Algorithmic Completeness of BSP Languages

Yoann Marquer, Frédéric Gava

Abstract

The Bulk-Synchronous Parallel (BSP) bridging model is a candidate for a simple and practical definition for High Performance Computing (HPC) algorithms. These BSP algorithms have been axiomatized, then captured by the operational semantics of the BSP Abstract State Machines (ASM-BSP), an extension of the ASM model of Gurevich. In this technical report, we define a minimal imperative language While-BSP, and prove that this model of computation fairly simulates the executions of the ASM-BSP machines, and reciprocally that ASM-BSP simulates While-BSP. Therefore, While-BSP is algorithmically complete for the BSP algorithms, and so does usual programming languages like Pregel, or imperative languages using the BSPLIB library.

Keywords. *ASM, BSP, Semantics, Algorithm, Simulation*

1 Introduction

1.1 Context of the work

Usual sequential imperative languages such as C or Java are said Turing-complete, which means that they can simulate (up to unbounded memory) the input-output relation of a Turing machine and thus compute (according to the Church Thesis) every effectively calculable functions.

But **algorithmic completeness** is a stronger notion than Turing-completeness. It focuses not only on the input-output (ie the functional) behavior of the computation but on the step-by-step (ie the algorithmic) behavior. Indeed, a model could compute all the desired functions, but some algorithms (ways to compute these functions) could be missing:

1. For example, a Turing machine with one tape can simulate a Turing machine with two tapes, so they are usually seen as equivalent. But it has been proven in [BBD+04] that the palindrome recognition, which can be done in $\mathcal{O}(n)$ steps (where n is the size of the word) with two tapes, requires at least $\mathcal{O}\left(n^2/\log(n)\right)$ steps with only one tape. Strictly speaking, that means that the algorithm in $\mathcal{O}(n)$ requires at least two tapes, even if its result can be simulated. Therefore, a Turing machine with two tapes is functionally equivalent to a Turing machine with one tape, but they are not algorithmically equivalent.

2. Another example is the computation of the greatest common divisor: it is known that an imperative language with only "for-loop" statements can compute all primitive recursive functions and not more. But it has also been proved [Col91] that such a language cannot compute the gcd with the smaller complexity (one speak of "ultimate obstinacy ") whereas it is possible using a "while" statement. Therefore, some algorithms cannot be written in a language even if another algorithm can be written which compute the same function.

The question is whether a model of computation is algorithmically complete or not. In other words, if this model can compute a set of functions, can it compute these functions in

every possible way ? This question rises interest because thus we could guarantee that the best possible algorithm can effectively be written in this model to compute the desired function.

The conceptual difficulty of this question is to clearly define what is meant by "algorithm", because even if a program or a computable functions are well-defined objects (by using the Church Thesis), usually the word "algorithm" is at most a vague concept, or a way to name programs written in pseudo-code. In other words, we need a Church thesis for algorithms. The Church Thesis can be seen as a consequence of the functional equivalence between models of computation like Turing machines, lambda-calculus or recursive functions. Unfortunately, their is no proof of algorithmic equivalence between proposed models for the algorithms, like the Abstract State Machines (ASM) of Gurevich [Gur00] or the recursive equations of Moschovakis [Mos01].

Nevertheless, the axiomatic approach of Gurevich appeared convincing for the community: instead of claiming that a particular model of computation captures in itself the behavior of the algorithms, Gurevich proposed **three postulates** for the sequential algorithms:

1. Sequential Time: a sequential algorithm computes step by step
2. Abstract State: a state of the execution can be abstracted as a (first-order) structure
3. Bounded Exploration: only a bounded amount of read or write can be done at each step of computation

The Gurevich Thesis states that every sequential algorithms (Algo) is captured by these three postulates. Thus, he proved in [Gur00] that the set of the sequential algorithms is exactly the set of his Abstract State Machines, therefore ASM = Algo. This proves that the operational approach of the ASM is equivalent to the axiomatization approach of the the sequential algorithms Algo.

In [Mar18], the first author proved that usual imperative programming languages are not only Turing-complete but also algorithmically complete. This has been done by proving that a minimal imperative language While is **algorithmically equivalent** to the Abstract State Machines of Gurevich: While $\simeq$ ASM, which means that the executions are the same, up to fresh variables and temporal dilation (more details are given at Section 6 p.11). Thus, we have While $\simeq$ Algo, and any programming language containing While commands is algorithmically complete (for sequential algorithms).

In the High Performance Computing (HPC) context, such a result is nowadays impossible to obtain. The main problem is the lack of definitions of what HPC computing formally is: the community lacks a consensual definition for the class of HPC algorithms. Using a **bridging model** [Val90] is a first step to this solution. It provides a conceptual bridge between the physical implementation of the machine and the abstraction available to a programmer of that machine. It also simplifies the task of the algorithm design, their programming and ensures a better portability from one system to another. And, most importantly, it allows the reasoning of the costs of the algorithms.

We conscientiously limit our work to the Bulk-Synchronous Parallel (BSP) bridging model [Bis04, SHM97] because it has the advantage of being endowed with a simple model of execution. We leave more complex HPC models to future work. Moreover, there are many different libraries and languages for programming BSP algorithms. The best known are the BSPLIB for C [HMS+98] or Java [SYK+10], BSML [BGG+10], Pregel [MAB+10] for big-data, etc.

In [MG18], the authors have axiomatized the set Algo_{BSP} of the BSP algorithms by extending the three postulates of Gurevich from a single processor X to a p-tuple of processors $\left(X^1, \ldots, X^p\right)$ working simultaneously, and adding a **fourth postulate** to take into account the BSP super-steps model of execution: a BSP algorithm alternates between a computation phase

when the processors compute independently from the others, and communication phase when the processors send or receive messages.

Then, we have extended the ASM model to p-tuple of processors by applying the ASM program to every processor at the same time, and assuming an abstract communication function when the processors has finished their computations. We proved that this extension ASM_{BSP} is the operational counterpart of the axiomatically defined BSP algorithms, in other words that $\text{ASM}_{\text{BSP}} = \text{ALGO}_{\text{BSP}}$.

1.2 Content of the work

In the Section 3 p.4 we remind our four postulates for ALGO_{BSP}, and in the Section 4 p.7 we remind the definition of ASM_{BSP}, with the theorem $\text{ASM}_{\text{BSP}} = \text{ALGO}_{\text{BSP}}$.

As in [Mar18], we think that a minimal imperative language is more convenient than a transition function to determine classes in time or space, or simply to be compared to more usual programming languages. Therefore, we define at the Section 5 p.9 the operational semantics of our core imperative programming language $\text{WHILE}_{\text{BSP}}$, which is our candidate for algorithmic completeness (for BSP algorithms).

We want to emphasize that our work is about a formal algorithmic model and not on architectures/machines or on formal verification of specific BSP algorithms/programs. Our work aims to be an intermediary between programming languages and an algorithm model, the class of BSP algorithms. Many definitions used here are well known to the ASM community.

We detail the notion of algorithmic equivalence at the Section 6 p.11. We prove in the Section 7 p.12 that ASM_{BSP} simulates $\text{WHILE}_{\text{BSP}}$, and we prove at the Section 8 p.15 that $\text{WHILE}_{\text{BSP}}$ simulates ASM_{BSP}.

Therefore we prove in this technical report that $\text{WHILE}_{\text{BSP}} \simeq \text{ASM}_{\text{BSP}}$. Because $\text{ASM}_{\text{BSP}} = \text{ALGO}_{\text{BSP}}$, that means that our core language $\text{WHILE}_{\text{BSP}}$ is algorithmically complete for the class of BSP algorithms. We discuss at the Section 9 p.19 details and explanations about the proof and the approach.

2 Preliminaries

Definition 2.1 (Structure)**.** A (first-order) **structure** X is given by:

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

In order to have a uniform presentation, as in [Gur00] we consider constant symbols of the signature as 0-ary function symbols, and relation symbols R as their indicator function χ_R. Therefore, every symbol in $\mathcal{L}(X)$ is a function. Moreover, partial functions can be implemented with a special symbol undef, and we assume in this paper that every signature contains the boolean type (at least true, false, $\neg$ and $\wedge$) and the equality.

We distinguish in the signature $\mathcal{L}(X) = \text{Stat}(X) \cup \text{Dyn}(X)$ the dynamical symbols $\text{Dyn}(X)$ which can be updated (see p.5) and the static symbols $\text{Stat}(X)$ which cannot.

Definition 2.2 (Term)**.** A **term** θ of $\mathcal{L}(X)$ is defined by induction:

1. If $c \in \mathcal{L}(X)$ has arity 0, then c is a term
2. If $f \in \mathcal{L}(X)$ has an arity $\alpha > 0$ and $\theta_1, \ldots, \theta_\alpha$ are terms, then $f(\theta_1, \ldots, \theta_\alpha)$ is a term

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

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

A **formula** F is a term with a particular form :

$$F \stackrel{\text{def}}{=} \text{true} \mid \text{false} \mid R(\theta_1, \ldots, \theta_\alpha) \mid \neg F \mid (F_1 \wedge F_2)$$

where R is a relation symbol (ie a function with output $\overline{\text{true}}^X$ or $\overline{\text{false}}^X$) and $\theta_1, \ldots, \theta_\alpha$ are terms. We say that a formula is true (respectively false) in X if $\overline{F}^X = \overline{\text{true}}^X$ (respectively $\overline{\text{false}}^X$).

Notice that we use only formulas without quantifier, so we don't need the notion of logical variables. Instead, we will use the word **variables** for dynamical symbols with arity 0.

3 Axiomatization of the BSP algorithms

In this section, we give an axiomatic presentation by defining BSP algorithms as the objects verifying four postulates, which are a "natural" extension of the three postulates of Gurevich for the sequential algorithms in [Gur00].

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

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

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

Instead of defining a set of **final states** for the algorithms, we will say that a state S_t of an execution is final if $\tau_A(S_t) = S_t$. Indeed, in that case the execution is $S_0, S_1, \ldots, S_{t-1}, S_t, S_t, \ldots$ so, from an external point of view the execution will seem to have stopped. We say that an execution is **terminal** if it contains a final state. In that case, its **duration** is defined by:

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

A parallel model of computation uses a machine with multiple computing units (processors, cores, *etc.*), which have their own memory. Therefore, a state S_t of the algorithm must be a p-tuple $S_t = \left(X_t^1, \ldots, X_t^p\right)$, where p is the number of computing units.

Notice that **the number of processors is not fixed for the algorithm**, so A can have states using different number of processors. In this report, we will simply consider that this

number is preserved during a particular execution. In other words: the number of processors is fixed by the initial state.

Notice also that we assume that **the processors are synchronous**, as opposed to the standard approach in [BG03], which is a simplification in order to obtain a more accessible model which deals with small steps of computation or communication, and not wide steps.

If $\left(X^1, \ldots, X^p\right)$ is a state of the algorithm A, then the structures $X^1, \ldots, X^p$ will be called processor memories or **local memories**. The set of the local memories of A will be denoted by $M(A)$.

Notice that, as in [Gur00], we assume that the relevant symbols are internalized by the computing units, so we assume that **the processors share the same signature** $\mathcal{L}(A)$. But we don't assume that they have the same universe. In fact, the data structures (ie the part of the language which does not depend on the execution) of two distinct processors are not identical but isomorphic.

Moreover, we are interested in the algorithm and not a particular implementation (for example the name of the objects), therefore in the following postulate we will consider the states up to multi-isomorphism:

Definition 3.1 (Multi-Isomorphism)**.** $\overrightarrow{\zeta}$ is a **multi-isomorphism** between two states $\left(X^1, \ldots, X^p\right)$ and $\left(Y^1, \ldots, Y^q\right)$ if $p = q$ and $\overrightarrow{\zeta}$ is a p-tuple of applications $\zeta_1, \ldots, \zeta_p$ such that for every $1 \leq i \leq p$, ζ_i is an isomorphism between X^i and Y^i.

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

1. The states of A are tuples of (first-order) structures with the same signature $\mathcal{L}(A)$, which is finite and contains the booleans and the equality
2. $S(A)$ and $I(A)$ are closed by multi-isomorphism
3. The transition function τ_A preserves the universes and the numbers of processors, and commutes with multi-isomorphisms

For a parallel algorithm A, let X be a local memory of A, $f \in \mathcal{L}(A)$ be a dynamic α-ary function symbol, and $a_1, \ldots, a_\alpha, b$ be elements of the universe $\mathcal{U}(X)$. We say that $(f, a_1, \ldots, a_\alpha)$ is a location of X, and that $(f, a_1, \ldots, a_\alpha, b)$ is an **update** on X at the location $(f, a_1, \ldots, a_\alpha)$.

For example, if x is a variable then $(x, 42)$ is an update at the location x. But symbols with arity $\alpha > 0$ can be updated too. For example, if f is a one-dimensional array, then $(f, 0, 42)$ is an update at the location $(f, 0)$.

If u is an update then $X \oplus u$ is a new structure of signature $\mathcal{L}(A)$ and universe $\mathcal{U}(X)$ such that the interpretation of a function symbol $f \in \mathcal{L}(A)$ is:

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

For example, in $X \oplus (f, 0, 42)$, every symbol has the same interpretation than in X, except maybe for f because $\overline{f}^{X \oplus (f,0,42)}(0) = 42$ and $\overline{f}^{X \oplus (f,0,42)}(a) = \overline{f}^{X}(a)$ otherwise. We precised "maybe" because it may be possible that $\overline{f}^{X}(0)$ is already 42. Indeed, if $\overline{f}^{X}(\overrightarrow{a}) = b$ then the update $(f, \overrightarrow{a}, b)$ is said **trivial** in X, because nothing has changed: in that case $X \oplus (f, \overrightarrow{a}, b) = X$.

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

$b \neq b'$. We assume in that case that the entire set of updates clashes:

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

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

Let $\overrightarrow{X} = \left(X^1, \ldots, X^p\right)$ be a state of the parallel algorithm A. According to the transition function τ_A, the next state is $\tau_A\left(X^1, \ldots, X^p\right)$, which will be denoted by $\left(\tau_A(\overrightarrow{X})^1, \ldots, \tau_A(\overrightarrow{X})^p\right)$.

We denote by $\Delta^i(A, \overrightarrow{X}) \stackrel{\text{def}}{=} \tau_A(\overrightarrow{X})^i \ominus X^i$ the set of updates done by the i-th processor of A on the state $\overrightarrow{X}$, and by $\overrightarrow{\Delta}(A, \overrightarrow{X}) \stackrel{\text{def}}{=} \left(\Delta^1(A, \overrightarrow{X}), \ldots, \Delta^p(A, \overrightarrow{X})\right)$ the **multiset of updates** done by A on the state $\overrightarrow{X}$.

In particular, if a state $\overrightarrow{X}$ is final, then $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$, so $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \overrightarrow{\emptyset}$.

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

The third postulate states that only a bounded number of terms can be read or updated during a computation step:

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

This $T(A)$ is called the **exploration witness** of A. As in [Gur00], we assume (without lost of generality) that $T(A)$ is closed by subterms.

We said at the beginning of the section that these three postulates are a "natural" extension of the three postulates of Gurevich. Indeed, we proved in [MG18] that an object verifying our postulates 1, 2 and 3 with $p = 1$ processor in every initial state is a sequential algorithms in the sense of [Gur00].

For a BSP algorithm, the sequence of states is organized by using supersteps. Notably, the communication between the processor memories occurs only during a communication phase. In order to do so, a BSP algorithm A will use two functions comp_A and comm_A indicating if the algorithm runs computations or runs communications (followed by a synchronization):

Postulate 4 (Superstep Phases)**.** For every BSP algorithm A there exists two applications $\text{comp}_A : M(A) \to M(A)$ commuting with isomorphisms, and $\text{comm}_A : S(A) \to S(A)$, such that for every state $\left(X^1, \ldots, X^p\right)$:

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

A **BSP algorithm** is an object verifying these four postulates, and we denote by ALGO_{BSP} the set of the BSP algorithms.

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

We remind that a state $\overrightarrow{X}$ is said final if $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$. Therefore, according to the fourth postulate, a **final state** must be in a communication phase such that $\text{comm}_A(\overrightarrow{X}) = \overrightarrow{X}$.

4 The ASM-BSP Model

The four postulates of the section 3 p.4 define the BSP algorithms with an axiomatic point of view, but that does not mean that they have a model. Or, in other words, that they are defined in an operational point of view.

In the same way that Gurevich proved in [Gur00] that his model of computation ASM (Abstract State Machines) captures the set of the sequential algorithms, we proved in the technical report [MG18] that the ASM_{BSP} model captures the BSP algorithms.

Definition 4.1 (ASM Program)**.**

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

where F is a formula, f has arity α and t_1, …, t_α, t_0 are terms.

Notice that if $n = 0$ then $\texttt{par } \Pi_1 \parallel \ \ldots \ \parallel \Pi_n \texttt{ endpar}$ is the empty program. If in the command $\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}$ the program Π_2 is empty we will write simply $\texttt{if } F \texttt{ then } \Pi_1 \texttt{ endif}$.

Definition 4.2 (Terms Read by an ASM program)**.**

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

An ASM machine is a kind of Turing Machine using not a tape but an abstract structure X:

Definition 4.3 (ASM Operational Semantics)**.**

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

Notice that the semantics of the `par` command is a set of updates done simultaneously, which differs from an usual imperative framework.

As for BSP algorithms, a state of an ASM_{BSP} machine will be a p-tuple of processors $\left(X^1, \ldots, X^p\right)$. We assume that ASM_{BSP} programs are SPMD (Single Program Multiple Data), which means that at each step of computation the program Π is executed individually on each processor. Therefore an ASM program Π induces a multiset of updates and a transition function:

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

If $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$, then every processor has finished its computation steps. In that case we assume that there exists a communication function to ensure the communications between processors:

Definition 4.4 (ASM_{BSP})**.** An **ASM_{BSP} machine** A is a triplet $(S(A), I(A), \tau_A)$ such that:

1. $S(A)$ is a set of tuples of structures with the same signature $\mathcal{L}(A)$, which is finite and contains the booleans and the equality, and $S(A)$ is closed by multi-isomorphism
2. $I(A) \subseteq S(A)$ is closed by multi-isomorphism, and is the set of the initial states
3. $\tau_A : S(A) \to S(A)$ verifies that there exists an ASM program Π and an application $\text{comm}_A : S(A) \to S(A)$ such that:

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

4. comm_A verifies that:
   (a) For every state $\overrightarrow{X}$ such that $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$, comm_A preserves the universes and the number of processors, and commutes with multi-isomorphisms
   (b) There exists a finite set of terms $T(\text{comm}_A)$ such that for every state $\overrightarrow{X}$ and $\overrightarrow{Y}$ with $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$ and $\tau_\Pi(\overrightarrow{Y}) = \overrightarrow{Y}$, if they coincide over $T(\text{comm}_A)$ then $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \overrightarrow{\Delta}(A, \overrightarrow{Y})$.

We denote by ASM_{BSP} the set of the ASM_{BSP} machines. As for the BSP algorithms, a state $\overrightarrow{X}$ is said final if $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$. So, by the previous definition, if $\overrightarrow{X}$ is **final** then $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$ and $\text{comm}_A(\overrightarrow{X}) = \overrightarrow{X}$.

The last conditions about the communication function may seem arbitrary, but they are required to ensure that the communication function is not a kind of magic device. We discussed some issues and we constructed an example of such communication function in [MG18].

Proposition 4.5 (Computations of BSP Algorithms are BSP ASMs)**.** *For every* BSP *algorithm* A*, there exists an* ASM *program* Π_A *such that for every state* $\overrightarrow{X}$ *in a computation phase:*

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

This ASM program Π_A has a **normal form**:

```
      if F_1 then Π_1
else  if F_2 then Π_2
⋮
else  if F_c then Π_c
      endif ... endif
```

where for every local memory X from a state in a computing phase, one and only one of these formulas $F_1, \ldots, F_c$ is true, and the ASM programs $\Pi_1, \ldots, \Pi_c$ are simultaneous update commands `par` $u_1 \parallel \ldots \parallel u_m$ `endpar` producing distinct non-clashing sets of non-trivial updates.

By using this proposition, we proved in [MG18] that the axiomatic presentation of ALGO_{BSP} and the operational presentation of ASM_{BSP} corresponds to the same set of objects:

Theorem 4.6. $\text{ALGO}_{\text{BSP}} = \text{ASM}_{\text{BSP}}$

Corollary 4.7. *Every* ASM_{BSP} *program has a normal form.*

Proof. According to the previous theorem, an ASM_{BSP} is a BSP algorithm, and every BSP algorithm is captured by an ASM_{BSP} with a program in normal form. Thus, our initial ASM_{BSP} is equal to an ASM_{BSP} with a program in normal form. □

5 The While-BSP Model

In this section we present a SPMD (Single Program Multiple Data) version of a WHILE language, because this programming language is minimal. Indeed, the WHILE programs are only sequential updates, conditionals and unbounded loop. So, if $\text{WHILE}_{\text{BSP}}$ is algorithmically equivalent to the BSP algorithms then every imperative language containing these control flow commands are algorithmically complete (in the sense of our definition p.12) for the BSP algorithms.

As in ASM_{BSP}, this $\text{WHILE}_{\text{BSP}}$ language uses first-order structures as data structures, and not a particular architecture: this is not a concrete language but an abstract one. Any concrete data structure which can be used for ASM_{BSP} can be used for $\text{WHILE}_{\text{BSP}}$, and vice versa. In other words, in this paper we prove that both models of computations are equivalent, up to data structures. So, our main theorem p.14 and p.18 is a statement about control flow, and not data structures.

Moreover, as in ASM_{BSP}, processors of a $\text{WHILE}_{\text{BSP}}$ program will compute locally until the machine requires a communication step, which will be indicated with the command `comm`:

Definition 5.1 (WHILE Programs)**.**

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

where F is a formula, f has arity α and θ_1, …, θ_α, θ_0 are terms.

Similarly to the ASM programs, we write only `if` F $\{P\}$ for `if` F $\{P\}$ `else` $\{$`end`$\}$. Moreover, the composition of commands $c; P$ is extended to **composition of programs** $P_1; P_2$ by $\texttt{end}; P_2 \stackrel{\text{def}}{=} P_2$ and $(c; P_1); P_2 \stackrel{\text{def}}{=} c; (P_1; P_2)$.

The operational semantics of WHILE is formalized by a state transition system, where a state of the system is a pair $P \star X$ of a WHILE program P and a structure X, and a transition is determined only by the head command and the current structure:

Definition 5.2 (Operational Semantics of WHILE)**.**

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

$$\texttt{if } F \texttt{ } \{P_1\} \texttt{ else } \{P_2\}; P_3 \star X \succ P_i; P_3 \star X \quad \text{where } \begin{cases} i = 1 & \text{if } F \text{ is true in } X \\ i = 2 & \text{otherwise} \end{cases}$$

$$\texttt{while } F \texttt{ } \{P_1\}; P_2 \star X \succ P \star X \quad \text{where } \begin{cases} P = P_1; \texttt{while } F \texttt{ } \{P_1\}; P_2 & \text{if } F \text{ is true in } X \\ P = P_2 & \text{otherwise} \end{cases}$$

This transition system is deterministic. We denote by $\succ_t$ the succession of t steps. The only states without successor are $\texttt{comm}; P \star X$ and $\texttt{end} \star X$. We say that a WHILE program P **terminates locally** on a processor X, which will be denoted by $P \downarrow X$, if there exists t, P' and X' such that $P \star X \succ_t \texttt{comm}; P' \star X'$ or $P \star X \succ_t \texttt{end} \star X'$. This t will be denoted by $\text{time}(P, X)$, and this X' by $P(X)$. If P does not terminate locally on X, which will be denoted by $P \uparrow X$, then we assume that $\text{time}(P, X) = \infty$. We proved in [Mar18] that:

Proposition 5.3 (Composition of Programs)**.** *$P_1; P_2 \downarrow X$ if and only if $P_1 \downarrow X$ and $P_2 \downarrow P_1(X)$, such that:*

1. $P_1; P_2(X) = P_2(P_1(X))$
2. $\text{time}(P_1; P_2, X) = \text{time}(P_1, X) + \text{time}(P_2, P_1(X))$

For every $t \leq \text{time}(P, X)$, there exists a unique P_t and X_t such that $P \star X \succ_t P_t \star X_t$. This P_t will be denoted by $\tau_X^t(P)$, and this X_t by $\tau_P^t(X)$. Notice that τ_P is not a transition function in the sense of the first postulate p.4 because $\tau_P^t(X) \neq \tau_P^1 \circ \cdots \circ \tau_P^1(X)$. We denote by $\Delta(P, X)$ the succession of updates made by P on X:

$$\Delta(P, X) \stackrel{\text{def}}{=} \bigcup_{0 \leq t < time(P,X)} \tau_P^{t+1}(X) \ominus \tau_P^t(X)$$

Unlike the ASM, only one update can be done at each step. Therefore, if $P \downarrow X$ then $\Delta(P, X)$ is finite. Moreover, we say that P is **without overwrite** on X if $\Delta(P, X)$ is consistent (see p.5). We proved in [Mar18] that:

Lemma 5.4 (Updates without overwrite)**.** *If $P \downarrow X$ without overwrite then $\Delta(P, X) = P(X) \ominus X$.*

As in ASM_{BSP}, processors of a $\text{WHILE}_{\text{BSP}}$ program will compute locally until the beginning of the communication phase. The operational semantics of $\text{WHILE}_{\text{BSP}}$ is formalized by a state transition system, where a state of the system is a pair $\overrightarrow{P} \star \overrightarrow{X}$ of a p-tuple $\overrightarrow{P} = \left(P^1, \ldots, P^p\right)$ of WHILE programs, and a p-tuple $\overrightarrow{X} = \left(X^1, \ldots, X^p\right)$ of structures.

$\overrightarrow{P}$ will be said in a **computation phase** if there exists $1 \leq i \leq p$ such that $P^i \neq \texttt{end}$ and for every P, $P^i \neq \texttt{comm}; P$. Otherwise, $\overrightarrow{P}$ will be said in a **communication phase**. We define:

$$\begin{array}{l} \overrightarrow{\tau}_{\overrightarrow{X}}(\overrightarrow{P}) \stackrel{\text{def}}{=} \left(\tau_{X^1}(P^1), \ldots, \tau_{X^p}(P^p)\right) \\ \overrightarrow{\tau}_{\overrightarrow{P}}(\overrightarrow{X}) \stackrel{\text{def}}{=} \left(\tau_{P^1}(X^1), \ldots, \tau_{P^p}(X^p)\right) \end{array}$$

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

$$\begin{array}{ll} P \star X \succ \tau_X(P) \star \tau_P(X) & \text{if } P \star X \text{ has a successor} \\ P \star X = \tau_X(P) \star \tau_P(X) & \text{otherwise} \end{array}$$

Definition 5.5 ($\text{WHILE}_{\text{BSP}}$)**.** A **$\text{While}_{\text{BSP}}$ machine** M is a quadruplet $(S(M), I(M), P, \text{comm}_M)$ verifying that:

1. $S(M)$ is a set of tuples of structures with the same signature $\mathcal{L}(M)$, which is finite and contains the booleans and the equality
2. $I(M) \subseteq S(M)$, and the initial states of the transition system have the form $P \star X$, where $X \in I(M)$
3. P is a WHILE program with terms from $\mathcal{L}(M)$
4. $\text{comm}_M : S(M) \mapsto S(M)$ is an application which preserves the universes and the number of processors, and admits an exploration witness as in the definition 4.4 p.8.

The operational semantics of $\text{WHILE}_{\text{BSP}}$ is defined by:

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

where $\overrightarrow{\text{next}}(P^1, \ldots, P^p) = \big(\text{next}(P^1), \ldots, \text{next}(P^p)\big)$, with $\text{next}(\texttt{comm}; P) = P$ and $\text{next}(\texttt{end}) = \texttt{end}$.

Notice that, according to this definition, a state $\overrightarrow{X}$ of a $\text{WHILE}_{\text{BSP}}$ machine M is **final** if every program of the state transition system is `end` and if $\text{comm}_M(\overrightarrow{X}) = \overrightarrow{X}$.

6 Fair Simulation

In this section, we call **model of computation** a set of programs associated with an operational semantics, like our definition of ASM_{BSP} p.8 and our definition of $\text{WHILE}_{\text{BSP}}$ p.10.

Sometimes a literal identity can be proven between two models of computation. For example, Serge Grigorieff and Pierre Valarcher proved in [GV12] that classes of Evolving MultiAlgebras (a variant of Gurevich's ASMs) can be identified to Turing Machines, Random Access Machines or other sequential models of computation. But generally, only a simulation can be proven between two models of computation.

We say that a model of computation M_1 simulates another model M_2 if for every program P_2 of M_2 there exists a program P_1 of M_1 producing "similar" executions from the initial states. By "similar" we mean up to a bounded number of fresh variables and up to temporal dilation, concepts which are introduced in the following examples:

Example 6.1 (Fresh Variables)**.** In these common examples, in order to simulate the same behavior, a **fresh variable** is introduced in the code:

1. `loop` n $\{P\}$ can be simulated by $i := 0$; `while` $i \neq n$ $\{P;\ i := i + 1;\}$, by using a fresh variable i.
2. An exchange $x \leftrightarrow y$ between two variables x and y can be simulated by $v := x;\ x := y;\ y := v$, by using a fresh variable v.

So, the signature $\mathcal{L}_1$ of the simulating program is an extension of the signature $\mathcal{L}_2$ of the simulated program, in other words $\mathcal{L}_1 \supseteq \mathcal{L}_2$. The signature must remain finite, so $\mathcal{L}_1 \setminus \mathcal{L}_2$ must be finite, but this is not enough. Using a function symbol with arity $\alpha > 0$ would be unfair because it could store an unbounded amount of information. So, in our definition 6.5 p.12 of the simulation we will only assume that $\mathcal{L}_1 \setminus \mathcal{L}_2$ is a finite set of variables. Moreover, to prevent the introduction of new information, we will assume that these "fresh" variables are uniformly initialized (ie their value is the same in the initial processors, up to isomorphism).

Definition 6.2 (Restriction of Structure)**.** Let X be a structure with signature $\mathcal{L}_1$, and let $\mathcal{L}_2$ be a signature such that $\mathcal{L}_1 \supseteq \mathcal{L}_2$. The **restriction** of X to the signature $\mathcal{L}_2$ will be denoted $X|_{\mathcal{L}_2}$ and is defined as a structure of signature $\mathcal{L}_2$ such that $\mathcal{U}(X|_{\mathcal{L}_2}) = \mathcal{U}(X)$ and for every $f \in \mathcal{L}_2$ we have $\overline{f}^{X|_{\mathcal{L}_2}} = \overline{f}^{X}$. In the other way, X will be called an **extension** of $X|_{\mathcal{L}_2}$.

Example 6.3 (Temporal Dilation)**.** During every "step" of a Turing machine, the state of the machine is updated, a symbol is written in the cell, and the head may move left or right. But the notion of elementary action is very arbitrary. We may consider either a machine M_3 requiring three steps to do these three elementary actions, or a machine M_1 requiring only one step to do

the three aspects of one multi-action. An execution $\overrightarrow{X}$ of M_3 corresponds to an execution $\overrightarrow{Y}$ of M_1 if for every three steps of M_3 the state is the same than M_1:

M_3	$X_0, X_1, X_2, X_3, X_4, X_5, X_6, X_7, X_8, \ldots$
M_1	$Y_0, \quad Y_1, \quad Y_2, \quad \ldots$

In other words, if M_3 is three times faster than M_1, these execution may be considered equivalent. For every t, we have $X_{3t} = Y_t$, so we say that there is a **temporal dilation** $d = 3$. In this example, the temporal dilation is uniform for every program, but in the definition 6.5 p.12 this temporal dilation may depend on the simulated program.

Example 6.4 (Ending Time)**.** Notice that the temporal dilation is not sufficient to ensure that the final states are the same. Indeed, in the previous example, even if Y_t is terminal for M_1, we have an infinite loop for M_3:

M_3	$\ldots, X_{3t}, X_{3t+1}, X_{3t+2}, X_{3t}, X_{3t+1}, X_{3t+2}, \ldots$
M_1	$\ldots, \quad Y_t, \quad Y_t, \quad \ldots$

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

Definition 6.5 (Fair Simulation)**.** Let M_1 and M_2 be two models of computation. We say that M_1 **simulates** M_2 if for every program P_2 of M_2 there exists a program P_1 of M_1 such that:

1. $\mathcal{L}(P_1) \supseteq \mathcal{L}(P_2)$, and $\mathcal{L}(P_1) \setminus \mathcal{L}(P_2)$ is a finite set of fresh variables depending only on P_2, and uniformly initialized

and if there exists $d > 0$ and $e \geq 0$ depending only on P_2 such that for every execution $S'_0, S'_1, S'_2, \ldots$ of P_2 there exists an execution $S_0, S_1, S_2, \ldots$ of P_1 verifying that:

2. for every $t \in \mathbb{N}$, $S_{d \times t}|_{\mathcal{L}(P_2)} = S'_t$
3. $\text{time}(P_1, S_0) = d \times \text{time}(P_2, S'_0) + e$

If M_1 simulates M_2 and M_2 simulates M_1, then these models of computation will be said **algorithmically equivalent**, which will be denoted by $M_1 \simeq M_2$.

Notice that the parameters of the simulation, ie the fresh variables, the temporal dilation and the ending time, depends on what is simulated and not on a particular state of the execution.

Moreover, if M_1 and M_2 are algorithmically equivalent, and M_2 is a set of algorithms (as the set of BSP algorithms defined at the section 3 p.4) then we will say that the model of computation M_1 is **algorithmically complete** for the class of algorithms M_2.

Because in this report we prove p.14 and p.18 that $\text{WHILE}_{\text{BSP}} \simeq \text{ASM}_{\text{BSP}}$, and because we proved in [MG18] that $\text{ASM}_{\text{BSP}} = \text{ALGO}_{\text{BSP}}$, that means that our minimal model $\text{WHILE}_{\text{BSP}}$ defined p.10 is algorithmically complete for the BSP algorithms.

7 ASM-BSP simulates While-BSP

The simplest idea to translate a WHILE program P into an ASM program is to add a variable for the current line of the program and follow the flow of execution. We detail in [Mar18] why this naive solution does not work as intended. Instead, we will add a bounded number of boolean variables $b_{P_1}, \ldots, b_{P_k}$, where $P_1, \ldots, P_k$ are the programs occurring during a possible execution of the simulated program P.

The set of these programs will be called the **control flow graph** $\mathcal{G}(P)$ of the program P. Before defining it we must introduce the following notation:

$$\mathcal{G}(P_1); P_2 \stackrel{\text{def}}{=} \{P; P_2 \mid P \in \mathcal{G}(P_1)\}$$

Definition 7.1 (Control Flow Graph)**.**

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

We prove in [Mar18] that $\text{card}(\mathcal{G}(P)) \leq \text{length}(P) + 1$, where the length of a WHILE program P is defined in the usual way. Therefore, the number of boolean variables b_{P_i}, where $P_i \in \mathcal{G}(P)$, is bounded by a number depending only of P.

A processor X of the $\text{WHILE}_{\text{BSP}}$ machine M will be simulated by the ASM machine A by using the same processor X but with new symbols $\mathcal{L}(A) = \mathcal{L}(M) \cup \{b_{P_i} \mid P_i \in \mathcal{G}(P)\}$ such that one and only one of the b_{P_i} is true. This extension of the structure X will be denoted by $X[b_{P_i}]$.

An initial processor X for the WHILE program P will be simulated by the processor $X[b_P]$, and during the execution the boolean variable which is true at a given step of a given processor is determined by the translation Π_P of the program P:

Definition 7.2 (Translation of a WHILE Program)**.**

$$\Pi_P \stackrel{\text{def}}{=} \texttt{par } \|_{P_i \in \mathcal{G}(P)} \texttt{ if } b_{P_i} \texttt{ then } [\![P_i]\!]^{\text{ASM}} \texttt{ endif endpar}$$

where the translation $[\![P_i]\!]^{\text{ASM}}$ of the first step of P_i is defined at the table 1 p.14.

Notice that we distinguished two cases for the translation of the `while` command. Indeed, if we did not, then the ASM would have tried to do two updates $b_{\texttt{while } F \texttt{ \{}P_1\texttt{\};}P_2} :=$ false and $b_{P_1;\texttt{while } F \texttt{ \{}P_1\texttt{\};}P_2} :=$ true. But, in the case where the body $P_1 = \texttt{end}$, both would have clashed and the ASM would have stopped. This behavior is not compatible with the WHILE program, which would had looped forever. To preserve this behavior, we modified the translation and added a fresh variable b_∞ which prevents the ASM from terminating. An other solution would have been to forbid empty body for the `while` commands in the definition 5.1 p.9, as it is done in [Mar18] and in the submitted version of this report.

Therefore, we can prove that the ASM program Π_P simulates step-by-step the WHILE program P:

Proposition 7.3 (step-by-step Simulation of a WHILE program)**.** *For every* $0 \leq t < \text{time}(P, X)$*:*

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

$$
\begin{array}{rcl}
[\![\texttt{end}]\!]^{\mathrm{ASM}} & \stackrel{\text{def}}{=} & \texttt{par endpar} \\
[\![\texttt{comm}; P]\!]^{\mathrm{ASM}} & \stackrel{\text{def}}{=} & \texttt{par endpar} \\
& & \quad \texttt{par } b_{f(\theta_1,\ldots,\theta_\alpha):=\theta_0;P} := \text{false} \\
[\![f(\theta_1,\ldots,\theta_\alpha) := \theta_0; P]\!]^{\mathrm{ASM}} & \stackrel{\text{def}}{=} & \quad\quad \| \; f(\theta_1,\ldots,\theta_\alpha) := \theta_0 \\
& & \quad\quad \| \; b_P := \text{true} \\
& & \texttt{endpar} \\
& & \quad \texttt{par } b_{\texttt{if } F \texttt{ \{}P_1\texttt{\} else \{}P_2\texttt{\};}P_3} := \text{false} \\
& & \quad\quad \texttt{if } F \texttt{ then } b_{P_1;P_3} := \text{true} \\
[\![\texttt{if } F \texttt{ \{}P_1\texttt{\} else \{}P_2\texttt{\};} P_3]\!]^{\mathrm{ASM}} & \stackrel{\text{def}}{=} & \quad\quad \| \quad\quad \texttt{else } b_{P_2;P_3} := \text{true} \\
& & \quad\quad \texttt{endif} \\
& & \texttt{endpar} \\
& & \texttt{if } F \texttt{ then } b_\infty := \neg b_\infty \\
& & \qquad\qquad \texttt{par } b_{\texttt{while } F \texttt{ \{end\};}P} := \text{false} \\
[\![\texttt{while } F \texttt{ \{end\};} P]\!]^{\mathrm{ASM}} & \stackrel{\text{def}}{=} & \quad \texttt{else} \quad\quad \| \; b_P := \text{true} \\
& & \qquad\quad \texttt{endpar} \\
& & \texttt{endif} \\
& & \quad \texttt{par } b_{\texttt{while } F \texttt{ \{}c;P_1\texttt{\};}P_2} := \text{false} \\
& & \quad\quad \texttt{if } F \texttt{ then } b_{c;P_1;\texttt{while } F \texttt{ \{}c;P_1\texttt{\};}P_2} := \text{true} \\
[\![\texttt{while } F \texttt{ \{}c; P_1\texttt{\};} P_2]\!]^{\mathrm{ASM}} & \stackrel{\text{def}}{=} & \quad\quad \| \quad\quad \texttt{else } b_{P_2} := \text{true} \\
& & \quad\quad \texttt{endif} \\
& & \texttt{endpar}
\end{array}
$$

Table 1: Translation of one step of a WHILE program

Proof. The proof is made by case on $\tau_X^t(P)$, similarly to the proof in the long version of [Mar18], except that there is a new command `comm` and new terminal states $\texttt{comm}; P \star X$. □

Therefore, we can prove that $\mathrm{ASM}_{\mathrm{BSP}}$ simulates $\mathrm{WHILE}_{\mathrm{BSP}}$ in the sense of the definition 6.5 p.12, with at most $\mathrm{length}(P) + 2$ fresh variables, a temporal dilation $d = 1$ and an ending time $e = 0$. Such simulation is said to be **strictly step-by-step** (see the discussion at section 9 p.19).

Theorem 7.4. $\mathrm{ASM}_{\mathrm{BSP}}$ *simulates* $\mathrm{WHILE}_{\mathrm{BSP}}$.

Proof. Let $M = (S(M), I(M), P, \mathrm{comm}_M)$ be a $\mathrm{WHILE}_{\mathrm{BSP}}$ machine, and let $A = (S(A), I(A), \tau_A)$ be an $\mathrm{ASM}_{\mathrm{BSP}}$ machine such that:

- $S(A)$ is an extension of $S(M)$ by using the boolean variables b_{P_i} with $P_i \in \mathcal{G}(P)$, and b_∞.
- Every $X \in I(A)$ is a $X|_{\mathcal{L}(M)} \in I(M)$ such that only b_P is true.
- The ASM program of the machine A is the program Π_P defined p.13.
- comm_A is comm_M for the symbols of $\mathcal{L}(M)$, and for the boolean variables b_{P_i} and b_∞:
  - if $b_{\texttt{comm};P}$ is true in a processor then $b_{\texttt{comm};P}$ becomes false and b_P becomes true,
  - otherwise the boolean variables b_{P_i} and b_∞ remain unchanged.

According to the definition of the simulation p.12, there is three points to prove:

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

Until a communication phase, the ASM program Π_P is applied to every processor, and according to the proposition 7.3 the execution corresponds to the operational semantics of the WHILE programs defined p.9.

A processor terminates if $b_{\texttt{comm};P}$ or $b_{\texttt{end}}$ is true. When every processor has terminated, the state is in a communication phase and comm_A apply the communication comm_M and moves the boolean variables from $b_{\texttt{comm};P}$ to b_P, thus respecting the behavior of the application next p.10.

2. So, one step of the $\text{WHILE}_{\text{BSP}}$ machine M is simulated by $d = 1$ step of the ASM_{BSP} machine A.

3. Moreover, a state $\overrightarrow{X}$ of the ASM_{BSP} machine A is final if $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{X}$ and $\text{comm}_A(\overrightarrow{X}) = \overrightarrow{X}$, and this happens if and only if $b_{\texttt{end}}$ is true for every processor and $\text{comm}_M(\overrightarrow{X}) = \overrightarrow{X}$. Therefore, the ASM_{BSP} machine A stops if and only if the $\text{WHILE}_{\text{BSP}}$ machine M stops, and the ending time is $e = 0$.

□

8 While-BSP simulates ASM-BSP

We prove the simulation in two steps:

1. We translate the ASM program Π into an imperative program P_Π^{step} simulating one step of Π.

2. Then, we construct a WHILE program P_Π which repeats P_Π^{step} during the computation phase, and detects the communication phase by using a formula F_Π^{end}.

Because the ASM and WHILE programs have the same updates and conditionals, an intuitive translation $[\![\Pi]\!]^{\text{WHILE}}$ of an ASM program Π could be:

Definition 8.1 (Naive translation of one step of ASM)**.**

$$
\begin{aligned}
[\![f(\theta_1,\ldots,\theta_\alpha) := \theta_0]\!]^{\text{WHILE}} &\stackrel{\text{def}}{=} f(\theta_1,\ldots,\theta_\alpha) := \theta_0;\texttt{end}\\
[\![\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}]\!]^{\text{WHILE}} &\stackrel{\text{def}}{=} \texttt{if } F \texttt{ } \{[\![\Pi_1]\!]^{\text{WHILE}}\} \texttt{ else } \{[\![\Pi_2]\!]^{\text{WHILE}}\};\texttt{end}\\
[\![\texttt{par } \Pi_1 \parallel \ldots \parallel \Pi_n \texttt{ endpar}]\!]^{\text{WHILE}} &\stackrel{\text{def}}{=} [\![\Pi_1]\!]^{\text{WHILE}};\ldots;[\![\Pi_n]\!]^{\text{WHILE}}
\end{aligned}
$$

But an ASM program like $\texttt{par } x := y \parallel y := x \texttt{ endpar}$ which exchanges the values of the variables x and y is translated into $x := y; y := x;\texttt{end}$ which sets the value of x to the value of y, and leaves y unchanged. To capture the simultaneous behavior of the ASM programs, we need to substitute the terms read in Π (see the definition 4.2 p.7) by fresh variables:

Definition 8.2 (Substitution of a term by a variable)**.**

$$
\begin{aligned}
(f(\theta_1,\ldots,\theta_\alpha) := \theta_0)[v/\theta] &\stackrel{\text{def}}{=} f(\theta_1[v/\theta],\ldots,\theta_\alpha[v/\theta]) := \theta_0[v/\theta]\\
(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif})[v/\theta] &\stackrel{\text{def}}{=} \texttt{if } F[v/\theta] \texttt{ then } \Pi_1[v/\theta] \texttt{ else } \Pi_2[v/\theta] \texttt{ endif}\\
(\texttt{par } \Pi_1 \parallel \ldots \parallel \Pi_n \texttt{ endpar})[v/\theta] &\stackrel{\text{def}}{=} \texttt{par } \Pi_1[v/\theta] \parallel \ldots \parallel \Pi_n[v/\theta] \texttt{ endpar}\\
\text{where } \theta_1[v/\theta_2] &\stackrel{\text{def}}{=} \begin{cases} v & \text{if } \theta_1 = \theta_2 \\ \theta_1 & \text{otherwise} \end{cases}
\end{aligned}
$$

Because these variables are fresh, if θ_1 and θ_2 are distinct terms then $\Pi[v_1/\theta_1][v_2/\theta_2] = \Pi[v_2/\theta_2][v_1/\theta_1]$. Let $\theta_1, \ldots, \theta_r$ be the terms of $\mathrm{Read}(\Pi)$ defined p.7. For every term $\theta \in \mathrm{Read}(\Pi)$, we add to $\mathcal{L}(\Pi)$ a fresh variable v_θ, and we denote by $[\overrightarrow{v_\theta}/\overrightarrow{\theta}]$ the successive substitutions $[v_{\theta_1}/\theta_1] \ldots [v_{\theta_r}/\theta_r]$.

The WHILE program $v_{\theta_1} := \theta_1; \ldots; v_{\theta_r} := \theta_r; [\![\Pi]\!]^{\text{WHILE}}[\overrightarrow{v_\theta}/\overrightarrow{\theta}]$ captures the simultaneous behavior of the ASM program Π. But, according to our definition of the simulation p.12, one step of the ASM program Π must be translated into exactly d steps of the WHILE program. Thus, we must ensure that $[\![\Pi]\!]^{\text{WHILE}}[\overrightarrow{v_\theta}/\overrightarrow{\theta}]$ computes the same number of steps for every execution.

According to the corollary 4.7 p.8, we can assume that the ASM program Π is in normal form:

$$\begin{array}{ll} & \texttt{if } F_1 \texttt{ then } \Pi_1 \\ \texttt{else} & \texttt{if } F_2 \texttt{ then } \Pi_2 \\ \vdots & \\ \texttt{else} & \texttt{if } F_c \texttt{ then } \Pi_c \\ & \texttt{endif} \ldots \texttt{endif} \end{array}$$

where for every local memory X from a state in a computing phase, one and only one of these formulas $F_1, \ldots, F_c$ is true, and the programs Π_i have the form $\texttt{par } u_1^i \parallel \ldots \parallel u_{m_i}^i \texttt{ endpar}$ where $u_1^i, \ldots, u_{m_i}^i$ are m_i update commands, and produce distinct non-clashing sets of non-trivial updates (the proof is made in [MG18]).

Let $m = \max_{1 \leq i \leq c}\{m_i\}$. We pad (as in [FZG10]) every block of m_i update commands by using a $\texttt{skip } n$ command, defined by induction:

$$\begin{array}{lcl} \texttt{skip } 0 & \stackrel{\text{def}}{=} & \texttt{end} \\ \texttt{skip } n+1 & \stackrel{\text{def}}{=} & \texttt{if true } \{\texttt{end}\}; \texttt{skip } n \end{array}$$

The translation P_Π^{step} of one step of the ASM program Π is given explicitly at the table 2 p.17, and we prove that it requires exactly $r + c + m$ steps (which depends only of Π) for every extension X of a processor of the ASM_{BSP} machine with the fresh variables $\{v_\theta \mid \theta \in \mathrm{Read}(\Pi)\}$.

Proposition 8.3 (Translation of one step of an ASM program)**.**

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

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

Proof. The proof is done in [Mar18] by using the lemma 5.4 p.10. Indeed, P_Π^{step} is terminal because it contains no loop, and it is without overwrite on X because Π is in normal form. □

Therefore, we can prove by induction on t that for every processor X in a computation phase we have $\overbrace{P_\Pi^{\text{step}} \circ \cdots \circ P_\Pi^{\text{step}}}^{t \text{ times}}(X) = \tau_\Pi^t(X|_{\mathcal{L}(\Pi)})$. We aim to repeat P_Π^{step} until the end of the computation phase which is, according to the definition 4.4 p.8, the termination of Π.

Because every update block Π_i of the ASM program Π in normal form produces a non-clashing set of non-trivial updates, if Π can terminate then there exists a Π_i which is the empty program $\texttt{par endpar}$. Moreover, because the blocks produce distinct set of updates, this Π_i is the only empty block of updates in Π. Therefore, we can define the termination formula F_Π^{end}:

$\Pi =$	$P_\Pi^{\text{step}} \stackrel{\text{def}}{=}$
	$v_{\theta_1} := \theta_1;$
	$v_{\theta_2} := \theta_2;$
	$\vdots$
	$v_{\theta_r} := \theta_r;$
`if` F_1 `then`	`if` v_{F_1} {
`par` $f(\overrightarrow{\theta_1^1}) := \theta_1^1$	$f(\overrightarrow{v_{\theta_1^1}}) := v_{\theta_1^1};$
$\|\| f(\overrightarrow{\theta_2^1}) := \theta_2^1$	$f(\overrightarrow{v_{\theta_2^1}}) := v_{\theta_2^1};$
$\|\| \vdots$	$\vdots$
$\|\| f(\overrightarrow{\theta_{m_1}^1}) := \theta_{m_1}^1$	$f(\overrightarrow{v_{\theta_{m_1}^1}}) := v_{\theta_{m_1}^1};$
`endpar`	`skip` $(c+m)-(1+m_1)$ }
`else if` F_2 `then`	`else` { `if` v_{F_2} {
`par` $f(\overrightarrow{\theta_1^2}) := \theta_1^2$	$f(\overrightarrow{v_{\theta_1^2}}) := v_{\theta_1^2};$
$\|\| f(\overrightarrow{\theta_2^2}) := \theta_2^2$	$f(\overrightarrow{v_{\theta_2^2}}) := v_{\theta_2^2};$
$\|\| \vdots$	$\vdots$
$\|\| f(\overrightarrow{\theta_{m_2}^2}) := \theta_{m_2}^2$	$f(\overrightarrow{v_{\theta_{m_2}^2}}) := v_{\theta_{m_2}^2};$
`endpar`	`skip` $(c+m)-(2+m_2)$ }
$\vdots$	$\vdots$
`else if` F_c `then`	`else` { `if` v_{F_c} {
`par` $f(\overrightarrow{\theta_1^c}) := \theta_1^c$	$f(\overrightarrow{v_{\theta_1^c}}) := v_{\theta_1^c};$
$\|\| f(\overrightarrow{\theta_2^c}) := \theta_2^c$	$f(\overrightarrow{v_{\theta_2^c}}) := v_{\theta_2^c};$
$\|\| \vdots$	$\vdots$
$\|\| f(\overrightarrow{\theta_{m_c}^c}) := \theta_{m_c}^c$	$f(\overrightarrow{v_{\theta_{m_c}^c}}) := v_{\theta_{m_c}^c};$
`endpar`	`skip` $(c+m)-(c+m_c)$ }
`endif` ... `endif`	; `end` } ... ; `end` } ; `end`

Table 2: Translation P_Π^{step} of one step of the ASM program Π

Definition 8.4 (The termination formula)**.**

$$F_\Pi^{\text{end}} \stackrel{\text{def}}{=} \begin{cases} F_i & \text{if there exists } 1 \le i \le c \text{ such that } \Pi_i = \texttt{par endpar} \\ \text{false} & \text{otherwise} \end{cases}$$

Lemma 8.5 (Correctness of the termination formula)**.**

$$\min \left\{ t \in \mathbb{N} \mid \overline{F_\Pi^{end}} \overbrace{P_\Pi^{step} \circ \cdots \circ P_\Pi^{step}}^{t \text{ times}}(X) = true \right\} = \text{time}(\Pi, X|_{\mathcal{L}(\Pi)})$$

Proof. The proof is made by using the definition 8.4 p.17 of the termination formula, based on

the fact that Π is in normal form, and the proposition 8.3 p.16 establishing that the program P_Π^{step} simulates one step of the ASM program Π. $\square$

So, a computation phase of the ASM program Π can be simulated by a WHILE program like `while` $\neg F_\Pi^{\text{end}}$ $\{P_\Pi^{\text{step}}\}$; `end`.

According to the definition of the ASM_{BSP} p.8, the communication phase begins at the termination of the ASM program Π, and continues until Π can do the first update of the next computation phase, ie when F_Π^{end} becomes false.

The execution ends when F_Π^{end} is true and the communication function comm_A changes nothing. By using the exploration witness $T(\text{comm}_A)$ (defined p.8) of the communication function, the processor can locally know whether the communication function has updated it or not at the last step, but it cannot know if the communication function has globally terminated. Therefore, we use a fresh boolean $b^{\text{end}}_{\text{comm}_A}$ updated by the communication function comm_A itself to detect if the communication phase has terminated.

Let P_Π be the WHILE program defined at the table 3 p.18, which is the translation of the ASM program Π. Notice that `comm` is executed only if F_Π^{end} is true. That means that the fresh boolean $b^{\text{end}}_{\text{comm}_A}$ can be set to true only if F_Π^{end} is true and comm_A has terminated, which indicates the end of the execution.

$$
\begin{array}{l}
P_\Pi \stackrel{\text{def}}{=} \texttt{while } \neg b^{\text{end}}_{\text{comm}_A} \texttt{ \{} \\
\quad \texttt{if } F_\Pi^{\text{end}} \texttt{ \{} \\
\quad\quad \texttt{skip } (r+c+m)-1; \\
\quad\quad \texttt{comm;} \\
\quad\quad \texttt{end \}} \\
\quad \texttt{else \{} \\
\quad\quad P_\Pi^{\text{step}} \texttt{ \};} \\
\quad \texttt{end \};} \\
\texttt{end}
\end{array}
$$

Table 3: Translation P_Π of the ASM program Π

Theorem 8.6. $\text{WHILE}_{\text{BSP}}$ *simulates* ASM_{BSP}.

Proof. Let $A = (S(A), I(A), \tau_A)$ be an ASM_{BSP} machine, with an ASM program Π and a communication function comm_A, and let $M = (S(M), I(M), P, \text{comm}_M)$ be a $\text{WHILE}_{\text{BSP}}$ machine such that:

- $S(M)$ is an extension of $S(A)$ by using the fresh variables $v_{\theta_1}, \ldots, v_{\theta_r}$ with $\{\theta_1, \ldots, \theta_r\} = \text{Read}(\Pi)$, and the fresh boolean $b^{\text{end}}_{\text{comm}_A}$.
- Every $X \in I(M)$ is a $X|_{\mathcal{L}(A)} \in I(A)$, such that the fresh variables are initialized with the value of undef, and the fresh boolean $b^{\text{end}}_{\text{comm}_A}$ to false.
- The WHILE program P of the machine M is the program P_Π defined at the table 3 p.18.
- comm_M is comm_A for the symbols of $\mathcal{L}(A)$, and for the fresh variables:
  - comm_M leaves unchanged the variables v_θ for $\theta \in \text{Read}(\Pi)$
  - if comm_A changes nothing, then comm_M sets $b^{\text{end}}_{\text{comm}_A}$ to true

According to the definition of the simulation p.12, there is three points to prove:

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

Moreover, during the execution of P_Π:

- The `while` command checks in one step whether the execution has terminated or not.
- Then, the `if` command checks in one step whether F^{end}_Π is true or not which, according to the lemma 8.5 p.17, indicates whether Π has terminated or not:
  - If Π has terminated, then $(r+c+m)-1$ steps[1] are done by the `skip` command, then the head command of the execution becomes a `comm`, and the processor waits for the other processors to do the communication comm_M, which is done in one step. Moreover, if comm_A has terminated, $b^{\text{end}}_{\text{comm}_A}$ becomes true during comm_M.
  - If Π has not terminated then P^{step}_Π is executed in $r+c+m$ steps, and according to the proposition 8.3 p.16 simulates one step of the ASM program Π.

  Then the execution comes backs to the `while` command.

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

2. A computation step or a communication step of the ASM_{BSP} machine is simulated by exactly $d = 2+r+c+m$ steps of the $\text{While}_{\text{BSP}}$ machine.
3. A terminal state of the ASM_{BSP} machine is reached when Π has terminated for every processor and the communication function comm_A changes nothing. In such state, the `while` command checks $b^{\text{end}}_{\text{comm}_A}$ which was false during the entire execution, then the `if` command checks that F^{end}_Π is true, then `skip` $(r+c+m)-1$, then the communication is done and sets $b^{\text{end}}_{\text{comm}_A}$ to true. Then the `while` command verifies that the execution is terminated, and the program reaches the end after $e = d+1$ steps.

□

9 Discussion

In this section, as Gurevich did in [Gur99], we present the discussion about our result by using a dialog between the authors and Quisani, an imaginary scrupulous colleague:

AUTHORS: Therefore, by using the two parts of the theorem p.14 and p.18, we obtain in the sense of the definition 6.5 p.12 that:

$$\text{While}_{\text{BSP}} \simeq \text{ASM}_{\text{BSP}}$$

In other words, our imperative programming language $\text{While}_{\text{BSP}}$ defined p.10 is algorithmically equivalent to the BSP algorithms Algo_{BSP} axiomatized in [MG18], where we proved that $\text{ASM}_{\text{BSP}} = \text{Algo}_{\text{BSP}}$. And by algorithmically equivalent, we mean that the executions are the same, up to a bounded number of fresh variables, a constant temporal dilation, and a constant ending time.

[1]↑ By construction of the program in normal form, we have $c \geq 1$, because if Π cannot terminate there is at least one block of updates in Π, and otherwise there is at least the empty block of updates. So $(r+c+m)-1 \geq 0$.

QUISANI: Speaking of the simulation, even if the two parts of your theorem corresponds to your definition of the simulation, it appears that one way is easier than the other. Indeed, in the first part p.14 the temporal dilation was $d = 1$ and the ending time $e = 0$, which does not depend on the simulated program, whereas in the second part p.18 the temporal dilation was $d = 2 + r + c + m$ and the ending time $e = d + 1$.
Could we find **a better simulation with uniform parameters**, which means that they do not depend on the simulated program?

A: Not with the model of computation $\text{WHILE}_{\text{BSP}}$ defined p.10, because only one update can be done at every step of computation. But indeed, we can imagine an alternative definition of the $\text{WHILE}_{\text{BSP}}$ model, where `while` and `if` commands costs 0 step and tuple-updates like $(x, y) := (y, x)$ are allowed (for every size of tuples, not only couples).
In that case, the simulation could have been done with the same translation but with a uniform cost of $d = 1$ and $e = 1$. We can even get rid off the ending time by assuming that $b^{\text{end}}_{\text{comm}_A}$ is updated at the last step of the communication function, and not when the communication function changes nothing.
In a way, such alternate model is algorithmically closer to the ASM_{BSP} model, because there is a quasi-identity (only the fresh variables are necessary) between them. But in this report we preferred a minimal model of computation, which can be more easily compared to practical computing languages.

Q: Speaking of the cost model, you defined an alternation of computation phases and communication phases for the ASM_{BSP} and for the $\text{WHILE}_{\text{BSP}}$, but they don't coincide in the translation. Indeed, a computation step of an ASM_{BSP} is translated into several computation steps in $\text{WHILE}_{\text{BSP}}$ then an execution of the `comm` command. So, there are many more supersteps in the simulation of the ASM_{BSP} than in the execution of the ASM_{BSP} itself. Therefore, your simulation does not respect the cost model of the ALGO_{BSP} defined in [MG18].

A: Strictly speaking, you are perfectly right. And this cannot be avoided, at least for our definition of $\text{WHILE}_{\text{BSP}}$, because the communication function is called via the `comm` command, and the program P_Π simulating the ASM program Π internalizes the test required to know if the program does a computation step or a communication step, test which is done by the operational semantics and not the program in the definition 4.4 p.8 of ASM_{BSP}.
But the "computation steps" done by the $\text{WHILE}_{\text{BSP}}$ program during a communication phase are only tests (or updates done by the communication function itself) and not updates done by the program, so it may be misleading to call them proper computation steps.
If we call computation steps only the updates commands, communication steps the `comm` command and administrative steps the `if` and `while` command, and if we call computation phase a succession of computation or administrative steps, and communication phase a succession of communication or administrative steps, then the number of supersteps done by an ASM_{BSP} is preserved by our translation. Therefore, **the cost model is preserved**.

Q: Fine. But if you don't have an identity ($d = 1$ and $e = 0$) up to fresh variables, why do you bother to simulate one step in a constant number of step. Why not a polynomial number of steps ?

A: Indeed, usually a simulation is seen acceptable if the simulation costs $\mathcal{O}(f(n))$ steps, where f is a polynomial function and n is the duration for the simulated machine. But in this report we are not interested in the simulation of the function (the input-output behavior) but the simulation of the algorithm (the step-by-step behavior).
For example, a Turing machine with one tape can polynomially simulate a Turing machine with two tapes, so they are usually seen as equivalent. But it has been proven in [BBD+04] that the palindrome recognition, which can be done in $\mathcal{O}(n)$ steps (where n is the size of the word) with two tapes, requires at least $\mathcal{O}\left(n^2/\log(n)\right)$ steps with only one tape. Strictly speaking, that means that the algorithm in $\mathcal{O}(n)$ requires at least two tapes, even if its result can be polynomially simulated.
A consequence of the temporal dilation d and the ending time e is that we simulate an execution costing n steps by an execution costing $\mathcal{O}(n)$ steps, so the complexity in time is strictly preserved, but our simulation is even more restrictive (and so our main theorem is stronger).
Indeed, in this report we simulate not only the output and the duration (a functional simulation) but we simulate every step of the execution by d steps (an algorithmic simulation). So **we preserve the intentional behavior** and not only the extensional behavior of the simulated algorithm.

References

[BBD+04] Therese Biedl, Jonathan F. Buss, Erik D. Demaine, Martin L. Demaine, Mohammadtaghi Hajiaghayi, and Tomas Vinar. Palindrome recognition using a multidimensional tape. *Theoretical Computer Science*, 302:475–480, 2004. 1, 21

[BG03] Andreas Blass and Yuri Gurevich. Abstract state machines capture parallel algorithms. *ACM Trans. Comput. Logic*, 4(4):578–651, October 2003. 5

[BGG+10] Wadoud Bousdira, Frederic Gava, Louis Gesbert, Frederic Loulergue, and Guillaume Petiot. Functional parallel programming with revised bulk synchronous parallel ml. In *Proceedings of the 2010 First International Conference on Networking and Computing*, ICNC '10, pages 191–196, Washington, DC, USA, 2010. IEEE Computer Society. 2

[Bis04] Rob H. Bisseling. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI*. Oxford University Press, 2004. 2

[Col91] Loïc Colson. About primitive recursive algorithms. *Theoretical Computer Science*, 83:57–69, 1991. 1

[FZG10] Marie Ferbus-Zanda and Serge Grigorieff. Asm and operational algorithmic completeness of lambda calculus. *Fields of Logic and Computation*, 2010. 16

[Gur99] Yuri Gurevich. The sequential ASM thesis. 67:93, 01 1999. 19

[Gur00] Yuri Gurevich. Sequential abstract state machines capture sequential algorithms. *ACM Transactions on Computational Logic*, 2000. 2, 3, 4, 5, 6, 7

[GV12] Serge Grigorieff and Pierre Valarcher. Classes of algorithms: Formalization and comparison. *Bulletin of the EATCS*, (107), 2012. 11

[HMS+98] Jonathan M.D. Hill, Bill McColl, Dan C. Stefanescu, Mark W. Goudreau, Kevin Lang, Satish B. Rao, Torsten Suel, Thanasis Tsantilas, and Rob H. Bisseling. Bsplib: The bsp programming library. *Parallel Computing*, 24(14):1947 – 1980, 1998. 2

[MAB+10] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: A system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM. 2

[Mar18] Yoann Marquer. Algorithmic completeness of imperative programming languages. *Fundamenta Informaticae*, 2018. In Editing. 2, 3, 9, 10, 12, 13, 14, 16

[MG18] Yoann Marquer and Frédéric Gava. An ASM Thesis for BSP. Technical report, Laboratoire d'Algorithmique, Complexité et Logique, Université Paris-Est Créteil, February 2018. 2, 6, 7, 8, 12, 16, 19, 20

[Mos01] Yiannis N. Moschovakis. What is an algorithm? *Mathematics Unlimited*, 2001. 2

[SHM97] David B. Skillicorn, Jonathan M. D. Hill, and William F. McColl. Questions and answers about bsp. *Scientific Programming*, 6:249–274, 1997. 2

[SYK+10] Sangwon Seo, Edward J. Yoon, Jaehong Kim, Seongwook Jin, Jin-Soo Kim, and Seungryoul Maeng. Hama: An efficient matrix computation with the mapreduce framework. In *Proceedings of the 2010 IEEE Second International Conference on Cloud Computing Technology and Science*, CLOUDCOM '10, pages 721–726, Washington, DC, USA, 2010. IEEE Computer Society. 2

[Val90] Leslie G. Valiant. A bridging model for parallel computation. *Commun. ACM*, 33(8):103–111, August 1990. 2