "waits" for the communication phase, that corresponds to a possible load-balancing overhead of the algorithm. Nevertheless, during a communication phase, the communication function can force a processor to send new "data"; this situation can appear when a processor reads a value on another processor. More discussions about the function of communication are available in Q48-Q59 p. 30.

Fourthly, in this postulate we did not distinguish a communication step from a synchronization one, because both are globally done by the function $\mathbf{comm}_A$. A superstep is a phase of computation (made of computation steps) followed by a phase of communication (made of communication steps the last one being the barrier). The whole execution is thus a sequence of supersteps, and the number of supersteps is the same as in the BSP model. Finally, because a processor which has finished its computations steps waits for the other to finish, before the communication steps are completed globally, we have the intended cost model for BSP. More discussions about the function of communication could be find in Q4-Q6 p. 12 and in Q41-Q43 p. 23.

Example 3 *(Prefix computation).* Our example is a common logarithmic prefix computation that is, for an associative $\uplus$ operator, calculating, on each processor pid, $\sum_{i=1}^{pid} x_i = x_1 \uplus \cdots \uplus x_{pid}$ where each x_i is a value belonging to a processor i and where p is the number of processors. In a (informal) pseudo-code (where $\uplus$ is the interpretation of `op`) in a SPMD-like manner (single Program Multiple Data[17]), the algorithm can be written as:

```
Input: int x;
Output: r;
Init: int j:=0, r:=x, y:=undef;
  while (2^j < p) do
      if (pid > 2^j) then
         read(y,r,pid-2^j);
         Comm();
         r:=y op r;
      else
         Comm();
    endif;
    j:=j+1
  endwhile
```

In this example, the `read` elementary function allows the processor `pid` to perform a distant reading of the value of `r` in the processor `pid-2^j`, which will be done (and the value written in `y`) during the execution of the `Comm` function only. Each processor gets its final result in `r`. The parameter `pid` is interpreted as the value of the processor id (an integer between 1 and p). `op` is $\uplus$, and the other boolean or arithmetical constructors/operations are interpreted as usual. We also use an auxiliary boolean variable b, which is true during the communication phase, and such that the computation phase can begin only if b becomes false.

In this example, the local memories can be seen as tuples $(j_i, r_i, x_i, y_i, b_i)$ of the values for the variables $j_i = \overline{j}^{X^i}$, etc. The initial states are such that for every $i \in \{1, \cdots, p\}$, we have $j_i = 0$, $r_i = x_i$, $y_i = \mathbf{undef}$ and $b_i = \mathbf{true}$ (we force an empty phase of computation to run immediately the necessary communications), and they are distinguished from each other only by the number p of processors and the values x_i.
A transition step of this algorithm can be seen as a function $(j_1, r_1, x_1, y_1, b_1) \cdots (j_p, r_p, x_p, y_p, b_p) \mapsto (j'_1, r'_1, x_1, y'_1, b'_1) \cdots (j'_p, r'_p, x_p, y'_p, b'_p)$ such that for every $1 \leqslant i \leqslant p$, if $b = \mathbf{true}$ then $\mathbf{comm}_A$ makes the updates $b'_i = \mathbf{false}$ and (if $pid > 2^{j_i}$) $y'_i = r_{pid-2^{j_i}}$. Otherwise, $\mathbf{compu}_A$ makes the updates $j'_i = j_i + 1$, $b'_i = \mathbf{true}$ and (if $pid > 2^{j_i}$) $r'_i = y'_i \uplus r_i$. The transition stops when $2^j \geqslant p$.

Finally, by assuming a uniform interpretation of the usual constructors and operations, $T(A) = \{\mathbf{true}, p, pid, j, r, x, y, b\}$ is an exploration witness. Notice that in this example, we did not use any explicit data transfers (only the primitive `read`). We will explain in SubSection 2.7 p. 24 how to make them explicit.

We now prove that the set of BSP algorithms satisfies, during a computation phase, that every processor computes independently of the state of the other processors; there is thus a strict separation of these two phases. To do this, we first prove some lemma about the computation phase and then the communication function:

[17] Which means that the program is executed individually on each processor.

Lemma 3 *(Computing states are closed by multi-isomorphism). If the state $\left(X^1, \dots, X^p\right)$ is in a computing phase and multi-isomorphic to the state $\left(Y^1, \dots, Y^p\right)$, then $\left(Y^1, \dots, Y^p\right)$ is in a computing phase too.*

Proof (Sketch of). We assume by contradiction that for every $1 \leqslant i \leqslant p$ we have $\mathbf{compu}_A(Y^i) = Y^i$. The transition function (second postulate) and the computation function (fourth postulate) commute with multi-isomorphisms, so $\overrightarrow{\zeta}\left(\tau_A\left(X^1, \dots, X^p\right)\right) = \overrightarrow{\zeta}\left(X^1, \dots, X^p\right)$. By applying $\overrightarrow{\zeta}^{-1}$ on both sides, we have for every $1 \leqslant i \leqslant p$ that $\mathbf{compu}_A(X^i) = X^i$, which contradicts that $\left(X^1, \dots, X^p\right)$ is in a computing phase. □

We did not assume in the fourth postulate that the communication function commutes with multi-isomorphisms, because this is a corollary of the previous lemma and the second postulate:

Corollary 1 *(Properties of the communication). For every* BSP *algorithm A and for every state in a communication phase, $\mathbf{comm}_A$ preserves the universes, the number of processors, and commutes with multi-isomorphisms.*

Proof (Sketch of). Let $\overrightarrow{X}$ be a state in a communication phase, so (fourth postulate) $\mathbf{comm}_A(\overrightarrow{X}) = \tau_A(\overrightarrow{X})$, thus (second postulate) $\mathbf{comm}_A$ preserves the universes and the number of processors. Moreover, according to Lemma 3, $\overrightarrow{\zeta}(\overrightarrow{X})$ is in a communication phase too, so (second and fourth postulate) $\mathbf{comm}_A\left(\overrightarrow{\zeta}(\overrightarrow{X})\right) = \overrightarrow{\zeta}\left(\mathbf{comm}_A(\overrightarrow{X})\right)$. □

According to the third postulate, only the critical elements of the local memories matter to compute a step of the execution. So, we will assume[18] that $S(A)$ is *closed with respect to the exploration witness*, which means that if $\left(X^1, \dots, X^i, \dots, X^p\right)$ is a state and the structure Y^i has the same signature and critical elements as X^i, then $\left(X^1, \dots, Y^i, \dots, X^p\right)$ is a state too.

Proposition 1 *(No communication during computation phases). For every state $(X^1, \dots, X^p)$ and $(Y^1, \dots, Y^q)$ in a computation phase, if X^i and Y^j have the same critical elements then $\Delta^i(A, \overrightarrow{X}) = \Delta^j(A, \overrightarrow{Y})$.*

Proof (Sketch of). Because $\overrightarrow{X}$ and $\overrightarrow{Y}$ are in a computing phase, $\Delta^i(A, \overrightarrow{X}) = \mathbf{compu}_A(X^i) \ominus X^i$ and $\Delta^j(A, \overrightarrow{Y}) = \mathbf{compu}_A(Y^j) \ominus Y^j$. The proof is made by case: if $\mathbf{compu}_A(X^i) = X^i$ and $\mathbf{compu}_A(Y^j) = Y^j$, then $\Delta^i(A, \overrightarrow{X}) = \varnothing = \Delta^j(A, \overrightarrow{Y})$. Otherwise, we assume $\mathbf{compu}_A(Y^j) \neq Y^j$ (the other case being similar). Let $\overrightarrow{Z}$ be the p-tuple $\overrightarrow{X}$ where X^i has been replaced by Y^j. X^i and Y^j have the same critical elements and $S(A)$ is closed with respect to the exploration witness, so $\overrightarrow{Z}$ is a state. $\overrightarrow{X}$ and $\overrightarrow{Z}$ coincide over $T(A)$, so (third postulate) $\Delta^i(A, \overrightarrow{X}) = \Delta^i(A, \overrightarrow{Z})$. Moreover, because $\mathbf{compu}_A(Y^j) \neq Y^j$, $\overrightarrow{Z}$ is in a computing phase, so $\Delta^i(A, \overrightarrow{Z}) = \mathbf{compu}_A(Y^j) \ominus Y^j = \Delta^j(A, \overrightarrow{Y})$. □

Finally, notice that our postulates are a "natural" extension of those in [6]:

Lemma 4 *(A single processor is sequential). A* BSP *algorithm with a unique processor ($p = 1$) is a sequential algorithm. Thus,* $\text{ALGO}_{\text{SEQ}} \subseteq \text{ALGO}_{\text{BSP}}$.

Proof. By assuming $p = 1$ then our first three postulates are the same as those in [6]. For the last postulate, we assume that $\mathbf{comm}_A$ is the identity. □

2.3. Questions and answers

As in [22,47,48], we present the discussion about our results by using a dialog between the authors and a scrupulous colleague. We discuss also other related works that have guided our choices (with *pros and cons*). These parts are a bit long because our approach differs from the previous ones. We will sometimes refer directly to these discussions in order to make the reading more fluent.

Question 1: *Hello. I understand that you want to formally characterize subclasses of distributed algorithms (*HPC *context) but* BSP *seems a naive model of computation and not accurate for* HPC*. Why did you choose such a model?*

Yes, sometimes the BSP model is useless. For example, for irregular data-structures computations, being restricted by the supersteps may force to use complex heuristics; which is not as natural as spawning small processes of calculation. However, the ability of using the BSP's model of performance (cost model) for algorithmic study is the most important advantage. And this cost model comes from its structured execution model (the sequence of supersteps). Also, in the subdomain of HPC that is big-data, BSP is one of the most widely used model as the development of PREGEL-like frameworks [26] shows. By the

[18] This assumption is without cost, because we can construct the algorithm B with $S(B) \supseteq S(A)$ and prove results like the Lemma 2 p. 17 for B, then apply the result for the restricted set of states $S(A)$.

way, the *pros and cons* of the BSP bridging model are not the topic of this work and are largely discussed in [21,22] and on more recent publications.

Q2: *OK, for the underlying model of computation, but why did you choose this definition of algorithms (clearly inspired from [6]) and not another one?*

For this work, we chose the approach of [6] to formally describe the (small-steps, discrete time) algorithms and not the recursive equations of [7,8] because instead of claiming that a particular model of computation (the ASMs or the recursors) captures the behavior of the algorithms, the approach of [6] require only a few convincing postulates. There is also the model of [9], in which the authors define algorithms as a strategy and a tree of all the possible executions of elementary instructions (the events) on what is called "concrete data structures" or CDS (the values), with a denotational semantics. But notice that, unfortunately, there is no proof of algorithmic equivalence between the aforementioned models. We leave their study for the BSP framework to future work. There is also the category of algorithms of [50] but that is currently limited to primitive recursive equations.

Q3: *So why not simply use a BSP-Turing machine to define an algorithm?*

It is known that one-tape Turing machines could simulate every computable function. But in this paper we are interested in the step-by-step behavior of the algorithms, and not the input-output relation of the functions. Moreover, simulate algorithms by using a Turing-machine is a low-level approach, which does not describe the algorithm at its natural level of abstraction. Every algorithm assumes elementary operations which are not refined down to the assembly language by the algorithm itself. These operations are seen as oracular, which means that they produce the desired output in one step of computation.

Q4: *From a programming point of view, I think there is too many abstractions. For instance, when using BSPLIB, messages received at the past superstep are dropped, which behavior is not reflected by your function* $\mathbf{comm}_A$.

We want to be as general as possible. Perhaps a future library would allow reading data received n supersteps ago as in the BSP+ model of [31]. Moreover, the communication function, as an "elementary function", may realize some computations and is thus not a pure transmission of data. But the exploration witness prevents it to do everything: only a finite set of symbols can be updated. And we provide a realistic example of such a function which mainly corresponds to the BSPLIB's DRMA primitives (SubSection 2.7.4 p. 26).

Q5: *And so why is it not just a permutation of values to be exchanged?*

The communications can be used to model synchronous interactions with the environment (input/output or error messages, *etc.*) and therefore make appear or disappear values. Another example is the BSP-PRAM model of [31] where a sending message of a superstep can be received several supersteps after (which is only possible, using the BSPLIB, by keeping this kind of messages in specific buffers). The function of communication is another elementary function such as performing arithmetic operations (on binary or unary integers). Our result is independent of these functions and it is the user that defines its level of abstraction (the structures and their elementary functions, and on which machine they can be used).

Q6: *When using BSPLIB and other BSP libraries, I can switch between sequential computations and BSP ones. Why not model this kind of feature?*

The sequential parts can be modeled as purely asynchronous computations replicated and performed by all the processors. Or, one processor (typically the first one) is performing these computations while other processors are "waiting" with an empty computation phase.

Q7: *Why are you using (multi-)isomorphisms?*

Mainly because we are interested in the actual properties of the models of computation, not the naming conventions used for them. Interesting discussions of about isomorphisms are given in [47]. Moreover, this is required for the proofs as in [6], to prove by using the second postulate that a multi-isomorphic state is also a state, and thus be able to use the third postulate.

Q8: *Fine. But are you sure about your postulates? I mean, do they completely define BSP algorithms? Or not more?*

It is impossible to be sure because we are formalizing a concept that is currently only intuitive. But because our postulates are as general and simple as possible, we believe that they correctly capture the intuitive class of BSP algorithms. By the way, as in [6], we prove in the next section that our axiomatic presentation is identical to an operational presentation.

Q9: *So there is no hidden limitation, everything is alright?*

Of course not. As for sequential ASMs, recursive algorithms are not well characterized (as explain in [7]); that could be a problem if we want to characterize a hierarchical extension of BSP [33] where the overall machine is a tree of recursive BSP machines (different solutions seems possible but none strongly convinced us). The exploration witness is also a constraint (see SubSection 2.7 p. 24) but necessary for the results. Our work is also abstract and so any implementation can end up being difficult.

Q10: *Speaking abstraction and from a pragmatic point of view, it seems sufficient to be limited to a maximal number of processors, e.g.* $p \leqslant 2^{256}$*. So why considering an arbitrary number of processors in your results?*

When you are designing an algorithm (in your mind), you do not think to a limit (an arbitrary choice) number of processors and therefore you would prefer your algorithm to be defined for any number of processors (even with a property to this number such as to be a power of two) and thus for any BSP machine, in the present and in the future (notion of "*immortal* algorithms").

Q11: *But in [15,18,19,32], the authors give more general postulates about concurrent and/or distributed algorithms. For example, they also assume a infinite set as input (a set of processes that are pairs of agent name associate with a communicating sequential algorithm). Why not using their works by adding some restrictions to take into account the BSP model of execution?*

This is another solution. But we think that *ad hoc* restrictions on more general (and complex) postulates is not an appropriate approach to characterize the class of BSP algorithms by themselves. A model is better expressed at its natural level of abstraction, in order to highlight its particular properties (*e.g.* specific subclasses such that communications-oblivious algorithms [31]). For instance, the important topic of the cost model is inherent to a bridging model like BSP, and it is not clear how such restrictions could highlight this cost model. But your remark is valid, and we will continue to compare our work with their approach later.

Q12: *Speaking of the processors, why did you not formally define them? For instance, as a special abstract machine.*

This would amount to defining a BSP machine (center part of Fig. 2 p. 4) and not the BSP algorithms (right part of Fig. 2). In some descriptions of BSP, the performance parameter **r** does not even exist (this is also the case for the modern multi-BSP bridging model [33]): there is nothing to say about the processors since they are assumed to be homogeneous, only their number and "network" parameters are still relevant.

Q13: *About homogeneity, in Postulate 2 you assumed that every processor has the same signature* $\mathcal{L}(A)$*. But there are distributed algorithms for heterogeneous architectures, for instance where computing units may be CPUs or GPUs, which does not perform the same elementary operations.*

Indeed, in this paper we are not trying to capture all the distributed algorithms, only the BSP ones, and they are assumed to be homogeneous, which implies that they have the same elementary operations. For instance, you may think of a server farm with identical computing units.

Q14: *But even if your processors have the same signature, that does not imply that they have the same interpretation, nor even the same universe.*

Actually they cannot have the *same* universe, because (for instance) the digits manipulated by one processor cannot be the same digits manipulated by another processor, that would be shared memory.

Q15: *I wanted to mean different* occurrences *of the same universe, of course.*

OK, but from one processor to another the dynamical symbols (like the variables) and the parameters (*e.g.* the pid of the processor) cannot have the same value, or there would be no point for distributed computation.

Q16: *Indeed, but what about constructors and operations? Surely, to state that the processors are homogeneous require that the data structures are the same from one processor to another (up to isomorphism)?*

We agree that it would be a good definition. And in order to obtain such homogeneity, by using Postulate 2 you have to define the interpretation of the constructors and the operations with syntax-based equations as in the Examples 1 and 2 p. 7. For instance, every processor would work on its own copy of the binary integers. This path has been investigated in [12], but the homogeneity hypothesis is not actually required to prove our main contribution.

Q17: *So, in a way, your result is more effective than required, because you proved it for more than homogeneous processors and usual data structures.*

Indeed, but that means also that the presented results in this work will still hold even after the discovery of novel data structures.

Q18: *About novel objects, what do you think about subset synchronization?*

Subgroup synchronization [30] is a more complicated feature and has not been taken into account in this work. We know that it is relevant for modern HPC architectures such as clusters of multi-cores, and there exist specific bridging models (such as multi-BSP [33]) for it. Their execution models are more complex and largely different to BSP, and may lead to interesting future works.

Q19: *For now, you have just an abstract axiomatization of* BSP *algorithms. What is the link with actual programming?*

We will now follow [6] to provide an operational point of view with the notion of Abstract state Machines (ASMs). And in Section 3 p. 30, we will provide an imperative BSP programming language and common routines from the well-known BSPLIB (BSP programming in C) in SubSection 2.7 p. 24.

2.4. ASM-BSP captures the BSP algorithms

The four previous postulates define the BSP algorithms from an *axiomatic point of view* but that does not mean that they have a model [47] or, in other words, that they are defined from an *operational point of view*. In the same way that the model of computation ASM captures the set of sequential algorithms [6], we now prove that the ASM_{BSP} model captures the BSP algorithms.

2.4.1. Definition and operational semantics of ASM-BSP

Definition 8 *(ASM program [6]).*

$$\begin{aligned}\Pi \overset{\text{def}}{=}\ & f\,(\theta_1,\ldots,\theta_\alpha) := \theta_0\\ &|\ \textbf{if } F \textbf{ then } \Pi_1 \textbf{ else } \Pi_2 \textbf{ endif}\\ &|\ \textbf{par } \Pi_1 \,\|\cdots\|\, \Pi_n \textbf{ endpar}\end{aligned}$$

where f is a dynamical symbol with arity α; F is a formula; and $\theta_1, \ldots, \theta_\alpha, \theta_0$ are terms of the considered signature.

Notice that if $n = 0$ then **par** $\Pi_1\|\cdots\|\Pi_n$ **endpar** is the empty program. If in **if** F **then** Π_1 **else** Π_2 **endif** the program Π_2 is empty we will write simply **if** F **then** Π_1 **endif**. An ASM machine [6] is a kind of Turing machine using not a tape but an abstract structure X:

Definition 9 *(ASM operational semantics [6]).* The operational semantics $\Delta(\Pi, X)$ of an ASM program Π on a structure X is defined by induction as the following set of updates:

$$\begin{aligned}\Delta(f\,(\theta_1,\ldots,\theta_\alpha) := \theta_0, X) &\overset{\text{def}}{=} \left\{(f, \overline{\theta_1}^X, \ldots, \overline{\theta_\alpha}^X, \overline{\theta_0}^X)\right\}\\ \Delta(\textbf{if } F \textbf{ then } \Pi_1 \textbf{ else } \Pi_2 \textbf{ endif}, X) &\overset{\text{def}}{=} \Delta(\Pi_i, X)\\ &\quad\ \text{where}\begin{cases} i = 1 & \text{if } F \text{ is } \textbf{true} \text{ on } X\\ i = 2 & \text{otherwise}\end{cases}\\ \Delta(\textbf{par } \Pi_1 \,\|\ldots\|\, \Pi_n \textbf{ endpar}, X) &\overset{\text{def}}{=} \Delta(\Pi_1, X) \cup \cdots \cup \Delta(\Pi_n, X)\end{aligned}$$

Notice that the semantics of the `par` is a set of updates done simultaneously, which differs from an usual imperative framework. In order to construct an exploration witness for an ASM program Π, the set Read(Π) of the *read terms* and the set Write(Π) of the *written terms* are defined by induction:

```
if $(s = 0 \wedge 2^j < p)$ then par
                          if $2^j < pid$ then $\texttt{read}(y, r, pid - 2^j)$ endif
                        $\|$ $s := 1$
                        endpar
else
    if $(s = 2)$          then par
                          if $2^j < pid$ then $r := y \texttt{ op } r$ endif
                        $\|$ $j := j + 1$
                        $\|$ $s := 0$
                        endpar
endif endif
```

Fig. 4. An ASM program for the prefix computation.

Definition 10 *(Terms read/written by an ASM program).*

$$\begin{aligned}
\text{Read}\,(f\,(\theta_1, \ldots, \theta_\alpha) := \theta_0) &\overset{\text{def}}{=} \{\theta_1, \ldots, \theta_\alpha, \theta_0\} \\
\text{Read}\,(\textbf{if } F \textbf{ then } \Pi_1 \textbf{ else } \Pi_2 \textbf{ endif}) &\overset{\text{def}}{=} \{F\} \cup \text{Read}\,(\Pi_1) \cup \text{Read}\,(\Pi_2) \\
\text{Read}\,(\textbf{par } \Pi_1 \| \cdots \| \Pi_n \textbf{ endpar}) &\overset{\text{def}}{=} \text{Read}\,(\Pi_1) \cup \cdots \cup \text{Read}\,(\Pi_n)
\end{aligned}$$

$$\begin{aligned}
\text{Write}\,(f\,(\theta_1, \ldots, \theta_\alpha) := \theta_0) &\overset{\text{def}}{=} \{f\,(\theta_1, \ldots, \theta_\alpha)\} \\
\text{Write}\,(\textbf{if } F \textbf{ then } \Pi_1 \textbf{ else } \Pi_2 \textbf{ endif}) &\overset{\text{def}}{=} \text{Write}\,(\Pi_1) \cup \text{Write}\,(\Pi_2) \\
\text{Write}\,(\textbf{par } \Pi_1 \| \cdots \| \Pi_n \textbf{ endpar}) &\overset{\text{def}}{=} \text{Write}\,(\Pi_1) \cup \cdots \cup \text{Write}\,(\Pi_n)
\end{aligned}$$

As before, a state of an ASM$_{\text{BSP}}$ machine is a p-tuple of local memories $\overrightarrow{X} = (X^1, \ldots, X^p)$. We assume that ASM$_{\text{BSP}}$ programs work in a SPMD-like way, which means that at each step of computation the ASM$_{\text{BSP}}$ program Π is executed individually on each processor. Therefore Π induces a multiset of updates and a transition function τ_Π:

$$\begin{aligned}
\overrightarrow{\Delta}(\Pi, (X^1, \ldots, X^p)) &\overset{\text{def}}{=} (\Delta(\Pi, X^1), \ldots, \Delta(\Pi, X^p)) \\
\tau_\Pi\,(X^1, \ldots, X^p) &\overset{\text{def}}{=} (X^1 \oplus \Delta(\Pi, X^1), \ldots, X^p \oplus \Delta(\Pi, X^p))
\end{aligned}$$

We also extend the notation $\oplus$ to tuples by defining $\overrightarrow{X} \oplus \overrightarrow{\Delta}(\Pi, \overrightarrow{X}) = \tau_\Pi(\overrightarrow{X})$. Notice that we may have $\overrightarrow{X} \oplus \overrightarrow{\Delta}(\Pi, \overrightarrow{X}) = \overrightarrow{X}$ and $\overrightarrow{\Delta}(\Pi, \overrightarrow{X}) \neq \overrightarrow{\emptyset}$ if the updates are trivial. If $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$, then every processor has finished its computation steps. In that case we assume that there exists a communication function to ensure the communications between processors:

Definition 11. An ASM$_{\text{BSP}}$ machine M is a triplet $(S(M), I(M), \tau_M)$ such that:

(1) $S(M)$ is a set of p-tuples of structures with the same finite signature $\mathcal{L}(M)$ (containing the booleans and the equality); $S(M)$ and $I(M) \subseteq S(M)$ are closed by multi-isomorphism;
(2) $\tau_M : S(M) \mapsto S(M)$ verifies that there exists an ASM program Π and a function $\textbf{comm}_M : S(M) \mapsto S(M)$ such that:

$$\tau_M(\overrightarrow{X}) = \begin{cases} \tau_\Pi(\overrightarrow{X}) & \text{if } \tau_\Pi(\overrightarrow{X}) \neq \overrightarrow{X} \\ \textbf{comm}_M(\overrightarrow{X}) & \text{otherwise;} \end{cases}$$

(3) $\textbf{comm}_M$ verifies that:
  (1) For every state $\overrightarrow{X}$ such that $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$, $\textbf{comm}_M$ preserves the universes and the number of processors, and commutes with multi-isomorphisms;
  (2) There exists a finite set of terms $T(\textbf{comm}_M)$ such that for every state $\overrightarrow{X}$ and $\overrightarrow{Y}$ with $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$ and $\tau_\Pi(\overrightarrow{Y}) = \overrightarrow{Y}$, if they coincide over $T(\textbf{comm}_M)$ then $\overrightarrow{\Delta}(M, \overrightarrow{X}) = \overrightarrow{\Delta}(M, \overrightarrow{Y})$ (that is $\textbf{comm}_M(\overrightarrow{X}) \ominus \overrightarrow{X} = \textbf{comm}_M(\overrightarrow{Y}) \ominus \overrightarrow{Y}$).

We denote by ASM$_{\text{BSP}}$ the set of such machines. As for the BSP algorithms, a state $\overrightarrow{X}$ is said *final* if $\tau_M(\overrightarrow{X}) = \overrightarrow{X}$. So if $\overrightarrow{X}$ is final then $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$ and $\textbf{comm}_M(\overrightarrow{X}) = \overrightarrow{X}$.

The last conditions about the communication function may seem arbitrary, but they are required to ensure that the communication function is not a kind of magic device, and that it performs data-exchange (or even computations) step-by-step. We presented in this definition only the conditions required to prove Theorem 1 p. 18, and we construct an instance of $\textbf{comm}_M$ in Section 2.7.4 p. 26. More discussions about ASM$_{\text{BSP}}$ could be find in Q20-Q25 p. 20. Some related works are also discussed in Q29-Q35 p. 22 and in Q45-Q47 p. 24.

$$\left\{\begin{array}{l}
X_1 = \{s=\mathbf{0}, j=0, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_1\} \\
X_2 = \{s=\mathbf{0}, j=0, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_2\} \\
X_3 = \{s=\mathbf{0}, j=0, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_3\} \\
X_4 = \{s=\mathbf{0}, j=0, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_4\}
\end{array}\right.$$

$$\xrightarrow[\Pi]{}\left\{\begin{array}{l}
X_1 = \{s=\mathbf{1}, j=0, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_1\} \\
X_2 = \{s=\mathbf{1}, j=0, \mathrm{IdMsg}=\{(1, r, 2, y)\}, y=\mathbf{undef}, r=x_2\} \\
X_3 = \{s=\mathbf{1}, j=0, \mathrm{IdMsg}=\{(2, r, 3, y)\}, y=\mathbf{undef}, r=x_3\} \\
X_4 = \{s=\mathbf{1}, j=0, \mathrm{IdMsg}=\{(3, r, 4, y)\}, y=\mathbf{undef}, r=x_4\}
\end{array}\right.$$

$$\xrightarrow[\mathbf{comm}_M]{}\left\{\begin{array}{l}
X_1 = \{s=\mathbf{2}, j=0, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_1\} \\
X_2 = \{s=\mathbf{2}, j=0, \mathrm{IdMsg}=\emptyset, y=x_1, r=x_2\} \\
X_3 = \{s=\mathbf{2}, j=0, \mathrm{IdMsg}=\emptyset, y=x_2, r=x_3\} \\
X_4 = \{s=\mathbf{2}, j=0, \mathrm{IdMsg}=\emptyset, y=x_3, r=x_4\}
\end{array}\right.$$

$$\xrightarrow[\Pi]{}\left\{\begin{array}{l}
X_1 = \{s=\mathbf{0}, j=1, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_1\} \\
X_2 = \{s=\mathbf{0}, j=1, \mathrm{IdMsg}=\emptyset, y=x_1, r=x_1 \uplus x_2\} \\
X_3 = \{s=\mathbf{0}, j=1, \mathrm{IdMsg}=\emptyset, y=x_2, r=x_2 \uplus x_3\} \\
X_4 = \{s=\mathbf{0}, j=1, \mathrm{IdMsg}=\emptyset, y=x_3, r=x_3 \uplus x_4\}
\end{array}\right.$$

$$\xrightarrow[\Pi]{}\left\{\begin{array}{l}
X_1 = \{s=\mathbf{1}, j=1, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_1\} \\
X_2 = \{s=\mathbf{1}, j=1, \mathrm{IdMsg}=\emptyset, y=x_1, r=x_1 \uplus x_2\} \\
X_3 = \{s=\mathbf{1}, j=1, \mathrm{IdMsg}=\{(1, r, 3, y)\}, y=x_2, r=x_2 \uplus x_3\} \\
X_4 = \{s=\mathbf{1}, j=1, \mathrm{IdMsg}=\{(2, r, 4, y)\}, y=x_3, r=x_3 \uplus x_4\}
\end{array}\right.$$

$$\xrightarrow[\mathbf{comm}_M]{}\left\{\begin{array}{l}
X_1 = \{s=\mathbf{2}, j=1, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_1\} \\
X_2 = \{s=\mathbf{2}, j=1, \mathrm{IdMsg}=\emptyset, y=x_1, r=x_1 \uplus x_2\} \\
X_3 = \{s=\mathbf{2}, j=1, \mathrm{IdMsg}=\emptyset, y=x_1, r=x_2 \uplus x_3\} \\
X_4 = \{s=\mathbf{2}, j=1, \mathrm{IdMsg}=\emptyset, y=x_1 \uplus x_2, r=x_3 \uplus x_4\}
\end{array}\right.$$

$$\xrightarrow[\Pi]{}\left\{\begin{array}{l}
X_1 = \{s=\mathbf{0}, j=2, \mathrm{IdMsg}=\emptyset, y=\mathbf{undef}, r=x_1\} \\
X_2 = \{s=\mathbf{0}, j=2, \mathrm{IdMsg}=\emptyset, y=x_1, r=x_1 \uplus x_2\} \\
X_3 = \{s=\mathbf{0}, j=2, \mathrm{IdMsg}=\emptyset, y=x_1, r=x_1 \uplus x_2 \uplus x_3\} \\
X_4 = \{s=\mathbf{0}, j=2, \mathrm{IdMsg}=\emptyset, y=x_1 \uplus x_2, r=x_1 \uplus x_2 \uplus x_3 \uplus x_4\}
\end{array}\right.$$

Fig. 5. Example of an ASM$_{\text{BSP}}$ execution for the prefix computation.

Example 4. The algorithm p. 10 for the prefix computation can be written with the ASM program Π in the Fig. 4. In this example, $\mathtt{read}(y, r, pid - 2^j)$ is a syntactic sugar[19] for $\mathrm{IdMsg} := \mathrm{IdMsg} \cup \{(pid - 2^j, r, pid, y)\}$, where IdMsg is interpreted as a set (empty at the initialization) of message identifiers (i, x, j, y), where i is the identifier of the sending processor, x the read location, j the identifier of the receiving processor, and y the written location. More details are given in p. 25. When called, for every processor $X^1, \ldots, X^p$, the communication function $\mathbf{comm}_M$ performs the data exchanges, then empties the buffers $\overline{\mathrm{IdMsg}}^{X^1}, \ldots, \overline{\mathrm{IdMsg}}^{X^p}$.

Instead of using an auxiliary boolean b, we use an integer s (for "control state" [28]) initialized at 0 such that when $s = 0$ the (relevant) processors fill the communication buffer, when $s = 1$ the ASM program does nothing thus the communication function is called and updates the state s from 1 to 2, and finally when $s = 2$ the (relevant) processors computes the prefix by using the received value. As before, the computation stops when $2^j \geqslant p$.

We provide an example of execution in Fig. 5 p. 16 for four processors, by distinguishing the computation steps from the communication steps.

As before, by assuming a uniform interpretation of the usual constructors and operations, $T(\Pi) = \{\mathbf{true}, p, pid, s, j, \mathrm{IdMsg}, y, r\}$ is an exploration witness for the ASM program, and $T(\mathbf{comm}_M) = \{\mathbf{true}, pid, s, \mathrm{IdMsg}, \mathrm{Msg}, y, r\}$ is a witness for the communication function. A more complex witness is required in SubSection p. 24 for the constructive version of the communication function. Therefore, $T(M) = T(\Pi) \cup T(\mathbf{comm}_M) = \{\mathbf{true}, p, pid, s, j, \mathrm{IdMsg}, \mathrm{Msg}, y, r\}$ is an exploration witness for the ASM$_{\text{BSP}}$ machine.

2.4.2. The BSP-ASM thesis

In the same way as for the sequential algorithms [47], we will prove that the BSP algorithms and ASM$_{\text{BSP}}$ are the same set of objects. To do so, we first prove in Lemma 7 p. 17 that a transition of a processor in a computation step can be captured by an ASM program. Then, we prove in Corollary 2 p. 17 that this potentially infinite number of ASM programs can be reduced to a finite number. Finally, we merge these programs into one (in normal form) in order to prove in Proposition 2 p. 18 that ASM$_{\text{BSP}}$ captures the computation steps of BSP algorithms. Finally, in Theorem 1 p. 18, we prove that ALGO$_{\text{BSP}}$ = ASM$_{\text{BSP}}$.

[19] Using explicitly a buffer in ASMs may seem surprising to a BSPLIB programmer but this properly reflects the algorithm: we add data in memory that will be communicated later (even if an optimized BSP library would be able to perform a communication-computation overlapping during a superstep). For verification of algorithms using ASMs, an extension of the presented core-language is used in [27].

Let ζ be an isomorphism from the structure X to the structure Y, and let $(f, a_1, \ldots, a_\alpha, b)$ be an update of X. We denote by $\zeta(f, a_1, \ldots, a_\alpha, b)$ the update $(f, \zeta(a_1), \ldots, \zeta(a_\alpha), \zeta(b))$ of Y. We generalize this notation to a set of updates: $\zeta(\{u_1, \ldots, u_k\}) \overset{\text{def}}{=} \{\zeta(u_1), \ldots, \zeta(u_k)\}$, and prove the following lemma, that will be used to prove Theorem 1 p. 18.

Lemma 5 *(Isomorphic ASM). For every* ASM *program Π with signature $\mathcal{L}(X)$ and every isomorphism ζ from X: $\zeta\,(X \oplus \Delta(\Pi, X)) = \zeta(X) \oplus \Delta(\Pi, \zeta(X))$*

Proof (Sketch of). $\zeta\,(X \oplus \Delta(\Pi, X))$ and $\zeta(X) \oplus \Delta(\Pi, \zeta(X))$ have the same universe $\mathcal{U}(\zeta(X))$ and the same signature $\mathcal{L}(X)$, thus it remains to prove that they have the same interpretation for every symbol f. By definition of $\zeta(\Delta)$, $(f, \overrightarrow{a}, b) \in \Delta$ if and only if $(f, \overrightarrow{\zeta(a)}, \zeta(b)) \in \zeta(\Delta)$. Moreover, because ζ is an isomorphism (remark p. 8 on formulas), $a = b$ in X if and only if $\zeta(a) = \zeta(b)$ in $\zeta(X)$, so Δ is consistent if and only if $\zeta(\Delta)$ is consistent. Therefore (definition of the updates) $\overline{f}^{\zeta(X \oplus \Delta)}(\overrightarrow{\zeta(a)}) = \overline{f}^{\zeta(X) \oplus \zeta(\Delta)}(\overrightarrow{\zeta(a)})$. □

Lemma 6 *(Multi-isomorphic set of updates). If $\overrightarrow{\zeta}$ is a multi-isomorphism from the state $\overrightarrow{X}$ to the state $\overrightarrow{Y}$ then $\overrightarrow{\zeta}\,(\overrightarrow{\Delta}(A, \overrightarrow{X})) = \overrightarrow{\Delta}(A, \overrightarrow{Y})$.*

Proof (Sketch of). For every $1 \leqslant i \leqslant p$, we prove $\zeta_i\,(\Delta^i(A, \overrightarrow{X})) = \zeta_i\,(\tau_A(\overrightarrow{X})^i \ominus X^i) = \zeta_i\,(\tau_A(\overrightarrow{X})^i) \ominus \zeta_i\,(X^i) = \tau_A(\overrightarrow{Y})^i \ominus Y^i = \Delta^i(A, \overrightarrow{Y})$, by using the remark p. 8 on formulas, and that τ_A commutes with $\overrightarrow{\zeta}$. □

Lemma 7 *(Each transition is captured by an ASM). For every state $(X^1, \ldots, X^p)$ of the* BSP *algorithm A in a computation phase, and for every $1 \leqslant i \leqslant p$, there exists an* ASM *program $\Pi_i^{\overrightarrow{X}}$ such that $\mathrm{Read}(\Pi_i^{\overrightarrow{X}}) \subseteq T(A)$ and $\Delta(\Pi_i^{\overrightarrow{X}}, X^i) = \Delta^i(A, \overrightarrow{X})$.*

Proof (Sketch of). For every update $(f, a_1, \ldots, a_\alpha, a_0) \in \Delta^i(A, \overrightarrow{X})$ there exists (Lemma 1 p. 9) $t_1, \ldots, t_\alpha, t_0 \in T(A)$ such that for every $0 \leqslant j \leqslant \alpha$ we have $\overline{t_j}^{X^i} = a_j$. So, the ASM program $f(t_1, \ldots, t_\alpha) := t_0$ is interpreted by $(f, a_1, \ldots, a_\alpha, a_0)$ in X^i. Moreover, $\Delta^i(A, \overrightarrow{X})$ contains (Lemma 2 p. 9) a bounded number m of updates $(f^1, a_1^1, \ldots, a_{\alpha^1}^1, a_0^1)$, ..., $(f^m, a_1^m, \ldots, a_{\alpha^m}^m, a_0^m)$. Let $\Pi_i^{\overrightarrow{X}}$ be the following program:

```
par
    $f^1(\theta_1^1, \ldots, \theta_{\alpha^1}^1) := \theta_0^1$
  ‖ $f^2(\theta_1^2, \ldots, \theta_{\alpha^2}^2) := \theta_0^2$
  ⋮
  ‖ $f^m(\theta_1^m, \ldots, \theta_{\alpha^m}^m) := \theta_0^m$
endpar
```
□

To narrow the number of relevant states we use the finiteness of the exploration witness $T(A)$. For every processor X, we denote by E_X the equivalence relation on pairs (θ_1, θ_2) of terms in $T(A)$ defined by:

$$E_X(\theta_1, \theta_2) \overset{\text{def}}{=} \begin{cases} \textbf{true} & \text{if } \overline{\theta_1}^X = \overline{\theta_2}^X \\ \textbf{false} & \text{otherwise} \end{cases}$$

Corollary 2 *(Syntactically equivalent memories). For every* BSP *algorithm A and for every state $(X^1, \ldots, X^p)$ and $(Y^1, \ldots, Y^q)$ in a computing phase, if $E_{X^i} = E_{Y^j}$ then $\Delta(\Pi_i^{\overrightarrow{X}}, Y^j) = \Delta^j(A, \overrightarrow{Y})$.*

Proof (Sketch of). The proof is similar to [6] but is extended to tuples. Let $\tilde{Y}^j$ be the structure obtained by replacing in $\mathcal{U}(Y^j)$ the elements appearing both in $\mathcal{U}(X^i)$ and $\mathcal{U}(Y^j)$ by fresh values, and let Z^j be the structure obtained by replacing in $\mathcal{U}(\tilde{Y}^j)$ the critical elements of $\tilde{Y}^j$ by the critical elements of X^i. Because $E_{X^i} = E_{Y^j}$, this operation is well-defined. So (Definition 6 p. 8, proof in [12]) Y^j and Z^j are isomorphic. Thus (second postulate) $\overrightarrow{Z} = (Y^1, \ldots, Z^j, \ldots, Y^q)$, obtained by replacing Y^j by Z^j in $\overrightarrow{Y}$, is a state.

Because $\overrightarrow{Y}$ is in a computing phase (Lemma 3 p. 11) $\overrightarrow{Z}$ is in a computation phase too. X^i and Z^j have the same critical elements, so (Proposition 1 p. 11) we have $\Delta^i(A, \overrightarrow{X}) = \Delta^j(A, \overrightarrow{Z})$. Moreover (Lemma 7 p. 17) $\Delta(\Pi_i^{\overrightarrow{X}}, X^i) = \Delta^i(A, \overrightarrow{X})$ with $\mathrm{Read}(\Pi_i^{\overrightarrow{X}}) \subseteq T(A)$. Because X^i and Z^j have the same critical elements they coincide over $\mathrm{Read}(\Pi_i^{\overrightarrow{X}})$, so $\Delta(\Pi_i^{\overrightarrow{X}}, X^i) = \Delta(\Pi_i^{\overrightarrow{X}}, Z^j)$. Therefore $\Delta(\Pi_i^{\overrightarrow{X}}, Z^j) = \Delta^j(A, \overrightarrow{Z})$.

Let ζ be an isomorphism between Y^j and Z^j, so $\zeta(\Delta(\Pi_i^{\overrightarrow{X}}, Y^j)) = \Delta(\Pi_i^{\overrightarrow{X}}, Z^j)$ and (Lemma 6 p. 17) $\zeta(\Delta^j(A, \overrightarrow{Y})) = \Delta^j(A, \overrightarrow{Z})$. Therefore $\zeta(\Delta(\Pi_i^{\overrightarrow{X}}, Y^j)) = \zeta(\Delta^j(A, \overrightarrow{Y}))$, and by applying ζ^{-1} on both sides $\Delta(\Pi_i^{\overrightarrow{X}}, Y^j) = \Delta^j(A, \overrightarrow{Y})$. □

Proposition 2 *(BSP-ASMs capture computation phases). For every* BSP *algorithm A, there exists an* ASM *program Π_A in normal form (see [6] or below) such that for every state $\overrightarrow{X}$ in a computation phase: $\overrightarrow{\Delta}(\Pi_A, \overrightarrow{X}) = \overrightarrow{\Delta}(A, \overrightarrow{X})$.*

Proof (Sketch of). $T(A)$ is finite (Postulate 3) so there exists n such that $T(A) = \{t_1, \ldots, t_n\}$. For every local memory X, E_X is an equivalence relation for the elements of $T(A)$, so there is at most $c \leqslant 2^n$ relations E_X. So there exists $Y_1^{i_1}, \ldots, Y_c^{i_c}$ from states $\overrightarrow{Y_1}, \ldots, \overrightarrow{Y_c}$ in a computing phase, such that for every local memory X from a state in a computing phase there exists one and only one $1 \leqslant j \leqslant c$ such that $E_X = E_{Y_j^{i_j}}$.

For every relation $E_{Y_j^{i_j}}$, let F_j be the formula defined by:

$$F_j \overset{\text{def}}{=} \bigwedge_{1 \leqslant k, \ell \leqslant n} E_{k\ell} \text{ where } E_{k\ell} = \begin{cases} (t_k = t_\ell) & \text{if } E_{Y_j^{i_j}}(t_k, t_\ell) \text{ is } \textbf{true} \\ \neg(t_k = t_\ell) & \text{otherwise} \end{cases}$$

So F_j is **true** in a local memory X if and only if $E_X = E_{Y_j^{i_j}}$. Let $\Pi_1, \ldots, \Pi_c$ be the ASM programs obtained at the Lemma 7 p. 17 such that for every $1 \leqslant j \leqslant c$, $\Delta(\Pi_j, Y_j^{i_j}) = \Delta^{i_j}(A, \overrightarrow{Y_j})$. Let Π_A be the ASM program:

```
    if $F_1$ then $\Pi_1$
else if $F_2$ then $\Pi_2$
  ⋮
else if $F_c$ then $\Pi_c$
    endif ... endif
```

Let $\overrightarrow{X} = (X^1, \ldots, X^p)$ be a state in a computation phase, and $1 \leqslant i \leqslant p$. There exists one and only one $1 \leqslant j \leqslant c$ such that $E_{X_i} = E_{Y_j^{i_j}}$, so F_j is **true** in X_i and the other formulas are **false**, thus $\Delta(\Pi_A, X^i) = \Delta(\Pi_j, X^i)$. Because $E_{X_i} = E_{Y_j^{i_j}}$, we have (Corollary 2 p. 17) $\Delta(\Pi_j, X^i) = \Delta^i(A, \overrightarrow{X})$. Therefore, we proved for every $1 \leqslant i \leqslant p$ that $\Delta(\Pi_A, X^i) = \Delta^i(A, \overrightarrow{X})$. □

In Π_A, for every local memory X from a state in a computing phase, one and only one of these formulas $F_1, \ldots, F_c$ is **true**: they are called *guards*. An ASM program with this form, guards, and parallel blocks of updates Π_i (proof of Lemma 7 p. 17) producing distinct non-clashing sets of non-trivial updates, is said to be in *normal form*.

By using this proposition, we prove that the axiomatic presentation of ALGO_{BSP} and the operational presentation of ASM_{BSP} define the same set of objects:

Theorem 1. $\text{ALGO}_{\text{BSP}} = \text{ASM}_{\text{BSP}}$

Proof (Sketch of). The proof is made by mutual inclusion.

On the one hand, let A be the BSP algorithm $(S(A), I(A), \tau_A)$. According to the fourth postulate, there exists $\textbf{compu}_A$ and $\textbf{comm}_A$ such that for every state $\overrightarrow{X}$:

$$\tau_A(\overrightarrow{X}) = \begin{cases} \overrightarrow{\textbf{compu}_A}(\overrightarrow{X}) & \text{if } \overrightarrow{\textbf{compu}_A}(\overrightarrow{X}) \neq \overrightarrow{X} \\ \textbf{comm}_A(\overrightarrow{X}) & \text{otherwise} \end{cases}$$

where $\overrightarrow{\textbf{compu}_A}(X^1, \ldots, X^p) = (\textbf{compu}_A(X^1), \ldots, \textbf{compu}_A(X^p))$. Then, we use the Proposition 2 to prove that:

$$\tau_A(\overrightarrow{X}) = \begin{cases} \tau_{\Pi_A}(\overrightarrow{X}) & \text{if } \tau_{\Pi_A}(\overrightarrow{X}) \neq \overrightarrow{X} \\ \textbf{comm}_A(\overrightarrow{X}) & \text{otherwise} \end{cases}$$

According to the Corollary 1, $\textbf{comm}_A$ preserves the universes, the number of processors, and commutes with multi-isomorphisms. And the other properties are immediately true according to the first three postulates. Thus, $\textbf{comm}_A$ verifies the properties of the Definition 11 p. 15. Therefore A is a ASM_{BSP} machine.

On the other hand, let M be the ASM_{BSP} machine $(S(M), I(M), \tau_M)$. By definition, there exists an ASM program Π and a function $\textbf{comm}_M$ such that:

$$\tau_M(\overrightarrow{X}) = \begin{cases} \tau_\Pi(\overrightarrow{X}) & \text{if } \tau_\Pi(\overrightarrow{X}) \neq \overrightarrow{X} \\ \textbf{comm}_M(\overrightarrow{X}) & \text{otherwise} \end{cases}$$

We prove that M is a BSP algorithm by proving that it verifies the four postulates. The first postulate is straightforward. For the second, τ_Π preserves the universes and the number of processors, and (by using the Lemma 5 p. 17) τ_Π commutes with multi-isomorphisms. For the third, $T(\Pi) = \{\mathbf{true}\} \cup \mathrm{Read}(\Pi) \cup \mathrm{Write}(\Pi)$ (Definition 10 p. 15) is an exploration witness for τ_Π so $T(M) = T(\Pi) \cup T(\mathbf{comm}_M)$ is for M. For the fourth, let $\mathbf{compu}_M(X) = X \oplus \Delta(\Pi, X)$ for every local memory X. So $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{\mathbf{compu}_M}(\overrightarrow{X})$, and we have:

$$\tau_M(\overrightarrow{X}) = \begin{cases} \overrightarrow{\mathbf{compu}_M}(\overrightarrow{X}) & \text{if } \overrightarrow{\mathbf{compu}_M}(\overrightarrow{X}) \neq \overrightarrow{X} \\ \mathbf{comm}_M(\overrightarrow{X}) & \text{otherwise} \end{cases}$$

Therefore M is a BSP algorithm. □

Corollary 3. *Every* ASM$_{\text{BSP}}$ *program has an equivalent normal form.*

Proof. According to the previous theorem, an ASM$_{\text{BSP}}$ is a BSP algorithm, and every BSP algorithm is captured by an ASM$_{\text{BSP}}$ with a program (Proposition 2 p. 18) in normal form. □

Thus, we will assume in the Subsection 3.3.2 p. 35 that the considered ASM program will be in normal form. More discussions about the proofs could be find in Q26-Q28 p. 20 and about the overall methodology in Q36-Q40 p. 23.

Theorem 1 proves that the axiomatic presentation of ALGO$_{\text{BSP}}$ and the operational presentation of ASM$_{\text{BSP}}$ correspond to the same set of objects. We believe that our four postulates for ALGO$_{\text{BSP}}$ are as simple as possible, and convincing, thus that we captured properly the BSP bridging model described in the Subsection 2.1.1 p. 4. Moreover, the ASM$_{\text{BSP}}$ provides a model of computation for our formalization, even if we think that the imperative programming language presented p. 31 is more suited for algorithmic (costs, subclasses [13]) analysis.

Finally, it remains two more steps in order to claim that ASM$_{\text{BSP}}$ objects are the actual BSP algorithms: (1) To ensure (SubSection 2.6 p. 24) that the duration corresponds to the standard BSP cost model and; (2) To provide (SubSection 2.7 p. 24) a constructive example of a non-trivial function of communication.

2.5. Questions and answers

Q20: *Now that you have modeled the operational point of view, and before the following, you understand that we must be strongly convinced that your presentation is appropriate.* ASM$_{\text{BSP}}$ *is introduced in the Definition 11 p. 15 as the* ASM *programs satisfying the fourth postulate, but no syntactic classification or otherwise constructive criterion is given to distinguish those programs, from which the fourth postulate could have been proved. In that sense* ASM$_{\text{BSP}}$ *is as axiomatic as the postulates and nothing is gained. So, your first theorem p. 18 is immediate and trivial.*

The set of BSP algorithms is defined by the four postulates, not by programs. The set of the ASM$_{\text{BSP}}$ machines are defined by using a standard sequential ASM program on every processor during the computation phases, and a communication function verifying some properties (including an exploration witness) during the communication phases. The proof of the first theorem has similarities with the proof of the sequential case, but the extension to the p-tuples requires many technical details, which are proved in our paper and are far from trivial. Moreover, the communication function is a distinct object from the ASM program, and it has been abstracted for sake of generality. We prove indeed in the Subsection 2.7 p. 24 that our example of communication function for the read/write has an exploration witness, but this is only a particular case and our definition still require the hypothesis for the general case.

Q21: *I think a sequential simulation of the sequence of supersteps seems sufficient. Why do you use a new model of computation* ASM$_{\text{BSP}}$ *instead of* ASMs *only? Indeed, each processor can be seen as a sequential* ASM*. So, in order to simulate one step of a* BSP *algorithm using several processors, we could use pids to compute sequentially the next step for each processor with a single* ASM*.*

For a program Π of an ASM$_{\text{BSP}}$ machine, you are thinking of generating a ASM-like sequential program as:

```
(* repeat these statements as long as there are no changes *)
    $Y := X;$
    **forall** $i$ **in**$1...p$ **do** $\Pi(X[i]);$
    **if** $X = Y$ **then** $X := \mathbf{comm}(X);$
```

where Y and X are p-tuples.

Q22: *Yes, where* $\Pi(X[i])$ *is the execution of* Π *at processor i.*

This kind of simulation has been studied by the second author in [34] (with appropriate invariants and pre/post-conditions in order to manage the BSP computations as a common sequential computation) but in the context of proving the correctness of BSP programs not the algorithmic completeness.

Even if such a simulation exists between these models, a "sequentialization" (each processor, one after the other) of the BSP model of execution cannot be exactly the function of transition of the postulates: adding p in the exploration witness in order to remain with only *bounded* local changes of the states, whatever the initial states (and thus whatever the number of processors) is not sufficient; Indeed, that induces p to be a constant for the algorithm. In our work, p is only fixed for each execution (at the beginning, in an initial state), making the approach more general when modeling algorithms, and obtaining the notion of *immortal* algorithms. Thus, using a sequentialization forces that any computation done in parallel is executed p times, which changes the BSP costs. Moreover, this would contradict the algorithmic simulation, that will define in the next section.

Q23: *OK, but your Definition 11 p. 15 of the* ASM_{BSP} *uses a single program for every processor, but according to [17], only one* ASM *program using "parallel forall" constructs is necessary to model the parallel algorithms. You could apply it to simplify your characterization of the* BSP *algorithms.*

We could, but this would violate the bounded exploration postulate. Actually, they have a less restricted version of the postulate. In [17] they use not only ground terms but also multiset comprehension terms, and access terms in [18]. We are thinking that it is not better to do as much (model the BSP algorithms) with more (terms in the exploration witness). Actually, our approach does not require their metafinite model theory. Ours witness contain ground terms only.

Q24: *Why such a limitation?*

Because exploration witnesses of [6] are sufficient to bound the local computations of the processors.

Q25: *But, as stated in your second postulate, states are* p*-tuples of processors, so how do you handle manipulation on tuples or multisets?*

We do not have to. In our approach, tuples are manipulated by the operational semantics of ASM_{BSP}, not internally by the processors themselves. Therefore, we do not need to assume constructors and operations on tuples or multisets, nor raise functions in an *ad-hoc* "background postulate" [17].

Q26: *OK, but in order to define the semantics of* ASM *programs and to prove Proposition 2 p. 18, you need boolean constructors and operations. This could have been stated in a background postulate.*

Indeed, but we wanted to stick to the presentation of [6], therefore we assumed the boolean data structure not in a postulate but after Definition 3 p. 7.

Q27: *Fine. But could you explain why this is so important to verify the bounded exploration postulate from the sequential algorithms in a parallel computation framework?*

Every processor can be seen as a sequential machine (even if it communicates with others), therefore each processor individually should (intuitively) verify the sequential bounded exploration postulate. In particular, a shared memory approach seems to us either unrealistic or too restricted.

Q28: *Can you elaborate on that?*

If the shared memory does not depend on the number of processors, that would imply that (for implementation purpose) the same device should be used to manage communications between an unbounded number of processors. Therefore, a finite volume should contain an unbounded amount of information, which seems unrealistic. Therefore, we think a shared memory can be used only for a bounded number of processors, which is too restricted to model the parallel algorithms. Our approach of distributed (BSP) algorithms can handle any number of processors. Locally, every processor can be seen as a sequential machine, and the data are spread globally amongst them.

Q29: *So your model is not accurate for multi-cores machines?*

BSP is a bridging model so it aims to work on most relevant kind of machines. Shared memory machines could also be used in a way such that each processor only accesses a sub-part of the shared memory (which is then "private") and communications could be performed using a dedicated part of the memory. But, our model is independant to this potential limit of processors (which could be needed for some algorithms working on such machines). This is more an implementation issue.

Q30: *Back to the modified exploration witnesses of [17]. They can take into account "parallel loop" and they obtain models for* PRAM *and* MAPREDUCE*... And their results are also independent of "p"!*

In our work, we use the sequence of supersteps of the BSP bridging model to simplify the approach (we thus have not concurrent accesses to a shared memory). Even if the number of processors is unbounded, we assume that every processor works in small steps. Instead of defining a state of an execution by a meta-finite structure [17] (even if their use is mainly technical, they are needed for the definitions and proofs), we assume that a state is a p-tuple of structures. Instead of using a global program with "forall" commands [17], every processor runs its local program with "par" commands. Instead of relaxing the third postulate of [6], we assume it for every p-tuple of processors. We think that this approach leads to a simpler presentation of BSP algorithms, using only ground terms and tuples of common structures.[20] Again, if we used their exploration witnesses directly, we would have to introduce an artificial and uncommon limitation of [15,17] instead of having a natural extension of [6]. Adding the artificial limitations is more an implementation rather than a specification. We would not get the class of BSP algorithms at their right level of abstraction. And we want in future work to extend our postulates to take into account more complicated bridging models (bottom up strategy). And thus, the artificial limitations could be more and more complex. For instance, the MULTI-BSP bridging model, with its tree of nested machines, is already complicated enough without adding an extra layer of difficulty.

Notice that the second example of [17] is a PRAM machine simulation (with a succession of computations then communications). There is still a last drawback: their ASMs used an implicit shared memory (a shared structure) which is irrelevant for HPC computations. This flaw of the PRAM model was already criticized in [20]. Their third example is MAPREDUCE (which is very close to BSP in their implementation) [2] which uses again a single shared structure.

Q31: *So you admit that your model is fully distributed and that you have a set of sequential machines. So let us continue with related works. We can imagine that a distributed machine is defined as a set of pairs (a, Π_a) where a is the name of the machine/agent [39] and Π_a a sequential* ASM*. Reading your definition, I see only one Π and not "p" processing units as in the* BSP *model.*

We can found such an approach in [37] (with postulates). But using only a finite number of agents makes impossible to reason for any p (we insist again on immortal algorithms) and so we do not want such an arbitrary choice.

Q32: *But reading the work of [15,32,40] and even more recently in [16], there is not such a limitation! Their work thus allows to model distributed machines with direct (and concurrent) read/write of terms in [15] or using point-to-point primitives with buffers and queues of messages "à la* MPI*". That looks like the most general definition. Why not directly use their definitions in order to get the class of* BSP *algorithms?*

We want to clarify one thing. The ASM thesis comes from the fact that sequential algorithms work in small steps, that is steps of bounded complexity. But the number of processors (computing units) is unbounded for distributed/parallel algorithms, which motivated the work of [19] to define distributed algorithms with wide steps, that is steps of unbounded complexity. Hence the technicality of the presentation, and the unconvincing attempts to capture distributed algorithms [15]. A first attempt to simplify this work has been done in [18] and in [15,16].

And so you are absolutely right, our model is surely a kind of subclass of their works. But not completely. Our characterization (model) is largely different.

Firstly, we do not model a BSP computer, our work is about BSP algorithms (the right part of Fig. 2 p. 4). The ASM_{BSP} program contains the algorithm "code" which is applied on each processor. These are the postulates (axiomatic point of view) that characterize the class of BSP algorithms rather than a set of abstract machines (operational point of view). That is closer to the original approach of [6]. Notice that our exploration witness persists to be a finite set and thus, whatever p, only a bound number of symbol can be modified, which induces that some symbols are shared by the processors (but with different values on each X_i).

Secondly, the realism of using an infinite set of ASMs [15] is debatable for a bridging model such that BSP. But here is the remaining issue for our purpose. We recall that we want to define the class of BSP algorithms and to prove that an imperative BSP language is sufficient to program all the BSP algorithms with the right BSP cost. We could realize (or implement) a BSP algorithm as this set of communicating agents (with point-to-point primitives; they use sequential witnesses for the local machines/agents) and the BSP patterns would be simulated by these primitives and by adding, at the end, a synchronization algorithm of all the agents. But we would have a complex implementation rather than a simple characterization (at the right level of abstraction).

Q33: *Complex! Are you sure?*

Not at one hundred percent. But the works of [15,16] remains complex because they have to manage what is call the

[20] Which postulates are the simpler is always debatable but ours are "realistic" because they only use the intuitive notion of multi-isomorphism. We conjecture that our approach can be generalized to a broader class of bridging models [33,41], which we will investigate in future work.

"sequentially consistent" (again interleaving and concurrent accesses). They thus define $\mathcal{A}$-run, a function which selects the agents/ASMs that can perform *moves* (computations or communications) and the initial state for each of the agents. And this, at each step of the algorithm's $\mathcal{A}$-execution (using a kind of partial order). Thus, to prove that another model (*e.g.* a programming language) is equivalent to this axiomatization, it would be necessary to manage most of these constraints in order to do the proofs, which seems unnecessarily complicated. Moreover, the implementation (as an extra layer of code) would make surely complex the BSP cost analysis. We insist on the fact that our characterization is still limited to BSP, and, of course, if you want the most general possible characterization, you must take the more complex one of [15].

Q34: *Are there other differences? Your work remains different to their approach widely accepted and used by other researchers...*

Our model is still limited by assuming a unique program Π and thus a single exploration witness. This code is *the* BSP algorithm instead of having potential different programs on the processes. We take up the fact that agents (processors) share the same finite signature. We use a set of simple p-tuples while they use a set of agents (each using a set of initial states). We abstract the communications while they sometime use an explicit network topology (a direct graph [16]). They need a partial order, the $\mathcal{A}$-runs and $\mathcal{A}$-executions. These complex constructions are not required in our model because we assume that the initial states are p-tuples which thus contain the initial data and p, the number of processors (and the other BSP parameters).

Q35: *A different one, surely, but that your model is simpler is still to be debated.*

We can argue that, according to Lemma p. 11, a BSP algorithm with only one processor is a sequential algorithm. Therefore, our presentation can be seen as an extension of [6], unlike the usual presentations for parallel algorithms [17] or of distributed algorithms [16] that require more complicated terms for the exploration witness or more restricted constructors (*e.g.* the $\mathcal{A}$-runs).

Q36: *Is another model possible to characterize the* BSP *algorithms?*

Sure. This can be more useful for proving some properties. But that would be the same set, with another way to describe it.

Q37: *OK, but with only a single code, can you have all the* BSP *algorithms?*

We follow [35] about the difference between a PARallel composition of SEQuential actions (PAR of SEQ) and a SEQuential composition of PARallel actions (SEQ of PAR). Our ASM_{BSP} is SEQ(PAR). This leads to a *macroscopic* point of view[21] which is close to a specification. Being a SEQ(PAR) model allows a high level description of the BSP algorithms. In the next section, we will define the operational semantics of an imperative programming language (the practice versus the theoretical idea of an algorithm) which is PAR(SEQ). In general PAR(SEQ) is the implementation of a SEQ(PAR) language, so in that sense, our Π program is *the* BSP algorithm.

Q38: *So, why are you limited to* SPMD *computations?*

Different codes can be run by the processors by using conditionals on the "id" of the processors. For instance "if pid=0 then code1 else code2" for running "code1" (*e.g.* master part) only on processor 0. Again, we are *not* limited to SPMD computations. The ASM program Π *fully contains* the BSP algorithm, that is *all* the "actions" that can be performed by *any* processors, not necessarily the same instructions: each processor picks the needed instruction to execute but they could be completely different. Only the size of Π is finite due to the exploration witness. For instance, it is impossible to have a number of conditionals in Π that depends of p. Indeed, according to Proposition 1, during a computation phase, if two processors coincide over the exploration witness, then they will use the same code. And according to the third Postulate, the exploration witness is bounded. So, there exists only a bounded number c of possible subroutines during the computation phase, even if $p >> c$.

Moreover, the processors may not know their own ids and that our p-tuples are not ordered; we never use such a property: processors are organized like a set and we use tuples only for convenience of notation. We use p-tuples only to add the BSP execution model in the original postulates of [6].

Q39: *OK, but I cannot get the interleaving of computations. Your model of "distributed* ASM*" seems very synchronous! I mean, when defining the operational semantics of a concurrent language (or process calculus), the rules are applied one after the other, leading to*

[21] Take for instance a BSP sorting algorithm: first all the processors locally sort their own data, and then they perform some exchanges in order to have the elements sorted between them. This can be defined as a sequence of parallel actions, which is also independent of the number of processors.

the interleaving of computations.

The BSP model makes the hypothesis that the processors are *homogeneous* (uniform). So if one processor can perform one step of the algorithm, there is no reason to lock it just to highlight an interleaving. And if there is nothing to do, it does nothing until the *phase of communication*. Our execution model is thus largely "massively parallel" during the computation phases.

Q40: *But that is not realistic!*

Indeed. But our work is *not* an implementation of a BSP machine, it is about BSP algorithms (right part of the Fig. 2 p. 4). For us, a BSP algorithm is an "idea" of how to compute (in a BSP manner) to get a result, not a juxtaposition of machines. Obtain a correctness result about an MPI implantation of a BSP library (*e.g.* BSPLIB) on a physical distributed machine (*e.g.* a cluster) is an interesting work but is not the purpose of this paper.

Q41: *Now speaking about the communication, why apply several times the function of communication? When designing a BSP algorithm, I use once a specific routine (such as collective operations).*

As long as the exploration witness for the communication function is preserved, collective operations are possible. For instance, one broadcast operation in one step will be allowed, but it will not be allowed for a processor to receive an unbounded number of messages in one step of communication. But this behavior can nevertheless be simulated in several communication steps, *i.e.* a communication phase. For instance, collective operations can be simulated by a more elementary sequence of point-to-point communications (the BSP 1-relation). ASM is like a Turing machine: the exploration witness prevents it from performing all the communications in a single step. Our example of function of communication p. 24 performs exchanges until there are no more to be done. For instance, if the collective operation is a reduce(x) (for simplicity, with a specific operator + such as the string concatenation) then we could add these kinds of instructions: "buffer:=reduce(buffer,x,y)" to compute the reduce of "x" in "y". If the communications were made in a single step then there will be possibly $\sum_{i=1}^{p} x_i$ values on processors which is a unbound size of data that cannot be communicated in a single step without using magic devices (hardwares). By forcing (because of the exploration witness of the function of communication) that the data be transmitted one by one, we also preserve the BSP model of h-relations (SubSection 2.7 p. 24). But, in this example, reduce can perform an unbounded amount of work (p times "+" for any p) and thus does not correspond to a bounded (finite) number of read or written terms.

Q42: *So common BSP patterns of communications (e.g. BSPLIB primitives) are impossible?*

They are possible but they should be done step-by-step. We show in SubSection 2.7 p. 24 how to do this and how to get an exploration witness for algorithms using such elementary primitives. This is a little bit technical but keep in mind that our main goal is the algorithmic completeness of (common) imperative programming languages with BSP primitives, not to stay in the ASM framework.

Q43: *And what happens in case of runtime errors during communications?*

Typically, when one processor has a bigger number of supersteps than other processors, or when there is an out-of-bound sending or reading, or too much data to communicated (if necessary such as in MPI), it leads to a runtime error. The BSP function of communication can return a **undef**. That causes a stop of the operational semantics of the ASM_{BSP}.

Q44: *And for random reading of messages such as in the BSPLIB?*

We can say that such structures are typically heaps of messages. Random algorithms have also been studied in the ASM framework [27]: two executions, within the same environment, will lead to identical results. For determinism, adding the environments is a tricky, but not elegant, solution. Otherwise, truly formally studying randomized algorithms is a hard task (even for sequential computing) which is only in its infancy and is not relevant to this article.

Q45: *About structured parallelism. Is there other works on this subject?*

In [38], authors model the P3L set of skeletons.[22] That allows the analysis of P3L programs using standard ASM tools. Data-parallel skeletons are close to the BSP model (thinking of the MAPREDUCE framework [42]) but they are not as expressive and could induce asynchronous computations, so their cost model is not as well-established as the BSP one (nevertheless, skeletons are very useful for many parallel application).

[22] Parallel skeletons [4] is a paradigm where the programmer uses a set of patterns (the skeletons) in order to get a parallel execution of its algorithm.

Q46: *And what about task parallelism and load-balancing of threads such as in [29]?*

Our ASM$_{\text{BSP}}$ language does not directly provide such routines. But they can be simulated (hardly, we know it). Anyway, this is more a programming problem (or a architectural problem, for example, for a cloud framework) than formally defining BSP algorithms.

Q47: *You speak regularly of the BSP's costs notably for BSP algorithms but can you truly analyze them by using the ASM framework? Assuming any elementary primitives is perhaps too abstract for such a study.*

There are many papers about the cost analysis of programs (*e.g.* worst case execution time, WCET) and their complexity. They are less for ASM [43] for the reason that you are pointing at and on the fact that pure ASMs do not really structure codes (no sequence, only a single loop) as one would expect for algorithms. This is also why we prefer to use ASMs as intermediary objects from programs to algorithms. Let us explain our result about BSP costs and our realization of a function of communication.

2.6. Cost model property

We now show the link between the duration of an ASM$_{\text{BSP}}$ execution and the cost of a BSP algorithm A.

If the execution begins with a communication, we assume that no computation is done for the first superstep. We remind that a state $\overrightarrow{X_t}$ is in a computation phase if there exists $1 \leqslant i \leqslant p$ such that $\mathbf{compu}_A(X_t^i) \neq X_t^i$. The computation for every processor is done in parallel, step by step, and these steps are done *simultaneously*. By definition of the fourth postulate, when a processor has finished its computations it waits for the other processors, and the communication phase can only begin when all the processors have finished their computations. So, the cost in time of the computation phase is $w \overset{\text{def}}{=} \max_{1 \leqslant i \leqslant p}(w_i)$, where w_i is the number of steps done by the processor i (on local memory X^i) during the considered superstep.

Then the state is in a communication phase, when the messages between the processors are sent and received. Notice that $\mathbf{comm}_A$ may require several steps in order to communicate the messages, which contrasts with the usual approach in BSP where the communication actions of a superstep are considered as one unit. But this approach would violate the third postulate, so we had to consider a step-by-step communication approach, then consider these communication steps as one communication phase.

We assume that the data are communicable in time $\mathbf{g}$ for an exchange (a 1-relation, see the BSP model p. 4) at each step (each call of the transition function τ_A) of $\mathbf{comm}_A$. So, during the superstep, the communication phase requires $h \times \mathbf{g}$ steps, where h is the number of exchanges (or 1-relation):

Lemma 8 *(Order of exchanges). For every h-relation there exists a sequence of exchanges requiring at most h exchanges.*

Proof (Sketch of). A proof by induction (on the number of messages) could be made (step-by-steps and this sequence is costly to compute, one is presented in [44]) but pattern of exchanges like the well known Latin square (use in the TCP/IP implementation of the BSPLIB) or routing in specific networks could also be used as correct algorithms and for finding the sequence of exchanges. □

It remains to add the cost of the barrier (initialization of the network, synchronization of the processors and assurance that every message has been received) which is assumed in the usual BSP model to be a constant $\mathbf{L}$. Therefore, we obtained a cost property which is sound with the standard BSP cost model:

Proposition 3 *(Cost model for the BSP-ASMs). ASM$_{\text{BSP}}$ has the same cost model as the standard BSP bridging model.*

2.7. A realization of the communication function, and its exploration witness

Our previous result is only valid up to elementary functions and especially the $\mathbf{comm}_M$ one, which raises the question whether such a non trivial function exists.[23] We now describe how such a communication function should be done in the ASM context, and how to construct the needed exploration witness.

Such a function is rather naive since only point-to-point exchangings of a single data are considered. We also assume the function makes messages traveling on the "network" but we dot not describe such an object. Messages are sent letter by letter: it is the sequence of steps of $\mathbf{comm}_M$ that perform the h-relation. If the network can send blocks of data or broadcast

[23] Notice that we do not prove an implementation of a $\mathbf{comm}_M$ function using low level primitives such that the ones of MPI: it is an interesting but orthogonal work; it is closer to semantically correct implementation of a BSP machine and thus takes part in the left part of Fig. 2 p. 4. For such a work, we believe that using machine-correct proofs is profitable (*e.g.* with the coq proof assistant, https://coq.inria.fr/).

messages and thus involve more than two processors, a more efficient (but much more complicated) function can be given. But our function is sufficient because it involves the expected BSP cost for the traditional BSP programming primitives (the h-relation is preserved).

2.7.1. Generalities

We need the communication function to preserves the universes and the number of processors, and commutes with multi-isomorphisms. This condition will be met because, as we will see in the following, the communication function will manage the communications between processors by using a syntactical process. We need also to construct an exploration witness,[24] which is a bit technical. The main ideas are:

- The messages are sent letter by letter; the representation gives the formal size of the communicated values;
- The communications use two buffers: (1) IdMsg, to store the order of communication during a computation phase and (2) Msg, for the sending of the messages during a communication phase;
- If we assume that every processor can send or receive at most one message during an "exchange" (or 1-relation, see SubSection 2.1.2 p. 5), then the h-relation requires exactly h exchanges.

We want also to ensure that the local memories are only updated at the end of the communication phase. Indeed, in a state $\overrightarrow{X}$ during the communication phase, if a local memory is updated we may have $\tau_\Pi(\overrightarrow{X}) \neq \overrightarrow{X}$, so the computation phase may begin even if the communication phase is not completed. That would not fit the BSP model of execution. We could also use a boolean b_{comm} which is **true** during the communication phase, and such that the computation phase can begin only if b_{comm} becomes **false**. In that case, b_{comm} should be added to the exploration witness of the communication function.

In this section, we use the standard BSPLIB's DRMA primitives `bsp_get` and `bsp_put` only. We briefly recall how their work and we rename them read and write afterwards. Direct point-to-point sending of values as well as collective primitives have not been considered. They can be implemented but with a much more complicated presentation. Our purpose here is to prove the achievability of such a function of communication, not to find an efficient/correct one.

2.7.2. BSPlib's DRMA primitives

The BSPLIB [23] is a C library of communication routines. It offers routines for both message passing and DRMA primitives. We can also query some information about the machine: `int bsp_nprocs()` returns the number of processors $\mathbf{p}$ and `int bsp_pid()` returns the processor identifier which belongs to $0, \ldots, \mathbf{p}-1$. The barrier and all the communication are done using `void bsp_sync()` which blocks the node until all other nodes have called `bsp_sync` and all messages have been received.

The two main DRMA routines operations are: (1) `void bsp_get(int pid,const void *src,int offset, void *dst,int nbytes)` stands for distant reading access; it copies n bytes to the local memory address `dst` from the variable `src` at `offset` of the remote processor `pid`; (2) `void bsp_put(int pid,const void *src,void *dst, int offset, int nbytes)` stands for distant writing access; it copies n bytes from local memory `src` to `dst` at `offset` on remote processor `pid`.

For our ASM, reading and writing are *not* on buffers of bytes (memory area as in the C language) but on *variables*. For instance, in an ASM_{BSP} we can use the command write(x, j, y), which is syntactic sugar for $\text{IdMsg} := \text{IdMsg} \cup \{(pid, x, j, y)\}$, that stores in the buffer IdMsg the order to send the value of x from the current processor to processor j in the variable y. More generally, IdMsg contains the identifiers (i, x, j, y) of the messages, where i is the identifier of the sending processor, x the read location, j the identifier of the receiving processor, and y the written location. In the same way, a command (again, with syntactic sugar) read(x, j, y) of Π will be interpreted in the processor X^i by (j, y, i, x) and added to the buffer $\overline{\text{IdMsg}}^{X^i}$. For each processor, $\mathbf{comm}_M$ just move one message (if exists) from one buffer to another one of another processor and performs the appropriate update (described below). Finally, a sequence of $\mathbf{comm}_M$ functions correspond to a call of the BSPLIB's `bsp_sync`.

2.7.3. Serialization and characterization of terms

The *representation* θ_a of an element a in the local memory X is the unique term formed only by constructors (see p. 7) such that $\overline{\theta_a}^X = a$. For instance, the representation of 314 as a decimal number is $\underline{314}_{10}$, and its representation as a binary number is $\underline{100111010}_2$. Because a letter is not itself a term, for convenience we will add to the language $\mathcal{L}(M)$ a constant symbol $\underline{f}$ for every symbol $f \in \mathcal{L}(M)$, interpreted by $\overline{\underline{f}}^{X^i} = f$.

Definition 12 *(Sequentialization).* If θ is a term, let $\overset{\circ}{\overline{\theta}}$ be the sequence of the letters used to write it in the prefix notation. The sequentialization of a term θ, noted $\overset{\circ}{\underline{\theta}}$ is the same sequence but with the use of constant symbols $\underline{f}$.

[24] This will not be necessary for other BSP libraries if their programs can be simulated using the BSPLIB which is the case, to our knowledge, for most of them.

For instance, if f is a binary symbol, g is a unary symbol, and a and b are constants, then the sequence of letters of the term $\theta = f(f(b, g(a)), g(f(a, b)))$ is $\overset{\circ}{\overline{\theta}} = (f, f, b, g, a, g, f, a, b)$. So, its *sequentialization* is $\overset{\circ}{\underline{\theta}} = (\underline{f}, \underline{f}, \underline{b}, \underline{g}, \underline{a}, \underline{g}, \underline{f}, \underline{a}, \underline{b})$.

Therefore, $a \mapsto \overset{\circ}{\overline{\theta_a}}$ is the implementation in our framework of the *serialization* function (*e.g.* JAVA's `toString` method). We assume that every value which can be communicated must be *serializable*, which is already given because we assumed that every element must be representable. In order to assign a size to terms, we can use this serialization.

The *weight*, noted $\mathrm{w}(f)$, of a symbol f with arity $\alpha(f)$ is defined by $\mathrm{w}(f) \overset{\text{def}}{=} 1 - \alpha(f)$. Because for every symbol f we have $\alpha(f) \geqslant 0$, notice that $\mathrm{w}(f) \leqslant 1$. The weight of a word $f_1 \dots f_\ell$ of length ℓ is defined by: $\mathrm{w}(f_1 \dots f_\ell) \overset{\text{def}}{=} \sum_{i=1}^{\ell} \mathrm{w}(f_i)$. If $1 \leqslant i \leqslant \ell$, the word $f_i \dots f_\ell$ is called a suffix of the word $f_1 \dots f_\ell$.

Lemma 9 *(Characterization of a term). The word $f_1 \dots f_\ell$ is a term if and only if $\mathrm{w}(f_1 \dots f_\ell) = 1$ and for every suffix $f_i \dots f_\ell$, $\mathrm{w}(f_i \dots f_\ell) \geqslant 1$.*

Proof (Sketch of). The implication is proved by induction on the term θ, and the converse by induction on the length of the word $f_1 \dots f_\ell$. If $f_1 \dots f_\ell$ is the empty word, then $\ell = 0$ and $\mathrm{w}(f_1 \dots f_\ell) = \sum_{i=1}^{\ell} \mathrm{w}(f_i) = 0$. So, because the empty word is not a term, the equivalence holds in that case too. $\square$

2.7.4. *A function of communication à la BSPlib in the ASM framework*

We recall that we want to find the exploration witness for every program using BSPLIB-like DRMA primitives. So, a message from a processor i to a processor j will be sent letter by letter. At the end of the computation phase these identifiers are mixed together in three steps: (1) For every processors X^i and X^k (with $k \neq i$), the communication function moves all (i, x, j, y) from $\overline{\mathrm{IdMsg}}^{X^k}$ to $\overline{\mathrm{IdMsg}}^{X^i}$; (2) For every processor X^i, the communication function replaces $\overline{\mathrm{Msg}}^{X^i}$ by the set $\{(\overset{\circ}{\underline{\theta}}_{\overline{x}^{X^i}}, j, y) \mid (i, x, j, y) \in \overline{\mathrm{IdMsg}}^{X^i}\}$; (3) For every processor X^i, the communication function replaces $\overline{\mathrm{IdMsg}}^{X^i}$ by the empty set. Notice that the last two steps are purely local.

Then the processor X^i "sends" the message letter by letter to the processor X^j. Using our ASM$_{\mathrm{BSP}}$, it is only a "copy" from the structure X^i to X^j. If the message were reconstructed on X^j only at the end of the communication phase, then the term θ itself would be necessary, and the exploration witness could not be bounded. Instead, the value is also reconstructed step by step each time a letter is received.

For instance, to reconstruct the term $\theta = f(f(b, g(a)), g(f(a, b)))$, the processor X^i sends $\overset{\circ}{\underline{\theta}} = (\underline{f}, \underline{f}, \underline{b}, \underline{g}, \underline{a}, \underline{g}, \underline{f}, \underline{a}, \underline{b})$ one letter at a time from the end to the beginning, and the processor X^j reconstructs θ by using the following updates:

$$\begin{aligned}
&\underline{b} \text{ received} \Rightarrow x_1 := b\\
&\underline{a} \text{ received} \Rightarrow x_2 := a\\
&\underline{f} \text{ received} \Rightarrow x_1 := f(x_2, x_1)\\
&\underline{g} \text{ received} \Rightarrow x_1 := g(x_1)\\
&\underline{a} \text{ received} \Rightarrow x_2 := a\\
&\underline{g} \text{ received} \Rightarrow x_2 := g(x_2)\\
&\underline{b} \text{ received} \Rightarrow x_3 := b\\
&\underline{f} \text{ received} \Rightarrow x_2 := f(x_3, x_2)\\
&\underline{f} \text{ received} \Rightarrow x_1 := f(x_2, x_1)
\end{aligned}$$

At the end, we got $x_1 = f(f(b, g(a)), g(f(a, b)))$.

Firstly, notice that the number of variables used that way cannot be bounded. For instance, the term $f(\dots f(a_{n+1}, a_n) \dots, a_1)$ requires $n+1$ variables. So, instead of a variable "x_{nArg}", we will use a unary symbol $\mathrm{RcvVal}(\mathrm{nArg})$. We thus need to specify how the index nArg evolves. We set $\overline{\mathrm{nArg}}^{X^i} = 0$ for every local memory at the beginning of a communication phase. Then, each time a symbol $\underline{f}$ is received, nArg is updated by $\mathrm{nArg} := \mathrm{nArg} + \mathrm{w}(f)$, where $\mathrm{w}(f)$ is the weight of the symbol f. The weight of a word is the sum of the weight of its letters. We get that $\mathrm{nArg} \geqslant 1$ during the communication phase, and that for every term θ we have $\mathrm{w}(\theta) = 1$. Therefore, at the end of the communication phase, nArg is the number of terms received.

Secondly, the letters may be received for different locations. In fact, a letter $\underline{f}$ cannot be sent alone, the location (j, y) should be made explicit. Therefore, the messages traveling on the "network" have the form $(\underline{f}, j, y)$, and the symbols RcvVal and nArg should also depend on the location. So, in fact, RcvVal is a binary symbol, and nArg is a unary symbol.

If $\mathrm{WriteComm}(\Pi) = \{y_1, \dots, y_k\}$ is the set of the variables written in Π by communication primitives, then the locations are $\underline{y_1}, \dots, \underline{y_k}$. So, in our example, each time the processor X^j receive a message $(\underline{c}, j, y)$ where c is a constant symbol, the following update is made:

```
par
        $\text{nArg}(\underline{y}) := \text{nArg}(\underline{y}) + 1$
  $\|$ $\text{RcvVal}\left(\underline{y}, \text{nArg}(\underline{y}) + 1\right) := c$
endpar
```

And each time the processor X^j receive a message $(\underline{f}, j, y)$ where $\alpha(f) \geqslant 1$ the following update is made:

```
par
        $\text{nArg}(\underline{y}) := \text{nArg}(\underline{y}) + 1 - \alpha\left(\underline{f}\right)$
  $\|\text{RcvVal}\left(\underline{y}, \text{nArg}(\underline{y}) + 1 - \alpha\left(\underline{f}\right)\right) := f\begin{pmatrix}\text{RcvVal}\left(\underline{y}, \text{nArg}(\underline{y})\right), \dots, \\ \text{RcvVal}\left(\underline{y}, \text{nArg}(\underline{y}) + 1 - \alpha\left(\underline{f}\right)\right)\end{pmatrix}$
endpar
```

Therefore, at the end of the communication phase, $\text{nArg}(\underline{y})$ is the number of terms received at the location $\underline{y}$. if $\text{nArg}(\underline{y}) = 0$ the variable y does not require to be updated. If $\text{nArg}(\underline{y}) \geqslant 2$, more than one term have been received, so there might be a conflict. In that case, we assume that the variable y should not be updated.[25] So, at the end of the communication phase (the synchronization), the following updates are made for the variables $y_1, \dots, y_k \in \text{WriteComm}(\Pi)$:

```
par
    if $\text{nArg}(\underline{y_1}) = 1$ then $y_1 := \text{RcvVal}\left(\underline{y_1}, 1\right)$
  $\|$ $\vdots$
  $\|$ if $\text{nArg}(\underline{y_k}) = 1$ then $y_k := \text{RcvVal}\left(\underline{y_k}, 1\right)$
endpar
```

Therefore, in our example with the communication primitives read and write, the exploration witness $T(\mathbf{comm}_M)$ of the communication function is the closure by subterms of:

$$\begin{aligned}&\bigcup_{y \in \text{WriteComm}(\Pi)} \bigcup_{f \in \mathcal{L}(M)} \bigcup_{\alpha(f)=0} \left\{f, \text{RcvVal}\left(\underline{y}, \text{nArg}(\underline{y}) + 1\right)\right\}\\ &\bigcup_{\alpha(f) \geqslant 1} \left\{f\left(\text{RcvVal}\left(\underline{y}, \text{nArg}(\underline{y})\right), \dots, \text{RcvVal}\left(\underline{y}, \text{nArg}(\underline{y}) + 1 - \alpha\left(\underline{f}\right)\right)\right)\right\}\\ &\cup \left\{\mathbf{true}, \text{nArg}(\underline{y}) = 1, y, \text{RcvVal}\left(\underline{y}, 1\right)\right\}\end{aligned}$$

So, because $\text{WriteComm}(\Pi)$ and $\mathcal{L}(M)$ are finite, we have that $T(\mathbf{comm}_M)$ is finite too.

Proposition 4 *(Communication "à la BSPlib"). A function of communication with primitives for distant readings/writings (*DRMA*) "à la* BSPLIB*" can be constructed using* ASM$_{\text{BSP}}$*, with its exploration witness. It performs an h-relation at cost $\mathbf{g} \times h$ and endings the superstep at cost* $\mathbf{L}$*.*

This is possible using Lemma 8 p. 24. Notice that we did not specify how RcvVal is emptied at the beginning of the next communication phase. These are details which are left to the implementation and depend on the chosen data structure (we are still working up to elementary functions). We also not truly force an order of exchanges which would make the presentation even more complicated. In our example the latency $\mathbf{L}$ of the barrier includes, at the end of the communication phase, the assurance that every message has been received. The bandwidth $\mathbf{g}$ includes, at the beginning of the communication phase, the preparation of the messages themselves by a kind of serialization and, at the end of the communication phase, the update in the distant memories by the received messages (deserialization).

Notice that proving that an order of exchanges exists (Lemma 8) does not mean that the communication function can compute it efficiently. The communication function may perform a Latin square, or something more elaborate as in the implementation of the BSPLIB on some architectures [22]. But this is taken care of $\mathbf{L}$ which is, in the BSP model, the cost required to perform the barrier (synchronize the processors and managing the network that is the latency). In our realization

[25] This decision may be problematic if the same value has been sent several times, but we will not consider this problem here but a solution could be to distinguish them by using a counter (or tags).

of the function of communication, we did not want to force an order of exchanges because this would have made the presentation even more complicated.

Even if serialization is an internal calculation for the function (and so it counts in **g** only) as well as emptied the buffers of communication (in **L** only), one may argue that the fourth postulate allows the communication function to perform computations. To partially avoid it, we can assume that the terms in the exploration witness $T(M)$ can be separated between $T(\Pi)$ and $T(\mathbf{comm}_M)$ such that $T(\Pi)$ is for the states in a computation phase, and that for every update $(f, \overset{\circ}{\overline{a}}, b)$ of a processor X^i in a communication phase, either there exists a term $\theta \in T(\mathbf{comm}_M)$ such that $b = \overline{\theta}^{X^i}$, or there exists a variable $v \in T(\Pi)$ and a processor X^j such that $b = \overline{\theta_{\overline{v}^{X^j}}}^{X^i}$. To do a computation, a term like $x + 1$ is required, so the restriction to a variable prevents the computations of the terms in $T(\Pi)$. Of course, the last communication step should be able to write in $T(\Pi)$, and the final result should be also read in $T(\Pi)$.

2.8. Questions and answers

Q48: *Can I trust your realization of SubSection 2.7? Is it truly the* BSP*'s* DRMA *routines?*

Sorry to say we do not have the proof (there are possible flaws). To do such a proof, we must have a formal specification of these routines (such as a formal operational semantics) and, this specification must be accepted as valid by the community. Then we can "prove" that our realization is correct with respect to this specification. Otherwise, we would have just proven that our realization is correct with respect to our own specification, which amounts to say that our realization is a specification. Our realization is just an example of a function of communication of usual BSP routines and how to construct the exploration witness. Since it is unrealistic to want to run an $\mathrm{ASM}_{\mathrm{BSP}}$, it is not important that the function contains flaws, just that it is constructive. A much more interesting proof would be a machine-checked (with the COQ system) correct MPI implementation of the BSPLIB communications for a mainstream programming language such as C or JAVA.

Q49: *Is it why no mention is made of the* BSPLIB*'s registration routines "***bsp_push***" and "***bsp_pop***" in your realization?*

Indeed, adding such routines uselessly complicates the formalization for our purpose. For example, what happen when a distant variable is reading and, during the same phase of communication, deregistrating? That slightly differs of which BSP library is used [34].

Q50: *But, I do not see the common "***bsp_sync()***" in your realization, instead there is an uncommon sequence of a single communication function...*

As explained previously, we could transform this "**sync**" in a loop such as "**while** b_{comm} **do comm done**". Using a single "**sync**" (which performs all the communications) is impossible in our model because it will break the exploration witness which ensures that the maximal number of symbols communicated per processor is finite during all the executions (the machines can cause only a bounded local change on the states). If such a "**sync**" had existed then we would have an elementary primitive capable of exchanging in a single step an unbounded number of information (since it would be independent of the number of processors). This is impossible because ASMs work step-by-step and with bound primitives only (so the exploration witness). So our realization works one BSP's 1-relation at a time. This is not realistic in practice but sufficient to show its existence in a constructive way.

Q51: *When you defined informally the* BSP *algorithms in Section 2.1 p. 4, you said that during a computation phase the units "request* data transfers *to other units", not to communicate anything...*

The requests can be used to perform load-balancing, I/O operations and many other "data transfer" that the BSPLIB can not do. Thus we have abstracted the function of communication and let the user choose the one that suits its need (with related elementary functions, seen as oracular). To be constructive, we provided a realization of the BSPLIB's DRMA routines p. 24. Moreover, if the function of communication also performs some "computations" to manage the messages, it is as the management of datagrams in a network protocol.

Q52: *In that case, why is* $\mathbf{comm}_M$ *not able to compute, for example, the output of the algorithm? Which part of the condition forbids this?*

Its exploration witness, which ensures that the computations are performed step by step. Moreover, as in our example, we can separate the computations performed by the communication function from those of the processors.

Q53: *I think such* DRMA *primitives can be implemented in one step* $\mathbf{comm}_M$ *if we introduce arrays in the universe of a state. The exploration witness is still finite because all messages are stored in one array associated to one (special) symbol (with the needed primitives* *to extract values from an array).*

Actually, this has been investigated in the thesis of the first author. An array with an unbounded length is not a term, as opposed to the term(s) needed to implement an array with a fixed length. So, a fixed-size array can be sent by our communication function. In our example, we constructed a letter-by-letter communication function, but by sending messages of size n you can divide the number of required exchanges by n. But this n is a constant, so this will not really change the cost model for the communications.

Q54: *Speaking about the cost model, when reading your realization, I think you do not count all the moves of the identifiers (j, y, i, x)s made by* $\text{read}(y, i, x)$*s from processor X_j to X_i (for any i and $j \neq i$). So is the cost of computing and performing the h-relation of your function rather in $O(h)$?*

A message identifier is a message, so is already included in the h-relation. It is used to trigger the sending of the associated value (term) and therefore force a processor to throw communications. The preparation step (1) does not have to be proportional in h, for instance $c \times h$, to be added to the cost $\mathbf{g} \times h$ of transmitting the messages letter by letter, in order to obtain $(c + \mathbf{g}) \times h$, thus including the cost of the preparation in $\mathbf{g}$. The sending of the message identifiers are already included in h [21]. We did not detail this part because usually the message identifiers are seen as negligible, because their size is very small compared to the messages transmitted by truly values of BSP programs and negligible in the cost analysis of BSP algorithms [21].

Q55: *You say that the function of communication can force processors to throw new communications or worse computations. Is that a flaw or a strange feature?*

That is possible since the function of communication "has the hand" to the whole memory. And it is a desired feature for the distant readings (DRMA) because processors received the identifiers and then send the appropriate values. All these messages are part of the h-relation (even if some of them are negligible in practice). In a different kind of way, for the computation case of some BSP algorithms (such as the PREGEL-like ones [26], *i.e.* on graphs), the amount of data is distributed across the memories and thus some processors could have terminated their local computations (voting to "halt" [26]) and have no new message to send, but, it is not the global termination since possibly at least one processor can send new data to other processors thus relaunching the overall computation.

Q56: *So how are evaluated* **g** *and the h-relation? I mean you are sending both letters and identifiers.*

g depends of many architectural parameters. In C, on a beowulf cluster, you need also to take into account how data are buffered into a network card. For JAVA, time to serialize the objects is costly and is measured when benchmarking **g**. Thus only an over-approximation of **g** can be achieved. As we said in Q54 an identifier (i, x, j, y) is a message, as well as the n messages $(\underline{f}, j, y)$ used to send a term, where n is the size of the term. Let us call them "elementary messages" and let us assume, for sake of simplicity, that they have the fix size B (for instance, the size of the integers, 64 bits, is fixed for a given architecture[26]). For a given superstep and a given processor $0 \leqslant i < p$, let $h_i^{iden,in}$ (resp. $h_i^{iden,out}$) be the number of identifiers received (resp. sent) by i during the communication phase, and $h_i^{term,in}$ (resp. $h_i^{term,out}$) be the number of terms received (resp. sent) by i. Such sizes are fixed by the algorithm and the interpretation of the processor identifiers (which can be assumed to be uniform, see Section 2.2). Because the cost of the communication phase is $g \times h + L$ where g reflects the cost of performing a 1-relation (of the BSP machine executing the algorithm) and h the maximum number of words (sent or received) per processor, transmitting n messages of size 1 or 1 message of size n are seen as equivalent in the BSP model. Thus we have $h = \max_{0 \leqslant i < p}(h_i^{iden,in} + N.h_i^{term,in}, h_i^{iden,out} + N.h_i^{term,out})$. Actually, in our realization of the communication function, every processor performs one sending/receiving of elementary message per communication step. To summarize, up to implementation details (uniform interpretation of the processor identifiers, size of the data), h is given by the algorithm itself (the pattern of communication).

Q57: *Is it also the case for* **L***?*

Indeed. It also depends on some architectural parameters and on which routines are available (*e.g.* DRMA with or without buffering of data). It is the case for our realization where we update Msg and IdMsg.

Q58: *Is using a single function of communication not a problem? I think of a faulty algorithm that would use two different MPI's collective routines to terminate a superstep.*

[26] In the PUB [51], a BSPLIB-like library, B is the size of a datagram and a 1-relation is for B words in place of a single word; Notice that our realization of the communication is naive since letters are transmitted one by one in place of batching all the data (for a bulk sending, in order to optimize the use of the network) as it is done in most BSP libraries [23,51].

This is not a problem. We can imagine such a function working by cases and generating an **undef** otherwise which forces the ASMs to stop.

Q59: *Fine, you obtained a realization of the communication function for* BSP. *But this has been done by using an* ASM *program, not a* BSP *one.*

You are right. It is time to introduce you to the imperative programming language that we will use for the rest of the paper.

3. Algorithm completeness and imperative characterization of BSP

Because a programmer prefers to use a mainstream language to get his algorithms implemented, we present a SPMD extension of the standard IMP core-language. The IMP programs are only sequential updates, conditionals and unbounded loops, so this programming language can be seen as minimal. That means that the results of this section can be generalized to more common and complex imperative languages such as C or JAVA. As before, the IMP_{BSP} machines use first-order structures, and are not limited to a particular architecture. Any concrete data structure which can be used for ASM_{BSP} can be used for IMP_{BSP}, and *vice versa*. In other words, we are working *up to data structures and their elementary operations* (such as the function of communication) and by comparing the algorithmic expressiveness of the control flow statements. But before presenting the operational semantics of IMP_{BSP}, we first have to formalize the notion of *algorithm completeness*.

3.1. Algorithmic simulation and completeness [12]

Sometimes a literal identity can be proven between two models of computation [14]. But generally, only a *simulation* can be proven between them. We say that a computation model M_1 can simulate another computation model M_2 if for every program P_2 of M_2 there exists a program P_1 of M_1 producing "similar" executions from the initial states. By "similar" we mean up to a bounded number of *fresh variables* and up to *temporal dilation*.

3.1.1. Fresh variables

For instance, an exchange $x \leftrightarrow y$ between two variables x and y can be simulated by $v := x;\ x := y;\ y := v$, by using a *fresh variable* v. So, the signature $\mathcal{L}_1$ of the simulating program is an extension of the signature $\mathcal{L}_2$ of the simulated program, in other words $\mathcal{L}_1 \supseteq \mathcal{L}_2$.

The signature must remain finite, so $\mathcal{L}_1 \backslash \mathcal{L}_2$ must be a finite set of variables, but this is not enough. To prevent the introduction of an unbounded amount of information, we also assume that the "fresh" variables are *uniformly initialized*: their values are the same in the initial processors, up to isomorphism. To do so we introduce the notion of restriction of structures:

Definition 13 *(Restriction of structures).* Let X be a structure with signature $\mathcal{L}_1$, and let $\mathcal{L}_2$ be a signature such that $\mathcal{L}_1 \supseteq \mathcal{L}_2$. The *restriction* of X to the signature $\mathcal{L}_2$ will be denoted $X|_{\mathcal{L}_2}$ and is defined as a structure of signature $\mathcal{L}_2$ such that $\mathcal{U}(X|_{\mathcal{L}_2}) = \mathcal{U}(X)$ and for every $f \in \mathcal{L}_2$ we have $\overline{f}^{X|_{\mathcal{L}_2}} = \overline{f}^{X}$. Moreover, X will be called an *extension* of $X|_{\mathcal{L}_2}$.

3.1.2. Temporal dilation

During every "step" of a Turing machine, the state of the machine is updated, a symbol is written in the cell, and the head may move left or right. But the notion of elementary action is arbitrary. We may consider either a machine M_1 requiring three steps to do these three elementary actions, or a machine M_3 requiring only one step to do the three aspects of one multi-action. An execution $\vec{\mathcal{T}}$ of M_3 corresponds to an execution $\vec{\mathcal{S}}$ of M_1 if for every three steps of M_1 the state is the same as M_3:

$$\begin{array}{l|llllllllll} M_1 & \mathcal{S}_0, & \mathcal{S}_1, & \mathcal{S}_2, & \mathcal{S}_3, & \mathcal{S}_4, & \mathcal{S}_5, & \mathcal{S}_6, & \mathcal{S}_7, & \mathcal{S}_8, & \ldots \\ \hline M_3 & \mathcal{T}_0, & & & \mathcal{T}_1, & & & \mathcal{T}_2, & & & \ldots \end{array}$$

In other words, if M_1 is three times faster than M_3, these executions may be considered equivalent, so we will say that there is a temporal dilation $d = 3$. In this example, the dilation is uniform for every program, but this temporal dilation may depend on the simulated program (but not on an initial state).

Notice that the temporal dilation is not sufficient to ensure that the final states are the same. Indeed, in the previous example, $\mathcal{T}_t$ may be terminal for M_1, but we may have an infinite loop for M_1:

$$\begin{array}{l|llllllll} M_1 & \ldots, & \mathcal{S}_{3t}, & \mathcal{S}_{3t+1}, & \mathcal{S}_{3t+2}, & \mathcal{S}_{3t}, & \mathcal{S}_{3t+1}, & \mathcal{S}_{3t+2}, & \ldots \\ \hline M_3 & \ldots, & \mathcal{T}_t, & & & \mathcal{T}_t, & & & \ldots \end{array}$$

So, we add to the following definition an *ending time* e ensuring that the simulating program terminates if the simulated program has terminated:

Definition 14 *(Algorithmic simulation).* Let M_1 and M_2 be two machines. We say that M_1 *simulates* M_2 if for every program P_2 of M_2 there exists a program P_1 of M_1 such that: (1) $\mathcal{L}(P_1) \supseteq \mathcal{L}(P_2)$, and $\mathcal{L}(P_1) \setminus \mathcal{L}(P_2)$ is a finite set of fresh variables depending only on P_2 and uniformly initialized; and there exists $d > 0$ and $e \geqslant 0$ depending both only on P_2 such that for every execution $\vec{\mathcal{T}} = \mathcal{T}_0, \mathcal{T}_1, \dots$ of P_2 there exists an execution $\vec{\mathcal{S}} = \mathcal{S}_0, \mathcal{S}_1, \dots$ of P_1 verifying that (2) for every $t \in \mathbb{N}$, $\mathcal{S}_{d \times t}|_{\mathcal{L}(P_2)} = \mathcal{T}_t$, and (3) $\text{time}(P_1, \mathcal{S}_0) = d \times \text{time}(P_2, \mathcal{T}_0) + e$.

Notice that these parameters (*i.e.* the fresh variables, the temporal dilation d and the ending time e) depend on what is simulated and not on a particular state of the execution. If M_1 simulates M_2 and M_2 simulates M_1 then they are said *algorithmically equivalent* which is denoted by $M_1 \simeq M_2$. *Algorithm completeness* is when a model of computation is algorithmically equivalent to the entire *class* of algorithms [48] for the functions it computes. More discussions about the simulation could be find in Q62-Q67 p. 39.

3.2. An imperative characterization of BSP

An IMP$_{\text{BSP}}$ program is a common *sequence of commands*, which are standard *control flow statements* or *assignments*. These assignments can contain primitives for computation or operations on buffer to prepare the communications steps (*e.g.* BSP DRMA operations, see SubSection 2.7). The main difference with a usual sequential program is the primitive of communication:

Definition 15 *(Syntax of imperative programs).*

$$\begin{aligned}
\text{commands: } c &\overset{\text{def}}{=} f(\theta_1, \dots, \theta_\alpha) := \theta_0 \\
&\mid \textbf{if } F \ \{P_1\} \textbf{ else } \{P_2\} \\
&\mid \textbf{while } F \ \{P\} \\
&\mid \textbf{comm} \\
\text{programs: } P &\overset{\text{def}}{=} \textbf{end} \ \mid \ c; P
\end{aligned}$$

where F is a formula, f has arity α and $\theta_1, \dots, \theta_\alpha, \theta_0$ are terms. To improve readability and when there is no ambiguity, we write **if** F $\{P\}$ for the command **if** F $\{P\}$ **else** $\{\textbf{end}\}$. The composition of commands $c; P$ is extended to *composition of programs* $P_1 \fatsemi P_2$ with the following inductive rules: $(\textbf{end} \fatsemi P_2) \overset{\text{def}}{=} P_2$ and $(c; P_1) \fatsemi P_2 \overset{\text{def}}{=} c; (P_1 \fatsemi P_2)$.

A processor follows the operational semantics for of its IMP$_{\text{BSP}}$ program, and thus computes locally until it reaches a **comm** (in which case it waits for the global communication step) or an **end** command. The operational semantics of the local computations is formalized by a state transition system, where a state of the system is a pair $P \star X$ of a program P and a structure X, and a transition $\succ$ is determined only by the head command and the current structure:

Definition 16 *(Sequential operational semantics [12]).*

$$\begin{array}{rcll}
f(\theta_1, \dots, \theta_\alpha) := \theta_0; P \star X & \succ & P \star X \oplus (f, \overline{\theta_1}^X, \dots, \overline{\theta_\alpha}^X, \overline{\theta_0}^X) & \\
\textbf{if } F \ \{P_1\} \textbf{ else } \{P_2\}; P_3 \star X & \succ & P_1 \fatsemi P_3 \star X & \text{if } \overline{F}^X = \textbf{true} \\
\textbf{if } F \ \{P_1\} \textbf{ else } \{P_2\}; P_3 \star X & \succ & P_2 \fatsemi P_3 \star X & \text{otherwise} \\
\textbf{while } F \ \{P_1\}; P_2 \star X & \succ & P_1 \fatsemi \textbf{while } F \ \{P_1\}; P_2 \star X & \text{if } \overline{F}^X = \textbf{true} \\
\textbf{while } F \ \{P_1\}; P_2 \star X & \succ & P_2 \star X & \text{otherwise}
\end{array}$$

It is easy to prove that this transition system is deterministic. We denote by $\succ_t$ the succession of t steps. The states without successors are **comm**; $P \star X$ and **end** $\star X$. We say that a program P *terminates locally* on a local memory X, denoted by $P \overset{\mathbf{e}}{\curlyvee} X$, if there exists t and X' such that $P \star X \succ_t \textbf{end} \star X'$. Moreover, a program P *waits for a communication* on a local memory X, denoted by $P \overset{\mathbf{c}}{\curlyvee} X$, if there exists t, P' and X' such that $P \star X \succ_t \textbf{comm}; P' \star X'$. Because the state transition system is deterministic, these t and X' are unique, so t will be denoted by $\text{time}_{\text{seq}}(P, X)$, and X' by $P(X)$. If P does not terminate locally nor waits for a communication on X, denoted by $P \overset{\infty}{\curlyvee} X$, then we assume that $\text{time}_{\text{seq}}(P, X) = \infty$.

For every $0 \leqslant t \leqslant \text{time}_{\text{seq}}(P, X)$, there exists a unique P_t and X_t such that $P \star X \succ_t P_t \star X_t$. This P_t will be denoted by $\tau_X^t(P)$, and this X_t by $\tau_P^t(X)$. Notice that τ_P is not a transition function because $\tau_P^t(X) \neq \tau_P^1 \circ \dots \circ \tau_P^1(X)$. We denote by $\Delta(P, X)$ the succession of updates made by P on X:

$$\Delta(P, X) \overset{\text{def}}{=} \bigcup_{0 \leqslant t < \text{time}_{\text{seq}}(P, X)} \tau_P^{t+1}(X) \ominus \tau_P^t(X)$$

Unlike the ASM, only one update can be done at each step. Therefore, if $P \overset{\mathbf{e}}{\curlyvee} X$ or $P \overset{\mathbf{c}}{\curlyvee} X$ then $\Delta(P, X)$ is finite. Moreover, we say that P is *without overwrite* on X if $\Delta(P, X)$ is consistent.

Lemma 10 *(Updates without overwrite). If $P \overset{\mathbf{e}}{\curlyvee} X$ or $P \overset{\mathbf{c}}{\curlyvee} X$ without overwrite then $\Delta(P, X) = P(X) \ominus X$.*

Proof (Sketch of). The proof is made by induction on $\mathrm{time}_{\mathrm{seq}}(P, X)$. The only difference with the purely sequential case [12] is that programs can terminate on the **comm** statement. □

The operational semantics of IMP$_{\mathrm{BSP}}$ is formalized by a state transition system, where a state of the system is a pair $\vec{P} \star \vec{X}$ of a p-tuple $\vec{P} = \left(P^1, \ldots, P^p\right)$ of programs, and a p-tuple $\vec{X} = \left(X^1, \ldots, X^p\right)$ of local memories. $\vec{P}$ will be said in a *computation phase* if there exists $1 \leqslant i \leqslant p$ such that $P^i \neq \mathbf{end}$ and $P^i \neq \mathbf{comm}; P$. Otherwise, $\vec{P}$ will be said in a *communication phase*. We define:

$$\vec{\tau}_{\vec{X}}(\vec{P}) \overset{\mathrm{def}}{=} \left(\tau_{X^1}(P^1), \ldots, \tau_{X^p}(P^p)\right)$$
$$\vec{\tau}_{\vec{P}}(\vec{X}) \overset{\mathrm{def}}{=} \left(\tau_{P^1}(X^1), \ldots, \tau_{P^p}(X^p)\right)$$

where, for every $P \star X$, $\tau_X(P)$ and $\tau_P(X)$ are defined by:

$$P \star X \succ \tau_X(P) \star \tau_P(X) \ \text{ if } P \star X \text{ has a successor}$$
$$P \star X = \tau_X(P) \star \tau_P(X) \ \text{ otherwise}$$

For the following definition, we recall the use of $\ominus$ on p-tuples: we denote by $\vec{\tau}_{\vec{P}}(\vec{X}) \ominus \vec{X}$ the p-tuple $\left(\tau_{P^1}(X^1) \ominus X^1, \ldots, \tau_{P^p}(X^p) \ominus X^p\right)$.

Definition 17 *(Semantics of the BSP imperative core-language).* An IMP$_{\mathrm{BSP}}$ machine M is a quadruplet $(S(M), I(M), P_{init}, \mathbf{comm}_M)$ verifying that:

(1) $S(M)$ is a set of p-tuples of processors with the same finite signature $\mathcal{L}(M)$ containing the booleans and the equality;
(2) The initial states of the transition system have the form $\overrightarrow{P_{init}} \star \vec{X}$, where $\vec{X} \in I(M) \subseteq S(M)$ and $\overrightarrow{P_{init}} = (P_{init}, \ldots, P_{init})$;
(3) P_{init} is a program with terms from $\mathcal{L}(M)$;
(4) $\mathbf{comm}_M : S(M) \mapsto S(M)$ verifies (as in Definition 11 p. 15) that:
    (1) For every state $\vec{P} \star \vec{X}$ such that $\vec{P}$ is in a communication phase, $\mathbf{comm}_M$ preserves the universes and the number of processors, and commutes with multi-isomorphisms;
    (2) There exists a finite set of terms $T(\mathbf{comm}_M)$ such that for every state $\vec{P} \star \vec{X}$ and $\vec{Q} \star \vec{Y}$, if $\vec{P}$ and $\vec{Q}$ are in a communication phase and $\vec{X}$ and $\vec{Y}$ coincide over $T(\mathbf{comm}_M)$ then $\vec{\tau}_{\vec{P}}(\vec{X}) \ominus \vec{X} = \vec{\tau}_{\vec{Q}}(\vec{Y}) \ominus \vec{Y}$ (that is $\mathbf{comm}_M(\vec{X}) \ominus \vec{X} = \mathbf{comm}_M(\vec{Y}) \ominus \vec{Y}$).

The operational semantics $\vec{\succ}$ of IMP$_{\mathrm{BSP}}$ is defined by:

$$\vec{P} \star \vec{X} \,\vec{\succ} \begin{cases} \vec{\tau}_{\vec{X}}(\vec{P}) \star \vec{\tau}_{\vec{P}}(\vec{X}) & \text{if } \vec{P} \text{ is in a computation phase} \\ \overrightarrow{\mathbf{next}}(\vec{P}) \star \mathbf{comm}_M(\vec{X}) & \text{if } \vec{P} \text{ is in a communication phase} \end{cases}$$

where $\overrightarrow{\mathbf{next}}(P^1, \ldots, P^p) = \left(\mathbf{next}(P^1), \ldots, \mathbf{next}(P^p)\right)$, with $\mathbf{next}(\mathbf{comm}; P) = P$ and $\mathbf{next}(\mathbf{end}) = \mathbf{end}$.

As previously, a superstep is composed of a phase of computation followed by a phase of communication, and the whole execution of machine is a sequence of supersteps. Because $\succ$ and $\mathbf{comm}_M$ are deterministic, this transition system is also deterministic. We denote by $\vec{\succ}_t$ the succession of t steps. According to the previous definitions, the fixpoints[27] of the transition system are states $\vec{P} \star \vec{X}$ such that for every $1 \leqslant i \leqslant p$, $P_i = \mathbf{while}\ F\ \{\mathbf{end}\}; P'$ (in which case F is true in X_i) or $P_i = \mathbf{end}$, and such that if $\vec{P}$ is in a communication phase then $\mathbf{comm}_M(\vec{X}) = \vec{X}$.

A state $\vec{P} \star \vec{X}$ is said *final* if $\mathbf{comm}_M(\vec{X}) = \vec{X}$ and for every $1 \leqslant i \leqslant p$, $P_i = \mathbf{end}$. We say that, on local memories $\vec{X}$, the (initial) program P_{init} *terminates globally*, which will be denoted by $P_{init} \curlyvee \vec{X}$, if there exists t, $\vec{P'}$ and $\vec{X'}$ such that $\overrightarrow{P_{init}} \star \vec{X} \,\vec{\succ}_t\, \vec{P'} \star \vec{X'}$, where $\vec{P'} \star \vec{X'}$ is a final state. We denote by $\mathrm{time}(P_{init}, \vec{X})$ the smallest of those t, by assuming (as in the Definition 3 p. 6) potential infinite duration if P_{init} does not terminate globally[28] on $\vec{X}$. Moreover, because the reached state is a fixpoint, $\vec{X'}$ is unique, and will be denoted by $\overrightarrow{P_{init}}(\vec{X})$.

Notice that in IMP$_{\mathrm{BSP}}$, because the communication steps cannot begin until the end of the computation steps, we have for every computation step $\vec{P} \star \vec{X}$ (such that every P_i terminates locally or waits for a communication on X_i) that

[27] The states such that the structures and the "instruction pointer" of the programs cannot be changed anymore.
[28] There is an infinite number of supersteps or during one of them, at least, the computations of one processor diverges.

```
                 while (2^j < p) {
                     if (pid > 2^j) {
                         read(y, r, pid − 2^j);
                         while b_comm {comm; end};
                         b_comm := true;
                         r := y op r;
                         end
P_init  def=         } else {
                         while b_comm {comm; end};
                         b_comm := true;
                         end
                     };
                     j := j + 1;
                     end
                 }; end
```

Fig. 6. Imperative program for the prefix computation.

$\text{time}(\overrightarrow{P}, \overrightarrow{X}) = \text{time}(\overrightarrow{P'}, \overrightarrow{X'}) + \max_{i=1}^{p} \text{time}_{\text{seq}}(P_i, X_i)$, where $\overrightarrow{P'} \star \overrightarrow{X'}$ is the first communication step after $\overrightarrow{P} \star \overrightarrow{X}$. This is intended for the cost model, detailed in Subsection 2.6 p. 24.

Example 5. We can use the imperative program of Fig. 6 p. 33 for the prefix computation (see p. 10), by assuming as in SubSection 2.7 p. 24 a boolean b_{comm} initialized to **true** and updated to **false** when the communication function has finished to transmit the messages. As before, $\mathtt{read}(\mathtt{y}, \mathtt{r}, \mathtt{pid} - 2^{\mathtt{j}})$ is only syntactic sugar to manage the communication buffers. The other initializations and the execution are similar to Example 4 p. 16, except that the program evolves for every processor (which can be seen as an instruction pointer).

3.3. *Algorithmic completeness of a core-imperative BSP language*

Now that we have defined our core BSP programming language as a common imperative model of computation, we prove in the following subsection that ASM_{BSP} algorithmically simulates (Definition 14 p. 31) IMP_{BSP}, and in the SubSection 3.3.2 p. 35 that IMP_{BSP} algorithmically simulates ASM_{BSP}.

3.3.1. ASM_{BSP} *simulates* IMP_{BSP}

The simplest idea to translate an IMP_{BSP} program P into an ASM program is to add a variable for the current line of the program and follow the flow of execution. It has been showed in [12] why such a method raises some difficulties notably to define the translation inductively. Instead, we will add a *bounded* number of boolean variables $b_{P_1}, \dots, b_{P_k}$, where $P_1, \dots, P_k$ are the programs occurring during a possible execution of the simulated program P. The set of these programs will be called the control Flow Graph (CFG) $\mathcal{G}(P)$ of the program P, and differs from [12] only because of **comm**:

Definition 18 *(Control flow graph).*

$$
\begin{aligned}
\mathcal{G}(\textbf{end}) &\overset{\text{def}}{=} \{\textbf{end}\} \\
\mathcal{G}(\textbf{comm}; P) &\overset{\text{def}}{=} \{\textbf{comm}; P\} \\
&\quad \cup \ \mathcal{G}(P) \\
\mathcal{G}(f\,(\theta_1, \dots, \theta_\alpha) := \theta_0; P) &\overset{\text{def}}{=} \{f\,(\theta_1, \dots, \theta_\alpha) := \theta_0; P\} \\
&\quad \cup \ \mathcal{G}(P) \\
\mathcal{G}(\textbf{if } F\ \{P_1\} \textbf{ else } \{P_2\}; P_3) &\overset{\text{def}}{=} \{\textbf{if } F\ \{P_1\} \textbf{ else } \{P_2\}; P_3\} \\
&\quad \cup \ \mathcal{G}(P_1) \overset{*}{\rightharpoonup} P_3 \\
&\quad \cup \ \mathcal{G}(P_2) \overset{*}{\rightharpoonup} P_3 \\
&\quad \cup \mathcal{G}(P_3) \\
\mathcal{G}(\textbf{while } F\ \{P_1\}; P_2) &\overset{\text{def}}{=} \mathcal{G}(P_1) \overset{*}{\rightharpoonup} (\textbf{while } F\ \{P_1\}; P_2) \\
&\quad \cup \ \mathcal{G}(P_2)
\end{aligned}
$$

where $\mathcal{G}(P_1) \overset{*}{\rightharpoonup} P_2 \overset{\text{def}}{=} \{P \mathbin{⨟} P_2 \mid P \in \mathcal{G}(P_1)\}$.

$$
\begin{array}{rcl}
[\![\mathbf{end}]\!]^{\text{ASM}} & \overset{\text{def}}{=} & \mathbf{par}\ \ \mathbf{endpar} \\
[\![\mathbf{comm};\, P]\!]^{\text{ASM}} & \overset{\text{def}}{=} & \mathbf{par}\ \ \mathbf{endpar} \\
[\![f\,(\theta_1,\ldots,\theta_\alpha) := \theta_0;\, P]\!]^{\text{ASM}} & \overset{\text{def}}{=} & \begin{array}{l} \mathbf{par} \\ \quad b_{①} := \mathbf{false} \\ \parallel f\,(\theta_1,\ldots,\theta_\alpha) := \theta_0 \\ \parallel b_{②} := \mathbf{true} \\ \mathbf{endpar} \end{array} \\
[\![\mathbf{if}\ F\ \{P_1\}\ \mathbf{else}\ \{P_2\};\, P_3]\!]^{\text{ASM}} & \overset{\text{def}}{=} & \begin{array}{l} \mathbf{par} \\ \quad b_{③} := \mathbf{false} \\ \parallel \mathbf{if}\ F\ \mathbf{then}\ b_{④} := \mathbf{true}\ \mathbf{else}\ b_{⑤} := \mathbf{true}\ \mathbf{endif} \\ \mathbf{endpar} \end{array} \\
[\![\mathbf{while}\ F\ \{c;\, P_1\};\, P_2]\!]^{\text{ASM}} & \overset{\text{def}}{=} & \begin{array}{l} \mathbf{par} \\ \quad b_{⑥} := \mathbf{false} \\ \parallel \mathbf{if}\ F\ \mathbf{then}\ b_{⑦} := \mathbf{true}\ \mathbf{else}\ b_{⑧} := \mathbf{true}\ \mathbf{endif} \\ \mathbf{endpar} \end{array} \\
[\![\mathbf{while}\ F\ \{\mathbf{end}\};\, P_2]\!]^{\text{ASM}} & \overset{\text{def}}{=} & \begin{array}{l} \mathbf{if}\ F\ \mathbf{then}\ b_\infty := \neg b_\infty \\ \mathbf{else\ par} \\ \qquad\qquad b_{⑧} := \mathbf{true} \\ \qquad\quad \parallel b_{⑨} := \mathbf{false} \\ \qquad \mathbf{endpar} \\ \mathbf{endif} \end{array}
\end{array}
$$

where:

Assignment :

$① \overset{\text{def}}{=} f\,(\theta_1,\ldots,\theta_\alpha) := \theta_0;\, P$

$② \overset{\text{def}}{=} P$

Conditional :

$③ \overset{\text{def}}{=} \mathbf{if}\ F\ \{P_1\}\ \mathbf{else}\ \{P_2\};\, P_3$

$④ \overset{\text{def}}{=} P_1 \,\S\, P_3$

$⑤ \overset{\text{def}}{=} P_2 \,\S\, P_3$

Loop :

$⑥ \overset{\text{def}}{=} \mathbf{while}\ F\ \{c;\, P_1\};\, P_2$

$⑦ \overset{\text{def}}{=} c;\, P_1 \,\S\, \mathbf{while}\ F\ \{c;\, P_1\};\, P_2$

$⑧ \overset{\text{def}}{=} P_2$

$⑨ \overset{\text{def}}{=} \mathbf{while}\ F\ \{\mathbf{end}\};\, P_2$

Fig. 7. Translation of one step of a program.

By induction, we have $\text{card}(\mathcal{G}(P)) \leqslant \text{length}(P) + 1$, where the length (number of instructions) of a program P is defined by induction in the usual way (see [12] for a formal definition). Therefore, the number of boolean variables b_{P_i}, where $P_i \in \mathcal{G}(P)$, is bounded by a number depending only of P.

Each processor X of the IMP$_{\text{BSP}}$ machine M will be simulated by the ASM$_{\text{BSP}}$ machine A by using the same processor X but with new symbols $\mathcal{L}(A) = \mathcal{L}(M) \cup \{b_{P_i} \mid P_i \in \mathcal{G}(P)\}$ such that one and only one of the b_{P_i} is **true**. This extension of the structure X will be denoted by $X[b_{P_i}]$. An initial processor X for the program P_{init} will be simulated by the processor $X[b_{P_{init}}]$, and during the execution, the translation Π_P of the program P determines whether a boolean variable is **true** (or not) at a given step of a given processor:

Definition 19 *(Translation of a program P into an ASM program Π_P).*

$$
\Pi_P \overset{\text{def}}{=} \begin{array}{l} \mathbf{par} \\ \quad \mathbf{if}\ b_{P_1}\ \mathbf{then}\ [\![P_1]\!]^{\text{ASM}}\ \mathbf{endif} \\ \parallel \vdots \\ \parallel \mathbf{if}\ b_{P_k}\ \mathbf{then}\ [\![P_k]\!]^{\text{ASM}}\ \mathbf{endif} \\ \mathbf{endpar} \end{array}
$$

such that $\mathcal{G}(P) = \{P_1, \ldots, P_k\}$ and where the translation $[\![P_i]\!]^{\text{ASM}}$ of the first step of P_i is defined at the Fig. 7 p. 34. Notice that the **comm** is translated to an empty code because, in the ASM$_{\text{BSP}}$, the communication steps are always performed by the communication function when the local computation steps have terminated. Other statements are translated as in the sequential case.

Notice that we distinguished two cases for the translation of the **while** command. Indeed, if we did not, then the ASM would have tried to do two updates $b_{\mathbf{while}\ F\ \{P_1\};P_2} := \mathbf{false}$ and $b_{P_1;\mathbf{while}\ F\ \{P_1\};P_2} := \mathbf{true}$. But, in the case where the body $P_1 = \mathbf{end}$, both would have clashed and the ASM$_{\text{BSP}}$ would have stopped. This behavior is not compatible with the IMP$_{\text{BSP}}$ program, which would had looped forever. To preserve this behavior, we modified the translation and added a fresh variable b_∞ which prevents the ASM$_{\text{BSP}}$ from terminating. Another solution, but less general, would have been to forbid empty body for the **while** commands (as in [12]).

We can now prove that, during the computation phases, the ASM program Π_P simulates step-by-step the program P and so for each processor X:

Lemma 11 *(Step-by-step simulation of the sequential parts). For every* $0 \leqslant t < \text{time}_{\text{seq}}(P, X)$*:*

$$\tau_{\Pi_P}(\tau_P^t(X)[b_{\tau_X^t(P)}]) = \tau_P^{t+1}(X)[b_{\tau_X^{t+1}(P)}]$$

Proof (Sketch of). The proof is made by case on $\tau_X^t(P)$ and by induction on t, by using the fact that the $b_{\tau_X^t(P)}$ are fresh and that for every $P_{\mathcal{G}} \in \mathcal{G}(P)$, $\Delta(\Pi_P, X[b_{P_{\mathcal{G}}}]) = \Delta([\![P_{\mathcal{G}}]\!]^{\text{ASM}}, X[b_{P_{\mathcal{G}}}])$. The proof is as in the sequential case [12], except when the processor waits for a communication. □

Therefore, ASM$_{\text{BSP}}$ algorithmically simulates IMP$_{\text{BSP}}$ with at most length$(P) + 2$ fresh variables, a temporal dilation $d = 1$ and an ending time $e = 0$:

Proposition 5. ASM$_{\text{BSP}}$ *algorithmically simulates* IMP$_{\text{BSP}}$.

Proof (Sketch of). Let $M = (S(M), I(M), P_{init}, \mathbf{comm}_M)$ be an IMP$_{\text{BSP}}$ machine, and let $A = (S(A), I(A), \tau_A)$ be the ASM$_{\text{BSP}}$ machine[29] defined by: $S(A)$ is an extension of $S(M)$ by using the boolean variables b_{P_i} with $P_i \in \mathcal{G}(P)$, and b_∞; every $X \in I(A)$ is a $X|_{\mathcal{L}(M)} \in I(M)$ such that only b_P is **true**; the ASM program of the machine A is the program Π_P in Definition 19 p. 34; and $\mathbf{comm}_A$ is $\mathbf{comm}_M$ for the symbols of $\mathcal{L}(M)$, and for the new boolean variables it works as follows: if $b_{\mathbf{comm};P}$ is **true** in a processor then $b_{\mathbf{comm};P}$ becomes **false** and b_P becomes **true**; otherwise the boolean variables b_{P_i} and b_∞ remain unchanged. According to the Definition 14 p. 31 of an algorithmic simulation, there are three points to prove:

(1) $\mathcal{L}(A) = \mathcal{L}(M) \cup \{b_{P_i} \mid P_i \in \mathcal{G}(P)\} \cup \{b_\infty\}$, where card$(\mathcal{G}(P)) \leqslant$ length$(P) + 1$, so the number of fresh variables is finite and depends only of the simulated program P.

During a computation phase, the ASM program Π_P is applied to every processor, and according to the Lemma 11 the execution corresponds to the operational semantics of the IMP programs in the Definition 16 p. 31. A processor i terminates if $b_{\mathbf{comm};P}$ ($P_i \overset{\mathbf{c}}{\curlyvee} X_i$ case) or $b_{\mathbf{end}}$ ($P_i \overset{\mathbf{e}}{\curlyvee} X_i$ case) is **true** (for the current IMP$_{\text{BSP}}$ program P_i on processor i). When every processor has terminated, the state is in a communication phase and $\mathbf{comm}_A$ applies the communication function $\mathbf{comm}_M$ and updates the boolean variables from $b_{\mathbf{comm};P}$ to b_P, thus respecting the behavior of the function **next** (see Definition 17 p. 32).

(1) One step of the IMP$_{\text{BSP}}$ machine M is simulated by $d = 1$ step of the ASM$_{\text{BSP}}$ machine A (during both computation and communication phases);
(2) Moreover, a state $\overrightarrow{X}$ of the ASM$_{\text{BSP}}$ machine A is final if $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$ and $\mathbf{comm}_A(\overrightarrow{X}) = \overrightarrow{X}$, and this happens if and only if $b_{\mathbf{end}}$ is **true** for every processor and $\mathbf{comm}_M(\overrightarrow{X}) = \overrightarrow{X}$. Therefore, the ASM$_{\text{BSP}}$ machine A stops if and only if the IMP$_{\text{BSP}}$ machine M stops, and the ending time is $e = 0$. □

3.3.2. IMP$_{\text{BSP}}$ *simulates* ASM$_{\text{BSP}}$

We prove this second simulation (the reverse of the previous simulation) in two steps: (1) We translate (as in the sequential case) an ASM program Π into an imperative program P_Π^{step} simulating one step of Π; (2) Then, we construct an imperative program P_Π which repeats P_Π^{step} during the computation phase, and detects the communication phase by using a formula F_Π^{end}.

Because the ASM$_{\text{BSP}}$ and IMP$_{\text{BSP}}$ programs have the same updates and conditionals, a naive sequentialization $[\![\Pi]\!]^{\text{IMP}}$ of an ASM program Π does not work [12], because an ASM program like **par** $x := y \parallel y := x$ **endpar** which exchanges the values of the variables x and y would be translated into $x := y;\ y := x;$ **end** which sets the value of x to the value of y and leaves y unchanged. To capture the simultaneous behavior of the ASM programs, we need to substitute the terms $\theta_1, \ldots, \theta_r$ read in Π (where $r = \text{card}(\text{Read}(\Pi))$, see p. 15) by fresh variables $v_{\theta_1}, \ldots, v_{\theta_r}$, to obtain the program $[\![\Pi]\!]^{\text{IMP}}[\overrightarrow{v_\theta}/\overrightarrow{\theta}]$.

Thus, for every term $\theta \in \text{Read}(\Pi)$, we add to $\mathcal{L}(\Pi)$ a fresh variable v_θ. The program $v_{\theta_1} := \theta_1; \ldots; v_{\theta_r} := \theta_r;\ [\![\Pi]\!]^{\text{IMP}}[\overrightarrow{v_\theta}/\overrightarrow{\theta}]$ captures the simultaneous behavior of the ASM program Π. But, according to the Definition 14 p. 31 of the algorithmic simulation, one step of the ASM program Π must be translated into exactly d steps of the program. Thus, we must ensure that our translated program computes the same number of steps for every execution. According to the Corollary 3 p. 19, we can assume that the ASM program Π is in normal form:

[29] Notice that Π_P is an ASM program, and that because we extended the $\mathbf{comm}_M$ function (which verifies the third postulate) with a bounded number of fresh variables, $\mathbf{comm}_A$ has an exploration witness. Thus, A is actually an ASM$_{\text{BSP}}$ machine.

```
$\Pi =$
        if $F_1$ then
         par $f(\overrightarrow{\theta_1^1}) := \theta_1^1$
              $\vdots$
            $\|$
            $\|$ $f(\overrightarrow{\theta_{m_1}^1}) := \theta_{m_1}^1$
      endpar
      $\vdots$

else if $F_c$ then
         par $f(\overrightarrow{\theta_1^c}) := \theta_1^c$
              $\vdots$
            $\|$
            $\|$ $f(\overrightarrow{\theta_{m_c}^c}) := \theta_{m_c}^c$
      endpar
endif ... endif
```

```
$P_\Pi^{\text{step}} \overset{\text{def}}{=}$
   $v_{\theta_1} := \theta_1$;
        $\vdots$
   $v_{\theta_r} := \theta_r$;
   if $v_{F_1}$ {
     $f(\overrightarrow{v_{\theta_1^1}}) := v_{\theta_1^1}$;
          $\vdots$
     $f(\overrightarrow{v_{\theta_{m_1}^1}}) := v_{\theta_{m_1}^1}$;
     skip $(c+m) - (1+m_1)$}

   $\vdots$

else { if $v_{F_c}$ {
     $f(\overrightarrow{v_{\theta_1^c}}) := v_{\theta_1^c}$;
          $\vdots$
     $f(\overrightarrow{v_{\theta_{m_c}^c}}) := v_{\theta_{m_c}^c}$;
     skip $(c+m) - (1+m_c)$}
; end} ... ; end} ; end
```

Fig. 8. Translation P_Π^{step} of one step of the ASM program Π.

```
     if $F_1$ then $\Pi_1$
else if $F_2$ then $\Pi_2$
$\vdots$
else if $F_c$ then $\Pi_c$
     endif ... endif
```

where, for every processor X from a state in a computing phase, one and only one of these formulas $F_1, \ldots, F_c$ is **true**, and the programs Π_i have the form **par** $u_1^i \| \ldots \| u_{m_i}^i$ where $u_1^i, \ldots, u_{m_i}^i$ are m_i update commands, and Π_i produces distinct non-clashing sets of non-trivial updates. Let $m = \max_{1 \leqslant i \leqslant c}\{m_i\}$. We pad [36] every block of m_i update commands by using a `skip` n command, defined by:

$$\begin{aligned} \texttt{skip}\, 0 &\overset{\text{def}}{=} \textbf{end} \\ \texttt{skip}\, n+1 &\overset{\text{def}}{=} \textbf{if true}\, \{\textbf{end}\}; \texttt{skip}\, n \end{aligned}$$

The translation P_Π^{step} of one step of the ASM program Π is given explicitly at Fig. 8 p. 36, and we prove that it requires exactly $r + c + m$ steps (which depends only of Π) for every extension X of a processor of the ASM_{BSP} machine with the fresh variables $\{v_\theta \mid \theta \in \text{Read}(\Pi)\}$.

Lemma 12 *(Translation of one step of a sequential ASM program).*

$$\begin{aligned} (P_\Pi^{step}(X) \ominus X)|_{\mathcal{L}(\Pi)} &= \Delta(\Pi, X|_{\mathcal{L}(\Pi)}) \\ \text{time}_{\text{seq}}(P_\Pi^{step}, X) &= r + c + m \end{aligned}$$

where $r = card(Read\,(\Pi))$, c is the number of formulas in the ASM program Π in normal form, and m is the maximum number of updates per block of updates in Π.

Proof (Sketch of). The proof is similar to the sequential case [12]. The initialization of P_Π^{step} requires r steps. Because Π is in normal form, one and only one v_{F_i} is **true** in X. One step is required to enter on the block of v_{F_i}, the other conditionals are erased in $c-1$ steps. The updates of the block are $\Delta(\Pi, X|_{\mathcal{L}(\Pi)})$ and require m_i steps, then the `skip` $m - m_i$ commands require $m - m_i$ steps.

We have $\Delta(P_\Pi^{\text{step}}, X) = \{(v_\theta, \overline{\theta}^X)\}_{\theta \in \text{Read}(\Pi)} \cup \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$. P_Π^{step} terminates locally because it contains no loop nor **comm**, and is without overwrite on X because Π is in normal form. So (Lemma 10 p. 32) $\Delta(P_\Pi^{\text{step}}, X) = P_\Pi^{\text{step}}(X) \ominus X$. Therefore $(P_\Pi^{\text{step}}(X) \ominus X)|_{\mathcal{L}(\Pi)} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$. $\square$

```
$P_\Pi \overset{\text{def}}{=}$ while $\neg b^{\text{end}}_{\mathbf{comm}_A}$ {
    if $F^{\text{end}}_\Pi$ {
        skip $(r+c+m)-1$;
        comm;
        end
    } else {
        $P^{\text{step}}_\Pi$
    }; end
}; end
```

Fig. 9. Translation P_Π of the ASM program Π.

Therefore, we can prove by induction on t that for every processor X in a computation phase we have:

$$\overbrace{P^{\text{step}}_\Pi \circ \cdots \circ P^{\text{step}}_\Pi}^{t \text{ times}}(X) = \tau^t_\Pi(X|_{\mathcal{L}(\Pi)})$$

Thus, we want to repeat P^{step}_Π until the end of the computation phase which is, according to the Definition 9 p. 14, the termination of Π. Because every update block Π_i of the ASM program Π in normal form produce a non-clashing set of non-trivial updates, if Π can terminate then there exists a Π_i which is the empty program **par endpar**. Because the blocks produce distinct set of updates, this Π_i is the only empty block of updates in Π. So, we can define the termination formula F^{end}_Π:

Definition 20 *(The termination formula).*

$$F^{\text{end}}_\Pi \overset{\text{def}}{=} \begin{cases} F_i & \text{if there exists } 1 \leqslant i \leqslant c \\ & \quad\text{such that } \Pi_i = \textbf{par endpar} \\ \textbf{false} & \text{otherwise} \end{cases}$$

Lemma 13 *(Correctness of the termination formula).*

$$\inf\left\{ t \in \mathbb{N} \;\middle|\; \overline{F^{end}_\Pi}^{\overbrace{P^{step}_\Pi \circ \cdots \circ P^{step}_\Pi}^{t\ times}(X)} = \textbf{true} \right\} = \text{time}_{\text{seq}}(\Pi, X|_{\mathcal{L}(\Pi)})$$

Proof (Sketch of). The proof is similar to the sequential case [12], and based on the Lemma 12 which states that the program P^{step}_Π simulates one step of the ASM program Π. If Π does not terminate then the set is empty, thus the inf is ∞. □

So, a computation phase of the ASM program Π can be simulated by an IMP program like **while** $\neg F^{\text{end}}_\Pi$ $\{P^{\text{step}}_\Pi\}$; **end**. According to the definition of ASM$_{\text{BSP}}$, the communication phase begins at the termination of the ASM program Π, and continues until Π can do the first update of the next computation phase.

The execution ends when F^{end}_Π is **true** and the communication function $\mathbf{comm}_A$ changes nothing. By using the exploration witness $T(\mathbf{comm}_A)$ of the communication function, each processor can locally know whether the communication function has updated it or not at the last step, but it cannot know if the communication function has globally terminated. Therefore, we use a fresh boolean $b^{\text{end}}_{\mathbf{comm}_A}$ updated by the communication function $\mathbf{comm}_A$ itself to detect if the communication phase has terminated.

Let P_Π be the IMP program defined in Fig. 9, which is the translation of the ASM program Π. Notice that **comm** is executed only if F^{end}_Π is **true**. That means that the fresh boolean $b^{\text{end}}_{\mathbf{comm}_A}$ can be set to **true** only if F^{end}_Π is **true** and $\mathbf{comm}_A$ has terminated, which indicates the end of the execution. Therefore, IMP$_{\text{BSP}}$ algorithmically simulates ASM$_{\text{BSP}}$ with at most Read (Π) + 2 fresh variables, a temporal dilation $d = 2 + r + c + m$ and an ending time $e = d + 1$:

Proposition 6. IMP$_{\text{BSP}}$ *algorithmically simulates* ASM$_{\text{BSP}}$.

Proof (Sketch of). Let $A = (S(A), I(A), \tau_A)$ be an ASM$_{\text{BSP}}$ machine, with an ASM program Π and a communication function $\mathbf{comm}_A$. Let $M = (S(M), I(M), P_\Pi, \mathbf{comm}_M)$ be the IMP$_{\text{BSP}}$ machine defined by: $S(M)$ is an extension of $S(A)$ by using the fresh variables $v_{\theta_1}, \ldots, v_{\theta_r}$ with $\{\theta_1, \ldots, \theta_r\} = \text{Read}(\Pi)$, and the fresh boolean $b^{\text{end}}_{\mathbf{comm}_A}$; every $X \in I(M)$ is a $X|_{\mathcal{L}(A)} \in I(A)$, such that $v_{\theta_1}, \ldots, v_{\theta_r}$ are initialized with the value of **undef**, and $b^{\text{end}}_{\mathbf{comm}_A}$ to **false**; the IMP program P_Π is defined at the Fig. 9 p. 37; $\mathbf{comm}_M$ is $\mathbf{comm}_A$ for the symbols of $\mathcal{L}(A)$, leaves unchanged the variables v_θ for $\theta \in \text{Read}(\Pi)$, and if $\mathbf{comm}_A$ changes nothing then $\mathbf{comm}_M$ updates $b^{\text{end}}_{\mathbf{comm}_A}$ to **true**. According to the Definition 14 p. 31 of the simulation, there are three points to prove:

$$\begin{tikzcd}
\mathrm{ALGO}_{\mathrm{SEQ}} \arrow[r, equal, "{[6]}"] \arrow[d, "superstep"] & \mathrm{ASM}_{\mathrm{SEQ}} \arrow[r, leftrightsquigarrow, "{[12]}"] \arrow[d, "\mathrm{SEQ(PAR)}"] & \mathrm{IMP}_{\mathrm{SEQ}} \arrow[d, "\mathrm{PAR(SEQ)}"] \\
\mathrm{ALGO}_{\mathrm{BSP}} \arrow[r, equal] & \mathrm{ASM}_{\mathrm{BSP}} \arrow[r, leftrightsquigarrow] & \mathrm{IMP}_{\mathrm{BSP}}
\end{tikzcd}$$

Fig. 10. Diagrams of the models.

(1) $\mathcal{L}(M) = \mathcal{L}(A) \cup \{v_\theta \mid \theta \in \text{Read}\,(\Pi)\} \cup \{b^{\text{end}}_{\textbf{comm}_A}\}$, so we have $r+1$ fresh variables with $r = \text{card}(\text{Read}\,(\Pi))$, which depends only of the ASM program Π.

During the computation phase, P_Π is applied on every processor and works as follows: The **while** command checks in one step whether the execution has terminated or not; then, the **if** command checks in one step whether F^{end}_Π is **true** or not which (Lemma 13 p. 37) indicates whether Π has terminated or not. If Π has terminated, then $(r+c+m)-1$ steps are done by the `skip` commands, then the head command of the execution becomes a **comm**, and the processor waits for the other processors to do the communication $\textbf{comm}_M$, which is done in one step. Moreover, if $\textbf{comm}_A$ has terminated, $b^{\text{end}}_{\textbf{comm}_A}$ becomes **true** during $\textbf{comm}_M$. Otherwise, if Π has not terminated, P^{step}_Π is executed in $r+c+m$ steps, and (Lemma 12 p. 36) simulates one step of the ASM program Π. In any case, the execution comes backs to the **while** command.

Notice that the **comm** commands are after the `skip` in order to be the last things done by a processor during the simulation of a communication step of the ASM_{BSP} machine. This ensures the synchronization of the processors when some are in a computation phase and others are waiting the communication.

(2) Thus, a computation or a communication step of the ASM_{BSP} machine is simulated by exactly $d = 2+r+c+m$ steps of the IMP_{BSP} machine.

(3) A terminal state of the ASM_{BSP} machine is reached when Π has terminated for every processor and the communication function $\textbf{comm}_A$ changes nothing. In such state, the **while** command checks $b^{\text{end}}_{\textbf{comm}_A}$ which was **false** during the entire execution, then the **if** command checks that F^{end}_Π is **true**, then `skip` $(r+c+m)-1$, then the communication is done and sets $b^{\text{end}}_{\textbf{comm}_A}$ to **true**. Then the **while** command verifies that the execution is terminated, and the program reaches the end after $e = d+1$ steps. □

Therefore, according to the Proposition 5 p. 35 and the Proposition 6 p. 37, we obtain in the sense of the Definition 14 p. 31 that:

Theorem 2. $\text{IMP}_{\text{BSP}} \simeq \text{ASM}_{\text{BSP}}$

In other words, our core imperative programming language IMP_{BSP} is *algorithmically equivalent* to ASM_{BSP}. And by algorithmically equivalent, we mean that the executions are the same, up to a bounded number of fresh variables, a constant temporal dilation, and a constant ending time. In that sense, the BSP *cost model is preserved* by the translation (see Proposition 7 p. 40). More discussions about the overall result could be find in Q68-Q71 p. 40.

3.4. Questions and answers

Q60: *With all these different models and notations, I am a little lost. So what?*

We have proven, by using an algorithmic bi-simulation, that ASM_{BSP} and IMP_{BSP} are algorithmically equivalent. And thus that IMP_{BSP} is complete for the class of BSP algorithms. Fig. 10 p. 38 illustrates the use of the different models.

Q61: *So why do you use this plethoric number of definitions just to prove an obvious result?*

Well, It is the main drawback of our work: everything needs to be defined even the most intuitive concepts. But keep in mind that this algorithmic equivalence has never been proved before.

Q62: *Speaking of this simulation, even if both parts of your Theorem 2 p. 38 corresponds to your definition of the simulation, it appears that one way is easier than the other. Indeed, in the first part the temporal dilation was $d = 1$ and the ending time $e = 0$, which does not depend on the simulated program, whereas in the second part the temporal dilation was $d = 2+r+c+m$ and the ending time $e = d+1$. Could we find a better simulation with uniform parameters, which means that they do not depend on the simulated program?*

Not with IMP_{BSP} because only one update can be done at every step of computation. But indeed, we can imagine an alternative definition of the IMP_{BSP} model, where loops and conditionals commands cost 0 step and tuple-updates like

$(x, y) := (y, x)$ are allowed (for every size of tuples, not only couples). In that case, the simulation could have been done with the same translation but with a uniform cost of $d = 1$ and $e = 1$. We can even get rid of the ending time by assuming that $b^{\text{end}}_{\mathbf{comm}_A}$ is updated at the last step of the communication function, and not when the communication function changes nothing. In a way, such alternate model is algorithmically closer to the ASM$_{\text{BSP}}$ model, because there is a quasi-identity (only the fresh variables are necessary) between them. But in this work we preferred a minimal model of computation, which can be more easily compared to mainstream programming languages.

Q63: *Again, why do you not consider the number of processors as part of the input of the algorithm? For instance, a sorting algorithm will have an execution time depending of the size of the array. In this point of view, the total execution time could depend of the number of processors, and each step may be simulated by $d * \mathbf{p}$ steps.*

Indeed, the total number of steps depends on the size of the inputs, which includes $\mathbf{p}$. But the execution time should not be confused with the cost for simulating one step. If the BSP algorithm requires $f(n)$ steps, where n is the size of the inputs, then our simulation will require $d \times f(n) + e$ steps. Our simulation preserves the $O(f(n))$ in time, but this is not the main point. Our simulation is algorithmic because it preserves the step-by-step behavior of the BSP algorithm, by simulating each step by d steps, where d does not depend of the size of the inputs or the number of processors. Also, it is *not* possible to use p times the ASM's parallel updates because this would break the temporal dilation requirement (being a constant independant to the program and of the input data; but p is the size of the different p-tuples so is part of the input).

Q64: *I understand that what you call "algorithmically" is thus not modifying (intrinsically) the complexity of the algorithms (sequential or BSP). But do your simulation works only for the local computations of the BSP's supersteps?*

No, It works for the sequence of supersteps and thus for the overall computation. For all the models (ASM$_{\text{BSP}}$, ALGO$_{\text{BSP}}$ and IMP$_{\text{BSP}}$), what we called $\text{time}(A, S_0)$ is not only the sequential parts but all the computation and communication steps of the transition function: during local computations, processors compute in parallel (SEQ(PAR)) and then the communication is global. For IMP$_{\text{BSP}}$ we also use the notion of $\text{time}_{\text{seq}}(P, X)$ only during computations phases because programs are independant (PAR(SEQ)) but this local timing is "synchronised" with the global time.

Q65: *And the same for communications?*

Yes, bounded by the witness, the function of communication performs only one-step reductions (say 1-relations). We thus preserve the BSP costs, at worst with a constant factor d that is independant of p and of the algorithms.

Q66: *Fine. But if you do not have an identity why do you bother to simulate one step in a constant number of steps. Why not a polynomial (or a linear) number of steps?*

Indeed, usually a simulation is seen acceptable if the simulation costs $\mathcal{O}(f(n))$ steps, where f is a polynomial function and n is the duration for the simulated machine. But in this work we are not interested in the simulation of the function (the input-output behavior) but the simulation of the algorithm, the *step-by-step behavior*. A consequence of the temporal dilation d and the ending time e is that we simulate an execution costing n steps by an execution costing $\mathcal{O}(n)$ steps, so the complexity in time is strictly preserved, but our simulation is even more restrictive (and so our result is stronger). Indeed, in this work we simulate not only the output and the duration (a functional simulation) but we simulate every step of the execution by d steps (an algorithmic simulation). So *we preserve the intentional behavior* and not only the extensional behavior of the simulated algorithm.

Q67: *And for infinite algorithms?*

There are three mains cases for infinite durations. First, one processor (at least) performs an infinite computation (case where during a computation phase, $\exists i\ P_i \overset{\infty}{\curlyvee} X_i$ such that is $\text{time}_{\text{seq}}(P_i, X_i) = \infty$). Second, the calls of the function of communication never terminate (this is rather a flaw of defining such a function). Third, there is an infinite number of supersteps. In every case we have $\text{time}(P_{init}, \overrightarrow{X}) = \infty$. Because our simulation works step-by-step, the infinite computations are preserved with the same "costs".

Q68: *For translating IMP$_{\text{BSP}}$ programs into ASM$_{\text{BSP}}$ programs, why not using control state ASMs [28]?*

Indeed, our control Flow Graph (CFG) was quite similar. We used it because CFG is a well-known tool which works well for our proofs and for mainstream languages. But its use was only "internal" to the proofs, and therefore does not influence the final result.

Q69: *Fine. So can we deduce from your results that the BSPLIB is BSP algorithmically complete?*

Our result of algorithmic completeness is up to elementary functions so is independant of the level of abstraction chosen by the user and of the physical architectures of the machines (the user can choose any kind of machine, from Turing ones to the most modern processors). Thus, any BSP algorithm using communications such as the ones of the BSPLIB (permutation of data using distant writing/reading and sending of messages) can be programmed using the BSPLIB *with the right cost*. But other communications are possible (such as performing I/O or load-balancing or exchanging data of two non consequent supersteps as in the BSP+ model of [31]); using the BSPLIB only may force to simulate them and this simulation can be not algorithmic (increase in a non-constant way the number of supersteps). Anyway, an elementary function can be too much abstract (and unrealistic) but this is an orthogonal problem.

Q70: *I am happy to finally see a set of concurrent processes* PAR(SEQ) *as the semantics of* IMP_{BSP}. *So, you finally admit that it is necessary to reason in such a way?*

For programming BSPLIB-like algorithms yes. But, we prove in our context that PAR(SEQ) (IMP_{BSP}) is algorithmically equivalent to SEQ(PAR) (ASM_{BSP}). So both reasoning are correct. Nevertheless, there exist BSP programming languages, such as BSML [25], that are SEQ(PAR) (and even if the underlying implementation is PAR(SEQ), *e.g.* using MPI) and so it could be possible to also prove their algorithm completeness; so, it is not necessary, it is just common for most mainstream languages, and thus for programming BSPLIB-like algorithms.

Q71: *Now, speaking of the cost model, you defined an alternation of computation and communication phases for the* ASM_{BSP} *and for the* IMP_{BSP}, *but they do not coincide in the translation. Indeed, a communication step of an* ASM_{BSP} *is translated into several computation steps in* IMP_{BSP} *then an execution of the* **comm** *command. So, there are many more supersteps in the simulation of the* ASM_{BSP} *than in the execution of the* ASM_{BSP} *itself. Does your simulation truly respect the* BSP *cost model?*

Strictly speaking, you are perfectly right. And this cannot be avoided, at least for our definition of IMP_{BSP}, because the communication function is called *via* the **comm** command, and the program P_{Π} simulating the ASM program Π internalizes the test required to know if the program does a computation step or a communication step, test which is done by the operational semantics and not the program in the definition of ASM_{BSP}. But the "computation steps" done by the IMP_{BSP} program during a communication phase are only tests (or updates done by the communication function itself) and not updates done by the program, so it may be misleading to call them proper computation steps. If we call computation steps only the updates commands, communication steps the **comm** commands and administrative steps the **if** and **while** commands, and if we call computation phase a succession of computation or administrative steps, and communication phase a succession of communication or administrative steps, then the number of supersteps done by an ASM_{BSP} is preserved by our translation. Moreover the duration of the local computation are also proportional (up to the factor d), and the cost of the communications (h-relations) are exactly the same. Therefore, *the cost model is preserved*:

Proposition 7 *(Preservation of the Cost Model). The algorithmic simulation between* IMP_{BSP} *and* ASM_{BSP} *preserves the cost model (up to a constant temporal dilation).*

And we can apply such a proposition to the realization of the function of communication of SubSection 2.7 p. 24 and thus to BSPLIB-like programs.

Q72: *Finally, I would like to raise the question whether the lifting to* BSP *could be done uniformly for all bridging models?*

Having given these definitions would surely simplify an adaptation of this work to other bridging models. But we think that generalizing such a work for any bridging model seems unrealistic without having properly defined the notion of "all bridging models".

4. Conclusion

4.1. Summary of the contribution

A *bridging model* (see p. 4) provides a common level of *understanding* between hardware and software engineers. It provides software developers with an attractive escape route from the world of architecture-dependent parallel software [20]. The BSP bridging model allows the design of "*immortal*"[30] algorithms using a *realistic cost model* (and without any overspecification requiring the use of a large number of performance parameters) that can fit most distributed architectures. Many BSP algorithms has been designed and used with success in many domains [21]. The following equation summarizes the result:

[30] Independent of the architecture and thus working for any machine in the present and in the future as long as the BSP model can be adapted to it.

Theorem 3. $\text{IMP}_{\text{BSP}} \simeq \text{ASM}_{\text{BSP}} = \text{ALGO}_{\text{BSP}}$

More precisely, about the $=$ of the above equation, we have given an *axiomatic* definition of BSP algorithms by adding only one *postulate* to the sequential ones of [6] for sequential algorithms (which has been widely consensual). This new postulate is the definition of the *supersteps* and we abstract how *communication* is performed, not being restricted to a specific BSP library. The ASM_{BSP} are the operational viewpoints and they are used as an intermediary to the imperative BSP core-programming language IMP_{BSP}. We also show how the BSP costs of BSP algorithms can be obtained and preserved from execution (the duration) of ASM_{BSP} programs.

We answer previous criticisms by defining a convincing set of parallel algorithms, in a restricted model, running in a predictable time. This small addition allows a greater *confidence* in this formal definition compared to previous work: postulates of concurrent ASMs do not provide the same level of intuitive clarity as the postulates for sequential algorithms. We have thus revisited the problem of the "*parallel ASM thesis*" *i.e.*, to provide a *machine-independent* definition of BSP algorithms and a proof that these algorithms are faithfully captured by ASM_{BSP}. We also proved that the *cost model* is preserved, which is the main novelty and specificity of this work compared to the traditional work about distributed or concurrent ASMs.

Our work is relevant because it allows universality (immortal stands for BSP computing): all future BSP algorithms, whatever their specificities, up to elementary functions (such as the communication one), will be captured by our definitions. So, our ASM_{BSP} is not just another model, it is a *class model* [48], which contains all BSP algorithms (and not more).

Now speaking about the $\simeq$ of the above equation. The ASM framework allows the essence of computation to be formally described: the semantics, the cost and the intentional behavior of the algorithms. But it is not mainstream for coding algorithms, and programming languages are largely preferred in practice. We have proven that a core-programming language, call IMP_{BSP}, is algorithmically complete for the *class of* BSP *algorithms*. To do that, we have given an operational semantics of IMP_{BSP}, and by using an algorithmic simulation we proved that IMP_{BSP} is algorithmically equivalent to the ASM_{BSP}, up to elementary functions (primitives to manipulate structures and the function of communication which is used for the synchronization in both the BSP extension of ASMs and the imperative programming language *à la* BSPLIB; the result is thus independent of the architecture of the underlying machine). By algorithmically equivalent, we mean that the executions are the same, up to a bounded number of fresh variables, a constant temporal dilation, and a constant ending time. Such a simulation preserves the BSP costs of algorithms. To do this, we have given two (total) functions of compilation, one from ASM_{BSP} to this core language and another for the converse way. ASM_{BSP} thus served us well as intermediary between programming languages (usable by the algorithmicians) and the class of the BSP algorithms.

By extrapolation, JAVA/C $+$ BSPLIB is complete to program BSP algorithms, up to elementary primitives (*e.g.* on data-structures and/or how communicated values): by given a realization of the BSPLIB's DRMA operations, we have highlighted the BSPLIB's sub-class of BSP algorithms. And we have shown that the intended BSP costs of such algorithms are preserved. Notice that our work is still limited to BSP algorithms even if it is still sufficient for many HPC and big-data applications. That leads to interesting future works.

4.2. Future work

Firstly, how to adapt our work to a hierarchical extension of BSP [33] which is closer to modern HPC architectures? It is worth noticing that in this model, only memories are described (in a hierarchical way), never the processors. And for other models such as LOGP [41], *etc.*? Can we thus imagine a method that "automatically" characterizes a bridging model? (since there is, or will be, a plethoric number of bridging models).

Secondly, BSP is a bridging model between hardwares and softwares. It could be interesting to study such a link more formally (left part of Fig. 2 p. 4). For instance, can we prove that the primitives of a BSP language can truly "be BSP" on a typical cluster architecture?

Thirdly, there are many languages having a BSP-like model of execution, for instance PREGEL [26] for writing large-graph algorithms. There is also the BSP-RAM model of [31]. An interesting application would be to prove which ones are BSP algorithmically complete and which are not. MAPREDUCE is an interesting case: its model of execution is BSP but there are only limited routines. It has been proven in [42] that some restricted BSP algorithms can be efficiently simulated by the MAPREDUCE framework. But the author notes that using the available simulations, the asymptotic costs of the algorithms do increase. We believe that it is impossible (in general) to have such an equivalence. Thus, it seems that the language is not BSP complete. Such a formal proof can only be done by contradiction: assuming a "perfect" translation and exhibiting a contradiction. Even if it exists, such a proof would be hard to establish because one can imagine a very subtle translation. But the MAPREDUCE also benefits of an automatic load balancing of the individual tasks which is not possible using BSPLIB (especially for tasks with unknown execution time), and thus making the formal proof complex. However, we think, as [42], that a breadth first search algorithm is a good example for such a counter-example (the author of [42] has studied an efficient translation but not all the possible ones).

In any case, studying the BSP-RAM (such as communication-oblivious of [31]) would lead to a define strict subclasses of BSP algorithms. Studying the BSP$\lambda-$*calculus* (the foundation of BSML [25]) as was done for the $\lambda-$*calculus* [36] is another working track. Maybe the works of [7–9] would be preferable in that case.

Declaration of Competing Interest

We have no competing interest.

References

[1] S. Gorlatch, Send-receive considered harmful: myths and realities of message passing, ACM Trans. Program. Lang. Syst. 26 (1) (2004) 47–56.
[2] J. Dean, S. Ghemawat, MAPREDUCE: a flexible data processing tool, Commun. ACM 53 (1) (2010) 72–77.
[3] F. Cappello, A. Guermouche, M. Snir, On communication determinism in parallel HPC applications, in: Computer Communications and Networks, ICCCN, IEEE, 2010, pp. 1–8.
[4] H. González-Vélez, M. Leyton, A survey of algorithmic skeleton frameworks: high-level structured parallel programming enablers, Softw. Pract. Exp. 40 (12) (2010) 1135–1160.
[5] M.Y. Vardi, What is an algorithm?, Commun. ACM 55 (3) (2012) 5.
[6] Y. Gurevich, Sequential abstract-state machines capture sequential algorithms, ACM Trans. Comput. Log. 1 (1) (2000) 77–111.
[7] Y.N. Moschovakis, What is an algorithm?, in: B. Engquist, W. Schmid (Eds.), Mathematics Unlimited — 2001 and Beyond, Springer, 2001, pp. 919–936.
[8] Y.N. Moschovakis, The formal language of recursion, J. Symb. Log. 54 (4) (1989) 1216–1252.
[9] G. Berry, P.-L. Curien, Sequential algorithms on concrete data structures, Theor. Comput. Sci. 20 (1) (1982) 265–321.
[10] T.C. Biedl, J.F. Buss, E.D. Demaine, M.L. Demaine, M.T. Hajiaghayi, T. Vinar, Palindrome recognition using a multidimensional tape, Theor. Comput. Sci. 302 (1–3) (2003) 475–480.
[11] L. van den Dries, Generating the greatest common divisor, and limitations of primitive recursive algorithms, Found. Comput. Math. 3 (3) (2003) 297–324.
[12] Y. Marquer, Algorithmic completeness of imperative programming languages, Fundam. Inform. 168 (1) (2019) 51–77, https://doi.org/10.3233/FI-2019-1824, https://dr-apeiron.net/lib/exe/fetch.php/fr:recherche:fi-while-long.pdf.
[13] Y. Marquer, P. Valarcher, An Imperative Language Characterizing PTIME Algorithms, CSLI Publications, Studies in Weak Arithmetics, vol. 3, 2016, http://www.dr-apeiron.net/lib/exe/fetch.php/en:research:ploopc2.pdf.
[14] S. Grigorieff, P. Valarcher, Evolving multialgebras unify all usual sequential computation models, in: J.-Y. Marion, Thomas Schwentick (Eds.), Symposium on Theoretical Aspects of Computer Science, STACS, in: LIPIcs, Leibniz-Zentrum fuer Informatik, vol. 5, 2010, pp. 417–428.
[15] E. Börger, K. Schewe, Concurrent abstract state machines, Acta Inform. 53 (5) (2016) 469–492.
[16] E. Börger, K.-D. Schewe, Communication in abstract state machines, J. Univers. Comput. Sci. 23 (2) (2017) 129–145.
[17] F. Ferrarotti, K.-D. Schewe, L. Tec, Q. Wang, A new thesis concerning synchronised parallel computing – simplified parallel ASM thesis, Theor. Comput. Sci. 649 (2016) 25–53.
[18] K.-D. Schewe, Q. Wang, A simplified parallel ASM thesis, in: J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, E. Riccobene (Eds.), Abstract State Machines, Alloy, B, VDM (ABZ), in: LNCS, vol. 7316, Springer, 2012, pp. 341–344.
[19] A. Blass, Y. Gurevich, Abstract state machines capture parallel algorithms, ACM Trans. Comput. Log. 4 (4) (2003) 578–651.
[20] L.G. Valiant, A bridging model for parallel computation, Commun. ACM 33 (8) (1990) 103–111.
[21] R.H. Bisseling, Parallel Scientific Computation. A Structured Approach Using BSP and MPI, Oxford University Press, 2004.
[22] D.B. Skillicorn, J.M.D. Hill, W.F. McColl, Questions and answers about BSP, Sci. Program. 6 (3) (1997) 249–274.
[23] J.M.D. Hill, B. McColl, D.C. Stefanescu, M.W. Goudreau, K. Lang, S.B. Rao, T. Suel, T. Tsantilas, R. Bisseling, BSPLIB: the BSP programming library, Parallel Comput. 24 (1998) 1947–1980, http://www.bsp-worldwide.org/.
[24] K. Siddique, Z. Akhtar, Y. Kim, Y.-S. Jeong, E.J. Yoon, Investigating Apache HAMA: a bulk synchronous parallel computing framework, J. Supercomput. 73 (9) (2017) 4190–4205, https://hama.apache.org/.
[25] F. Loulergue, BSλ_p: functional BSP programs on enumerated vectors, in: J. Kazuki (Ed.), International Symposium on High Performance Computing, in: Lecture Notes in Computer Science, vol. 1940, Springer, 2000, pp. 355–363, http://traclifo.univ-orleans.fr/BSML/.
[26] G. Malewicz, M.H. Austern, A.J.C. Bik, J.C. Dehnert, I. Horn, N. Leiser, PREGEL: a system for large-scale graph processing, in: Management of Data, ACM, 2010, pp. 135–146, The free version: http://giraph.apache.org/.
[27] E. Börger, R.F. Stärk, Abstract State Machines: A Method for High-Level System Design and Analysis, Springer, 2003.
[28] E. Börger, High level system design and analysis using abstract state machines, in: Applied Formal Methods, FM-Trends 98, Springer, Berlin, Heidelberg, 1999.
[29] R. da, Rosa Righi, R. de Quadros Gomes, V.F. Rodrigues, C.A. da Costa, A.M. Alberti, L.L. Pilla, P.O.A. Navaux, MigPF: towards on self-organizing process rescheduling of bulk-synchronous parallel applications, Future Gener. Comput. Syst. 78 (2018) 272–286.
[30] C.W. Keßler, NestStep: nested parallelism and virtual shared memory for the BSP model, J. Supercomput. 17 (3) (2000) 245–262, http://www.ida.liu.se/~chrke55/neststep/index.html.
[31] A. Tiskin, The Design and Analysis of Bulk-Synchronous Parallel Algorithms, Ph.D. Thesis, Oxford University Computing Laboratory, 1998.
[32] A. Prinz, E. Sherratt, Distributed ASM-pitfalls and solutions, in: Y.A. Ameur, K. Schewe (Eds.), Abstract State Machines, Alloy, B, TLA, VDM, and Z (ABZ), vol. 8477, Springer, 2014, pp. 210–215.
[33] L.G. Valiant, A bridging model for multi-core computing, J. Comput. Syst. Sci. 77 (1) (2011) 154–166.
[34] J. Fortin, F. Gava, BSP-WHY: an intermediate language for deductive verification of BSP programs, in: High-Level Parallel Programming and Applications, HLPP, ACM, 2010, pp. 35–44.
[35] L. Bougé, The data parallel programming model: a semantic perspective, in: G. Perrin, A. Darte (Eds.), The Data Parallel Programming Model: Foundations, HPF Realization, and Scientific Applications, in: LNCS, vol. 1132, Springer, 1996, pp. 4–26.
[36] M. Ferbus-Zanda, S. Grigorieff, ASMS and operational algorithmic completeness of lambda calculus, in: A. Blass, N. Dershowitz, W. Reisig (Eds.), Fields of Logic and Computation, in: LNCS, vol. 6300, Springer, 2010, pp. 301–327.
[37] A. Glausch, W. Reisig, An ASM-characterization of a class of distributed algorithms, in: J-R. Abrial, U. Glässer (Eds.), Rigorous Methods for Software Construction and Analysis, Essays Dedicated to Egon Börger on the Occasion of His 60th Birthday, in: LNCS, vol. 5115, Springer, 2009, pp. 50–64.
[38] A. Cavarra, E. Riccobene, A. Zavanella, A formal model for the parallel semantics of P3L, in: ACM Symposium on Applied Computing, SAC, 2000, pp. 804–812.
[39] A. Cavarra, A data-flow approach to test multi-agent ASMS, Form. Asp. Comput. 23 (1) (2011) 21–41.
[40] K.-D. Schewe, F. Ferrarotti, L. Tec, Q. Wang, W. An, Evolving concurrent systems: behavioural theory and logic, in: Australasian Computer Science Week Multiconference, ACSW, 2017, pp. 1–10.
[41] D. Culler, R. Karp, D. Patterson, A. Sahay, K. Schauser, E. Santos, R. Subramonian, T. von Eicken, LogP: toward a realistic model of parallel computation, in: ACM SIGPLAN Symposium on Principles and Practises of Parallel Programming, 1993, pp. 1–12.
[42] M.F. Pace, BSP VS MAPREDUCE, in: H.H. Ali, Y. Shi, D. Khazanchi, M. Lees, G.D. van Albada, J.J. Dongarra, P.M.A. Sloot (Eds.), Computational Science, ICCS, in: Procedia Computer Science, vol. 9, Elsevier, 2012, pp. 246–255.

[43] M. Spielmann, Abstract State Machines: Verification Problems and Complexity, Ph.D. Thesis, RWTH Aachen University, Germany, 2000.
[44] Y. Marquer, F. Gava, An ASM thesis for BSP in HAL, https://hal.archives-ouvertes.fr/hal-01717647.
[45] Y. Marquer, F. Gava, Algorithmic completeness of BSP languages in HAL, https://hal.archives-ouvertes.fr/hal-01742406.
[46] Y. Marquer, F. Gava, Algorithmic completeness for BSP languages, in: International Conference on High Performance Computing & Simulation, HPCS, 4Pad Workshop, IEEE, 2018, pp. 740–747.
[47] Y. Gurevich, The sequential ASM thesis, Bull. Eur. Assoc. Theor. Comput. Sci. 67 (1999) 93–110.
[48] S. Grigorieff, P. Valarcher, Classes of algorithms: formalization and comparison, Bull. Eur. Assoc. Theor. Comput. Sci. 107 (2012) 95–127.
[49] P. Cégielski, I. Guessarian, Normalization of some extended abstract state machines, in: A. Blass, N. Dershowitz, W. Reisig (Eds.), Fields of Logic and Computation, vol. 6300, Springer, 2010, pp. 165–180.
[50] N.S. Yanofsky, Towards a definition of an algorithm, J. Log. Comput. 21 (2) (2011) 253–286.
[51] O. Bonorden, B.H.H. Juurlink, I. von Otte, I. Rieping, The Paderborn University BSP (PUB) library, Parallel Comput. 29 (2) (2003) 187–207.