Axiomatization and Imperative Characterization of Multi-BSP Algorithms: A Q&A on a Partial Solution

Frédéric Gava, Yoann Marquer

Axiomatization and Imperative Characterization of Multi-BSP Algorithms

A Q&A about a partial solution and some inherent problems

Frédéric Gava
LACL, University of Paris-East
Créteil, France
frederic.gava@univ-paris-est.fr

and

Yoann Marquer
Inria, Univ Rennes, CNRS, IRISA
Rennes, France
yoann.marquer@inria.fr

August 13, 2020

Abstract

MULTI-BSP is a new bridging model which takes into account hierarchical architectures. We discuss two questions about MULTI-BSP algorithms and their programming: (1) How do we get a formal characterization of the class of MULTI-BSP algorithms? (2) How can a programming language be proven algorithmically complete for such a class? Our solution is based on an extension of a BSP version of sequential Abstract State Machines (ASMs).

1 Introduction

1.1 Context of the work and background

Modern HPC architectures are made of hundreds of interconnected nodes each, with thousands of cores. Programming and reasoning on such *hierarchical* architectures is a daunting task without the use of some *high-level abstractions*.

1.1.1 Multi-processors algorithms and hierarchical architectures

As it has been shown in [2], the BSP model [4, 23] is not truly adapted for hierarchical architectures (design of portable and scalable algorithms, performance prediction, *etc.*). BSP then partially loses the fact of being an efficient *bridging model*. An update of this model is needed and must be defined with the same level of abstraction, and bridges most of the architectures as the original BSP model did. So a proposition has been made in [26] and it has been called MULTI-BSP. The intention is: "*with the comparison with the previous literature, our goal here is that of finding a bridging model that isolates the most fundamental issues of multi-core computing and allows them to be usefully studied in some detail.*" [26].

An important issue [1, 26] is the ability of designing *immortal algorithms*, which are algorithms that *scale* for *any machine* in the present and in the future as long as the bridging model can be adapted to it. In particular, these algorithms must work for any number of computing units (up to some constraints such as a number of processors equal to a power of two). This raises two questions: (1) can we *characterize* the *class* of the MULTI-BSP algorithms so that it is a natural extension of the class of the BSP algorithms [20]? And (2), can we prove that all the MULTI-BSP algorithms can be *programmed* using a programming language? If we can *intuitively* answer yes to these two questions or at least as soon as you see a MULTI-BSP algorithm or a programming language, *formally*, this requires more attention. Firstly, what is an algorithm [27] and secondly what is being algorithmically complete. And thus, what class do MULTI-BSP algorithms belong to?

1.1.2 Axiomatization and imperative characterization of sequential algorithms

The most known definition of *sequential* (small steps and discrete time) algorithms is the *axiomatic* presentation of [15, 16]. The main idea is that there is not any standard language that truly represents all algorithms. Three widely consensual *postulates* are used to define the infinite set of sequential algorithms: (1) *sequential time*, algorithms work step-by-step; (2) *abstract states*, algorithms are oracular and the steps only depend on primitives (elementary operations); (3) *bounded exploration*, each of these steps is finite. This axiomatic definition has been mapped to the notion of Abstract State Machine (ASM, also called *evolving algebras, a kind of Turing machine with the appropriate level of abstraction* [15, 16]). Every sequential algorithm can be captured by an ASM and every ASM is a sequential algorithm. That is $\text{ALGO}_{\text{SEQ}}=\text{ASM}_{\text{SEQ}}$.

Furthermore, the study of class models [14, 19, 20] allows us to classify what can or cannot be effectively programmed. Indeed, if it is known that mainstream languages such as C or JAVA are *Turing-complete*, which means that they can simulate the computation of any computable function (so the input-output of any Turing machine, up to an unbounded memory), what is called *algorithmic completeness* rather focuses on the *step-by-step* behavior that is, a model could compute all the desired functions, but some algorithms (ways to compute these functions) could be missing [14]. By using the aforementioned axiomatization of the algorithms and an *operational semantics* of an imperative core-language (IMP), it has been formally proved in [19] that $\text{ASM}_{\text{SEQ}} \simeq \text{IMP}$ that is mainstream languages are also algorithmically complete, which was only informally assumed so far.

In [20], we extended these results to BSP and proved that $\text{ALGO}_{\text{BSP}}=\text{ASM}_{\text{BSP}} \simeq \text{IMP}_{\text{BSP}}$. To do so, we add another postulate that arranges the steps into BSP's *supersteps* and in a way that the BSP algorithms no longer work on a single memory (a first-order structure) but on p-tuples of arbitrary sizes, on per computing unit. We also extend the ASMs so that they work on these p-tuples in a SEQ(PAR) manner [7]. ASM_{BSP} use internally a global *function of communication* (which is also working step-by-step) *abstracting* how communications are performed by any BSP library. Finally, we extend IMP in an SPMD (Single Program Multiple Data) fashion so that programs also work on p-tuples but in a PAR(SEQ) manner. IMP_{BSP} programs use an explicit call of a function of communication (*i.e.* a specific command). In this way, we define the class of BSP algorithms and prove that BSP languages (mostly C/JAVA+BSPLIB ones) are algorithmically complete to this class.

1.2 Content of the work

We are now interested in answering the question of whether such results can be extended to the MULTI-BSP model, that is

(Intended theorem) $\text{ALGO}_{\text{MULTI}}= \text{ASM}_{\text{MULTI}} \simeq \text{IMP}_{\text{MULTI}}$

That is proving an equivalence between an axiomatic definition of the MULTI-BSP algorithms and their operational points of view ($\text{ASM}_{\text{MULTI}}$) and, proving an algorithmic simulation between $\text{IMP}_{\text{MULTI}}$ and $\text{ASM}_{\text{MULTI}}$ (up to elementary operations, *e.g.* how to perform integer additions or how to manage the communications). Because the MULTI-BSP model is ("just") a hierarchical extension of BSP, it seems appropriate to ask the question of how to extend the previous results about BSP [20] in a "*minimal*" way rather than doing all the proofs from scratch. We will show that this is mostly feasible but that there are some points that require more important modifications due to the use of *nested supersteps*. It is to notice that some definitions will be slightly unsatisfying because there is a lack of description of how computations are performed on "nodes" in the MULTI-BSP model [26].

1.3 Outline

We mimic the works of [14, 15, 23] to organize the rest of the paper as a *dialog* between the *author* and a curious but equally *scrupulous colleague*. We made this uncommon choice of writing because this article is the *continuation* of [20] (which also contains a lot of FAQs) and it seems natural to ask ourselves the question of how to do such a work.

In Section 2 p.5, we first discuss MULTI-BSP algorithms (SubSection 2.1 p.5) and their axiomatization (SubSection 2.2 p.7). In Section 3 p.13, we then discuss their operational points of view as an extension of ASMs (SubSection 3.1 p.13) and the algorithmic simulation with a core programming language (SubSection 3.2 p.16). We finish this section by proving the intended theorem and that the *cost model is*

preserved. Some related work is presented in Section 4 p.22. Section 5 p.25 concludes and finishes with a brief outlook on future work.

Contents

2 An axiomatization of MULTI-BSP algorithms

Question 1: *Hello. Before we start, we probably need a common vocabulary. What should I read so that we can understand each other?*

We recommend the original papers about ASMs [15, 16]. The author presents the ideas behind ASMs (the wanted Church-Turing thesis for algorithms). The paper [14] is also a good introduction to the ideas of classes and algorithmic completeness whereas [23] is a perfect introduction to the BSP bridging model. And of course [26] is the reference for MULTI-BSP.

This work is the continuation of [20], so, to get a common *vocabulary*, it is strongly recommended to read it. All the notions used in this work are presented there (except what is dedicated to the MULTI-BSP model): first-order structures X (page. 6), signature, terms θ and universe (p. 7), interpretation of terms $\overline{\theta}^X$ (p. 7), isomorphism of structures (p. 8) and consistent updates Δ (p. 8), *etc.*

2.1 The MULTI-BSP model

Q2: *Could you remind me of the* MULTI-BSP *model which is less known than* BSP*?*

MULTI-BSP extends the BSP bridging model to take into account modern hierarchical architectures. There exist other extensions such as the one of [9] but MULTI-BSP describes hierarchical architectures in a simpler way. This model brings a *tree*-based view of nested *components* (sub-machines) where the lowest *levels*[1] are computing units (processors, the ***leaves***) with a small block of memory (*e.g.* caches) and every other level (the ***nodes***) contains a memory *only* (or a network). Inside a node-component, each memory can access other memories only with the use of an explicit global communication. The tree is of depth $\mathbf{d}$ and is assumed *balanced and homogeneous*. Fig. 1 illustrates the different components. A MULTI-BSP architecture has four parameters at each level $s \in \{1 \cdots \mathbf{d}\}$:

1. $\mathbf{p_s}$ is the number of sub-components inside level s;
2. $\mathbf{g_s}$ is the *bandwidth* between levels s and $s-1$; the ratio of the number of operations to the number of words that can be transmitted in a second;
3. $\mathbf{L_s}$ is the *synchronization cost* of all sub-components at $s-1$ level;
4. $\mathbf{m_s}$ is the amount of memory available for each component at level s.

Finally, there is the homogeneous local processing speed $\mathbf{r}$ of each of the processors (components of level 1) where a step on a word is considered as the unit of time ($\mathbf{g}_1 = 1$). A leaf does not have sub-components making $\mathbf{p}_1$ and $\mathbf{L}_1$ both equal to 0. For example, the BSP model with parameters ($\mathbf{p}$, $\mathbf{g}$, $\mathbf{L}$) where each basic unit has memory $\mathbf{m}$ would be modeled with [26] $\mathbf{d} = 2$ and $(\mathbf{p}_1 = 0, \mathbf{g}_1 = 1, \mathbf{L}_1 = 0, \mathbf{m}_1 = \mathbf{m})$, $(\mathbf{p}_2 = \mathbf{p}, \mathbf{g}_2 = \mathbf{g}, \mathbf{L}_2 = \mathbf{L}, \mathbf{m}_2)$ where $\mathbf{m}_2$ is, for example, the size of a slower but bigger memory. Another value for $\mathbf{m}_2$ could be n, the size of the problem (implemented as a virtual memory with a distributed file system). Assuming a specific architecture with data streams as input, $\mathbf{m}_2$ could also be ∞ and $\mathbf{g}_2$ could rely on the acquisition stream data rate.

Q3: *And what about the execution model of* MULTI-BSP *algorithms?*

A MULTI-BSP computer works in a *sequence of nested supersteps* where at a node level s, each superstep is made of the supersteps of levels $s-1$ and terminates by a synchronization (barrier) of all the sub-machines of level $s-1$ in favor of a data exchange between the m_{s-1} memories and m_s whereas on leaves (level 1), a superstep is made of computations only[2] (using the data accessible the local memories of leaves). Then, a new superstep can start at level s. Note that the definition requires synchronization of the directly below sub-components of a component, but no synchronization across branches that are separated in the component hierarchy. The *cost* of a MULTI-BSP algorithm is the sum of the costs of the supersteps of the root node, where the cost of each of these supersteps is (recursively) the maximal cost of the supersteps of the sub-components plus the needed communication plus the synchronization; And for each leaf, the cost is the computation time. And so on.

[1] The term of "stage" was used in place of "level" in [1, 2] because "level" (as layer or floor) is largely used in other domains in computer science notably in type systems that were used in [2].

[2] Regarding a level that is only composed by a memory capacity, that is to say a node, one processor of the branch is

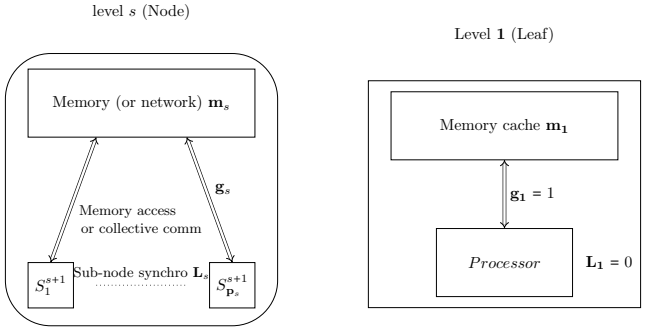

Figure 1: The MULTI-BSP components.

Q4: *Hum, the architecture is not exactly identical to the original work of [26]. Why?*

Indeed and these modifications do not change the way to write or analyze the MULTI-BSP algorithms. Firstly, we slightly change the indexes for convenience only. For example, the bandwidth parameter of the memory concerns the current level instead of the upper level as originally described in [26]. Secondly, in our purpose, each level has its own memory whereas in [26], the leaves are computing units only, an additional abstract level. This also simplifies the presentation. Thirdly, we natively allow *horizontal* communications to make the set of the BSP algorithms a *subset* of the MULTI-BSP ones and, in particular, with the idea of working in an *incremental* manner (as in [1]), that is using BSP programs inside MULTI-BSP ones.

Q5: *So your model do not have shared memories and concurrent accesses to them?*

Definitively not. Concurrent writing induces unnecessarily complicated semantics (non-deterministic behaviour) and even if we assume only concurrent reading, we will see later that this leads to an axiomatization too far away from that of [16]. From a certain point of view, our MULTI-BSP model is a kind of homogeneous tree organized D-BSP [25] model without dynamic subgroup synchronizations and where sub-components shared a memory.

Q6: *So, in your model, processors cannot read/write values from any level. And the communication phase at a given level i (the h-relation+synchronization L_i) is performed by the sub-components only and not by all the processors that belong to level i. What about the $G_i = g_i + g_{i1} + \cdots + g_1$ parameter (the cost of communication from level 1 to outside level i) of [26]? Can you elaborate on that? Because your model deviates from the original model of [26] where each memory, at a level i, is shared be all processors belong it.*

Absolutely. We are in a distributed memory model. There are some *pro* and *cons*. First, hierarchical-network clusters can be taken into account (assuming that some machines share their own memomies with other ones and where the accesses are, of course, only using communication). Second, a superstep that is using communication from level 1 to outside level i (the G_i parameters[3]) can be simulated by using only communication on the deeper levels $(i-1, \cdots, 1)$; that is closer to the recursive data decomposition of algorithms. And if processors share a given memory (say the L3 cache), then we can cut it into several parts. Using distributed memories is also closer to the original BSP model. Third, if there are too much cores, there may be a bootleneck making the G_i parameter inoperative when accessing certain memories (*e.g.* the RAM). On the contrary, the number of "simultaneous" accesses is reduced by considering those by sub-nodes only (during a communication phase). And for performing the communications (scattering/gathering data) where would be the intermediary needed values, on the lowest levels? And finally, how could processors access a memory (*e.g.* the RAM) without having caches copies (kind of miss-caches) made by the OS/architecture? Using only communications between levels makes this more explicit (assuming always enough memory when doing the local computations that is when processors are using the memories at level 1).

responsible for all the memory manipulations that are done to it: this processor (which is chosen depends on the implementation) performs the data exchanges such as gathering data, *etc.* We will show later the inherent problem of such a design.

[3]It is to notice that such parameters are defined but never explicitly used in[26] and only communication between consecutive levels are used in the algorithms.

But this model suffers for two main defects. First, if there are only communication between different levels, how does the system handle them? Scattering and gathering data needs a little of computation. The solution of [29] is using a "master" that is selecting a core to handle a given level. We will analyse this later. Second, there is no possible communication between processors of different sub-trees without going through all their top levels and thus using there synchronization mechanisms (unlike the model of [26]). This overcost must be taken into account when designing MULTI-BSP algorithms in this model.

Q7: *You actually define a* MULTI-BSP *computer and so what are algorithms?*

Definition 1 *(*MULTI-BSP *algorithms, informal) A* MULTI-BSP *algorithm is a computation, organized in a nested sequence of supersteps, that can stand on any* MULTI-BSP *machine*[4].

Now let us present a more formal definition.

2.2 Axiomatization of MULTI-BSP algorithms

Q8: *It does not seem to be moving too far. What is necessary to be modified in your previous formal axiomatization [20] of the* BSP *algorithms?*

Obviously the tree of nested components (memories) and the organization of the nested supersteps.

Q9: *That sounds too easy. I guess you must redefine all the previous postulates and the* ASM*s? And the imperative language too?*

You are right. But let us start with a word of caution and with the postulates. The first one is identical to the one of both sequential [16] and BSP algorithms [20]. It stipulates that there is no concurrency.

2.2.1 MULTI-BSP algorithms as state transition systems

Postulate 1 (Sequential Time) *A* MULTI-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. We recall [16, 20] that an *execution* of an algorithm A is an infinite sequence of states $\vec{S} = S_0, S_1, S_2, \ldots$ such that S_0 is an initial state and for every $t \in \mathbb{N}$, $S_{t+1} = \tau_A(S_t)$ (*accessible* states). And we will say that a state S_t of an execution $\vec{S}$ is *final* if $\tau_A(S_t) = S_t$ (*terminal* execution). The *duration* is

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

Every algorithm can access *elementary operations* (*a.k.a. primitives*) which only depend of the architecture.

Q10: *Is it what is called "intrinsically oracular"?*

Right. This is an essential feature. For example, a MULTI-BSP algorithm could be different whether a broadcasting primitive is available or is simulated by point-to-point sending of data. We are not interested in a specific library nor a particular machine but to all possible MULTI-BSP algorithms.

Q11: *And so what happens to* MULTI-BSP *architectures. Are they also oracular?*

Basically yes. MULTI-BSP algorithms will manipulate *states* as d-nested-tuples (called *trees* in the rest of the paper) of structures. Each structure represents the *available memory* of a component which could only be accessible using communicating primitives (nodes), except the ones of computing units (leaves) where computations could also occur. The related definitions about structures of [16, 20] will subsequently be used for each component. Let us define the trees.

[4] Assuming, if necessary, some properties on the performance parameters such as $\mathbf{p_2}$ is even or, on most modern and realistic machines, that $\forall s \in \{1 \ldots \mathbf{d}\}$, $\mathbf{m}_s \gg \mathbf{m}_{s-1}$, *etc.*

2.2.2 States as MULTI-BSP Trees

Definition 2 (MULTI-BSP Trees) *Trees are made of nodes and leaves. Trees are defined by induction*

$$T \stackrel{\text{def}}{=} \langle X \rangle \mid \left\langle X \middle| T_1, \ldots, T_p \right\rangle$$

where $p > 0$ (the number of subcomponents at the level) and X is the **memory** *(structure) of the component. $T_1, \ldots, T_p$ (noted $\vec{T}$ in the following) are the p* **branches** *(subtrees) of the node.*

We assume that for each leaf $\langle Y \rangle$, a processor exists and can only access without communication to the Y memory (local computations), whereas nodes $\langle X | T_1, \ldots, T_p \rangle$ can only communicate between two consecutive levels. We also assume that each memory is unique, for instance that there exists a symbol **id** interpreted differently on every memory. To remain consistent with Subsection 2.1 p.5, the leaves and their associated processors are homogeneous. Moreover, in the following (and for simplicity and when it is unambiguous), by assuming $p \geq 0$, $\langle X | T_1, \ldots, T_p \rangle$ or $\langle X | \vec{T} \rangle$ may denote both leaves and nodes.

Q12: *You provided a classical yet general definition for trees, but in Subsection 2.1 p.5 you assumed the trees to be balanced.*

Indeed this is an important assumption for the MULTI-BSP algorithms (cost analysis, *optimality* [26] and algorithm design such as for the distribution and *load-balancing* of data), but our proofs do not require this hypothesis and thus we wanted to prove a more general result. If we wanted the MULTI-BSP d-trees to be balanced, we could have defined them inductively as $T \stackrel{\text{def}}{=} \langle X \rangle^1 \mid \langle X | T_1^n, \ldots, T_p^n \rangle^{n+1}$, where n is the level.

Now we can define some simple relations on trees such as subtree, structural equality and position of a subtree in a tree.

Definition 3 (Subtree relation) *For every S and T (trees), we define inductively $S \trianglelefteq T$, S is a subtree of T, as*

$$\begin{array}{c} S \trianglelefteq S \\ S \trianglelefteq T_i \;\Rightarrow\; T \trianglelefteq \left\langle X \middle| T_1, \ldots, T_i, \ldots, T_p \right\rangle \end{array}$$

Notice that because each memory is unique, if $S \trianglelefteq \langle X | T_1, \ldots, T_p \rangle$ then there exists a unique i such that $S \trianglelefteq T_i$. Finally, for the second postulate, we need to define the notion of "*preserving* the structure of a tree" and tree-isomorphism.

Definition 4 (Tree-similarity (structurally))

$$\begin{array}{c} \langle X \rangle \triangleq \langle Y \rangle \\ \left\langle X \middle| T_1, \ldots, T_p \right\rangle \triangleq \left\langle Y \middle| U_1, \ldots, U_q \right\rangle \\ \text{if } p = q \text{ and for every } 1 \leq i \leq p, T_i \triangleq U_i \end{array}$$

To put it simply, the trees are the same but with potentially different memory content on their components. This notation is not restricted to trees of structures, and will be used also in Definition 6 for trees of functions. For example, $\left\langle X^1 \middle| \langle X^2 \rangle, \langle X^3 \rangle \right\rangle$ is *not* similar to $\left\langle Y^1 \middle| \langle Y^2 \rangle, \langle Y^3 \rangle, \langle Y^4 \rangle \right\rangle$.

Definition 5 (Position in a tree) *For every similar pair of trees $T \triangleq U$ and every subtree $S \trianglelefteq T$, we define inductively $\mathbf{pos}(S, T; U)$ as the subtree of U that is in the same position as S is in T:*

$$\begin{array}{c} \mathbf{pos}(S, S; U) \stackrel{\text{def}}{=} U \\ \mathbf{pos}\left(S, \langle X | \vec{T} \rangle; \langle Y | \vec{U} \rangle\right) \stackrel{\text{def}}{=} \mathbf{pos}(S, T_i; U_i) \end{array}$$

where T_i is the unique (see Definition 3) T_j amongst $T_1, \ldots, T_p$ such that $S \trianglelefteq T_j$

In [20] we defined multi-isomorphisms for tuples of structures, that should have been named "tuple-isomorphisms". For this paper, we define tree-isomorphisms.

Definition 6 (Tree-isomorphism) *A* tree-function Φ *is defined inductively as a tree of functions:*

$$\Phi \stackrel{\text{def}}{=} \langle \varphi \rangle \mid \langle \varphi | \Phi_1, \ldots, \Phi_p \rangle$$

where $p > 0$ and φ is a standard function, such that for every tree $T \triangleq \Phi$:

$$\begin{array}{lll} \text{if } \Phi = \langle \varphi \rangle & \text{then } \Phi\langle X \rangle & = \langle \varphi(X) \rangle \\ \text{if } \Phi = \langle \varphi | \Phi_1, \ldots, \Phi_p \rangle & \text{then } \Phi\langle X | T_1, \ldots, T_p \rangle & = \langle \varphi(X) | \Phi_1(T_1), \ldots, \Phi_p(T_p) \rangle \end{array}$$

For every T and U (trees), the tree-function Φ is a tree-isomorphism *between T and U if 1) $T \triangleq \Phi \triangleq U$, and 2) for every subtree $\langle X|\vec{T}\rangle \Subset T$ and $\langle Y|\vec{U}\rangle = \mathbf{pos}\Big(\langle X|\vec{T}\rangle, T; U\Big)$ there exists $\langle \varphi|\vec{\Phi}\rangle \triangleq \langle X|\vec{T}\rangle$ such that φ is an isomorphism between X and Y.*

To put it simply, we apply an isomorphism to each memory (structure) of the tree. For example, $\Big\langle X^1\Big|\langle X^2\rangle, \langle X^3\rangle\Big\rangle$ is isomorph to $\Big\langle Y^1\Big|\langle Y^2\rangle, \langle Y^3\rangle\Big\rangle$ if each X^i is isomorph to each Y^i. Notice that if Φ is a tree-isomorphism then Φ^{-1}, defined as Φ where all the φ have been replaced by their reciprocal φ^{-1}, is also a tree-isomorphism which is the reciprocal of Φ.

Q13: *What does isomorphism in this context mean?*

We are interested in the actual properties of all the models of computation, not the naming conventions used for them, as in [16]. Thus, in the following, we always reason up to isomorphism.

Postulate 2 (Abstract States) *For every* MULTI-BSP *algorithm A:*

1. *The states of A are trees of structures with the same finite signature $\mathcal{L}(A)$ (containing at least the booleans, the equality and a uniquely interpreted* **id** *(identifier) symbol);*
2. *$S(A)$ and $I(A)$ are closed by tree-isomorphisms and subtrees;*
3. *The transition function τ_A preserves tree structures and the universes, and commutes with tree-isomorphisms.*

Q14: *Are you defining a fixed size for the trees?*

No. For each state, d (the maximal depth of a tree) and each p_i are fixed for each execution only, abstracting the run of the algorithm on different MULTI-BSP machines and thus making the approach general when modeling algorithms (the notion of *immortal* algorithm). In the same way, the structures of different initial states contain potentially different values, abstracting the run of different input (initial) data.

Q15: *Ok and what does "preserves tree structures" mean?*

By "preserves tree structures" we mean that if T is a state (thus is a tree) then $\tau_A(T)$ is a tree such that $\tau_A(T) \triangleq T$. For every subtree $S \Subset T$ we define $\tau_A(T)_S$, the (unique) subtree of $\tau_A(T)$ which is in the same position in $\tau_A(T)$ as S is in T.

Definition 7 (Next subtree)

$$\tau_A(T)_S \stackrel{\text{def}}{=} \mathbf{pos}(S, T; \tau_A(T))$$

Q16: *Also, what does "closed and commute" by/with "tree-isomorphisms" or "subtrees" mean?*

Let Z be a set of trees. Z is closed by tree-isomorphims means that if Φ is a tree-isomorphism and $T \in Z$ then $\Phi(T) \in Z$. Z is closed by subtrees means that if $S \Subset T$ and $T \in Z$ then $S \in Z$. τ_A commutes with tree-isomorphisms means that if Φ is a tree-isomorphism and T is a state then $\tau_A(\Phi(T)) = \Phi(\tau_A(T))$.

Q17: *Could you also briefly remind me of the concept of universe?*

We recall [16, 20] that if a structure X has *signature* $\mathcal{L}(A)$ and *universe* $\mathcal{U}(X)$, the *update* $(f, \vec{a}, b)$ where $f \in \mathcal{L}(A)$ and $\vec{a}, b \in \mathcal{U}(X)$ is defined as in [20, 16], and so is the structure $X \oplus (f, \vec{a}, b)$ with the same signature and universe as X, and where all the *symbols* are *interpreted* as in X except f that now has value b in $f(\vec{a})$. The update is said *trivial* if nothing has changed. As in [20, 16], the notation $X \oplus \Delta$ is extended to tuples of updates Δ that *clash* if inconsistent. Moreover, if the structures X and Y have the same signature and universe, then $\Delta = Y \ominus X$ denotes the unique set of non-trivial updates such that $Y = X \oplus \Delta$.

Definition 8 (Updates of a leaf) *As in [16, 20], the set of updates done to the memory of the leaf $\langle X\rangle$ is $\Delta(A, X) \stackrel{\text{def}}{=} \tau_A(X) \ominus X$.*

In [20] the states were tuples $\vec{X}$ of structures and we defined $\Delta^i(A, \vec{X})$ as the i-th component of the tuple $\tau_A(\vec{X}) \ominus \vec{X}$. But in the MULTI-BSP context, where the states are trees, we define updates of subtrees.

Definition 9 (Updates of a subtree) *For every state T, the set of updates done for a subtree $\langle X|\vec{T}\rangle \Subset T$ is $\Delta(A, T)_{\langle X|\vec{T}\rangle} \stackrel{\text{def}}{=} Y \ominus X$, where $\langle Y|\vec{U}\rangle = \tau_A(T)_{\langle X|\vec{T}\rangle}$.*

2.2.3 MULTI-BSP algorithms work step-by-step

Q18: *And now, do processors compute locally (and nodes communicate) on every isomorphic data-structure without limitation?*

Sorry to say no again. We want the algorithms to work *step-by-step*. Indeed, 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* [16] (and thus opposed to the bridging model approach).

Q19: *Are you speaking of the exploration witnesses à la Gurevich [16]?*

Yes. We need such a postulate to forbid algorithms to work unrealistically[5]. It is as those of sequential and BSP algorithms. For the rest of the paper, we also need to define the notion of *coincidence*.

Definition 10 (Coincidence over terms) *Let A be a* MULTI-BSP *algorithm and Θ be a set of terms of $\mathcal{L}(A)$. We say that two states U and V of A* coincide over Θ *if $U \stackrel{\circ}{=} V$ and for every $\langle X|\overrightarrow{U}\rangle \trianglelefteqslant U$ and $\langle Y|\overrightarrow{V}\rangle = \mathbf{pos}\Big(\langle X|\overrightarrow{U}\rangle, U; V\Big)$ we have for every $\theta \in \Theta$ that $\overline{\theta}^X = \overline{\theta}^Y$.*

where $\overline{\theta}^X$ denotes the interpretation [20] (p. 7) of the term θ in the structure (memory) X.

Postulate 3 (Bounded exploration) *For every* MULTI-BSP *algorithm A there exists a finite set $\Theta(A)$ of terms (closed by subterms) such that for every state U and V, if they coincide over $\Theta(A)$ then for every pair of subtrees S and W such that $S \trianglelefteqslant U$ and $W = \mathbf{pos}(S, U; V)$ we have $\Delta(A, U)_S = \Delta(A, V)_W$.*

$\Theta(A)$ is called the *exploration witness* of A and contains at least the boolean symbol **true**. The interpretations of the terms in $\Theta(A)$ are called the *critical elements*.

Q20: *What can we do with these elements?*

We can prove that every value in an update is a critical element.

Lemma 1 (Critical Elements) *Let A be a* MULTI-BSP *algorithm and U be a state. For every subtree $S \trianglelefteqslant U$, if $(f, \overrightarrow{a}, b) \in \Delta(A, U)_S$ then $\overrightarrow{a}, b$ are critical elements.*

Proof (sketch). The proof (by contradiction and by using a fresh variable in the witness) is similar to the ones of [16, 20]. □

That implies that for every step of the computation, for a given memory (structure), only a bounded (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) *Let A be a* MULTI-BSP *algorithm and U be a state. For every subtree $S \trianglelefteqslant U$, card$(\Delta(A, U)_S)$ (cardinality) is bound.*

Proof (sketch). The proof is similar to the one in [16, 20]. □

2.2.4 MULTI-BSP algorithm nested computations

Q21: *That is thus close to the original work of [16]. But, I currently do not see anything about the nested superstep organization of* MULTI-BSP *algorithms.*

This is our last postulate. The communication between local memories occurs only during a communication phase. To do so, a MULTI-BSP algorithm A will use two functions $\mathbf{compu}_A$ and $\mathbf{comm}_A$ whether, during the execution of A, a branch runs computations or communications (it is *working*). The function $\mathbf{comm}_A$ allows the communication between the memories of a component. So we need first to define the set of leaves (*resp.* nodes) for the $\mathbf{compu}_A$ (*resp.* $\mathbf{comm}_A$) function.

Definition 11 (All leaves and nodes) *The (potentially infinite) sets of the single memories (from leaves) and tuples of memories (from nodes) of a* MULTI-BSP *algorithm A are defined as:*

$$Leaves(A) \stackrel{\text{def}}{=} \bigcup_{T \in S(A)} \{X \mid \langle X \rangle \trianglelefteqslant T\}$$

$$Nodes(A) \stackrel{\text{def}}{=} \bigcup_{T \in S(A)} \left\{(X_0, X_1, \ldots, X_p) \;\middle|\; p > 0 \wedge \left\langle X_0 \middle| \langle X_1|\overrightarrow{T_1}\rangle, \ldots, \langle X_p|\overrightarrow{T_p}\rangle \right\rangle \trianglelefteqslant T\right\}$$

where $p + 1$ is called the arity of the tuple.

[5] We insist that such witnesses are not used to prove that the executions are finite; It is not a kind of measurement as in the Hoare logic.

To put it simply, we recursively add all the memory components (from nodes and leaves) of all the subtrees of all the states (trees) of a MULTI-BSP algorithm A. We can now give the last postulate.

Postulate 4 (Nested superstep phases) *For every* MULTI-BSP *algorithm A there exists a pair of functions $\boldsymbol{compu}_A : Leaves(A) \to Leaves(A)$ and $\boldsymbol{comm}_A : Nodes(A) \to Nodes(A)$ preserving the arity, such that by induction on every state $\langle X|\overrightarrow{T}\rangle$:*

$$\begin{array}{rcl} \tau_A\langle X\rangle & = & \langle \boldsymbol{compu}_A(X)\rangle \\ \tau_A\left\langle X_0\middle|\langle X_1|\overrightarrow{T_1}\rangle,\ldots,\langle X_p|\overrightarrow{T_p}\rangle\right\rangle & = & \begin{cases} \left\langle X_0\middle|\tau_A\langle X_1|\overrightarrow{T_1}\rangle,\ldots,\tau_A\langle X_p|\overrightarrow{T_p}\rangle\right\rangle \\ \quad \textit{if } \exists 1 \le i \le p, \tau_A\langle X_i|\overrightarrow{T_i}\rangle \neq \langle X_i|\overrightarrow{T_i}\rangle \\ \left\langle X'_0\middle|\langle X'_1|\overrightarrow{T_1}\rangle,\ldots,\langle X'_p|\overrightarrow{T_p}\rangle\right\rangle \\ \quad \textit{otherwise} \end{cases} \end{array}$$

where $p > 0$ and $\boldsymbol{comm}_A(X_0, X_1, \ldots, X_p) = \left(X'_0, X'_1, \ldots, X'_p\right)$, which performs the communications of the memories of a component.

From a certain point of view, $\mathbf{compu}_A$ is the call of a single processor's operation whereas $\mathbf{comm}_A$ is a single collective data transfer.

Q22: *Some explanations are needed.*

Of course. The function τ_A is close to the BSP one of [20] except that it is now defined recursively (on each level) and thus for each component. τ_A performs computations on leaves only and communications within the upward memory components otherwise. Notice that τ_A induces that the function of communication modifies the structures if a shifting (upward or downward the work in the tree) of level is needed. As intended for the MULTI-BSP model of execution, for a given level, such an evaluation is feasible only if there is no work in the lower levels (they finish their supersteps). Otherwise, nothing is done; That forbids concurrent memory manipulations from different levels.

Q23: *I do not see the* BSP*'s sequences of computations and communications as in your previous postulates about* BSP *[20]. Is this normal?*

For a node controlling only $p > 0$ leaves, we have:

$$\tau_A\left\langle X_0\middle|\langle X_1\rangle,\ldots,\langle X_p\rangle\right\rangle = \begin{cases} \left\langle X_0\middle|\langle \mathbf{compu}_A(X_1)\rangle,\ldots,\langle \mathbf{compu}_A(X_p)\rangle\right\rangle \\ \quad \text{if } \exists 1 \le i \le p, \mathbf{compu}_A(X_i) \neq X_i \\ \left\langle X'_0\middle|\langle X'_1\rangle,\ldots,\langle X'_p\rangle\right\rangle \\ \quad \text{otherwise, where } p > 0 \text{ and} \\ \quad \mathbf{comm}_A(X_0, X_1, \ldots, X_p) = \left(X'_0, X'_1, \ldots, X'_p\right) \end{cases}$$

which corresponds closely to the fourth postulate [20] for BSP algorithms (the sequence of supersteps), especially when considering a BSP machine as a MULTI-BSP machine with a single node of p processors.

Q24: *It is strange that there is no computation on nodes because most communication patterns need a little computation. How do you do that?*

It is the responsibility of the communication function to (implicitly) perform such computations because it is *oracular*. In "practice" [1, 30], the needed computations that are done in the upper memories focus on data management and one selected (by the implementation/architecture) processor of a branch can do such a work. We will discuss this issue later.

Q25: *A bit of cheating but let us move on. What is thus a* MULTI-BSP *algorithm?*

A MULTI-BSP *algorithm* is an object verifying these four postulates, and we denote by $\text{ALGO}_{\text{MULTI}}$ the (infinite) set of MULTI-BSP algorithms.

Q26: *And now, what can we deduce from all this?*

Now, we can prove that the set of MULTI-BSP algorithms satisfies a strict separation of the phases. To do so, we first prove some lemmas about the computation and communication *phases*. So for convenience, we will define such phases for branches.

Definition 12 (Computation and Communication phases) *A leaf $\langle X\rangle$ is said to be in a **computation phase** if $\boldsymbol{compu}_A(X) \neq X$. A node $\left\langle X_0\middle|\langle X_1|\overrightarrow{T_1}\rangle,\ldots,\langle X_p|\overrightarrow{T_p}\rangle\right\rangle$ is said to be in a **computation phase** if there exists $1 \le i \le p$ such that $\tau_A\langle X_i|\overrightarrow{T_i}\rangle \neq \langle X_i|\overrightarrow{T_i}\rangle$. Otherwise, the node is said to be in a **communication phase**.*

This requires some remarks. Firstly, we did not specify the function $\mathbf{comm}_A$ in order to be generic (oracular spirit). Secondly, 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. That can also happen between different nodes during the communication phases: one branch (sub-component) needs more computations/communications than other branches. Thirdly we do not distinguish a communication step from a synchronization step, because both are globally done, at a given level, by the function $\mathbf{comm}_A$. A *superstep* is a manipulation of the memories of each component until the sub-components finish their own supersteps.

Lemma 3 (Computation phases are closed by subtree) *Let A be a* MULTI-BSP *algorithm, and T be a state. If a subtree $S \trianglelefteq T$ is in a computation phase, then T is also in a computation phase.*

Proof. The proof is done by induction on the subtree relation. If $S = T$ then T is in a computation phase by hypothesis. If $S \trianglelefteq T_i$ with $T = \langle X | T_1, \ldots, T_i, \ldots, T_p \rangle$ and $p > 0$, then by induction hypothesis $T_i = \langle X_i | U_1, \ldots, U_q \rangle$ is in a computation phase. So, either $q = 0$ (it is a leaf) and $\mathbf{compu}_A(X_i) \neq X_i$, or $\tau_A\langle X_i | U_1, \ldots, U_q \rangle = \langle X_i | \tau_A(U_1), \ldots, \tau_A(U_q) \rangle$ and there exists $1 \leq j \leq q$ such that $\tau_A(U_j) \neq U_j$. In any case, $\tau_A(T_i) \neq T_i$, so T is also in a computation phase. □

Notice that in general we do not have $\tau_A(T)_S = \tau_A(S)$. For instance, two leaves $\langle X_1 \rangle$ and $\langle X_2 \rangle$ may have finished their computations so we have $\tau_A\langle X_1 \rangle = \langle X_1 \rangle$, but they may be updated by the communication function so $\tau_A\langle X | \langle X_1 \rangle, \langle X_2 \rangle \rangle_{X_1} \neq X_1$. But this is the case for subtrees in a computation phase.

Lemma 4 (Computing states) *Let A be a* MULTI-BSP *algorithm, and T be a state. For every leaf $S \trianglelefteq T$ in a computation phase: $\tau_A(T)_S = \tau_A(S)$.*

Proof. The proof is done by induction on T. If $T = \langle X \rangle$ is a leaf, then the only subtree is T itself. Let us assume now that $T = \langle X | T_1, \ldots, T_p \rangle$ is a node. If $S = T$ then by definition of $\mathbf{pos}$ we have $\tau_A(T)_S = \tau_A(T) = \tau_A(S)$. If there exists $1 \leq i \leq p$ such that $S \trianglelefteq T_i$ then by induction hypothesis $\tau_A(T_i)_S = \tau_A(S)$. Because the subtree $S \trianglelefteq T$ is in a computation phase, by Lemma 3, so do T. So $\tau_A(T) = \langle X | \tau_A(T_1), \ldots, \tau_A(T_p) \rangle$ and thus, by definition of $\mathbf{pos}$, $\tau_A(T)_S = \tau_A(T_i)_S$. Hence $\tau_A(T)_S = \tau_A(S)$. □

Corollary 1 (Computing leaves) *Let A be a* MULTI-BSP *algorithm, and T be a state. For every leaf $\langle X \rangle \trianglelefteq T$ in a computation phase: $\Delta(A, T)_X = \Delta(A, X)$.*

Proof. By definition, $\Delta(A, T)_X = \tau_A(T)_X \ominus X$. Because $\langle X \rangle$ is in a computation phase, by Lemma 4, $\tau_A(T)_X = \tau_A(X)$. So $\Delta(A, T)_X = \Delta(A, X)$. □

Thus, a leaf in a computation phase ignores all the processors above.

Technical Lemma 1 (Properties of the computation) *For every* MULTI-BSP *algorithm A, $\boldsymbol{compu}_A$ preserves the universes and commutes with isomorphisms.*

Proof (sketch). The proof is similar to the one of [20]. □

Lemma 5 (Computing states are closed by tree-isomorphisms) *Let A be a* MULTI-BSP *algorithm, Φ be a tree-isomorphism and $T \triangleq \Phi$ be a state. If T is in a computation phase, then $\Phi(T)$ is in a computation phase too.*

Proof. If $T = \langle X \rangle$ is a leaf, we assume by contradiction that $\Phi\langle X \rangle$ is not in a computation phase, so $\tau_A(\Phi\langle X \rangle) = \Phi\langle X \rangle$. By the second postulate, $\tau_A(\Phi\langle X \rangle) = \Phi(\tau_A\langle X \rangle)$. So $\Phi(\tau_A\langle X \rangle) = \Phi\langle X \rangle$, and by applying Φ^{-1} on both sides we have $\tau_A\langle X \rangle = \langle X \rangle$ that contradicts that $\langle X \rangle$ is in a computation phase. If $T = \langle X | T_1, \ldots, T_p \rangle$ is a node, we assume by contradiction that $\Phi(T) = \langle \varphi(X) | \Phi_1(T_1), \ldots, \Phi_p(T_p) \rangle$ is not in a computation phase, so for every $1 \leq i \leq p$, $\tau_A(\Phi_i(T_i)) = \Phi_i(T_i)$. By the second postulate, $\tau_A(\Phi_i(T_i)) = \Phi_i(\tau_A(T_i))$, so by applying Φ_i^{-1} we obtain that for every $1 \leq i \leq p$, $\tau_A(T_i) = T_i$, which contradicts that T is in a computation phase.

□

Technical Lemma 2 (Shifting of level) *Let A be a* MULTI-BSP *algorithm, and T be a state. If a subtree $S \trianglelefteq T$ has finished its communication phase, then: (1) if $S = \langle X | T_1, \ldots, T_p \rangle$ then the next phase (communication or computation) will be on the T_i ; exclusise or (2) if exist $U = \langle X | T_1, \ldots, S, \ldots, T_p \rangle$ then the next phase (communication or computation) will be on U.*

Proof (sketch). The proof is done by induction on T and by case using the fourth postulat: if S is a leaf (end of the computation phase) then U is a node (the level is "up"); otherwise, S is a node, and depending of $\mathbf{comm}_A$, either one of the T_i has been modified such that a new phase helds on T_i (the level is "down") or U has beeen modified such the next communication phase helds on U. □

By abuse of language, we will say that the shifting of level (where the next phases take place in the tree, that is the flow of execution is changing of level) can only be directly downward (resp. upward) and we say "*down*" (resp. "*up*"). We will say that the level is ± 1 depending if it is "down" or "up". Notice that we did not assume in the fourth postulate that the communication function commutes with tree-isomorphisms, because this is a corollary (proved as in [20]) of the second postulate and Lemma 5.

Corollary 2 (Properties of communication) *For every* MULTI-BSP *algorithm A and for every state in a communication phase, $\boldsymbol{comm}_A$ preserves the universes and commutes with tree-isomorphisms.*

Proof. Let $T = \left\langle X_0 \middle| \langle X_1 | \overrightarrow{T_1} \rangle, \ldots, \langle X_p | \overrightarrow{T_p} \rangle \right\rangle$ be a node not in a computation phase, so

$$\tau_A(T) = \left\langle X_0' \middle| \langle X_1' | \overrightarrow{T_1} \rangle, \ldots, \langle X_p' | \overrightarrow{T_p} \rangle \right\rangle,$$

where $\mathbf{comm}_A(X_0, \ldots, X_p) = \left(X_0', \ldots, X_p'\right)$. By the second postulate, τ_A preserves the universes, so does $\mathbf{comm}_A$. Let Φ be a tree-isomorphism. Because T is not in a computation phase, by Lemma 5 (with Φ^{-1}), $\Phi(T)$ is not in a computation phase either, so

$$\tau_A(\Phi(T)) = \left\langle \varphi_0(X_0)' \middle| \langle \varphi_1(X_1)' | \overrightarrow{\Phi_1(T_1)} \rangle, \ldots, \langle \varphi_p(X_p)' | \overrightarrow{\Phi_p(T_p)} \rangle \right\rangle,$$

where $\mathbf{comm}_A(\varphi_0(X_0), \ldots, \varphi_p(X_p)) = \left(\varphi_0(X_0)', \ldots, \varphi_p(X_p)'\right)$. Moreover, by the second postulate, $\tau_A(\Phi(T)) = \Phi(\tau_A(T)) = \Phi\left\langle X_0' \middle| \langle X_1' | \overrightarrow{T_1} \rangle, \ldots, \langle X_p' | \overrightarrow{T_p} \rangle \right\rangle$, so for every $0 \leq i \leq p$, $\varphi_i(X_i)' = \varphi_i(X_i')$, and thus $\mathbf{comm}_A$ commutes with tree-isomorphisms. □

Q27: *Could you justify that your work extends the one about* BSP*?*

Our postulates are a "natural" extension of those in [16, 20].

Proposition 1 (Hierarchy of models) *A* BSP *algorithm with a unique processor ($p = 1$) is a sequential algorithm. A* MULTI-BSP *algorithm with a single node controlling only leaves is a* BSP *algorithm. Thus:*

$$\text{ALGO}_{\text{SEQ}} \subseteq \text{ALGO}_{\text{BSP}} \subseteq \text{ALGO}_{\text{MULTI}}$$

Proof (sketch). $\text{ALGO}_{\text{SEQ}} \subseteq \text{ALGO}_{\text{BSP}}$ is proven in [20]. Regarding the second inclusion, by ignoring the node and considering the leaves as a tuple, we obtain the same postulates (see Question 23 p.11) as in [20]. The hypothesis on **id** in the second postulate and on the subtrees in the third postulate correspond to the location in the tuples of [20]. □

Q28: *That is not a surprise and I am glad that this is the case. And what about the realization of a function of communication ($\boldsymbol{comm}_A$) as in [20]?*

This function "simulates" the BSPLIB's DRMA routines within the ASM framework and we exhibited its exploration witness. That allows a constructive result for BSP because any algorithm using such routines can be truly analysis. This construction matters since BSPLIB is a standard and common library. Sadly, such a library currently does not exist for MULTI-BSP and so we cannot do this work yet right now.

Q29: *So for now, you have just an abstract axiomatization. What is the link with actual programming?*

We will now follow [16, 20] to provide an operational point of view with the notion of Abstract State Machines (ASMs). And then, we will provide an imperative MULTI-BSP programming core-language and exhibit formally their links.

3 Imperative characterization of MULTI-BSP algorithms

3.1 $\text{ASM}_{\text{MULTI}}$ and the MULTI-BSP algorithms

As for the sequential case [16] and the BSP case [20], we first define the $\text{ASM}_{\text{MULTI}}$ machines and then we prove that they "capture" [15] the MULTI-BSP algorithms, or in other words, that MULTI-BSP algorithms and $\text{ASM}_{\text{MULTI}}$ are the same mathematical objects.

3.1.1 Definition and operational semantics of $\text{ASM}_{\text{MULTI}}$

Q30: *For the computing phases, are you using standard sequential* ASM*s?*

Indeed. Sequential ASMs have been defined in [15] to give an operational model to the sequential (small-steps and discrete time) algorithms. We also used them in [20] for the BSP algorithms. In fact, an ASM program contains the full algorithm by giving, at each step of the computation, what needs to be modified in the structures (memories). We refer to [20] (p. 14) for a formal definition of ASM programs Π and the sequential operational semantics $\Delta(\Pi, X)$ (p. 14). We also used such multisets of updates ([20] p. 15) and the transition function τ_Π induced by Π on leaves. We assume that $\text{ASM}_{\text{MULTI}}$ programs work in a SPMD-like way, which means that at each step of the computation, the $\text{ASM}_{\text{MULTI}}$ program Π is executed individually on each computing unit.

Definition 13 *An* $\text{ASM}_{\text{MULTI}}$ *machine* M *is a triplet* $(S(M), I(M), \tau_M)$ *such that:*

1. $S(M)$ *is a set of trees of structures with the same finite signature* $\mathcal{L}(M)$ *(containing at least the booleans, the equality and a uniquely interpreted* **id** *symbol);* $S(M)$ *and* $I(M) \subseteq S(M)$ *are closed by tree-isomorphisms and subtrees;*

2. *There exists an* ASM *program* Π *and a communication function* $\boldsymbol{comm}_M : Nodes(M) \to Nodes(M)$ *such that* $\tau_M : S(M) \mapsto S(M)$ *is defined by induction as:*

$$\begin{array}{rcl} \tau_M\langle X\rangle & = & \langle \tau_\Pi(X)\rangle \text{ (call to the } \Pi \text{ program)} \\ \tau_M\left\langle X_0 \middle| \langle X_1|\overrightarrow{T_1}\rangle, \ldots, \langle X_p|\overrightarrow{T_p}\rangle \right\rangle & = & \begin{cases} \left\langle X_0 \middle| \tau_M\langle X_1|\overrightarrow{T_1}\rangle, \ldots, \tau_M\langle X_p|\overrightarrow{T_p}\rangle \right\rangle \\ \quad \text{if } \exists 1 \le i \le p, \tau_M\langle X_i|\overrightarrow{T_i}\rangle \neq \langle X_i|\overrightarrow{T_i}\rangle \\ \left\langle X'_0 \middle| \langle X'_1|\overrightarrow{T_1}\rangle, \ldots, \langle X'_p|\overrightarrow{T_p}\rangle \right\rangle \\ \quad \text{otherwise} \end{cases} \end{array}$$

   where $p > 0$ *and* $\boldsymbol{comm}_M(X_0, X_1, \ldots, X_p) = \left(X'_0, X'_1, \ldots, X'_p\right)$.

3. $\boldsymbol{comm}_M$ *verifies that:*

   (1) For every state $\left\langle X_0 \middle| \langle X_1|\overrightarrow{T_1}\rangle, \ldots, \langle X_p|\overrightarrow{T_p}\rangle \right\rangle$ *in communication phase, i.e. such that* $\forall 1 \le i \le p$, $\tau_M\langle X_i|\overrightarrow{T_i}\rangle = \langle X_i|\overrightarrow{T_i}\rangle$, $\boldsymbol{comm}_M$ *preserves the universes and the arity, and commutes with tree-isomorphisms;*

   (2) There exists a finite set of terms $\Theta(\boldsymbol{comm}_M)$ *such that for every state* U *and* V *in communication phase, if they coincide over* $\Theta(\boldsymbol{comm}_M)$ *then for every subtrees* $S \trianglelefteq U$ *and* $W = \mathbf{pos}(S, U; V)$ *we have* $\Delta(M, U)_S = \Delta(M, V)_W$.

We denote by $\text{ASM}_{\text{MULTI}}$ the set of such machines. As for the MULTI-BSP algorithms, a state T (a tree) is said *final* if $\tau_M(T) = T$. If U is a state of an $\text{ASM}_{\text{MULTI}}$ machine M with an ASM program Π, and $X \trianglelefteq U$ is a leaf, then $\Delta(\Pi, U)_X$ will denote $\Delta(M, U)_X$ without ambiguity. The last two conditions about the communication function may seem arbitrary, but they are required to ensure that it is not a kind of *magic* device, and that it performs *data-exchanges* step-by-step.

Q31: *If I understand well, some components may wait for lower or upper level components to finish their own supersteps. How do you manage that?*

It is the role of the machine to do that. A solution, with the help of a fresh boolean, is to let the machine force the computing units to do empty steps (**par endpar** *i.e.* no update). That corresponds to a busy-looping (spinning). Another solution is considering that it is an implementation issue of the machine: a variety of system calls can block processes (locks, mutex, *etc.*). Notice that the machine is also responsive to shifting of level if necessary.

3.1.2 $\text{ASM}_{\text{MULTI}}$ captures MULTI-BSP algorithms

Q32: *The semantics of your* ASM*s is close to the superstep postulate and that is not a surprise since the* ASM *thesis stipulates that* ASM*s and algorithms are the same objects. In the* MULTI-BSP *case, how are you going to redo that?*

In the same way as for the BSP algorithms [20]. We first used the fact that the transitions of a

processor in a computation step can be captured by an ASM program (Lemma 6). Then, we prove in Corollary 3 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 that $\text{ASM}_{\text{MULTI}}$ captures the computation steps of MULTI-BSP algorithms. Finally, we prove that $\text{ALGO}_{\text{BSP}} = \text{ASM}_{\text{BSP}}$.

Lemma 6 (Each Local Transition is Captured by an ASM) *Let A be a MULTI-BSP algorithm. For every state U and leaf $X \trianglelefteq U$ in a computation phase, there exists an ASM program Π_X such that $Read(\Pi_X) \subseteq \Theta(A)$ and $\Delta(\Pi_X, X) = \Delta(A, X)$.*

Proof (sketch). The proof is similar as in the sequential case [16, 19], and uses Lemma 1 for critical elements, and Lemma 2 to obtain a finite program Π_X. □

As in [15, 20], to narrow the number of relevant states we use the *finiteness* of the exploration witness $\Theta(A)$. For every memory X, we denote by E_X the equivalence relation on pairs (θ_1, θ_2) of terms in $\Theta(A)$ defined by:

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

Corollary 3 (Syntactically equivalent memories) *Let A be a MULTI-BSP algorithm, U and V be two states, and $X \trianglelefteq U$ and $Y \trianglelefteq V$ be two leaves. If X and Y are in a computation phase and $E_X = E_Y$ then:*

$$\Delta(\Pi_X, V)_Y = \Delta(A, V)_Y$$

Proof (sketch). By definition of the operational semantics of $\text{ASM}_{\text{MULTI}}$, $\Delta(\Pi_X, U)_X = \Delta(\Pi_X, X)$ and $\Delta(\Pi_X, V)_Y = \Delta(\Pi_X, Y)$. Because X and Y are in a computation phase, by Corollary 1 p.12, $\Delta(A, U)_X = \Delta(A, X)$ and $\Delta(A, V)_Y = \Delta(A, Y)$. Then the proof is identical to the sequential case [16]. □

Proposition 2 ($\text{ASM}_{\text{MULTI}}$ capture Computations) *For every MULTI-BSP algorithm A, there exists an ASM program Π_A such that for every state U and leaf $X \trianglelefteq U$ in a computation phase:*

$$\Delta(\Pi_A, U)_X = \Delta(A, U)_X$$

Proof (sketch). The proof is similar to [16, 20] by using Corollary 3 to obtain subprograms and the finite exploration witness in the third Postulate to obtain a finite global program. □

Theorem 1 $\text{ALGO}_{\text{MULTI}} = \text{ASM}_{\text{MULTI}}$

Proof (sketch). The proof is similar to [16, 20] and is made by mutual inclusion. 1) A MULTI-BSP algorithm A is an $\text{ASM}_{\text{MULTI}}$ machine with the same (initial and standard) states and communication function, and using the program Π_A from Proposition 2 during the computation phases (and using technical Lemma 1, such computations preserve the universes and commute with tree-isomorphisms). Moreover, according to Corollary 2 the communication function preserves the universes and commutes with tree-isomorphisms. Finally, the other properties from Definition 13 are verified according to the postulates. 2) An $\text{ASM}_{\text{MULTI}}$ machine M with an ASM program Π verifies the fourth postulate, in particular its exploration witness is the closure by subterms of $\Theta(\Pi) \cup \Theta(\mathbf{comm}_M)$, and thus is a MULTI-BSP algorithm. □

Q33: *It is a little long. Is this really useful?*

Yes. But to our purpose, ASMs are *intermediary* objects only. Algorithmic completeness of programming languages is the main goal. How communications are organized (in a nested manner) is defined in both postulates and $\text{ASM}_{\text{MULTI}}$ in a similar way so that does not modify how computations are performed. It is the communication function that decides how to shift levels and how terms are sent (depending on the content of the memories/structures which, of course, have been modified by the leaves during the execution of the program Π itself).

3.2 Programming MULTI-BSP algorithms and algorithmical completeness

Q34: *That is a curious computation model to let an abstract function manage the level and how to send data. Any programmer's code would rather contain explicit shifting of level and letting the function of communication order the data. Can you elaborate on that?*

Indeed. The ASM program must modify the structures in order to initiate a shift of levels. Furthermore, the ASM execution flow is close to "*low level*" assembly codes and thus lacks *control flow* structures for programming algorithms in as usual. So most programmers naturally want a finer control and prefer using a programming language to get their algorithms implemented. We now present an SPMD extension of the standard IMP core-language. The IMP programs are common *sequences of commands*, which are standard *control flow statements* (conditionals and unbounded loop) or *assignments* (sequential updates), so this programming language can be seen as minimal.

3.2.1 Semantics of a core imperative MULTI-BSP language

Q35: *Why not directly choose a mainstream language such as* C *or* JAVA*?*

There are too many *constructions* on such languages and thus when doing proofs on them, some cases may be missed. A theorem prover such as COQ can help you to do that but that is not the goal of this work. A core language is sufficient because we can apply the following results to every mainstream language with at minimal such control flows. Let us introduce you to $\text{IMP}_{\text{MULTI}}$, our core imperative language [19] (inspired by the library of [30]).

Definition 14 (Syntax)

$$\begin{array}{rrl} \textit{commands: } c & \stackrel{\text{def}}{=} & f(\theta_1, \ldots, \theta_\alpha) := \theta_0 \\ & & \mid \textbf{if } F \ \{P_1\} \textbf{ else } \{P_2\} \\ & & \mid \textbf{while } F \ \{P\} \\ & & \mid \textbf{comm} \mid \textbf{up} \mid \textbf{down} \\ \textit{programs : } P & \stackrel{\text{def}}{=} & \textbf{end} \mid c; P \end{array}$$

where F is a formula ([20] p. 7), f has arity α and θ_0, θ_1, …, θ_α are terms. **comm** is used to perform one step of communication whereas **up** and **down** are used to shift of level. To improve readability we do not write the **end** when there is no ambiguity.

Q36: *I imagine better primitives such as the use of synchronizing primitives (like* BSP*'s* `bsp_sync()`*) to control the shifts of levels (flow up and down) on the* MULTI-BSP *trees and communications between levels such as* `upward(value v)` *and* `downward(value v,int id)` *[30]. What about such primitives?*

These primitives are for programs that are executed at each level of the d-trees, even on nodes. Assuming that only "scattering and gathering" of values are allowed as "computations" on nodes, the function of communication $\mathbf{comm}_M$ can simulate them easily. Again, we use such an abstract function in order not to be restricted by a particular library. Take for example the `upward` routine. It allows us to upward values from the memories of a level i to the memory of i–1. In IMP_{BSP}, we should have a code such as:

$$\textbf{up};\ \textbf{while}\ \textit{notEmpty}(\text{buffer})\ \{\ \textbf{if}\ \textit{manage}(\textbf{id})\ \{\textbf{comm}_M;\}\ \}$$

where the $buffer$ contains the values to upward and *manage* is a function that determines if the leaf handles or not the node at the given level (being the "master" [30]). In our simulation, on each leaf, the $\text{ASM}_{\text{MULTI}}$ program will run the above code where the values from the buffers are communicated step-by-step by the $\mathbf{comm}_M$.

Q37: *And what about recursive calls on the d-trees of [2] or [30]?*

Both are interesting programming languages that both allow to clearly distinguish what is executed on the nodes or leaves and how to send data between levels. They can be simulated by the use of the **down**/**up** primitives. A stack can also be used to keep track of the recursive calls of [2].

Q38: *So, what is the formal operational semantics of such a core language?*

The operational semantics of the local computations is formalized by a state transition system [19], where a state of the system is a pair $P \star X$ of a program P and a memory (structure) X, and a transition $\succ$ is determined only by the head command and the current structure. Here two examples of rule [20] (p. 31) (where $\oplus$ stands for an update of structure and $\overline{\theta_\alpha}^X$ for an interpretation of a term θ in the structure X):

$$f(\theta_1,\dots,\theta_\alpha) := \theta_0; P \star X \quad \succ \quad P \star X \oplus (f, \overline{\theta_1}^X, \dots, \overline{\theta_\alpha}^X, \overline{\theta_0}^X)$$

$$\textbf{while } F\ \{P_1\}; P_2 \star X \quad \succ \quad P_1 \fatsemi \textbf{while } F\ \{P_1\}; P_2 \star X \qquad \text{if } \overline{F}^X = \textbf{true}$$

Other rules are described in [20] (p. 31) and are without surprise. It is easy to show that this transition system is deterministic. The local states without successors are $\textbf{sync}; P \star X$ (where **sync** is **comm** or **up** or **down**) and $\textbf{end} \star X$. We use the same notation as in [20] about $\succ_t$ (the succession of t steps) and $\Delta(P, X)$ (the succession of updates made by P on X).

Q39: *Standard. But for true* MULTI-BSP *programs?*

Firstly, for a tree T, we note $\vec{\widetilde{P}}_T$ (*resp.* $\vec{\widetilde{\imath}}_T$ and $\vec{\widetilde{T}}$) the tuple of programs $(P_1,\dots,P_\beta)$ (*resp.* level identifiers, structures) where each P_i (*resp.* $\imath_i$, X_i) is a program (*resp.* identifier, structure) of a leaf of the tree T in a standard breadth-first traversal (in order of the **id**). The operational semantics of $\text{IMP}_{\text{MULTI}}$ is also formalized by a state transition system, where a state of the system is a triple $\vec{\widetilde{\imath}}_T \star \vec{\widetilde{P}}_T \star T$ where T is a tree of structures. The level identifiers (that take values from 1 to d) will be used to handle in which level the program manipulates a node and so, in different branches of T, these identifiers could be different. In practice, a component can choose a *single representative* leaf to handle the computations needed by the communications (contrary to this semantics where all processors on the same branch have a code that runs and *manages* the communication of the node at a given nested-superstep). It is to notice that the **comm**, **down** and **up** are *synchronous* and *global* to all the computing units of the same branch (accordling to the level identifiers); So if one diverges the overall computing diverges and if one performs a routine different than those of others then the overall machine fails.

If for a leaf $\langle X\rangle$, $\vec{\widetilde{\imath}}_{\langle X\rangle} = (1)$, then $\vec{\widetilde{P}}_{\langle X\rangle}$ is said to be in a **computation phase**. For a node $\langle X|\vec{T}\rangle = T$, if $\vec{\widetilde{P}}_T = (P_1,\dots,P_\beta)$, we say that $\vec{\widetilde{P}}_T$ is also in a computation phase if $P_i \neq \textbf{sync}; P \star X$, where **sync** = **comm**, **up** or **down**. Otherwise the tree is in a communication phase. We define $\textbf{next}(\textbf{sync}; P) = P$ and $\textbf{next}(\textbf{end}) = \textbf{end}$. We also note $\overrightarrow{\textbf{next}}(\dots, P^i, \dots) = (\dots, \textbf{next}(P^i), \dots)$. We denote by $\vec{\succ}_s$ the reduction at node level s which is defined as $\vec{\succ}$ in [20] (p. 32) but where we are using $\succ^s$ instead of $\succ$ and $\succ^s$ is defined as follow:

$$\begin{array}{ll} P \star X \succ^s \tau_X(P) \star \tau_P(X) & \text{if } P \star X \text{ has a successor (without being an update) and } s \neq 1 \\ P \star X = \tau_X(P) \star \tau_P(X) & \text{otherwise} \end{array}$$

Definition 15 (A semantics machine for the MULTI-BSP **core-language)** *An* $\text{IMP}_{\text{MULTI}}$ *machine* M *is a quintuplet* $(S(M), I(M), P_{init}, \boldsymbol{comm}_M, \vec{\widetilde{\imath}}_{I(M)})$*:*

1. *$S(M)$ is a set of trees of structures with the same finite signature $\mathcal{L}(M)$ (with at least the booleans, the equality and a uniquely interpreted* **id** *symbol);*
2. *The initial states of the transition system have the form $\vec{\widetilde{\imath}}_T \star \vec{\widetilde{P}}_T \star T$, where $T \in I(M) \subseteq S(M)$ and $\vec{\widetilde{P}}_T = (P_{init}, \cdots, P_{init})$ (P_{init} on each leaf);*
3. *P_{init} is a program with terms from $\mathcal{L}(M)$;*
4. *$\boldsymbol{comm}_M : Nodes(M) \to Nodes(M)$ verifies (as in Definition 13) that:*
   - *(1) For every state $\vec{\widetilde{\imath}}_T \star \vec{\widetilde{P}}_T \star T$ such that $\vec{\widetilde{P}}$ is in a communication phase, $\boldsymbol{comm}_M$ preserves the universes and the arity (size of tuples), and commutes with tree-isomorphisms;*
   - *(2) There exists a finite set of terms $\Theta(\boldsymbol{comm}_M)$ such that for every state $\vec{\widetilde{\imath}}_T \star \vec{\widetilde{P}}_T \star T$ and $\vec{\widetilde{\jmath}}_U \star \vec{\widetilde{Q}}_U \star U$, if $\vec{\widetilde{P}}_U$ and $\vec{\widetilde{Q}}_U$ are in a communication phase, $T \triangleq U$ and T and U coincide over $\Theta(\boldsymbol{comm}_M)$ then for each $S = \left\langle X_0 \middle| \langle X_1|\vec{T_1}\rangle, \cdots, \langle X_p|\vec{T_p}\rangle \right\rangle \trianglelefteq T$ and $V = \textbf{pos}(S, T; U) = \left\langle Y_0 \middle| \langle Y_1|\vec{U_1}\rangle, \cdots, \langle Y_p|\vec{U_p}\rangle \right\rangle$ we have $\boldsymbol{comm}_M(X_0, X_1, \cdots, X_p) \ominus (X_0, X_1, \cdots, X_p) = \boldsymbol{comm}_M(Y_0, Y_1, \cdots, Y_p) \ominus (Y_0, Y_1, \cdots, Y_p)$, where $\ominus$ is defined in [20] (p. 25).*

Q40: *Are they limitations on the $\vec{\widetilde{\imath}}_T$? They look like any identifier and so processors can do whatever.*

Indeed, but the operational semantics will forget to do whatever. If there is not computation from

an initial state, it is just an unless algorithm.

Q41: *And why such a definition of* $\boldsymbol{comm}_M$*?*

As explained in [20], such limitations are used to forbid this function to do whatever, especially an unbounded reduction on the structures.

We denote by $level(S,T)$ at which level S is in T if $S \lessdot T$. For every $T \triangleq U$ and $s \in 1 \dots d$ (a level identifier, where d is the depth), the operational semantics $\overset{\rightsquigarrow}{\succ}$ of $\text{IMP}_{\text{MULTI}}$ denoted $\overset{\rightsquigarrow}{I}_T \star \overset{\rightsquigarrow}{P}_T \star T \overset{\rightsquigarrow}{\succ} \overset{\rightsquigarrow}{J}_U \star \overset{\rightsquigarrow}{Q}_U \star U$ is defined with three cases:

Communication case: $S = \left\langle X_0 \middle| \langle X_1|\overrightarrow{T_1}\rangle, \dots, \langle X_p|\overrightarrow{T_p}\rangle \right\rangle \lessdot T$ where $\overset{\rightsquigarrow}{I}_S = (s, \cdots, s)$ and $level(S,T) = s$ and $\overset{\rightsquigarrow}{P}_S = (P_1, \cdots, P_\beta)$ (where for all i $P_i = \textbf{comm}; P$), then $\overset{\rightsquigarrow}{I}_S = \overset{\rightsquigarrow}{J}_S$ and $\overset{\rightsquigarrow}{Q}_V = \overrightarrow{\textbf{next}}(\overset{\rightsquigarrow}{P}_S)$ and if $V = \left\langle Y_0 \middle| \langle Y_1|\overrightarrow{U_1}\rangle, \dots, \langle Y_p|\overrightarrow{U_p}\rangle \right\rangle = \textbf{pos}(T, S; U)$ then $U_1 = T_1 \cdots U_p = T_p$ and $(Y_0, Y_1, \cdots, Y_p) = \textbf{comm}_M(X_0, X_1, \cdots, X_p)$;

Shifting case: $S = \langle X_0|\overrightarrow{T}\rangle \lessdot T$ where $\overset{\rightsquigarrow}{I}_S = (s, \cdots, s)$ and $level(S,T) = s$ and $\overset{\rightsquigarrow}{P}_S = (P_1, \cdots, P_\beta)$ (where for all i $P_i = \textbf{up}; P$ or $P_i = \textbf{down}; P$), then $\overset{\rightsquigarrow}{Q}_V = \overrightarrow{\textbf{next}}(\overset{\rightsquigarrow}{P}_S)$ and if $\langle V|\overrightarrow{U}\rangle = \textbf{pos}(T, S; U)$ then $\overrightarrow{U} = \overrightarrow{T}$ and $\overset{\rightsquigarrow}{J}_S = \overset{\rightsquigarrow}{I}_S \pm 1$ (for each s_i of $\overset{\rightsquigarrow}{I}_S$) according that **down** (*resp.* **up**) is the next command and if $s \neq 1$ (*resp.* $s \neq d$);

Computation case: If $S = \langle X \rangle$ then $\overset{\rightsquigarrow}{I}_S = \overset{\rightsquigarrow}{J}_S = (1)$ and if $\langle Y \rangle = \textbf{pos}(T, B; U)$ and if $\overset{\rightsquigarrow}{P}_S = P$ then $Y = \tau_P(X)$ and $\overset{\rightsquigarrow}{Q}_{\langle Y \rangle} = \tau_X(P)$. If $S = \langle X_0|\overrightarrow{T}\rangle \lessdot T$ where $\overset{\rightsquigarrow}{I}_S = (s, \cdots, s)$ and $level(S,T) = s$ and $\overset{\rightsquigarrow}{P}_S = (P_1, \cdots, P_\beta)$ (where for all i $P_i \neq \textbf{sync}; P$) then $\overset{\rightsquigarrow}{P}_S \star \overset{\rightsquigarrow}{S} \overset{\rightsquigarrow}{\succ}_s \overset{\rightsquigarrow}{Q}_V \star \overset{\rightsquigarrow}{V}$ if $V = \textbf{pos}(T, S; U)$.

A state $\overset{\rightsquigarrow}{I}_T \star \overset{\rightsquigarrow}{P}_T \star T$ is said *final* if all components are in a communication phase (performing nothing). We say that, on local memories (tree) T and initial level identifier $\overset{\rightsquigarrow}{I}_T$, the (initial) program P_{init} *terminates globally* if there exists t, $\overrightarrow{P'}$ and U such that $\overset{\rightsquigarrow}{I}_T \star \overrightarrow{P_{init}} \star X \overset{\rightsquigarrow}{\succ}_t \overset{\rightsquigarrow}{I'}_U \star \overrightarrow{P'} \star U$, where $U \triangleq T$ and $\overrightarrow{P'} = \overset{\rightsquigarrow}{P'}_U$ and it is a final state. We denote by $\text{time}(P_{init}, I, X)$ the smallest of those t, by assuming (as in the Definition 2.2.1) potential infinite duration if P_{init} does not terminate globally (there is an infinite number of nested supersteps or during one of them, at least, the computations of one processor diverges). Moreover, because the reached state is a fixpoint, U is unique, and will be denoted by $\overrightarrow{P_{init}}(I, X)$.

Q42: *Why select all the leaves of a branch to perform the node communications rather than use a single leaf (a master) as in [2] or [30]?*

To be sure that they are all at the right level during communication. That is thus only for semantics convenience. Which leaf is the "master" and truly manipulated the memory is implementation dependent. Moreover, that allows us to formally specify when the "waiting" is not implemented with system calls but with a busy-looping.

Q43: *So, are there computations on the nodes in your model?*

As explained before, in our version of the MULTI-BSP model, nodes are memories only and so yes, only communications can occur. But how these communications are managed (and what are they doing) is thus the responsibility of the $\textbf{comm}_M$ function. For our core imperative language, to allow executing an unbound number of calls of this function and choosing how to shift levels, "while" and "if" statements are possible on nodes but no assignment (local modification of the structure/memory which can be considered as a "distant writing").

Q44: *You forbid local assignments (on a leaf) when it is indicated that it is a node reduction (communication).*

Indeed. Without that, during any nested superstep, a leaf can modify its own memory and thus not respecting the MULTI-BSP model of execution.

Q45: *But then it is impossible for the* $\boldsymbol{comm}_M$ *function to be well executed. For example, if* $\boldsymbol{comm}_M$ *is inside a loop in order to send a complex data in several steps (e.g partitioned a list), then there is no way to write the current memory of the leaf to manage this number of steps. And it is unrealistic to use only the node memory to perform such a work. Could you elaborate on that?*

You are right. By leaving a control to the programmer over the communications, it is necessary to "execute" some codes. Because leaf updates are not allowed, we need to allow the function of communications to perform some modifications on the memories of the leaves. But these updates must not allow the processors to enter again a local computing phase because that would break the MULTI-BSP

execution model. These modifications must be used for the communications only. To do that we first note $descendants(T) = (X_1, \cdots, X_\beta)$ all the leaf structures (in a breadth-first order of the **id**) of a tree T which is not a parent of leaves. If $\forall X_i \in descendants(T)\ \tau_\Pi(X_i) = X_i$ then every computing unit has finished its computation steps. Then, we modified $\mathbf{comm}_M$ so that now $(Y_0, Y_1, \cdots, Y_p, descendants(U)) = \mathbf{comm}_M(X_0, X_1, \cdots, X_p, descendants(T))$ so that $\forall X \in (descendants(T)), \tau_P(X) = \varnothing$ (a new local computing phase is still impossible). Finally, to keep an equivalence with your previous models, we need the same slight change of the definition of $\mathbf{comm}_A$ in the postulates as well as in the $\text{ASM}_{\text{MULTI}}$ machine. Notice that the previous results are still valid, such a modification does not affect how the computations are organized, only what the communication can change. But we agree that this is an *unsatisfactory trick* but it is necessary if you want to be able to do several communication steps or user-programmed shifts of level.

Q46: *Would not that hide some communications between the nodes and the leaves?*

Admitting this new communication functions, yes. But that could be part of the $\mathbf{g}_i$ parameters. Notice that the semantics forbids assignments during the node superstep, only $\mathbf{comm}_A$ can modifying the structures. For example, such modifications could be used when *serializing* values (mainly an internally graph traversing) when using functional or object programming languages [2]. One can also imagine some restrictions on $\mathbf{comm}_A$. We currently prefer to stay as general as possible. We will discuss this issue in future work. Now let us present our final result.

3.2.2 Algorithmic completeness of a core-imperative MULTI-BSP language

Now that we have defined our core MULTI-BSP programming language as a common imperative model of computation, we prove in the following that $\text{ASM}_{\text{MULTI}}$ algorithmically simulates $\text{IMP}_{\text{MULTI}}$ and *vice-versa*.

Q47: *"Algorithmically simulates", what is it?*

We say that a computation model M_1 can *algorithmically 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.

Definition 16 (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) $\mathcal{L}(P_1) \supseteq \mathcal{L}(P_2)$, $\mathcal{L}(P_1) \setminus \mathcal{L}(P_2)$ is a finite set of fresh variables depending only on P_2, and uniformly initialized; there exists $d > 0$ and $e \geq 0$ depending both only on P_2 such that for every execution $\vec{T} = \mathcal{T}_0, \mathcal{T}_1, \ldots$ of P_2 there exists an execution $\vec{S} = \mathcal{S}_0, \mathcal{S}_1, \ldots$ of P_1 verifying that for every (2) $t \in \mathbb{N}$, $\mathcal{S}_{d\times t}|_{\mathcal{L}(P_2)} = \mathcal{T}_t$ and (3) $\text{time}(P_1, \mathcal{S}_0) = d \times \text{time}(P_2, \mathcal{T}_0) + e$.*

Notice that the parameters d and 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 [14] for the functions it computes. More details (notably about fresh variables and uniformly initialized) could be find in [20] (p. 30).

Q48: *So is the cost preserved using such an algorithmic simulation?*

Indeed. Simply because an algorithmic simulation [19, 20] only grows any step (computation or communication) of the simulated model by a *constant* factor. Thus durations (the cost of the algorithms that is all the computations and commmunications) grow only in a linear fashion.

Q49: *So you must now transform any $\text{ASM}_{\text{MULTI}}$ machine into an $\text{IMP}_{\text{MULTI}}$ one, and vice-versa?*

Yes. In fact, some modifications of the past transformations of [20] are necessary and such transformations are not trivial and must be treated carefully.

Q50: *So first, could you present the transformation of $\text{IMP}_{\text{MULTI}}$ programs into $\text{ASM}_{\text{MULTI}}$ ones ($\text{ASM}_{\text{MULTI}}$ simulates $\text{IMP}_{\text{MULTI}}$)?*

This transformation uses a control flow graph (CFG) $\mathcal{G}(P)$ [19, 20] of any $\text{IMP}_{\text{MULTI}}$ program P in order to follow the flow of execution (recall that $\text{ASM}_{\text{MULTI}}$ programs have a single loop) and gets a *bound* number of booleans (each per CFG's vertices) to manage this control. The computation of this CFG ([20] p. 33) does not require any change as well as the step-by-step simulation of the sequential parts [20].

Except that there are now two new kinds of vertices, those of the **up** and **down** that are generated as for the **comm** in [20].

Q51: *Which is to say?*

Each processor of the $\text{IMP}_{\text{MULTI}}$ machine M will be simulated by the $\text{ASM}_{\text{MULTI}}$ machine A by using the same processor 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 (where each of these booleans is a vertice of the CFG). Notably, there is the $b_{\mathbf{end}}$ boolean that indicates if there exists or not new instructions to perform in the program P. The ASM program generated from this set of booleans is as in [20] except there are now two new additional function cases that are **up** and **down**, each generating nothing as for the **comm** case.

Q52: *And thus, how the communications are managed, notably the "shifts" of levels?*

In the previous definitions of Section 2, we make the hypothesis that, for a MULTI-BSP algorithm A, after a call of the function of communication $\mathbf{comm}_A$, the machine, by reading the memories only, is able to shift (± 1) levels (see Lemma 2 p.12). We will construct such $\mathbf{comm}_A$ as $\mathbf{comm}_A = \cup_{P \in \mathcal{G}(P)} \mathbf{comm}_{A^P}$ where $\mathbf{comm}_{A^P}$ is defined inductively on $\mathcal{G}(P)$ [20] for any program P (that is using the booleans of the CFG which is thus an example of the aforementioned hidden computations of the communications). We use $[\![P]\!]^{\text{ASM}}$, the translation of the first step of a program P [20], where by construction only one b_P is true before each step; That forces only one new $b_{P'}$ to be true (except for a final state). The function of communication $\mathbf{comm}_{A^P}$ is thus inductively build to be as follows: for a given b_P, if the boolean $b_{P'}$ is set to true (where $P \equiv I; P'$ and I is an instruction) then $\mathbf{comm}_{A^P}$ is built inductively on $[\![P']\!]^{\text{ASM}}$; Moreover and by case of I, if I is an update $(f(\theta_1, \ldots, \theta_\alpha)) := \theta_0$ then do not execute it. Otherwise, if I is **comm** then $\mathbf{comm}_{A^P}$ is $\mathbf{comm}_M$ and that ends the induction. If I is **down** (*resp.* **up**) that also ends the induction and the $\vec{I}_T$ (where T is the node where the communication is currently performed) of the $\text{IMP}_{\text{MULTI}}$ machine are updated accordingly in order to get the simulation (we thus need a fresh variable s on each leaf structure that simulated the $\vec{I}_T$ of the $\text{IMP}_{\text{MULTI}}$ machine; this variable is modified only by $\mathbf{comm}_A$). And all b_P are now false in order not to throw a computing phase before the $\mathbf{comm}_M$.

Q53: *Have you got an example of an* $\text{IMP}_{\text{MULTI}}$ *scheme of communication*

Take for example the `upward(value v)` routine [30] and `bsp_sync()`. It allows us to upward values from the memories of a level i to the memory of $i-1$. In IMP_{BSP}, for `bsp_sync()`, we should have a code such as:

while *notEmpty*(buffer){
  if *manage*(**id**){$\mathbf{comm}_M$;}
};
up;

where the $buffer$ contains the values to upward and *manage* is a function that determines if the leaf manages or not the node at the given level (being the "master" or not [30]). In our simulation, on each leaf, the $\text{ASM}_{\text{MULTI}}$ program will run the above code where the values from the buffers are communicated step-by-step by the $\mathbf{comm}_A$. And finally, we up levels.

Q54: *So, are you running the program on nodes in an abstract manner until reaching a barrier (a communication phase)?*

That sums it up well. Such an execution is *costly*[6] but this cost is inside the $\mathbf{comm}_A$ which is thus not to take into account in the simulation. This is not perfect but allows the following result.

Proposition 3 $\text{ASM}_{\text{MULTI}}$ *algorithmically simulates* $\text{IMP}_{\text{MULTI}}$ *with at most length*$(P)+2$ *fresh variables, a temporal dilation* $d = 1$ *and an ending time* $e = 0$.

Proof (sketch). The proof is similar to the one of [20] (number of fresh variables) because the computing phases are managed in the same way. Only the communications differ. Let M be an $\text{IMP}_{\text{MULTI}}$ machine, and let A be the $\text{ASM}_{\text{MULTI}}$ machine defined as in [20] (except the communication function). According to the definition of an algorithmic simulation [19, 20], there are three points to prove:

1. The number of fresh variables is not modified by the function of communications and thus stay finite as in [20];

[6] Note that in [2, 29], only small computations are expected on nodes so such a trick is not an aberration.

2. Moreover, a state T of the $\text{ASM}_{\text{MULTI}}$ machine A is final if $\tau_\Pi(T) = T$ and $\forall \langle X | \overrightarrow{T} \rangle \leqslant T$, $\mathbf{comm}_{A^P}(X, \overrightarrow{T}) = (X, \overrightarrow{T})$ and this happens if and only if $b_{\mathbf{end}}$ is **true** for every processor and $\mathbf{comm}_{A^P}(X, \overrightarrow{T}) = (X, \overrightarrow{T})$ where P is the empty program. Therefore, the $\text{ASM}_{\text{MULTI}}$ machine A stops if and only if the $\text{IMP}_{\text{MULTI}}$ machine M stops, and the ending time is $e = 0$;

3. When every processor of a subtree V of T has terminated, the state is in a communication phase and the communication function $\mathbf{comm}_M$ updates the boolean variables from $b_{\mathbf{comm};P}$ to b_P, thus respecting the behavior of the function **next**. So, one step of the $\text{IMP}_{\text{MULTI}}$ machine M is simulated by $d = 1$ step of the $\text{ASM}_{\text{MULTI}}$ machine A (during computation phases). Next, for the communication phases, the $\text{IMP}_{\text{MULTI}}$ machine M is $\overrightarrow{\mathcal{I}}_T \star \overrightarrow{P}_T \star T \ \overrightarrow{\succ} \overrightarrow{\mathcal{J}}_U \star \overrightarrow{Q}_U \star U$. We have $descendants(V) = \{s \cdots s\} \subset \overrightarrow{\mathcal{I}}_T$ and there are two cases; First M is shifting of level and the generation of $\mathbf{comm}_A$ (described in Question 52 p.20) induces (by induction) a single call of $\mathbf{comm}_A$ making $d = 1$ and the semantics of $\overrightarrow{\succ}$ respects the behavior of the function **next**; Second M is performing a commmunication phase and then $V = \left\langle X_0 \middle| \langle X_1 | \overrightarrow{T_1} \rangle, \ldots, \langle X_p | \overrightarrow{T_p} \rangle \right\rangle \leqslant T$ again both M and A perform a single call of the function of communication on the $X_0, X_1, \cdots, X_p$ making $d = 1$.

□

Q55: *And now for the second simulation?*

We prove this second simulation (the reverse of the previous simulation) in two steps: (1) We translate (as in [20]) 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}.

Q56: *Could you recall P_Π^{step} please?*

Of course. Because the ASM and IMP programs do not have the same kind of loops and updates, a naive *sequentialization* of an ASM program Π does not work [19]. A solution is provided in [19] and used in [20] (p. 35) for BSP programs. We also use such a solution[7] for the sequential computations and we note it P_Π^{step}.

Q57: *So for the sequential parts there is no change. And for the communications?*

According to the definition of $\text{ASM}_{\text{MULTI}}$, a communication phase begins at the termination of the ASM program Π, and continues (recursively) until Π can do the first update of the next computation phase.

```
while ¬b^end_comm_A {
  if F^end_Π {
    skip (r + c + m)⨾
    if needDown {down; comm;}
      else if needUp {up; comm;}
      else {skip (r + c + m) ⨾ comm;}
    }
  } else { P^step_Π }
}
```

Figure 2: Translation P_Π of Π.

As in BSP [20], with the use of the communication function's exploration witness $\Theta(\mathbf{comm}_A)$, each processor can locally know whether $\mathbf{comm}_A$ has updated the memory or not at the last step. But it cannot know if the communications have globally terminated. Therefore, we add a fresh boolean $b^{\text{end}}_{\mathbf{comm}_A}$ updated by $\mathbf{comm}_A$ itself to detect if the communication phase has terminated. We also use two fresh booleans *needUp* and *needDown* that are updated to true by respectively the **up** and **down** functions and both booleans are used to go through the d-trees. Those booleans are *disjoint* in order not to up and down at the same time or being false both if only a horizontal communication is needed. We make the hypothesis that $\mathbf{comm}_A$ is also able to set these two booleans to false after a shift of level. So the code does not make updates on the nodes except those which are necessary by **comm** (resp. **down** and **up**) and thus do not throw a computing phase before communications are done.

Q58: *Finally, what is this second transformation?*

Let P_Π be the IMP program defined in Fig. 2, which is the translation of the $\text{ASM}_{\text{MULTI}}$ program Π where P_Π^{step} is defined in [20] (p. 37). Notice that the communications are executed only if F_Π^{end} is **true** [20] (p. 38). We note $r = card(\text{Read}(\Pi))$, c is the number of formulas in the ASM program Π and m is the maximum number of updates per block of updates ([20] p. 18) in Π. We obtain the following result.

Proposition 4 $\text{IMP}_{\text{MULTI}}$ *algorithmically simulates* $\text{ASM}_{\text{MULTI}}$ *with at most* $Read(\Pi) + 4$ *fresh variables, a temporal dilation* $d = 2 + r + c + m$ *and an ending time* $e = d + 1$

[7] The idea is to transform each non-clashing sets of non-trivial updates ([20] p. 17) of any ASM program Π into an appropriate sequence of common updates ([20] p. 36).

Proof (sketch). The proof is again similar to the one of [20] (number of fresh variables) because the computing phases are as in [20] and only the communication phases differ. Let A be an $\text{ASM}_{\text{MULTI}}$ machine, with an ASM program Π and a communication function $\mathbf{comm}_A$. Let M be the $\text{IMP}_{\text{MULTI}}$ machine defined as in [20] (same fresh variables+$needUp$ and $needDown$). If $\mathbf{comm}_A$ changes nothing then $\mathbf{comm}_M$ updates $b^{\text{end}}_{\mathbf{comm}_A}$ (resp. $needUp$ and $needDown$) accordongly. As above, there are three points to prove:

1. There are only two new variables (compared to [20]) so the result;
2. The computing phases are managed as in [20]: the **while** command checks in one step whether the execution has terminated or not (as in [20]); then, the **if** command checks in one step whether F^{end}_{Π} is true or not which (as in [20]) indicates whether Π has terminated or not. Notice that the **comm** commands are the ones in order to be the last things done by a processor during the simulation of a communication step of the $\text{ASM}_{\text{MULTI}}$ machine. This ensures the synchronization of the processors in the same branch when some are in a computation phase and others are not modified; That also enables the shift of levels (down or up, technical Lemma 2). It is only the function of communication that is able to perform such a shift and thus we must force it to be called. Thus, each step of the $\text{ASM}_{\text{MULTI}}$ machine is simulated by exactly $d = 2 + r + c + m$ steps of the $\text{IMP}_{\text{MULTI}}$ machine.
3. A terminal state of the $\text{ASM}_{\text{MULTI}}$ machine is reached when Π has terminated for every processor and the communication function $\mathbf{comm}_A$ changes nothing. In such a state, the **while** checks $b^{\text{end}}_{\mathbf{comm}_A}$ which was **false** during the entire execution, then the **if** checks that F^{end}_{Π} is **true**, then `skip` $(r + c + m)$, then some nested communications are done and sets $b^{\text{end}}_{\mathbf{comm}_A}$ to **true**. Then the **while** verifies that the execution is terminated, and the program reaches the end after $e = d+1$ steps.

□

Q59: *And now?*

According to the previous propositions, we obtain the following result.

Theorem 2 $\text{IMP}_{\text{MULTI}} \simeq \text{ASM}_{\text{MULTI}}$

Q60: *Could you sum up?*

Our core imperative language $\text{IMP}_{\text{MULTI}}$ is **algorithmically equivalent** to $\text{ASM}_{\text{MULTI}}$ which is the set $\text{ALGO}_{\text{MULTI}}$ of MULTI-BSP algorithms. In that sense, the MULTI-BSP *cost model is preserved* by the translation as it is discussed in [20] (for the BSP context). And any programming language with at least the features of $\text{IMP}_{\text{MULTI}}$ is MULTI-BSP complete that is it allows us to program any MULTI-BSP algorithm. This result succeeds the work of [20], where it has been proved that $\text{IMP}_{\text{BSP}} \simeq \text{ASM}_{\text{BSP}} = \text{ALGO}_{\text{BSP}}$, which also succeeds the work of [19] where $\text{IMP} \simeq \text{ASM}_{\text{SEQ}} = \text{ALGO}_{\text{SEQ}}$.

Q61: *It looks consistent. Now, it seems necessary to end this discussion.*

Ok. We will first discuss the works that inspired this paper and then we will conclude this discussion by summarizing what we can learn from this contribution and what we can do in the future.

4 Related Work

Q62: *So, just a few more questions. You used a particular model to define the algorithms but would not there be other ones?*

Let us see. As a formalization of an intuitive notion, there can be no formal proof that ASMs truly formalize what they are intended to. But the algorithms of all (small-steps, discrete time) sequential computation models considered up to now have been proved to be faithfully emulated by ASMs [6, 14, 15]. ASM is thus the most used model but right, some authors propose other models. To our knowledge, the other models are the *recursive equations* of [21], the *concrete data structures* of [3] and the *category* of algorithms of [28]. Unfortunately, there is no proof of equivalence between the aforementioned models. We are here using ASMs in line with our previous works [19, 20] but it would require further investigation. Notably, if we want results about *functional languages.*

Q63: *Are you thinking of the* BSML *language (*BSP *functional programming) and your previous work about* MULTI-BSP *that is* MULTI-ML *[2]?*

There is also the work of [11] as a first bridge between algorithms and $\lambda-$calculus, the kernel of

proof assistants such as COQ[8]. MULTI-ML allows MULTI-BSP programming in ML. Proving its algorithm completeness would be great. But that could be a difficult task. Indeed, even if it has to be feasible as for the λ-calculus [11], some details such as the exchange of functions still pose a problem.

Q64: *Are there the same difficulties for other* MULTI-BSP*-like languages?*

If you are thinking of [17, 29] or [18], that is surely not the case. Mostly because only memory blocks are communicated and they dot not have particular meaning. Perhaps the sending of objects can disturb the semantics but the true difficulty is giving an operational semantics to the C++ language which is an orthogonal work.

Q65: *And for other bridging models? I have the impression that we could write an article by model, which would not be finally very interesting in fact.*

You are right. As said in [26]: "*The goal here is to identify a bridging model on which the community can agree, one which would influence the design of both software and hardware. It will always be possible to have performance models that reflect a particular architecture in greater detail than does any bridging model, but such models are not among our goals here.*". There are thus many papers about bridging models notably with subgroup synchronizations or hierarchical memories/networks such as [8, 9, 10, 13, 25, 35]. Each has its own specificities and so having all of them in a single set of postulates seems impossible except using the (too) most general ones for distributed computing of [5, 22].

Q66: *So is it the end of this research?*

The MULTI-BSP model is general enough to take into account most (rather) modern hierarchical HPC architectures. For instance, the models of [13, 31, 32] can be seen as a two levels MULTI-BSP machine without a global memory (the root node's one). But as we have just seen, the MULTI-BSP model is not clean about the potential computations on nodes. Moreover, GPUs (that could induce an heterogeneous d-tree) are not taken into account which is sad for modern HPC architectures. We can imagine an interesting future work: designing another bridging model, giving its axiomatization and implementing a programming language which is proven to be algorithmically complete with this model. And this new language would have primitives to better manage the computation on the non-leaf parts.

Q67: *In [13], the author criticizes* MULTI-BSP *for this overlayer of difficulty. So, would not that be too complicated?*

Hierarchical models have drawbacks. It is more difficult to prove optimally in hierarchical models than in flat ones. [31] has also analyzed that some typical *numerical* algorithms cannot take advantage of a 2-level model. This is due to too much communication in the slower networks. Portability can also be more difficult when implementing languages dedicated to these models. Especially if we want a framework/tool to be able to run a single algorithm on a cluster of multi-cores+GPUs. For the case of [13], we can argue that if we can scatter the data/computations in two folds (one for the machines, another for the cores), we must be able to continue this decomposition for more levels (recursively). On the other hand, in [1], we have analyzed that when algorithms need much more computation than communication, hierarchical programs can be much more efficient than flat ones (but such programs are still more difficult to write and design, which is not a surprise).

Q68: *Back to the postulates, what about the parallel thesis of [12]? Are there any possible connections between your work and theirs?*

As explained in [20], even if the work is impressive, we think that such postulates are too much expressive: they can capture *concurrent writing* into a shared memory or asynchronous computations with unbounded number of processors. They define much more complicated exploration witnesses to manage that. Such witnesses are not necessary for the purpose of BSP-like bridging models and, if we want to use such a work, we would have to introduce some new and artificial (*ad-hoc*) postulates in order to limit the expressiveness which is not a natural way to capture algorithms as it is intended in [16].

Q69: *Maybe just using the postulates (using sets of the form* (a, π_a) *(where* a *is a process name and* π_a *a sequential* ASM*) of [5, 22] for the axiomatizations is a simple solution. I mean, having such a pair on leaves (simulating the processors) and only memory/structures on nodes. That would be closer to the definition of a* MULTI-BSP *machine of [26]. But you say that these postulates are complex but yours*

[8] In such a calculus, a proof of $\forall x{:}X\ \exists y{:}Y\ P(x,y)$ corresponds to a functional program that computes a y for any x such that $P(x,y)$; There is thus an underlying algorithm "inside" the program.

become more and more complex. What can you say?

Because d-trees need recursive definitions, they are a little complex to define. But the ideas behind them are not. A function of neighborhood is used in [5] to represent the network: a graph of communicating processes. But what is truly complex in [5, 22] is the use of infinite sets of processes[9], a mix between postulates and ASMs and, essentially, that the barriers must be exhibited as an implemented function not as a primitive. We also discuss the problem in [20].

Q70: *But in [33], they used such a solution for* BSP *and without such a mix. Can you compare you approach to theirs?*

Firstly and as explained in [20], the authors use complex postulates (those which are necessary to get an axiomatization of all the distributed algorithms where each global step of computation manipulates an infinite set) and then add new postulates to limit their work to BSP. That is not a natural way to get an *intuitive* class: five postulates and a signature restriction are necessary compared to our 4 simple postulates. Secondly and as they explained in their conclusion " It is also no problem to relax the assumption of a fixed number of processors." (necessary condition to get immortal algorithms). But it will not be as easy as they say. Because they use a "share" `barrier` location which is "free" in the environment. And such a location could be modified by the machine in a single step with the values of p other locations. And the signature admitting p different symbols of location information about the barrier. If p is not fixed, this will induce a problem for the exploration witness (finiteness). Thirdly, each processor uses a *monitored* location which is not clearly defined. Fourthly, the sequence of the supersteps (one crucial notion of the BSP model) is a consequence (theorem) of their postulates (implementation) not in the natural definition as in our work. Finally, they only allow sending data and thus their model of execution is, in order, a phase of computations, one of sending, one for receiving and doing a barrier. But the BSP model does not force such an order between the data exchange: an h-relation is where each processor sends *or* receives a maximum of h words of data. Thus, they have proven a functional (input-output) but not an algorithmic (step-by-step) characterization, as opposed to Gurevich's claim to capture classes of algorithms [15] , and not only classes of functions [14].

Q71: *And their* BSP-ASM *machine?*

To perform a synchronization, an additional pair (a, π_a) is used. It has an ASM rule which is syntactically bounded by p (read the p local information about the barrier). The only way to get an unbound p is using the more "complex" witnesses of [12]. So, it is necessary to merge all these postulates to get the "simple" BSP model of computations. Such witnesses are unnecessary in our work because of the use of p-tuples (or d-trees). Note that our machines (ASM_{BSP} and $\text{ASM}_{\text{MULTI}}$) abstract how the barrier is done by the use of the **comm** function. This is not to be seen as cheating but as leaving the machines the way to implement the BSP's synchronization unit (software or real physical card).

Q72: *They also criticize your own work with two objections. The first one is "In a nutshell* Marquer/Gava *provide an axiomatisation with four postulates, the first three of which are essentially* Gurevich's *postulates for sequential algorithms with the only change that states are restricted to a specific form. Hence, these three postulates imply that* BSP *algorithms are special sequential algorithms...If* BSP *algorithms were sequential, then by* Gurevich's *sequential* ASM *thesis they would be captured by sequential* ASM*s." [33].*

They are thinking to a kind of diagonalization. But there is not a constant and fixed sized p (nor for the d-trees in our works) in our initial sets of p-tuples. There are many p and our postulates only say that an execution must not change the value of p defined by an initial state (as a p-tuple). So if we can simulate an ASM_{BSP} with a sequential ASM, there will be a contradiction. We can imagine an ASM that performs the assignments on each of the p structures of any of the p-tuple where p is unbound. But this violates the exploration witness where only a constant number of assignments are allowed. On the other hand, the algorithmic simulation works with constants in order to preserve the costs of the algorithms. Thus p (unbounded) computations could not be algorithmically simulated by a single machine. By continuing such a reasoning , a sequential machine can also simulate all models of distribution by simulating the non-determinism of concurrent send/received using finite sets of structures which is non-sense.

Q73: *And for their second objection? "In particular, there would also be an* ASM *rule capturing the*

[9] This permits the distributed algorithm to be added to a concurrent system, so working for correctness of systems and not considering the algorithms as independent and abtract objects; We prefer programming languages for reasoning about correctness.

behaviour in a communication phase. However, their fourth postulate claims just an arbitrary state transition function for the communication phase, and this function appears again in the definition of an ASM_{BSP}*, i.e. no rule capturing the communication is derived."*

They define the communications using send/receiving operations on p^2 number of channels (p channels per processor). The barriers are simulated using a "protocol" with the use of different communications depending if the pid of a processor is even or odd. They are thus limited by BSP algorithms that use such operations (that also allows them to simulate all collective routines *à la* MPI). But again, if considering a BSP algorithm with DRMA routines (and shared memories), one processor could write in its shared location and then the p other processors could read this value. Using their send/receiving implementation only, the emitter processor will send p times the value. The BSP cost is the same but not the algorithm. Because we could not know which BSP libraries will be available in the future, we use an abstract function **comm** performing the communications step-by-step. We provide in [20] an example of such a function for DRMA routines in the ASM context: how the exploration witness could be exhibited for such algorithms. For example, this forces to serialize the terms as finite lists of elementary data and use such lists to estimate the h-relations.

Q74: *Ok, but if your axiomatization admits* (a, π_a) *on leaves and structures only on node, you can truly exhibit the processors, which is not the case in your work?*

Indeed. But when designing an algorithm, are you truly thinking about *how* does a processor compute? We think that algorithm design is mainly independent from the processors. And finally, in the original paper of MULTI-BSP [26], computing units are added as an abstract overlayer, as us in a way. There is nothing said about how units compute locally. Only the global organization in (nested) super-steps is described.

Q75: *And what about the work of [34]? The authors define* PRAM *algorithms where each state is a finite sets of "cells" (processors) accessing a global (share) memory using a single and replicated program. That is closer to your axiomatization and your work. For the* MULTI-BSP *purpose, we can imagine that their global memory is now organized in a tree-like fashion. Can you elaborate on that?*

Indeed. But a detail (nevertheless important) separates us: their postulates limit the algorithms to the use of a finite number of non-concurrent accesses to this finite share global memory of "localizations" (kind of memory locations). Note that such postulates do not limit the memory size, only the number of locations and thus the number of possible running cells to accesses to these localizations. Using our postulates, we are not limited by such a number because we are using distributed memories (but each of these memories is also locally limited by the number of locations) even if we are using DRMA locations [20]. Nevertheless, by limiting the elementary operations (roughly only on algebraic data where the size of each term is the number of constructors/letters), they give the possibility to study the complexity of PRAM which is a nice result.

Q76: *In both of these works [12, 34], shared memories are used. That is closer to the original model of* MULTI-BSP *[26], so why not adapt them to* MULTI-BSP*?*

As explain in [20], if any finite (in the number of locations) shared memory does not depend on the number of processors, the memory should contain an unbounded amount of information accesses, which seems unrealistic (as well as an infinite memory). 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* if condidering exploration witnesses *à la* Gurevich (otherwise, complex witnesses as in [12] must be considered). This should not be confused with the study of machines where it is obvious that each of them has a finite number of processors or with some PRAM complexity results where any algorithm executing on a machine with a given number of processors can also be executed on another machine with another number of processors; algorithms should be defined for *any number* of processors. So, let us conclude.

5 Conclusion

5.1 Summary of the Contribution

A *bridging model* provides a common level of *understanding* between hardware and software engineers. The BSP bridging model has been studied for a long time and many BSP algorithms have been designed

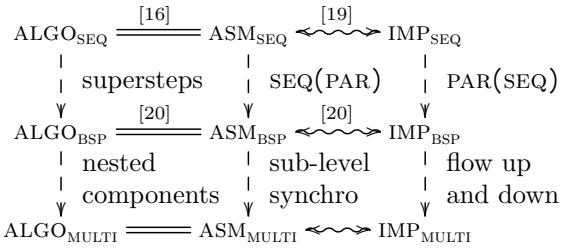

Figure 3: Diagram of the models.

and used with success in many domains [4]. But it can no longer be considered sufficient for modern architectures. If we consider a bridging model as a tool for the design of "*immortal*" algorithms, a more complete model must be considered. The MULTI-BSP, a natural extension of BSP, is currently a better candidate to do so.

In this discussion we showed how to extend the work of [20] in order to take into account the specifies of the MULTI-BSP model, notably the recursive decomposition of the components and thus of the supersteps. We give *four postulates* only to axiomatize the set of the MULTI-BSP algorithms by slightly modifying those for BSP [20]. This allows a greater *confidence* in the results; we also give the *operational* point of view in the form of ASMs and finally prove the *algorithmical completeness* of a core MULTI-BSP language by using an algorithmic simulation of the programs with the aforementioned ASMs. We can deduce from this that the *cost model* is preserved because of the algorithmic simulation: the number of small steps of the algorithms are identical, up to a constant factor. This is the main novelty and specificity of this work compared to the traditional work about distributed or concurrent ASMs. We also show the problems of how to *manage* computations on nodes that are only memories; notably, this requires an uncommon reading of data at any level from the computing units. Fig. 3 illustrates the result diagram and the following theorem summarizes the results of this work:

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

Q77: *Do you keep the* SEQ(PAR) *and* PAR(SEQ) *[7] designs of your past work?*

Of course. The ASM program still contains the algorithm and, following the machine, defines the computations to perform on the leaves. On the other hand, each $\text{IMP}_{\text{MULTI}}$ program performs (independently) some computations on the leaves and some subparts of the programs manage the communications.

Q78: *Without wishing to be demeaning, your contribution contains an obvious flaw: the* MULTI-BSP *model makes the hypothesis of limited memories (*$\mathbf{m_s}$*) but the first-order structures do not. Can you elaborate on that?*

We have to admit. But this limitation of memories poses other problems such as: how to compute the *size* of the objects? What size do the programs take in the memories? What is the extra cost for miss-caches? And above all, can we really *prohibit an excessive consumption* within all these memories without it becoming unmanageable? For the size of the objects, as in the computation of the h-relations in [20], we can count the size of first-order terms using their number of letters. We can also prohibit the use of too big terms (or too many of them) by always counting their sizes. But forbidding computing (performing an update) because of a lack of memory (and thus, returning `undef` or leaving the structure unchanged) seems meaningless especially when using terms in place of block of memories; that it is too much dependent on the underlying used programming language and thus we cannot obtain any general result on this subject. It seems thus an unrealistic solution and for the future, the only pragmatic way seems to trust the algorithmists.

Q79: *Finally, you are still working up to elementary operations. So, could you be sure that the cost model is not too abstract? And you do not provide any constructive example of a function of communication as in [20].*

Algorithms are inherently oracular and so are MULTI-BSP ones. We do not care if one of the primitives has too high an execution time or is unrealistic (in that case, the algorithm is just not defined at the right level of abstraction), we count the number of steps only. Some works have begun in this area

[15]. For the communication, we observe that there is a cruel lack of standard MULTI-BSP libraries and so we prefer to wait rather than to work on a function that will not reflect a real implementation.

5.2 Future Work

Q80: *Let us imagine possible future work. Why do you bother not to have truly computations on the nodes? It seems to be more of an unnecessary constraint. Can we imagine a better hierarchical model?*

There is no computation on nodes and our use of any communication function is a trick that is not satisfactory in general: for high-level languages such as JAVA or ML, scattering or gathering data is performed on complex data-structures. That needs specific computations that need to be exhibited. So yes, it is indeed a constraint because how it is carried out such computation seems to happen for free. [26] remarks that we can use the $\mathbf{G_i}$ for distant reading (from computing units to nodes) but that is also not satisfactory because the $\mathbf{G_i}$ constants are used for global (synchronous) communication inside a component rather than for reading on distant components.

Another solution is possible: having d-trees recursively inside d-tree nodes (in place of memories only). That can allow us to better take into account computations that could be needed on nodes because a deep d-tree can be a single leaf which could handle the role of a *representative* computing unit (and assuming a special parameter to this memory access). Moreover, if a deep d-tree can contain by itself some nodes, we freely get a natural model of GPUs used at a specific level (case of modern HPC architectures).

Q81: *This gives the impression that there will always be a need for new bridging models. I mean, adding* GPU*s,* FGPA*s, etc.*

And to our knowledge, nobody has proved the equivalence (if it exists) of the different models of formalization of the sequential algorithms (and even more for BSP/MULTI-BSP algorithms). Nor formally defined interesting sub-classes of the aforementioned models: polynomial times, communication-oblivious [24], *etc.* It is actually quite motivating.

Q82: *And could we imagine a proof of such results using a theorem prover such as* COQ*?*

That is a good idea. Notice that we are not aware of such a work even for the sequential case. There are surely interesting things to perform.

Q83: *Yes. A lot of work to do. A final question. Again, you admit never to use the memory sizes. What are your plans to solve this problem?*

In the real world, the unlimited-resources abstraction is absurd (and that explains the use of memory sizes in the MULTI-BSP model) but it remains hard to limit memories. But for restricted models (*e.g.* real-time programs or those that were statically analysised) we can surely obtain an estimation of the memory consumption and some finer results.

References

[1] Allombert, V., Gava, F.: Programming BSP and MULTI-BSP algorithms in ML. The Journal of Supercomputing (2019). To appear

[2] Allombert, V., Gava, F., Tesson, J.: MULTI-ML: Programming MULTI-BSP algorithms in ML. Journal of Parallel Programming **45**(2), 340–361 (2017)

[3] Berry, G., Curien, P.: Sequential algorithms on concrete data structures. Theor. Comput. Sci. **20**, 265–321 (1982)

[4] Bisseling, R.H.: Parallel Scientific Computation. A structured approach using BSP and MPI. Oxford University Press (2004)

[5] Börger, E., Schewe, K.: Communication in abstract state machines. J. UCS **23**(2), 129–145 (2017)

[6] Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)

[7] Bougé, L.: 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, *LNCS*, vol. 1132, pp. 4–26. Springer (1996)

[8] Cappello, F., Fraigniaud, P., Mans, B., Rosenberg, A.L.: An Algorithmic Model for Heterogeneous Hyper-clusters: Rationale and Experience. Int. J. Found. Comput. Sci. **16**(2), 195–215 (2005)

[9] Cha, H., Lee, D.: H-BSP: A hierarchical BSP computation model. Journal of Supercomputing **18**(2), 179–200 (2001)

[10] Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., von Eicken, T.: Logp : Toward a realistic model of parallel computation. ACM SIGPLAN Symposium on Principles and Practises of Parallel Programming pp. 1–12 (1993)

[11] Ferbus-Zanda, M., Grigorieff, S.: Asms and operational algorithmic completeness of lambda calculus. In: A. Blass, N. Dershowitz, W. Reisig (eds.) Fields of Logic and Computation, *LNCS*, vol. 6300, pp. 301–327. Springer (2010)

[12] Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing simplified parallel {ASM} thesis. Theoretical Computer Science **649**, 25–53 (2016)

[13] Gerbessiotis, A.V.: Extending the BSP model for multi-core and out-of-core computing: MBSP. Parallel Computing **41**, 90–102 (2015)

[14] Grigorieff, S., Valarcher, P.: Classes of algorithms: Formalization and comparison. Bulletin of the EATCS **107**, 95–127 (2012)

[15] Gurevich, Y.: The sequential ASM thesis. Bulletin of the EATCS **67**, 93 (1999)

[16] Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. **1**(1), 77–111 (2000)

[17] Hamidouche, K., Mendonca, F.M., Falcou, J., de Melo, A.C.M.A., Etiemble, D.: Parallel smith-waterman comparison on multicore and manycore computing platforms with BSP++. Journal of Parallel Programming **41**(1), 111–136 (2013)

[18] Keßler, C.W.: NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model. The Journal of Supercomputing **17**(3), 245–262 (2000)

[19] Marquer, Y.: Algorithmic completeness of imperative programming languages. Fundamenta Informaticae **168**(1), 51–77 (2019) `https://dr-apeiron.net/lib/exe/fetch.php/fr:recherche:fi-while-long.pdf`

[20] Marquer, Y., Gava, F.: Axiomatization and characterization of BSP algorithms. Journal of Logical and Algebraic Methods in Programming **109**(1), 1–43 (2019) `https://hal.archives-ouvertes.fr/hal-01742406/document`

[21] Moschovakis, Y.N.: What is an algorithm? In: B. Engquist, W. Schmid (eds.) Mathematics Unlimited — 2001 and Beyond, pp. 919–936. Springer (2001)

[22] Schewe, K.D., Ferrarotti, F., Tec, L., Wang, Q., An, W.: Evolving concurrent systems: Behavioural theory and logic. In: Australasian Computer Science Week Multiconference (ACSW), pp. 1–10 (2017)

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

[24] Tiskin, A.: The Design and Analysis of Bulk-Synchronous Parallel Algorithms. Ph.D. thesis, Oxford University Computing Laboratory (1998)

[25] de la Torre, P., Kruskal, C.P.: Submachine locality in the bulk synchronous setting. In: Euro-Par'96, 1123-1124. Springer Verlag (1996)

[26] Valiant, L.G.: A bridging model for multi-core computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)

[27] Vardi, M.Y.: What is an algorithm? Commun. ACM **55**(3), 5 (2012)

[28] Yanofsky, N.S.: Towards a definition of an algorithm. J. Log. Comput. **21**(2), 253–286 (2011)

[29] Yzelman, A.N., Bisseling, R.H.: An Object-oriented Bulk Synchronous Parallel Library for Multicore Programming. Concurrency and Computation: Practice and Experience **24**(5), 533–553 (2012)

[30] Yzelman, A.N., Bisseling, R.H., Roose, D., Meerbergen, K.: Multicorebsp for C: A high-performance library for shared-memory parallel programming. International Journal of Parallel Programming **42**(4), 619–642 (2014)

[31] Martin, J. M. R., Tiskin, A. V.: BSP algorithm design for hierarchical supercomputers. Unpublished manuscript. `http://www.dcs.warwick.ac.uk/~tiskin/pub/2001/bsp2.ps`

[32] Gava, F., Loulergue, F.: A Functional Language for Departmental Metacomputing. Parallel Processing Letters. **15**(3), 289–304 (2005)

[33] Ferrarotti, F., Gonzlez, S., Klaus-Dieter, S.: BSP abstract state machines capture bulk synchronous parallel computations. Science of Computer Programming **184**, 1-24 (2019)

[34] Dershowitz, N., Falkovich-Derzhavetz, E.: On the parallel computation thesis Logic Journal of the IGPL **24**(3), 346–374 (2016)

[35] Li, C., Hains, G.: SGL: towards a bridging model for heterogeneous hierarchical platforms IJHPCN **7**(2), 139–151 (2012)