An Axiomatization for BSP Algorithms

Yoann Marquer and Frédéric Gava(✉)

Laboratory of Algorithms, Complexity and Logic (LACL),
University of Paris-East, Créteil, France
yoann.apeiron.marquer@gmail.com, gava@u-pec.fr

Abstract. The Gurevich's thesis stipulates that sequential Abstract State Machines (ASMs) capture the essence of sequential algorithms. On another hand, the Bulk-Synchronous Parallel (BSP) bridging model is a well known model for HPC algorithm design. It provides a conceptual bridge between the physical implementation of the machine and the abstraction available to a programmer of that machine. The assumptions of the BSP model are thus provide portable and scalable performance predictions on most HPC systems. We follow Gurevich's thesis and extend the sequential postulates in order to intuitively and realistically capture BSP algorithms.

Keywords: BSP · ASM · Parallel algorithm · HPC · Postulates
Cost model

1 Introduction

1.1 Context of the Work

Nowadays, HPC (High Performance Computing) 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. The problem arises from the difficulty to get a *taxonomy* of computer architectures and frameworks: there is a zoo of definitions of systems, languages, paradigms and programming models. Indeed, in the HPC community, several terms could be used to designate the same thing, so that misunderstandings are easy. We can cite parallel patterns [5] versus algorithmic skeletons [8]; shared memory (PRAM) versus thread concurrency and Direct ReMote Access (DRMA); asynchronous send/receive routines (MPI, http://mpi-forum.org/) versus communicating processes (π-calculus).

In the sequential world, it is easier to classify programming languages within their paradigm (functional, object oriented, *etc.*) or by using some properties of the compilers (statically or dynamically typed, abstract machine or native code execution). This is mainly due to the fact that there is an overall consensus on what sequential computing is. For them, *formal semantics* have been often studied and there are now many tools for testing, debugging, cost analyzing, software engineering, *etc.* In this way, programmers can implement sequential algorithms using these languages, which *characterize* properly the sequential algorithms.

J. Vaidya and J. Li (Eds.): ICA3PP 2018, LNCS 11336, pp. 72–88, 2018.
https://doi.org/10.1007/978-3-030-05057-3_6

This consensus is only fair because everyone *informally* agrees to what constitutes a sequential algorithm. And now, half a century later, there is a growing interest in defining *formally* the notion of algorithms [10]. Gurevich introduced an *axiomatic* presentation (largely machine independent) of the sequential algorithms in [10]. The main idea is that there is no language that truly represents all sequential algorithms. In fact, every algorithmic book presents algorithms in its own way and programming languages give too much detail. An axiomatic definition [10] of the algorithms has been mapped to the notion of Abstract State Machine (ASM, a kind of Turing machine with the appropriate level of abstraction): Every sequential algorithm can be captured by an ASM. This allows a common vocabulary about sequential algorithms. This has been studied by the ASM community for several years.

A parallel computer, or a multi-processor system, is a computer composed of more than one processor (or unit of computation). It is common to classify parallel computers (Flynn's taxonomy) by distinguishing them by the way they access the system memory (shared or distributed). Indeed, the memory access scheme influences heavily the programming method of a given system. Distributed memory systems are needed for computations using a large amount of data which does not fit in the memory of a single machine.

The three *postulates* for sequential algorithms are mainly consensual. Nevertheless, to our knowledge, there is not such a work for HPC frameworks. First, due to the zoo of (informal) definitions and second, due to a lack of realistic *cost models* of common HPC architectures. In HPC, the cost measurement is not based on the complexity of an algorithm but is rather on the execution time, measured using empirical *benchmarks*. Programmers are benchmarking load balancing, communication (size of data), *etc.* Using such techniques, it is very difficult to explain why one code is faster than another and which one is more suitable for one architecture or another. This is regrettable because the community is failing to obtain some rigorous characterization of sub-classes of HPC algorithms. There is also a lack of studying algorithmic completeness of HPC languages. This is the basis from which to specify what can or cannot be effectively programmed. Finally, taking into account all the features of all HPC paradigms is a daunting task that is unlikely to be achieved [9]. Instead, a *bottom up strategy* (from the simplest models to the most complex) may be a solution that could serve as a basis for more general HPC models.

1.2 Content of the Work

Using a *bridging model* [20] is a first step to this solution because it simplifies the task of algorithm design, programming and simplifies the reasoning of *cost* and ensures a better *portability* from one system to another. A bridging model is an abstract model of a computer which provides a conceptual bridge between the physical implementation of the machine and the *abstraction* available to a programmer of that machine. We conscientiously limit our work to the Bulk-Synchronous Parallel (BSP) bridging model [1,18] because it has the advantage of being endowed with a simple model of execution. We leave more complex models to future work. Moreover, there are many different libraries and languages for programming BSP algorithms, for example, the BSPLIB for C [11] or JAVA [17], BSML [?], PREGEL [12] for big-data, *etc.*

Concurrent ASMs [3] try to capture the more general definition of asynchronous and distributed computations. We promote a rather different "bottom-up" approach consisting of restricting the model under consideration, so as to better highlight the algorithm execution time (which is often too difficult to assess for general models) and more generally to formalize our algorithms of a bridging model at their natural level of abstraction, instead of using a more general model then restrict it with an arbitrary hypothesis.

As a basis to this work, we first give an axiomatic definition of BSP algorithms (ALGO_{BSP}) with only 4 postulates. Then we extend the ASM model [10] of computation (ASM_{BSP}) for BSP. Our goal is to define a convincing set of parallel algorithms running in a predictable time and construct a model that computes these algorithms only. This can be summarized by $\text{ALGO}_{\text{BSP}}=\text{ASM}_{\text{BSP}}$. An interesting and novel point of this work is that the BSP cost model is preserved.

1.3 Outline

Many definitions used here are well known to the ASM community. Recalling all of them would be too long but they are available in the online technical report [22].

The remainder of this paper is structured as follows: In Sect. 2 we first recall the BSP model and define its postulates; Secondly, in Sect. 3, we give the operational semantics of ASM_{BSP} and finally, we give the main result. Section 4 concludes, gives some related work and a brief outlook on future work.

2 Characterizing BSP Algorithms

2.1 The BSP Bridging Model of Computation

As the RAM model provides a unifying approach that can *bridge* the worlds of sequential *hardware* and *software*, so Valiant sought [20] for a unifying model that could provide an effective (and universal) bridge between parallel hardware and software. A *bridging* model [20] allows to reduce the gap between an abstract execution (programming an algorithm) and concrete parallel systems (using a compiler and designing/optimizing a physical architecture).

The *direct mode* BSP model [1,18] is a *bridging* model that simplifies the programming of various parallel architectures using a certain level of abstraction. The assumptions of the BSP model are to provide *portable* and *scalable* performance predictions on HPC systems. Without dealing with low-level details of HPC architectures, the programmer can thus focus on algorithm design only. The BSP bridging model describes a parallel architecture, an execution model for the algorithms, and a cost model which allows to predict their performances on a given BSP architecture.

A BSP computer can be specified by $\mathbf{p}$ *uniform* computing units (**processors**), each capable of performing one elementary operation or accessing a local memory in one time unit. Processors communicate by sending a data to every other processor in $\mathbf{g}$ time units (gap which reflects network bandwidth inefficiency), and a barrier mechanism is able to synchronise all the processors in $\mathbf{L}$ time units ("latency" and the ability of the network to deliver messages under a continuous load). Such values, along with the processor's speed (*e.g.* Mflops) can be empirically determined by executing benchmarks.

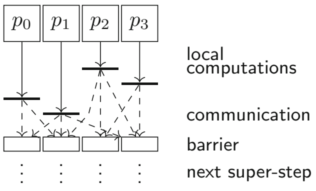

Fig. 1. A BSP super-step.

The time $\mathbf{g}$ is thus for collectively delivering a 1-relation which is a collective exchange where every processor receives/sends at most one word. The network can deliver an h-relation in time $\mathbf{g}\times h$. A BSP computation is organized as a *sequence* of **supersteps** (see Fig. 1). During a superstep, the processors may perform computations on local data or send messages to other processors. Messages are available for processing at their destinations by the next superstep, and each superstep is ended with the *barrier synchronisation* of the processors.

The execution time (cost) of a super-step s is the sum of the maximal of the local processing, the data delivery and the global synchronisation times. It is expressed by the following formula: $\mathrm{Cost}(s) = w^s + h^s \times \mathbf{g} + \mathbf{L}$ where $w^s = \max_{0\leq i<\mathbf{p}}(w_i^s)$ (where w_i^s is the local processing time on processor i during superstep s), and $h^s = \max_{0\leq i<\mathbf{p}}(h_i^s)$ (where h_i^s is the maximal number of words transmitted or received by the processor i). Some papers rather use the sum of words for h_i^s but modern networks are capable of sending while receiving data. The total cost (execution time) of a BSP algorithm is the sum of its super-step costs.

2.2 Axiomatic Characterization of BSP Algorithms

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

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

We follow [10] in which states, as first-order structures, are full instantaneous descriptions of an algorithm.

Definition 1 (Structure). *A (first-order) structure X is given by:*

1. *A (potentially infinite) set $\mathcal{U}(X)$ called the* ***universe*** *(or domain) of X*
2. *A finite set of function symbols $\mathcal{L}(X)$ called the* ***signature*** *(language) 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 an application: $\mathcal{U}(X)^\alpha \to \mathcal{U}(X)$*

In order to have a uniform presentation [10], we considered 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 $\mathcal{L}(X)$ is 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 the boolean type ($\neg$, $\wedge$) and the equality. We also distinguish dynamic symbols whose interpretation may change from one state to another, and static symbols which are the elementary operations.

Definition 2 (Term). *A term 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 $\theta_1, \ldots, \theta_\alpha$ are terms, then $f\left(\theta_1, \ldots, \theta_\alpha\right)$ is a term*

The interpretation $\overline{\theta}^X$ of a term θ in a structure X is defined by induction on θ:

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

A **formula** F is a term with the particular form $\textbf{true}|\textbf{false}|R\left(\theta_1, \ldots, \theta_\alpha\right)|\neg F$ $|(F_1 \wedge F_2)$ where R is a relation symbol (ie a function with output $\overline{\textbf{true}}^X$ or $\overline{\textbf{false}}^X$) and $\theta_1, \ldots, \theta_\alpha$ are terms. We say that a formula is true (resp. false) in X if $\overline{F}^X = \overline{\textbf{true}}^X$ (resp. $\overline{\textbf{false}}^X$).

A BSP algorithm works on independent and uniform computing units. Therefore, a state S_t of the algorithm A must be a tuple $\left(X_t^1, \ldots, X_t^p\right)$. 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, the number of processors). In this paper, 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.

If $\left(X^1, \ldots, X^p\right)$ is a state of the algorithm A, then the structures $X^1, \ldots, X^p$ will be called **processors** or **local memories**. The set of the *independent* local memories of A will be denoted by $M(A)$. We now define the BSP algorithms as the objects verifying the four presented postulates. The computation for every processor is done in parallel and step by step.

An **execution** of A is a sequence of states $S_0, S_1, S_2, \ldots$ such that S_0 is an initial state and for every $t \in \mathbb{N}$, $S_{t+1} = \tau_A(S_t)$. Instead of defining a set of *final states* for the algorithms, we will say that a state S_t of an execution is **final** if $\tau_A(S_t) = S_t$, that is the execution is: $S_0, S_1, \ldots, S_{t-1}, S_t, S_t, \ldots$ We say that an execution is **terminal** if it contains a final state.

We are interested in the algorithm and not a particular implementation (eg, the variables' names), therefore in the postulate we will consider the states up to multi-isomorphism.

Definition 3 (Multi-isomorphism). *$\overrightarrow{\zeta}$ is a multi-isomorphism between two states $\left(X^1, \ldots, X^p\right)$ and $\left(Y^1, \ldots, Y^q\right)$ if $p = q$ and $\overrightarrow{\zeta}$ is a p-tuple of applications*

$\zeta_1, \ldots, \zeta_p$ such that for every $1 \leq i \leq p$, 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 $\mathcal{L}(A)$;*
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 $a_1, \ldots, a_\alpha$, b be elements of the universe $\mathcal{U}(X)$. We say that $(f, a_1, \ldots, a_\alpha)$ is a location of X, and that $(f, a_1, \ldots, a_\alpha, b)$ is an **update** on X at the location $(f, a_1, \ldots, a_\alpha)$. For example, 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 example, 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 $\overrightarrow{a} = a_1, \ldots, a_\alpha$. For example, in $X \oplus (f, 0, 42)$, every symbol has the same interpretation than 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** 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:

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

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 $Y = X \oplus \Delta$. This Δ is called the **difference** between the two local memories, and is denoted by $Y \ominus X$.

Let $\overrightarrow{X} = \left(X^1, \ldots, X^p\right)$ 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 $(\tau_A(\overrightarrow{X})^1, \ldots, \tau_A(\overrightarrow{X})^p)$. 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 $\overrightarrow{\Delta}(A, \overrightarrow{X}) \overset{\text{def}}{=} (\Delta^1(A, \overrightarrow{X}), \ldots, \Delta^p(A, \overrightarrow{X}))$

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 $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \overrightarrow{\emptyset}$.

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

Postulate 3 (Bounded Exploration for Processors). *For every* BSP *algorithm A there exists a finite set $T(A)$ of terms 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 $1 \leq i \leq p$, we have $\Delta^i(A, \overrightarrow{X}) = \Delta^i(A, \overrightarrow{Y})$.*

$T(A)$ is called the **exploration witness** [10] of A. 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 the symbol "**true**" should always be in the exploration witness [10]. The interpretations of the terms in $T(A)$ are called the **critical elements** and we prove in [22] that every value in an update is a critical element:

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

That implies that for every step of the computation, for a given processor, only a bounded number of terms are read or written (amount of work).

Lemma 2 (Bounded Set of Updates). *For every state $\left(X^1, \ldots, X^p\right)$ of the algorithm A, for every $1 \leq i \leq p$, $|\Delta^i(A, \overrightarrow{X})|$ is bounded.*

Notice that for the moment we make no assumption on the communication between processors. Moreover, these three postulates are a "natural" extension of the ones of [10]. And by "natural", we mean that if we assume that $p = 1$ then our postulates are exactly the same:

Lemma 3 (A Single Processor is Sequential). *A* BSP *algorithm with a unique processor ($p = 1$) is a sequential algorithm. Therefore* $\text{ALGO}_{\text{SEQ}} \subseteq \text{ALGO}_{\text{BSP}}$. *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 $comp_A$ and $comm_A$ indicating if A runs computations or if it runs communications.*

Postulate 4 (Supersteps phases). *For every* BSP *algorithm A there exists two applications $comp_A : M(A) \to M(A)$ commuting with isomorphisms, and $comm_A : S(A) \to S(A)$, such that for every state $\left(X^1, \ldots, X^p\right)$:*

$$\tau_A\left(X^1, \ldots, X^p\right) = \begin{cases} \left(comp_A(X^1), \ldots, comp_A(X^p)\right) & \text{if } \exists 1 \leq i \leq p \\ & \text{such that } comp_A(X^i) \neq X^i \\ comm_A\left(X^1, \ldots, X^p\right) & \text{otherwise} \end{cases}$$

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 $\left(X^1,\ldots,X^p\right)$ will be said in a **computation phase** if there exists $1 \leq i \leq p$ such that $\text{comp}_A(X^i) \neq X^i$. Otherwise, the state will be said in a **communication phase**.

This requires some remarks. First, at every computation step, every processor which has not terminated performs its local computations. Second, we do not specified the function comm_A in order to be generic about which BSP library is used. We discuss in Sect. 3.3 the difference between comm_A and the usual communication routines in the BSP community.

Remember that a state $\overrightarrow{X}$ is said to be final if $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$. Therefore, 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.

We prove that the BSP algorithms satisfy, during a computation phase, that every processor computes independently of the state of the other processors:

Lemma 4 (No Communication during Computation Phases). *For every states $\left(X^1,\ldots,X^p\right)$ and $\left(Y^1,\ldots,Y^q\right)$ in a computing phase, if X^i and Y^j have the same critical elements then $\Delta^i(A,\overrightarrow{X}) = \Delta^j(A,\overrightarrow{Y})$.*

2.3 Questions and Answers

Why not using a BSP-Turing machine to define an algorithm?
It is known that standard Turing machines could simulate every algorithm. But we are here interested in the step-by-step behavior of the algorithms, and not the input-output relation of the functions. In this way, there is not a literal identity between the axiomatic point of view (postulates) of algorithms and the operational point of view of Turing machines. Moreover, simulating 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.

But I think there is too much abstractions: When using BSPLIB, *messages received at the past superstep are dropped. Your function comm_A does not show this fact.*
We want to be as general as possible. Perhaps a future library would allow reading data received n supersteps ago as the BSP+ model of [19]. Moreover, the communication function may realize some computations and is thus not a pure transmission of data. But the exploration witness forbids doing whatever: only a finite set of symbols can be updated. And we provide a realistic example of such a function which mainly correspond to the BSPLIB's primitives [22].

And 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.

And 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.

In [2,3,15,16]*, the authors give more general postulates about concurrent and/or distributed algorithms? Why not using their works by adding some restrictions to take into account the* BSP *model of execution?*
It is another solution. But we think that the restrictions on "more complex" postulates is not a natural characterization of the BSP algorithms. It is better for a model to be expressed at its natural level of abstraction in order to highlight its own properties. For example, there is the problematic of the cost model which is inherent to a bridging model like BSP: It is not clear how such restrictions could highlight the cost model.

Fine. But are you sure about your postulates? I mean, are they completely (and not more) defined BSP *algorithms?*
It is impossible to be sure because we are formalizing a concept that is currently only intuitive. But as they are general and simple, we believe that they correctly capture this intuitive idea. We prove in the next section that a natural operational model for BSP characterizes exactly those postulates.

Would not that be too abstract? The BSP *model is supposed to be a bridging model.*
We treat algorithms at their natural level of abstraction, and not as something to refine to machines: We explicitly assume that our primitives may not be elementary for a typical modern architecture (but could be so in the future) and that they can achieve a potentially complex operation in one step. This makes it possible to get away from a considered hardware model and makes it possible to calculate the costs in time (and in space) in a given framework which can be variable according to what is considered elementary. For example, in an Euclidean algorithm, it is either the Euclidean division that is elementary or the subtraction. If your BSP algorithm uses elementary operations which can not be realized on the BSP machine considered, then you are just not at the right level abstraction. Our work is still valid for *any* level of abstraction.

3 BSP-ASM Captures the BSP Algorithms

The four previous postulates define the BSP algorithms from an axiomatic viewpoint but that does not mean that they have a model, 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 the sequential algorithms [10], we prove in this section that the ASM_{BSP} model captures the BSP algorithms.

3.1 Definition and Operational Semantics of ASM-BSP

Definition 4 (ASM Program [10])

$$\begin{aligned}\Pi \stackrel{\text{def}}{=} & f\,(t_1,\ldots,t_\alpha) := t_0\\ & \mid \texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}\\ & \mid \texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}\end{aligned}$$

where f has arity α; F is a formula; θ_1, …, θ_α, θ_0 are terms of $\mathcal{L}(X)$. Notice that if $n = 0$ then `par` $\Pi_1\|\ldots\|\Pi_n$ `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 [10] is thus a kind of Turing machine using not a tape but an abstract structure X.

Definition 5 (ASM Operational Semantics)

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

Notice that the semantics of the `par` is a set of updates done simultaneously, which differs from an usual imperative framework. A state of a ASM$_{\text{BSP}}$ machine is a p-tuple of memories $(X^1, \ldots, X^p)$. We assume that the ASM$_{\text{BSP}}$ programs are SPMD (Single Program Multiple Data) 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 $\overrightarrow{\Delta}$ and a transition function τ_Π:

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

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 communication between processors.

Definition 6. *An* ASM$_{\text{BSP}}$ *machine M is a triplet $(S(M), I(M), \tau_M)$ such that:*

1. *$S(M)$ is a set of tuples of structures with the same finite signature $\mathcal{L}(M)$; $S(M)$ and $I(M) \subseteq S(M)$ are closed by multi-isomorphism;*
2. *$\tau_M : S(M) \mapsto S(M)$ verifies that there exists a program Π and an application $comm_M : S(M) \mapsto S(M)$ such that:*

$$\tau_M(\overrightarrow{X}) = \begin{cases} \tau_\Pi(\overrightarrow{X}) \textit{ if } \tau_\Pi(\overrightarrow{X}) \neq \overrightarrow{X}\\ comm_M(\overrightarrow{X}) \textit{ otherwise}\end{cases}$$

3. $comm_M$ verifies that:
   (1) For every state $\overrightarrow{X}$ such that $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$, $comm_M$ preserves the universes and the number of processors, and commutes with multi-isomorphisms
   (2) There exists a finite set of terms $T(comm_M)$ 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 $T(comm_M)$ then $\overrightarrow{\Delta}(M, \overrightarrow{X}) = \overrightarrow{\Delta}(M, \overrightarrow{Y})$.

We denote by ASM$_{\text{BSP}}$ the set of such machines. As before, 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 $\text{comm}_M(\overrightarrow{X}) = \overrightarrow{X}$.

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. For example, without these conditions, we could imagine that comm_M may compute the output of the algorithm in one step, or solve the halting problem. Moreover, we construct an example of comm_M in [22] (Section D).

3.2 The BSP-ASM Thesis

We prove that ASM$_{\text{BSP}}$ captures the computation phases of the BSP algorithms in three steps. First, we prove that during an execution, each set of updates is the interpretation of an ASM program (Lemma 8 p.16 [22]). Then, we prove an equivalence between these potentially infinite number of programs (Lemma 9 p.17). Finally, by using the third postulate, we prove in Lemma 10 p.18 that there is only a bounded number of relevant programs, which can be merged into a single one.

Proposition 1 (BSP-ASMs capture Computations of BSP Algorithms). *For every* BSP *algorithm A, there exists an* ASM *program Π_A such that for every state $\overrightarrow{X}$ in a computation phase: $\overrightarrow{\Delta}(\Pi_A, \overrightarrow{X}) = \overrightarrow{\Delta}(A, \overrightarrow{X})$.*

Theorem 1. ALGO$_{\text{BSP}}$=ASM$_{\text{BSP}}$ *(The proof is available in [22], Section C p.20).*

3.3 Cost Model Property and the Function of Communication

There is two more steps in order to claim that ASM$_{\text{BSP}}$ objects are the BSP bridging model algorithms: (1) To ensure that the duration corresponds to the standard cost model and; (2) To solve issues about the communication function.

Cost Model. 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 $1 \leq i \leq p$ such that $\text{comp}_A(X_t^i) \neq X_t^i$. The computation for every processor is done in parallel, step by step. So, the cost in time of the computation phase is $w \overset{\text{def}}{=} \max_{1 \leq i \leq p}(w_i)$, where w_i is the number of steps done by the processor i (on processor X^i) during the superstep.

Then the state is in a communication phase, when the messages between the processors are sent and received. Notice that 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 actions as one communication phase. ASM$_{\text{BSP}}$ exchanges terms and we show in [22] how formally define the size of terms. But we can imagine a machine that must further decompose the terms in order to transmit them (in bits for example). We just assume that the data are communicable in time $\mathbf{g}$ for a 1-relation.

So, during the superstep, the communication phase requires $h \times \mathbf{g}$ steps. It remains to add the cost of the synchronization of the processors, which is assumed in the usual BSP model to be a parameter $\mathbf{L}$. Therefore, we obtained a cost property which is sound with the standard BSP cost model.

A Realization of the Communication. An example of a communication function for the standard BSPLIB's primitives `bsp_get`, `bsp_put`, `bsp_send` `bsp_move` is presented in [22] (Section D).

Proposition 2 (Communication). *A function of communication, with routines for distant readings/writings and point-to-point sendings, performing an h-relation and requiring at most h exchanges can be designed using* ASM.

One may argue that the last postulate allows the communication function to do computations. To avoid it, we assume that the terms in the exploration witness $T(M)$ can be separated between $T(\Pi)$ and $T(\text{comm}_M)$ such that $T(\Pi)$ is for the states in a computation phase, and that for every update $(f, \overrightarrow{a}, b)$ of a processor X^i in a communication phase, either there exists a term $t \in T(\text{comm}_M)$ such that $b = \overline{t}^{X^i}$, or there exists a variable $v \in T(\Pi)$ and a processor X^j such that $b = \overline{t_{\overline{v}^{X^j}}}^{X^i}$ (**representation** presented in Section D p.24). To do a computation, a term like $x+1$ is required, so the restriction to a variable prevents the computations of the terms in $T(\Pi)$. Or course, the last communication step should be able to write in $T(\Pi)$, and the final result should be read in $T(\Pi)$.

4 Conclusion and Future Work

4.1 Summary of the Contribution

A bridging model 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*" (efficient and portable) parallel algorithms using a *realistic* cost model (and without any overspecification requiring the use of a large number of parameters) that can fit most distributed architectures. It has been used with success in many domains [1].

We have given an axiomatic definition of BSP algorithms by adding only one postulate to the sequential ones for sequential algorithms [10] which has been widely accepted by the scientific community. Mainly this postulate is the call of a function of communication. We abstract how communication is performed, not be restricting to a specific BSP library. We finally answer previous criticisms by defining a convincing set of parallel algorithms running in a predictable time.

Our work is relevant because it allows universality (immortal stands for BSP computing): all future BSP algorithms, whatever their specificities, will be captured by our definitions. So, our ASM$_{\text{BSP}}$ is not just another model, it is a class model, which contains all BSP algorithms.

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. But our work is limited to BSP algorithms even if it is still sufficient for many HPC and big-data applications. 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$_{\text{BSP}}$. We also prove 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.

4.2 Questions and Answers About this Work

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 by using an* ASM*.*
Even if such a simulation exists between these two models, what you mean, a "sequentialization" (each processor, one after the other) of the BSP model of execution, cannot be exactly the function of transition of the postulates. Moreover, in order to stay bounded, having p exploration witness (one for each sequential ASM) induces p to be a constant for the algorithm. In our work, p is only fixed of each execution, making the approach more general when modeling algorithms.

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, just another way to describe it.

So, reading the work of [3]*, a distributed machine is defined as a set of pairs* (a, Π_a) *where a is the name of the machine and* Π_a *a sequential* ASM*. Reading your definition, I see only one* Π *and not "*p*" processors as in the* BSP *model. I thus not imagine a* BSP *computer as it is.*
You are absolutely right but we do not model a BSP computer, our work is about BSP algorithms. The ASM$_{\text{BSP}}$ program contains the algorithm which is used on each "processor" (a first-order structure as explain before). 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 [10]. We also want to point out that, unlike [3], we are not

limited to a finite (fixed) set of machines: In our model, an algorithm is defined for $p = 1, 2, 1000$, *etc.* And we are not limited to point-to-point communications.

Ok, but with only a single code, you cannot have all the parallel algorithms...
We follow [4] 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$_{\text{BSP}}$ is SEQ(PAR). This leads to a *macroscopic* point of view[1] which is close to a specification. Being a SEQ(PAR) model allows a high level description of the BSP algorithms.

So, why are you limited to SPMD *computations?*
Different codes can be run by the processors using conditionals on the "id" of the processors. For example "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 there could be completely different. Only the size of Π is finite due to the exploration witness. For example, it is impossible to have a number of conditionals in Π that depends of p. Indeed, according to Lemma 4, during a computation phase, if two processors coincide over the exploration witness, then they will use the same code. And according to Postulate 3, the exploration witness is bounded. So, there exists only a bounded number c of possible subroutines during the computation phase, even if $p \gg c$.

Notice that processors may not know their own ids and there is no order in p-tuples; We never use such a property: Processors are organized like a set and we use tuples only for convenience of notation. We are using p-tuples just to add the BSP execution model in the original postulates of [10].

Ok, but I cannot get the interleavings of the computations as in [3]*? Your model seems very synchronous!*
The BSP model makes the hypothesis that the processors are 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 "asynchronous" during the computation phases.

Speaking about communication, why apply several times the function of communication? When designing a BSP *algorithm, I use once a collective operation!*
An ASM is like a Turing machine. It is not possible to perform all the communications in a single step: The exploration witness forbids doing this. Our function of communication performs some exchanges until there are no more.

[1] Take for example a BSP sorting algorithm: First all the processors locally sort there own data, and then, they perform some exchanges in order to have the elements sorted between them. One defines it as a sequence of parallel actions and being also independent to the number of processors.

What happens in case of runtime errors during communications?
Typically, when one processor has a bigger number of super-steps than other processors, or when there is an out-of-bound sending or reading, it leads to a runtime error. The BSP function of communication can return a $\perp$ value. That causes a stop of the operational semantics of the ASM$_{\text{BSP}}$.

4.3 Related Work

As far as we know, some work exists to model distributed programs using ASMs [15] but none to convincingly characterize BSP algorithms. In [6], authors model the P3L set of skeletons. That allows the analyze of P3L programs using standard ASM tools but not a formal characterization of what P3L is and is not.

The first work to extend ASMs for concurrent, distributed, agent-mobile algorithms is [2]. Too many postulates are used making the comprehension hard to follow or worse (loss of confidence). A first attempt to simplify this work has been done in [16] and again simplified in [7] by the use of multiset comprehension terms to maintain a kind of bounded exploration. Then, the authors prove that ASMs captures these postulates. Moreover, we are interested in distributed (HPC) computations more than parallel (threading) ASMs.

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 (or computing units) is unbounded for parallel algorithms, which motivated the work of [2] to define parallel algorithms with wide steps, that is steps of unbounded complexity. Hence the technicality of the presentation, and the unconvincing attempts to capture parallel algorithms [3].

Extending the ASMs for distributed computing is not new [3]. We believe that these postulates are more general than ours but we think that our extension still remains simple and natural for BSP algorithms. The authors are also not concerned about the problem of axiomatizing classes of algorithms using a cost model which is the heart of our work and the main advantage of the BSP model.

4.4 Future Work

This work leads to many possible work. First, how to adapt our work to a hierarchical extension of BSP [21] which is closer to modern HPC architectures?

Second, BSP is a bridging model between hardwares and softwares. It could be interesting to study such a link more formally. For example, can we prove that the primitives of a BSP language can truly "be BSP" on a typical cluster architecture?

Thirdly, we are currently working on extending the work of [13] in order to give the BSP algorithmic completeness of a BSP imperative programming language. There are some concrete applications: There are many languages having a BSP-like model of execution, for example PREGEL [12] for writing large-graph algorithms. An interesting application is proving which are BSP algorithmically complete and are not. BSPLIB programs are intuitively BSP. MAPREDUCE is a good candidate to be *not* [14]. Similarly, one can imagine proving which languages are too expressive for BSP. MPI is intuitively one of them. Last, the first author is working on postulates for more general distributed algorithm *à la* MPI.

In any case, studying the BSP-RAM (such as the communication-oblivious of [19]) or MAPREDUCE, would led to define subclasses of BSP algorithms.

References

1. Bisseling, R.H.: Parallel Scientific Computation: A Structured Approach Using BSP and MPI. Oxford University Press, Oxford (2004)
2. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Trans. Comput. Log. **4**(4), 578–651 (2003)
3. Börger, E., Schewe, K.-D.: Concurrent abstract state machines. Acta Inf. **53**(5), 469–492 (2016)
4. Bougé, L.: The data parallel programming model: a semantic perspective. In: Perrin, G.-R., Darte, A. (eds.) The Data Parallel Programming Model. LNCS, vol. 1132, pp. 4–26. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-61736-1_40
5. Cappello, F., Snir, M.: On communication determinism in HPC applications. In: Computer Communications and Networks (ICCCN), pp. 1–8. IEEE (2010)
6. Cavarra, A., Zavanella, A.: A formal model for the parallel semantics of p3l. In: ACM Symposium on Applied Computing (SAC), pp. 804–812 (2000)
7. Ferrarotti, F., Schewe, K.-D., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing –simplified parallel ASM thesis. Theor. Comput. Sci. **649**, 25–53 (2016)
8. González-Vélez, H., Leyton, M.: A survey of algorithmic skeleton frameworks. Softw. Pract. Exp. **40**(12), 1135–1160 (2010)
9. Gorlatch, S.: Send-receive considered harmful: myths and realities of message passing. ACM TOPLAS **26**(1), 47–56 (2004)
10. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. **1**(1), 77–111 (2000)
11. Hill, J.M.D., McColl, B., et al.: BSPLIB: the BSP programming library. Parallel Comput. **24**, 1947–1980 (1998)
12. Malewicz, G., et al.: PREGEL: a system for large-scale graph processing. In: Management of data, pp. 135–146. ACM (2010)
13. Marquer, Y.: Algorithmic completeness of imperative programming languages. Fundamenta Informaticae, pp. 1–27 (2017, accepted)
14. Pace, M.F.: BSP vs MAPREDUCE. Procedia Comput. Sci. **9**, 246–255 (2012)
15. Prinz, A., Sherratt, E.: Distributed ASM- pitfalls and solutions. In: Ait Ameur, Y., Schewe, K.D. (eds.) ABZ 2014. Lecture Notes in Computer Science, vol. 8477, pp. 210–215. Springer, Heidelberg (2014)
16. Schewe, K.-D., Wang, Q.: A simplified parallel ASM thesis. In: Derrick, J., et al. (eds.) ABZ 2012. LNCS, vol. 7316, pp. 341–344. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-30885-7_27
17. Seo, S., et al.: HAMA: an efficient matrix computation with the MAPREDUCE framework. In: Cloud Computing (CloudCom), pp. 721–726. IEEE (2010)
18. Skillicorn, D.B., Hill, J.M.D., McColl, W.F.: Questions and answers about BSP. Sci. Program. **6**(3), 249–274 (1997)

19. Tiskin, A.: The design and analysis of bulk-synchronous parallel algorithms. PhD thesis. Oxford University Computing Laboratory (1998)
20. Valiant, L.G.: A bridging model for parallel computation. Comm. ACM **33**(8), 103–111 (1990)
21. Valiant, L.G.: A bridging model for multi-core computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)
22. Marquer, Y., Gava, F.: An ASM thesis for BSP. Technical report (2018). https://hal.archives-ouvertes.fr/hal-01717647