An ASM Thesis for BSP

Yoann Marquer and Frédéric Gava

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

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

1 Introduction

1.1 Context of the work

Nowadays, HPC (High Performance Computing) is the *norm* in many areas but it remains *as difficult* to have well defined paradigms and a common vocabulary as it is in the traditional sequential world. The problem arises from the difficulty to get a *taxonomy* of computer architectures and frameworks: there is a zoo of definitions of systems, languages, paradigms and programming models. Indeed, in the HPC community, several terms could be used to designate the same thing, so that misunderstandings are easy. We can cite parallel patterns [5,10] versus algorithmic skeletons [9]; shared memory (PRAM) versus thread concurrency and Direct ReMote Access (DRMA); asynchronous send/received routines (MPI, http://mpi-forum.org/) versus communicating processes (π-calculus).

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

This consensus is only fair because everyone *informally* agrees to what constitutes a sequential algorithm. And now, half a century later, there is a growing interest in defining *formally* the notion of algorithms [11]. Gurevich introduced an *axiomatic* presentation (largely machine independent) of the sequential algorithms in [11]. The main idea is that there is no language that truly represents all sequential algorithms. In fact, every algorithmic book presents the algorithm

in its own way and programming languages give too much detail. An axiomatic definition [11] of the algorithms has been mapped to the notion of Abstract State Machine (ASM, a kind of Turing machine with the appropriate level of abstraction): Every sequential algorithm can be computed by an ASM. This allows a common vocabulary about sequential algorithms. This has been studied by the ASM community for several years.

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

The set of *postulates* for sequential algorithms has been widely accepted by the scientific community. Nevertheless, to our knowledge, there is not such a work for HPC frameworks. First, due to the zoo of (informal) definitions and second, due to a lack of realistic *cost models* of common HPC architectures. In HPC, the cost measurement is not based on the complexity of an algorithm but is rather on the execution time, measured using empirical *benchmarks*. Programmers are benchmarking load balancing, communication (size of data), *etc.* Using such techniques, it is very difficult to explain why one code is faster than another and which one is more suitable for one architecture or another. This is regrettable because the community is failing to obtain some rigorous definitions of what HPC algorithms are. There is also a lack of studying algorithmic completeness of HPC languages. This is the basis from which to specify what can or cannot be effectively programmed. Finally, taking into account all the features of all HPC paradigms is a daunting task that is unlikely to be achieved [10]. Instead, a *bottom up strategy* (from the simplest models to the most complex) may be a solution that could serve as a basis for more general HPC models.

1.2 Content of the work

Using a *bridging model* [23] is a first step to this solution because it simplifies the task of the algorithm design, their programming and simplifies the reasoning of *cost* and ensures a better *portability* from one system to another. In computer science, a bridging model is thus an abstract model of a computer which provides a conceptual bridge between the physical implementation of the machine and the *abstraction* available to a programmer of that machine. We conscientiously limit our work to the Bulk-Synchronous Parallel (BSP) bridging model [2,21] because it has the advantage of being endowed with a simple model of execution. We leave more complex models to future work. Moreover, there are many different libraries and languages for programming BSP algorithms. The best known are the BSPLIB for C [12] or JAVA [20], BSML [13], PREGEL [14] for big-data, *etc.*

Concurrent ASMs try to capture the more general definition of asynchronous and distributed computations. We promote a rather different "bottom-up" approach consisting of restricting the model under consideration, so as to better

take into account the physical architectures and in particular to highlight the algorithm execution time, which is often too difficult to assess for general models.

As a basis to this work, we must give first an axiomatic definition of BSP algorithms in the spirit of [11,15]. Basically, four postulates will be necessary. With such postulates, we can extend the ASMs of [11] to take into account the BSP model of computation. Our goal is to define a convincing set of parallel algorithms running in a predictable time and construct a model computing these algorithms only. This can be summarized by the $\text{ALGO}_{\text{BSP}} = \text{ASM}_{\text{BSP}}$. An interesting and novel point of this work is that the BSP cost model is preserved.

1.3 Outline

The remainder of this paper is structured as follows: In Section 2 we first recall the BSP model of computation and define the postulates; Secondly, in Section 3, we give the operational semantics of ASM_{BSP} and finally, we give the main result. Section 4 concludes the paper by giving some questions with their answers (notably about the related work) and a brief outlook on future work.

2 Characterizing BSP algorithms

2.1 The BSP bridging model of computation

As the RAM model provides a unifying approach that can *bridge* the worlds of sequential *hardware* and *software*, so Valiant sought [23] for a unifying model that could provide an effective (and universal) bridge between parallel hardware and software. A *bridging* model [23] allows to reduce the gap between an abstract execution (programming an algorithm) and concrete parallel systems (using a compiler and designing/optimizing a physical architecture).

The *direct mode* BSP model [2,21] is a *bridging* model that simplifies the programming of various parallel architectures using a certain level of abstraction. The assumptions of the BSP model are to provide *portable* and *scalable* performance predictions on HPC systems. Without dealing with low-level details of parallel architectures, the programmer can thus focus on algorithm design. The BSP bridging model describes a parallel architecture, an execution model, and a cost model which allows to predict the performance of a BSP algorithm on a given architecture. We now recall each of them.

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

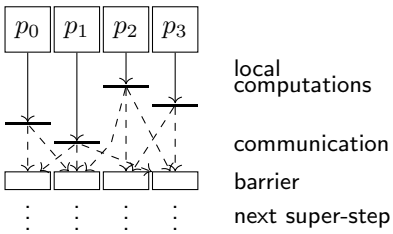

Fig. 1. A BSP super-step.

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

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

More comments on BSP are available in the appendix (Section B).

2.2 Axiomatic characterization of BSP algorithms

We follow [11] in which states are full instantaneous descriptions of an algorithm that can be conveniently formalized as first-order structures.

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

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

In order to have a uniform presentation [11], we considered constant symbols in $\mathcal{L}(X)$ as 0-ary function symbols, and relation symbols R as their indicator function χ_R. Therefore, every symbol in $\mathcal{L}(X)$ is a function. Moreover, partial functions can be implemented with a special symbol *undef*, and we assume in this paper that every $\mathcal{L}(X)$ contains the boolean type ($\neg$, $\wedge$) and the equality.

Definition 2 (Term). *A term of $\mathcal{L}(X)$ is defined by induction :*

1. *If c has arity 0, then c is a term*
2. *If f has an arity $\alpha > 0$ and $t_1, \ldots, t_\alpha$ are terms, then $f(t_1, \ldots, t_\alpha)$ is a term*

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

1. *If $t = c$ is a constant symbol, then $\overline{t}^X \stackrel{\text{def}}{=} \overline{c}^X$*
2. *If $t = f(t_1, \ldots, t_\alpha)$ where f is a symbol of the language $\mathcal{L}(X)$ with arity $\alpha > 0$ and $t_1, \ldots, t_\alpha$ are terms, then $\overline{t}^X \stackrel{\text{def}}{=} \overline{f}^X(\overline{t_1}^X, \ldots, \overline{t_\alpha}^X)$*

A **formula** F is a term with the particular form $\mathit{true} \,|\, \mathit{false} \,|R\,(t_1, \ldots, t_\alpha)\,|\neg F$ $|(F_1 \wedge F_2)$ where R is a relation symbol (ie a function with output $\overline{\mathit{true}}^X$ or $\overline{\mathit{false}}^X$) and $t_1, \ldots, t_\alpha$ are terms. We say that a formula is true (resp. false) in X if $\overline{F}^X = \overline{\mathit{true}}^X$ (resp. $\overline{\mathit{false}}^X$). These notions are fully detailed in the appendix (Section A).

We now define the BSP algorithms as the objects verifying four postulates. The computation for every processor is done in parallel and step by step.

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 $\vec{S} = S_0, S_1, S_2, \ldots$ such that S_0 is an initial state and for every $t \in \mathbb{N}$, $S_{t+1} = \tau_A(S_t)$.

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 will say that an execution is **terminal** if it contains a final state. In that case, its **duration** is defined by:

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

The BSP model defines the machine with multiple processors which have their own memory. Therefore, a state S_t of the algorithm must be a p-tuple $(X_t^1, \ldots, X_t^p)$[1]. Notice that p is not fixed for the algorithm, so A can have states using different number of processors. In this paper, 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.

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

Definition 3 (Multi-Isomorphism).

$\vec{\zeta}$ 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 $\vec{\zeta}$ is a p-tuple of applications $\zeta_1, \ldots, \zeta_p$ such that for every $1 \leq i \leq p$, ζ_i is an isomorphism between X^i and Y^i.

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

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

[1] To simplify, we annotate units from 1 to p and not, as usual in HPC, from 0 to $p-1$.

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

$$\overline{f}^{X \oplus u}(\overrightarrow{a}) \stackrel{\text{def}}{=} \begin{cases} b & \text{if } u = (f, \overrightarrow{a}, b) \qquad (\text{we note } \overrightarrow{a} = a_1, \ldots, a_\alpha) \\ \overline{f}^{X}(\overrightarrow{a}) & \text{otherwise} \end{cases}$$

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.

If $\overline{f}^{X}(\overrightarrow{a}) = b$ then the update $(f, \overrightarrow{a}, b)$ is said **trivial** in X, because nothing has changed. Indeed, if $(f, \overrightarrow{a}, b)$ is trivial in X then $X \oplus (f, \overrightarrow{a}, b) = X$.

If Δ is a set of updates then Δ is **consistent** if it does not contain two distinct updates with the same location. Notice that if Δ is inconsistent, then there exists $(f, \overrightarrow{a}, b), (f, \overrightarrow{a}, b') \in \Delta$ with $b \neq b'$. 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 $\Delta = \{(f, \overrightarrow{a}, b) \mid \overline{f}^{Y}(\overrightarrow{a}) = b \text{ and } \overline{f}^{X}(\overrightarrow{a}) \neq b\}$ of non trivial updates such that $Y = X \oplus \Delta$. This Δ is called the **difference** between the two local memories, and is denoted by $Y \ominus X$.

Let $\overrightarrow{X} = \left(X^1, \ldots, X^p\right)$ be a state of A. According to the transition function τ_A, the next state is $\tau_A(\overrightarrow{X})$, which will be denoted by $(\tau_A(\overrightarrow{X})^1, \ldots, \tau_A(\overrightarrow{X})^p)$. We denote by $\Delta^i(A, \overrightarrow{X}) \stackrel{\text{def}}{=} \tau_A(\overrightarrow{X})^i \ominus X^i$ the set of updates done by the i-th processor of A on the state $\overrightarrow{X}$, and by $\overrightarrow{\Delta}(A, \overrightarrow{X}) \stackrel{\text{def}}{=} (\Delta^1(A, \overrightarrow{X}), \ldots, \Delta^p(A, \overrightarrow{X}))$ the "multiset" of updates done by A on the state $\overrightarrow{X}$. In particular, if a state $\overrightarrow{X}$ is final, then $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$, so $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \overrightarrow{\emptyset}$.

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

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

$T(A)$ is called the **exploration witness** [11] of A. The interpretations of the terms in $T(A)$ are called the **critical elements**, and we prove, in Section C of the appendix, that every value in an update is a critical element:

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

That implies that for every step of the computation, for a given processor, only a bounded number of terms are read or written (amount of work). In other words, each processor individually is a sequential algorithm.

Lemma 2 (Bounded Set of Updates). *For every state* $(X^1, \ldots, X^p)$ *of the* BSP *algorithm* A*, for every* $1 \leq i \leq p$*,* $\#\Delta^i(A, \overrightarrow{X})$ *is bounded, where* $\#U$ *is the number of elements of the set* U*.*

Notice that for the moment we make no assumption on the communication between processors. Moreover, these three postulates are a "natural" extension of the ones of [11]. And by "natural", we mean that if we assume that $p = 1$ then our postulates are exactly the same:

Lemma 3 (A Single Processor is Sequential). *An algorithm verifying the first three postulates and with only one processor is a sequential algorithm.*

We organize the sequence of states into **supersteps**. 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 A runs computations or runs communications (followed by a barrier).

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

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

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

This requires some remarks. First, not only one processor performs the local computations but all who can. Second, we do not specified the function comm_A in order to be generic about which BSP library is used. We discuss in Section 3.3 the difference between comm_A and the usual communication routines in the BSP community. The communication function comm_A keeps p.

Remembering that a state $\overrightarrow{X}$ is said to be final if $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$. Therefore, according to the fourth postulate, $\overrightarrow{X}$ must be in a communication phase which is like a final phase that would terminate the whole execution as found in MPI.

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

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

3 BSP-ASM captures the BSP algorithms

The four previous postulates define the BSP algorithms from an axiomatic viewpoint but that does not mean that they have a model, or in, other words, that they are defined from an operational point of view. In the same way that the model of computation ASM captures the set of the sequential algorithms [11], we prove in this section that the ASM_{BSP} model captures the BSP algorithms.

3.1 Definition and operational semantics of ASM-BSP

Definition 4 (ASM Program [11]).

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

where: f has arity α; F is a formula; t_1, …, t_α, t_0 are terms of $\mathcal{L}(X)$.

Notice that if $n = 0$ then $\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar}$ is the empty program. If in $\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}$ the program Π_2 is empty we will write simply $\texttt{if } F \texttt{ then } \Pi_1 \texttt{ endif}$. An ASM machine [11] is a kind of Turing machine using not a tape but an abstract structure X:

Definition 5 (ASM Operational Semantics).

$$\begin{aligned} \Delta(f\,(t_1,\dots,t_\alpha) := t_0, X) &\stackrel{\text{def}}{=} \left\{\left(f, \overline{t_1}^X, \dots, \overline{t_\alpha}^X, \overline{t_0}^X\right)\right\} \\ \Delta(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}, X) &\stackrel{\text{def}}{=} \Delta(\Pi_i, X) \\ & \qquad \textit{where} \begin{cases} i = 1 \textit{ if } F \textit{ is true on } X \\ i = 2 \textit{ otherwise} \end{cases} \\ \Delta(\texttt{par } \Pi_1 \| \dots \| \Pi_n \texttt{ endpar}, X) &\stackrel{\text{def}}{=} \Delta(\Pi_1, X) \cup \dots \cup \Delta(\Pi_n, X) \end{aligned}$$

Notice that the semantics of the `par` is a set of updates done simultaneously, which differs from an usual imperative framework. A state of a ASM_{BSP} machine is a p-tuple of memories $(X^1, \dots, X^p)$. We assume that the ASM_{BSP} programs are SPMD (Single Program Multiple Data) which means that at each step of computation, the ASM_{BSP} program Π is executed individually on each processor. Therefore Π induces a multiset of updates $\overrightarrow{\Delta}$ and a transition function τ_Π:

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

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

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

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

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

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

The last conditions about the communication function may seem arbitrary, but they are required to ensure that the communication function is not a kind of magic device. For example, without these conditions, we could imagine that comm_M may compute the output of the algorithm in one step, or solve the halting problem. Moreover, we presented in this definition the conditions required to prove the main theorem, but we discuss some issues in Section 3.3, and we construct an example of such communication function in the appendix (Section D).

3.2 The BSP-ASM thesis

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

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

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

Proof. (**Sketch**). The full proof available in the appendix p.33. It is made by mutual inclusion. On the one hand, let A be the BSP algorithm $(S(A), I(A), \tau_A)$. According to the fourth postulate, there exists comp_A and comm_A such that for every state $\overrightarrow{X}$:

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

where $\overrightarrow{\text{comp}_A}\left(X^1, \ldots, X^p\right) = \left(\text{comp}_A(X^1), \ldots, \text{comp}_A(X^p)\right)$. Then, we use the the Proposition 1 to prove that:

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

According to the Lemma 5 p.13, comm_A preserves the universes, the number of processors, and commutes with multi-isomorphisms. And the other properties are immediately true according to the first three postulates. Therefore A is a ASM_{BSP} machine.

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

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

We prove that M is a BSP algorithm by proving that it verifies the four postulates. The first postulate is straightforward. The second requires the Lemma 7 p.15. For the third, we prove that $T(\Pi) = \{true\} \cup \mathrm{Read}(\Pi) \cup \mathrm{Write}(\Pi)$ (Definition 4 p.17) is an exploration witness for τ_Π so $T(M) = T(\Pi) \cup T(\mathrm{comm}_M)$ is for M. For the fourth, we set $\mathrm{comp}_M(X) = X \oplus \Delta(\Pi, X)$ for every local memory X. So $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{\mathrm{comp}_M}(\overrightarrow{X})$, and we have:

$$\tau_M(\overrightarrow{X}) = \begin{cases} \overrightarrow{\mathrm{comp}_M}(\overrightarrow{X}) \text{ if } \overrightarrow{\mathrm{comp}_M}(\overrightarrow{X}) \neq \overrightarrow{X} \\ \mathrm{comm}_M(\overrightarrow{X}) \text{ otherwise} \end{cases}$$

Therefore M is a BSP algorithm. □

3.3 Cost model property and the function of communication

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

Cost model. If the execution begins with a communication, we assume that no computation is done for the first superstep. We remind that a state $\overrightarrow{X_t}$ is in a computation phase if there exists $1 \leq i \leq p$ such that $\mathrm{comp}_A(X_t^i) \neq X_t^i$. The computation for every processor is done in parallel, step by step, and these steps are synchronized. So, the cost in time of the computation phase is $w \stackrel{\text{def}}{=} \max_{1 \leq i \leq p}(w_i)$, where w_i is the number of steps done by the processor i (on memory X^i) during the superstep.

Then the state is in a communication phase, when the messages between the processors are sent and received. Notice that comm_A may require several steps in order to communicate the messages, which contrasts with the usual approach in BSP where the communication actions of a superstep are considered as one unit. But this approach would violate the third postulate, so we had to consider a step-by-step communication approach, then consider these actions as one communication phase. $\mathrm{ASM}_{\mathrm{BSP}}$ exchanges terms and we show in the appendix how formally define the size of terms. But we can imagine a machine that must further decompose the terms in order to transmit them (in bits for example). We just assume that the data are communicable in time $\mathbf{g}$ for a 1-relation.

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

A realization of the communication. An example of a communication function for the standard BSPLIB's primitives (described in appendix Section D p.36) read (`bsp_get`), write (`bsp_put`), send (`bsp _send`) and rcv (`bsp_move`) is presented in Section D. The main difficulty is to assign an exploration witness to the communications.

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

One may argue that the last postulate allows the communication function to do computations. To avoid it, we assume that the terms in the exploration

witness $T(M)$ can be separated between $T(\Pi)$ and $T(\text{comm}_M)$ such that $T(\Pi)$ is for the states in a computation phase, and that for every update $(f, \overrightarrow{a}, b)$ of a processor X^i in a communication phase, either there exists a term $t \in T(\text{comm}_M)$ such that $b = \bar{t}^{X^i}$, or there exists a variable $v \in T(\Pi)$ and a processor X^j such that $b = \overline{t_{\bar{v}^{X^j}}}^{X^i}$ (**representation** presented in the appendix, section D p.36). To do a computation, a term like $x+1$ is required, so the restriction to a variable prevents the computations of the terms in $T(\Pi)$. Or course, the last communication step should be able to write in $T(\Pi)$, and the final result should be read in $T(\Pi)$.

4 Conclusion and Future Work

4.1 Summary of the Contribution

In computer science, a bridging model provides a common level of *understanding* between hardware and software engineers. It provides software developers with an attractive escape route from the world of architecture-dependent parallel software [23]. The BSP bridging model allows the design of "*immortal*" (efficient and portable) parallel algorithms using a *realistic* cost model (and without any over-specification requiring the use of a large number of parameters) that can fit most distributed architectures. It has been used with success in many domains [2].

We have given an axiomatic definition of BSP algorithms by adding only one postulate to the sequential ones for sequential algorithms [11] which has been widely accepted by the scientific community. Mainly this postulate is the call of a function of communication. We abstract how communication is performed, not be restricting to a specific BSP library. We finally answer previous criticisms by defining a convincing set of parallel algorithms running in a predictable time.

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

This small addition allows a greater *confidence* in this formal definition compared to previous work: Postulates of concurrent ASMs do not provide the same level of intuitive clarity as the postulates for sequential algorithms. But our work is limited to BSP algorithms even if it is still sufficient for many HPC and big-data applications. We have thus revisited the problem of the "*parallel ASM thesis*" *i.e.*, to provide a machine-independent definition of BSP algorithms and a proof that these algorithms are faithfully captured by ASM_{BSP}. We also prove that the *cost model* is preserved which is the main novelty and specificity of this work compared to the traditional work about distributed or concurrent ASMs.

4.2 Questions and answers about this work

Why not use a BSP*-Turing machine to simulate a* BSP *algorithm?*

For sequential computing, it is known that Turing machines could simulate every algorithm or any program of any language but without a constant factor

[1]. In this way, there is not an algorithmic equivalence between Turing machines and common sequential programming languages.

Why do you use a new language ASM_{BSP} *instead of using* ASM*s only? Indeed, each processor can be seen as a sequential* ASM*. So, in order to simulate one step of a* BSP *algorithm using several processors, we could use pids to compute sequentially the next step for each processor by using an* ASM*.*

But if you have p processors, then each step of the BSP algorithm will be simulated by p steps. This contradicts a temporal dilation [15]: Each step should be simulated by d steps, where d is a constant depending only on the simulated program. In that case, the simulation of a BSP algorithm by a sequential ASM would require that p is constant, which means that our simulation would hold only for a fixed number of processors, and not for every number.

Why are you limited to SPMD *computations?*

Different codes can be run by the processors using conditionals on the "id" of the processors. For example "if pid=0 then code1 else code2" for running "code1" (*e.g.* master part) only on processor 0.

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

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

What happens in case of runtime errors during communications?

Typically, when one processor has a bigger number of super-steps than other processors, or when there is an out-of-bound sending or reading, it leads to a runtime error. The BSP function of communication can return a $\bot$ value. That causes a stop of the operational semantics of the ASM_{BSP}.

When using BSPLIB*, messages received at the past superstep are dropped. Your communication function does not show this fact.*

We want to be as general as possible. Perhaps a future library would allow reading data received n supersteps ago. Moreover, the communication function may realize some computations and is thus not a pure transmission of data. But the exploration witness forbids doing whatever. And we provide a realistic example of such a function which mainly correspond to the BSPLIB's primitives.

What about related work?

As far as we know, some work exists to model distributed programs using ASMs [17] but none to convincingly characterize BSP algorithms. In [6], authors model the P3L set of skeletons. That allows the analyze of P3L programs using standard ASM tools but not a formal characterization of what P3L is and is not.

The first work to extend ASMs for concurrent, distributed, agent-mobile algorithms is [3]. Too many postulates are used making the comprehension hard to follow or worse (loss of confidence). A first attempt to simplify this work has been done in [18] and again simplified in [8] by the use of multiset comprehension terms to maintain a kind of bounded exploration. Then, the authors prove

that ASMs captures these postulates. Moreover, we are interested in distributed (HPC) computations more than parallel (threading) ASMs.

We want to clarify one thing. The ASM thesis comes from the fact that sequential algorithms work in small steps, that is steps of bounded complexity. But the number of processors (or computing units) is unbounded for parallel algorithms, which motivated the work of [3] to define parallel algorithms with wide steps, that is steps of unbounded complexity. Hence the technicality of the presentation, and the unconvincing attempts to capture parallel algorithms [4].

In our work, we use the sequence of supersteps of the BSP bridging model to simplify the approach. Even if the number of processors is unbounded, we assume that every processor works in small step. Instead of defining a state of an execution by a meta-finite structure, we assume that a state is a p-tuple of structures. Instead of using a global program with "forall" commands [8], every processor runs its local program with "par" commands. Instead of relaxing the third postulate, we assume it for every processor. This approach leads to a simpler presentation, using only ground terms and tuples of ordinary structures. Notice that the second example of [8] is PRAM. There is still a last drawback: Their ASMs used an implicit share memory which is irrelevant for HPC computations. This flaw of the PRAM model was already criticized in [23].

Extending the ASMs for distributed computing is not new [4]. For example the works of [7,19] about multi-agents and data-flow programs. We believe that these postulates are more general than ours but we think that our extension still remains simple and natural for BSP computing. The authors are also not concerned about the problem of algorithm completeness using a cost model which is the heart of our work and the main advantage of the BSP model.

4.3 Future Work

This work leads to many possible work. First, how adapting our work to a hierarchical extension of BSP [24] which is closer to modern HPC architectures?

Second, we are currently working on extending the work of [15] in order to give the BSP algorithmic completeness of a BSP imperative programming language. There are some concrete applications: There are many languages having a BSP-like model of execution, for example PREGEL [14] for writing large-graph algorithms. An interesting application is proving which are BSP algorithmically complete and are not. BSPLIB programs are intuitively BSP. PREGEL is a good candidate to be *not* BSP if we cannot dynamically change the graph (most recent feature). Indeed, a short-path computation using PREGEL needs n super-steps (where n is the shorter path) because a node could only communicate with its neighborhood, whereas a $\log(p)$ super-steps exists [22]. MAPREDUCE is also an interesting use case [16]. Similarly, on can imagine proving which languages are too expressive for BSP. MPI is intuitively one of them. Last, the first author is working on postulates for more general distributed algorithm *à la* MPI.

References

1. T. C. Biedl, *et al.* Palindrome Recognition Using a Multidimensional Tape. *Theor. Comput. Sci.*, 302(1-3):475–480, 2003.

2. R. H. Bisseling. *Parallel Scientific Computation. A Structured Approach Using* BSP *and* MPI. Oxford University Press, 2004.
3. A. Blass and Y. Gurevich. Abstract State Machines Capture Parallel Algorithms. *ACM Trans. Comput. Log.*, 4(4):578–651, 2003.
4. E. Börger and K.-D. Schewe. Concurrent Abstract State Machines. *Acta Inf.*, 53(5):469–492, 2016.
5. F. Cappello and M. Snir. On Communication Determinism in HPC Applications. In *Computer Communications and Networks (ICCCN)*, pages 1–8. IEEE, 2010.
6. A. Cavarra and A. Zavanella. A Formal Model for the Parallel Semantics of p3l. In *ACM Symposium on Applied Computing (SAC)*, pages 804–812, 2000.
7. A. Cavarra. A Data-Flow Approach to Test Multi-agent ASMs *Formal Asp. Comput.*, 23(1):21–41, 2011.
8. F. Ferrarotti, K.-D. Schewe, L. Tec, and Q. Wang. A New Thesis Concerning Synchronised Parallel Computing Simplified Parallel ASM Thesis. *Theoretical Computer Science*, 649:25–53, 2016.
9. H. González-Vélez and M. Leyton. A Survey of Algorithmic Skeleton Frameworks. *Software, Practrice & Experience*, 40(12):1135–1160, 2010.
10. S. Gorlatch. Send-receive Considered Harmful: Myths and Realities of Message Passing. *ACM TOPLAS*, 26(1):47–56, 2004.
11. Y. Gurevich. Sequential Abstract-state Machines Capture Sequential Algorithms. *ACM Trans. Comput. Log.*, 1(1):77–111, 2000.
12. J. M. D. Hill, B. McColl, D. C. Stefanescu, M. W. Goudreau, K. Lang, S. B. Rao, T. Suel, T. Tsantilas, and R. Bisseling. BSPLIB: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.
13. W. Bousdira, F. Gava, L. Gesbert, F. Loulergue and G. Petiot: Functional Parallel Programming with Bulk Synchronous Parallel ML. ICNC, pages 191–196, 2010
14. G. Malewicz, et *al.* PREGEL: A System for Large-scale Graph Processing. In *Management of data*, pages 135–146. ACM, 2010.
15. Y. Marquer. Algorithmic Completeness of Imperative Programming Languages. *Fundamenta Informaticae*, accepted, pages 1–27, 2017.
16. M. F. Pace. BSP vs MAPREDUCE. In *Computational Science (ICCS)*, volume 9 of *Procedia Computer Science*, pages 246–255. Elsevier, 2012.
17. A. Prinz and E. Sherratt. Distributed ASM- Pitfalls and Solutions. In *ABZ* conference, LNCS, volume 8477, pages 210–215. Springer, 2014.
18. K.-D. Schewe and Q. Wang. A Simplified Parallel ASM Thesis. In *ABZ* conference, LNCS, volume 7316, pages 341–344. Springer, 2012.
19. K.-D. Schewe, F. Ferrarotti, L. Tec, Q. Wang, and W. An. Evolving Concurrent Systems: Behavioural Theory and Logic. In *Australasian Computer Science Week Multiconference (ACSW)*, pages 1–10, 2017.
20. S. Seo, E. J. Yoon, J.-H. Kim, S. Jin, J.-S. Kim, and S. Maeng. HAMA: An Efficient Matrix Computation with the MAPREDUCE Framework. In *Cloud Computing (CloudCom)*, pages 721–726. IEEE, 2010.
21. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
22. A. Tiskin. *The Design and Analysis of Bulk-Synchronous Parallel Algorithms.* PhD thesis, Oxford University Computing Laboratory, 1998.
23. L. G. Valiant. A Bridging Model for Parallel Computation. *Comm. of the ACM*, 33(8):103–111, 1990.
24. L. G. Valiant. A Bridging Model for Multi-core Computing. *J. Comput. Syst. Sci.*, 77(1):154–166, 2011.

Proofs and other comments in http://lacl.fr/gava/tr-asm.pdf

A Preliminaries of the postulates

In in section, we recall some reminders about fundamental concepts of the formalization. We consider states as first-order structures whose vocabulary or signature is a finite set of function symbols. Furthermore, it is common to say that all convenient data structures (integers, graphs, sets, *etc.*) [15] are representable by **first-order structures**. See the definitions in Section p.4.

Using structures has also the advantage to sufficiently abstracting algorithms; Each algorithm can access **elementary operations** that could be executed. For example, the Euclid algorithm is different whether we use a (native) division or simulate it using subtractions. In the same manner, a BSP algorithm could be different whether a broadcasting primitive exists or is simulated by lower-level communicating primitives such as point-to-point sending of data. These elementary operations are also called **primitives** and depend only on the architecture. Our work is thus independent of a given architecture as in the spirit of a bridging model.

In order to have a uniform presentation, as in [11] we considered 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. In practice, the universe will be assumed infinite, for example if the data structure contains at least integers. We also assume that all the functions are total and represent partial function as total by using a special symbol *undef*. The symbols of the signature $\mathcal{L}(X)$ are distinguished between:

1. *Dyn* X the set of **dynamic symbols**, whose interpretation can change during an execution, like a variable[2] x;
2. *Stat* X the set of **static symbols**, which have a fixed interpretation during an execution. They are also distinguished between:
   (a) Init(X), the set of **parameters**, whose interpretation depends only on the initial state, like an array in a sorting algorithm; The symbols depending on the initial state are the dynamic symbols and the parameters, so we call them the **inputs**.

   The other symbols have a uniform interpretation in every state (up to isomorphism, see the Definition p.17), and they are also distinguished between:
   (b) Cons(X) the set of **constructors** (`true` and `false` for the booleans, 0 and S for the unary integers, *etc.*)
   (c) Oper(X) the set of **operations** ($\neg$ and $\wedge$ for the booleans, $+$ and $\times$ for the integers, *etc.*)

We assume in this paper that every signature contains the boolean type (*true*, *false*, $\neg$ and $\wedge$) and the equality. A term t is said well-typed in X if $\bar{t}^X \neq \overline{undef}^X$.

Example 1 (Booleans $\mathbb{B}$). The constructors are *true* and *false*, interpreted by two distinct values $\overline{true}^X$ and $\overline{false}^X$. The basic operations are the unary symbol $\neg$

[2] We assume there is no logical variables and every term is closed, so a "variable" is a dynamical symbol with arity 0.

and the binary symbol $\wedge$, defined by:

$$\begin{aligned}
\neg^X(\overline{true}^X) &\stackrel{\text{def}}{=} \overline{false}^X \\
\neg^X(\overline{false}^X) &\stackrel{\text{def}}{=} \overline{true}^X \\
\overline{\wedge}^X(\overline{true}^X, \overline{true}^X) &\stackrel{\text{def}}{=} \overline{true}^X \\
\overline{\wedge}^X(\overline{true}^X, \overline{false}^X) &\stackrel{\text{def}}{=} \overline{false}^X \\
\overline{\wedge}^X(\overline{false}^X, \overline{true}^X) &\stackrel{\text{def}}{=} \overline{false}^X \\
\overline{\wedge}^X(\overline{false}^X, \overline{false}^X) &\stackrel{\text{def}}{=} \overline{false}^X
\end{aligned}$$

All of the other logical connectives can be defined with $\neg$ and $\wedge$.

We assume in this paper that every signature contains the booleans and the equality, a binary symbol $=$ interpreted as $\equiv^X(a,a) = \overline{true}^X$ and $\equiv^X(a,b) = \overline{false}^X$ otherwise. We also assume that every element $a \neq \overline{undef}^X$ of the universe $\mathcal{U}(X)$ is representable, which means that there exists a unique term t_a formed only by constructors such that $\overline{t_a}^X = a$. This t_a is called the **representation** of a. This can be proven for every usual data structure [15] but this is not the point of this paper. In particular, we must assume that an element has only one type. For example, the binary integers use a different copy of $\mathbb{N}$ than the decimal integers. In other words $\overline{\underline{314}_{10}}^X \neq \overline{\underline{100111010}_2}^X$ but of course there is a bijection between the two copies. The **size** of an element is the length of its representation, in other words the number of constructors necessary to write it. For example $|\overline{\underline{314}_{10}}^X| = 3$ and $|\overline{\underline{100111010}_2}^X| = 9$. These sizes will be used to compute the super-steps costs of the BSP algorithms.

Example 2 (Unary integers $\mathbb{N}_1$). The constructors are the constant symbol $\underline{0}$ and the unary symbol S, interpreted respectively by 0 and $n \mapsto n+1$. Therefore, the terms have the form $S^n\underline{0}$, and are interpreted by $\overline{S^n\underline{0}}^X = n$.

Example 3 (Integers $\mathbb{N}_b$ in base $b \geq 2$). The constructors are the constant symbols $c_0^b, \ldots, c_{b-1}^b$, and the unary symbols $f_0^b, \ldots, f_{b-1}^b$, interpreted by :

1. $\overline{c_i^b}^X \stackrel{\text{def}}{=} i$
2. $\overline{f_i^b}^X(a) \stackrel{\text{def}}{=} \begin{cases} \overline{undef}^X & \text{if } a = \overline{c_0^b}^X \\ a \times b + i & \text{otherwise} \end{cases}$

Therefore, the terms formed only by constructors have the form $f_{i_0}^b \ldots f_{i_{n-1}}^b c_n^b$, which we denote by $\underline{i_n \ldots i_0}_b$ and call the expansion of the number in base b.

For example, the decimal expansion of $314 = (3 \times 10 + 1) \times 10 + 4$ is $\underline{314}_{10} = f_4^{10} f_1^{10} c_3^{10}$, and in the same way its binary expansion is $\underline{100111010}_2$. Notice that because of the condition $\overline{f_i^b}^X(\overline{c_0^b}^X) = \overline{undef}^X$ the expansion of a number cannot begin by a 0.

Operations can be added to the integers data structures, like the multiplication $\times$, depending of the basic operations considered for the algorithm.

Definition 7 (Formula). *A formula F is a term with a particular form:* $F \stackrel{\text{def}}{=} \mathit{true} \mid \mathit{false} \mid R(t_1, \ldots, t_\alpha) \mid \neg F \mid (F_1 \wedge F_2)$ *where R is a relation symbol, a function with output $\overline{\mathit{true}}^X$ or $\overline{\mathit{false}}^X$), and $t_1, \ldots, t_\alpha$ are terms.*

We say that a formula is true (resp. false) in X if $\overline{F}^X = \overline{\mathit{true}}^X$ (resp. $\overline{\mathit{false}}^X$).

Definition 8 (Isomorphism). *Let X and Y be two structures with the same signature $\mathcal{L}$, and let $\zeta : \mathcal{U}(X) \to \mathcal{U}(Y)$ be an application. ζ is an isomorphism between X and Y if :*

1. *ζ is surjective*
2. *For every symbol $c \in \mathcal{L}$ with arity 0, $\zeta(\overline{c}^X) = \overline{c}^Y$*
3. *For every $f \in \mathcal{L}$ with arity $\alpha > 0$, and for every $a_1, \ldots, a_\alpha \in \mathcal{U}(X)$, $\zeta(\overline{f}^X(a_1, \ldots, a_\alpha)) = \overline{f}^Y(\zeta(a_1), \ldots, \zeta(a_\alpha))$*

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

B The BSP bridging model of computation

B.1 The notion of bridging model.

The traditional von Neumann (RAM) model has always served as the main model for designing sequential algorithm (Fig 2). It has also served as a reference model for hardware design. In the context of parallel algorithm design, no such ubiquitous model exists. The PRAM model (shared memory) allows a theoretical reasoning about parallel algorithms by highlighting the parallelism's degree of problems. Nevertheless, it makes a number of assumptions that cannot be fulfilled in HPC applications and hardware; mainly because the cost of communication is greater than that of computation and the number of processors is limited.

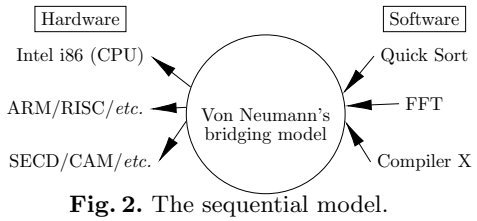

Fig. 2. The sequential model.

In computer science, a bridging model is thus an abstract model of a computer which provides a conceptual bridge between the physical implementation of the machine and the abstraction available to the programmer of that machine; in other words, it is intended to provide a common level of understanding between hardware and software engineers. A bridging model provides software developers with an attractive escape route from the world of architecture-dependent parallel software. A successful bridging model is one which can be efficiently implemented in reality and efficiently targeted by programmers; in particular, it should be possible for a compiler to produce good code from a typical high-level language. The term was introduced in [23], which argued that the strength of the von Neumann model was largely responsible for the success of computing. The paper goes on to develop the BSP model as an analogous model for parallel computing. Even with rapidly changing technology and architectural ideas, hardware designers can still share the common goal of realizing efficient von Neumann machines, without having to be too concerned about the software that is going to be executed. Thus, the von Neumann model is the connecting bridge that enables programs from the diverse and chaotic world of software to run efficiently on machines from the diverse and chaotic world of hardware. An analogous bridging model for parallel computation is thus needed.

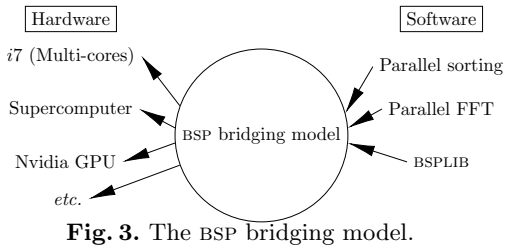

Fig. 3. The BSP bridging model.

As the von Neumann model provides a unifying approach that can bridge the worlds of sequential hardware and software, so Valiant [23] sought for a unifying model that could provide an effective bridge between parallel hardware and software. The BSP model has been introduced to better reflect the hardware design features of mainstream parallel computers, through the direct mode of BSP (assumed in this paper, Fig 3). The BSP model allows for efficient algorithm design without any over-

specification requiring the use of a large number of parameters. The underlying parallel computer implementation is similarly not overspecified. Each processor can communicate directly with every other processor, providing complete control over how the data is distributed between the processors in every superstep.

To evaluate an architecture-independent model of parallel computation such as BSP is to consider it in terms of all of its properties which means (a) its usefulness as a basis for the design and analysis of algorithms; (b) its applicability across the whole range of general-purpose architectures and its ability to provide efficient, scalable performance on them; (c) its support for the design of fully-portable programs; and (d) software engineering tools such as those for correctness or debug can be easily adapted to programs of this bridging model.

Take for example, a proof of correctness of a GPU-like program. Although interesting in itself it cannot be used directly for clusters of PCs. A bridging model has the advantage that if a program is correct, then this is the case for "all" physical architectures. Note that it is also the case for portable libraries such as MPI but algorithm design would be clearly architecture independent, which will be not the case using a bridging model. Moreover, it is known and accepted that correctness of programs is more costly in terms of work than just programming and designing algorithms. Hence the choice in this work of the BSP bridging model to provide both portability for proofs and a model for algorithmic design and efficient programs.

There exist other bridging models of parallel computations but only BSP is widely used and accepted for algorithm design. The BSP model has also been used with success in a wide variety of problems. A complete book of numerical algorithms is [2] and many other algorithms can be found in the literature.

B.2 The BSP model

The BSP architecture. We recall that a BSP computer is formed by 3 main components: (1) A set of **p** homogeneous pairs of processors-memories (units); (2) A communication network to exchange messages between these units of computations; (3) A global synchronization unit to execute global synchronization barriers. A wide range of actual architectures can be seen as BSP computers. Clusters of PCs and multi-cores, GPUs, *etc.* can be thus considered as BSP computers. Share memory machines could also be used in a way such that each processor only accesses a sub-part of the shared memory (which is then "private") and communications could be performed using a dedicated part of the memory.

The BSP model ignores the particular topology of the underlying machine; this rules out any use of network locality in algorithm design. The model only considers two levels of locality, local (inside the processor) and remote (outside a processor), with remote access usually being more expensive than local ones.

The execution model. A BSP program is logically executed as a sequence of *super-steps*, each of which is divided into three successive disjointed *phases*: (1) Each unit only uses its local data to perform *sequential* computations and

to request *data transfers* to other units; (2) The network delivers the requested data; (3) A *barrier* occurs, making the transferred data available for the next super-step. This *structured* model enforces a strict *separation* of communication and computation: during a super-step, no communication between the processors is allowed apart from transfer requests; only at the synchronization *barrier* is information actually exchanged. Messages sent in the previous superstep are available at the destination only at the start of the next superstep.

For performance, a BSP library can also send messages during the computation phase, but this is hidden to programmers. The best know examples being the BSPLIB for the C language and HAMA [20] for JAVA. A MPI program using *collective operations* only can also be viewed as a BSP program.

The cost model. The BSP model gives us a cost model that is both tractable and accurate. The *performance* of a BSP computer is characterized by 4 *parameters*: (1) The local processing speed **r**; (2) The number of processors **p**; (3) The time **L** required for a barrier; (4) The time **g** for collectively delivering a 1-relation which is a collective exchange where every processor receives/sends at most one word. The network can deliver an h-relation in time $\mathbf{g} \times h$.

The execution time (cost) of a super-step s is the sum of the maximal local processing time, the data delivery and the global synchronization times. The total cost (execution time) of a BSP algorithm is the sum of its super-step costs.

The partitioning of the data is a crucial issue. In fact, the choice of a distribution is one of the main means of influencing the performance of the algorithm. This leads to an emphasis on problem dependent techniques of data partitioning, instead of on hardware dependent techniques that take network topologies into account. The algorithm designer who is liberated from such hardware considerations may concentrate on exploiting the essential features of the problem.

B.3 Advantages and Disadvantages

This *structured* model of parallelism enforces a strict separation of communication and computation. This execution policy has three main advantages. Firstly, it removes non-determinism and guarantees the absence of deadlocks. This is also simply the most visible aspect of a parallel model that shifts the responsibility for timing and synchronisation issues from the applications to the communications library[3]. Secondly, it allows for an accurate model of performance prediction based on the throughput and latency of the network, and on the speed of processors. This performance prediction model can even be used at runtime to dynamically make decisions, for instance to choose whether to communicate in order to re-balance data, or to continue an unbalanced computation. Furthermore, barriers have also a number of attractions: it is harder to introduce the possibility of livelock, since barriers do not create circular data dependencies. Barriers also permit novel forms of fault tolerance. Third, because any BSP algorithm is

[3] BSP libraries are generally implemented using MPI or low level routines of the given specifics architectures.

organized as a sequence of supersteps, this makes it straightforward to extend techniques for constructing sequential algorithms to/from BSP algorithms.

The BSP model considers communication actions *en masse*. This is less flexible than asynchronous messages, but easier to debug since there are many communication actions in a parallel program, and their interactions are typically complex. Bulk sending also provides better performances since, from an implementation point of view, grouping communication together in a separate program phase permits a global optimisation of the data exchange by the communications library. Moreover it is easy to measure during the execution of a BSP program, time speeding to communicate and to synchronise by just adding chronos before and after the primitive of synchronization. This facility is mainly uses to compare different programs.

However, on most distributed multicore architectures, barriers are often expensive when the number of processors dramatically increases. Using BSP, programmers and designers have to keep in mind that some patterns are not BSP friendly: For example, BSP does not effectively manage pipeline and master/slave paradigms (farm of processes). Even if the BSP cost analysis proposes a method of estimating the work of a parallel algorithm, such a technique is not widespread: it seems that too many programmers still prefer using asynchronous send/received primitives (with potential data-races, deadlocks, *etc.*) instead of a structured model. We follow [10] that this manner of programming, as is the case with the "goto" statement, will be phased out in time (we can already see this for big-data frameworks such as PREGEL [14], MAPREDUCE [20], *etc.*).

Moreover, the BSP model, as a coarse-grained model of computation, has been proved to be very appropriate for problems with regular data-structures, and so, for problems based on domain decomposition. For irregular structures, some heuristics can improve the balance efficiently. But that is not as natural as spawning small processes (or threads) of calculation of each irregular part of the structure. That is the main inconvenient of the model.

B.4 Programming using BSP

BSP is thus defined as a distributed memory model with point-to-point communication between processors. The computation is divided into supersteps separated by global synchronization steps, and packets sent in one superstep are assumed to be delivered at the beginning of the next superstep. Each processor can communicate directly with every other processor, providing complete control over how the data is distributed between the processors in every superstep.

We now present the main libraries for BSP programming. The reader can thus understand why our communication function in ASM presented in Section D is sufficient to express most (all?) typical BSP algorithms. We do not present all the primitives but the most important ones only.

The standard BSPlib. The BSPLIB[4] [12] is a C library of communication routines. It aims to support the development of parallel algorithms based on the BSP model. It offers routines for both message passing (BSMP) and remote memory access (DRMA). The development of shared-memory based possibilities is justified by the intense use of DRMA routines in numerical algorithms. It is interesting to note that about 20 routines are available only, compared to the more than 200 that composed MPI. The BSPLIB is thus easier to understand and to use, without lack of expressivity for programming HPC architectures.

Within a BSP computation, we can query some information about the machine: `int bsp_nprocs()` returns the number of processors $\mathbf{p}$ and `int bsp_pid()` returns the processor identifier which belongs to $0, ..., \mathbf{p} - 1$. The barrier is done using `void bsp_sync()` which blocks the node until all other nodes have called `bsp_sync` and all messages sent to it have been received.

Sending a packet (in a buffering mode) is done using `void bsp_send(int pid, const void tag,const void  payload,int payload_bytes)`. The routine is based on the idea of a two-part message. A fixed-length part carries tag information that will help the receiver to interpret the message (`tag`); a variable-length part contains the main data `payload`. The length of the tag (`payload_bytes`) is required to be fixed during any particular superstep, but can vary between supersteps. The destination buffer of a processor may be viewed as a queue where incoming messages are enqueued in an arbitrary order. If the message is not accessed within the superstep it is removed from the buffer.

The programmer can know the number of received messages as well as the total size of received data (in bytes) using the routine `void bsp_qsize(int packets, int accum_nbytes)`. This routine works on the queue of received messages. To receive a message, the user should use the procedures `void bsp_get_tag(int status, void tag)` and `void bsp_move(void payload,int reception_bytes)`. `bsp_get_tag` returns the tag of the first message in the queue and the size of the corresponding payload (status is -1 if the queue is empty). `bsp_move` copies the payload of the first message of the system queue, *i.e.* the buffer call payload, and removes it from the queue. Then, the system will advance to the next message.

Registering or deleting a variable from global access is done using:
`void bsp_push_reg (const void ident, int size)` and
`void bsp_popregister(const void ident)`. Due to the SPMD structure of BSP programs, if p instances share the same name, they will not, in general, have the same physical address. To allow BSP programs to be executed correctly, the BSPLIB provides a mechanism for relating these various addresses by creating associations called registrations (not show here). Now, the two DRMA routines operations (Fig 4) are: (1) `void bsp_get(int pid,const void src,int offset, void dst,int nbytes)` stands for global reading access; It copies n bytes to the local memory address —`dst` from the variable `src` at `offset` of the remote processor pid; (2) `void bsp_put(int pid,const void src,void dst,int offset, int nbytes)` stands for global writing access; It copies n bytes from local memory `src` to `dst` at `offset` on remote processor `pid`. It is important to note that

[4] http://www.bsp-worldwide.org/

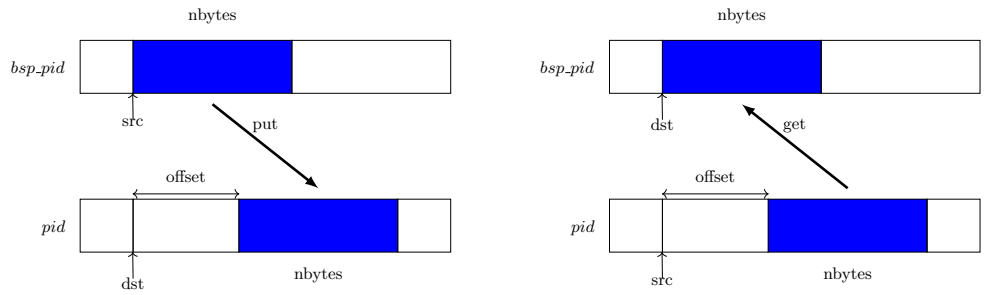

Fig. 4. DRMA BSP operations: "put" and "get".

the `get` and `put` operations are executed during the synchronisation step and all `get` are served before a `put` overwrites a value.

The Paderborn University BSP is another library for C which is close to the BSPLIB. For BSP computing, the main differences come from sending routines. Sending a single message (buffer) can be done using `void bsp_send(int dest, void* buffer, int size)`. After the calling, the buffer may be overwritten or freed. Each processor can access the received messages of type `t_bspmsg`. This can be done using `t_bspmsg* bsp_findmsg(t_bsp* bsp, int id, int index)` where `id` is the "id" of the source-node and `index` of the message. To access to the message, we need `void* bspmsg_data(t_bspmsg* msg)` which returns a pointer to the sending block of data and `int bspmsg_size(t_bspmsg* msg)` which returns its size.

The BSPLIB has been recently implemented for multi-core[5] and GPUs [6]. The PUB has been implementation for Java[7] and in the context of big-data[8] [20].

Other libraries Another way is using the standard MPI notably the collective operations [2]. But it is the responsible of the programmer to follow the BSP model when using asynchronous sending (potential deadlock). The number of routines of MPI is huge and understanding all their interactions is for expert.

NestStep[9] is a programming language, *à la* C, dedicated to BSP. It adds a virtual shared memory where the memory consistency is relaxed (but deterministic) during supersteps. The main method of a program is executed by all available processors. The main primitives of NestStep are the following: `step`: executes a statement in parallel; `combine` gather the results of the parallel execution (barrier); `forall` allows to parallel write (with barrier) to a shard array.

[5] http://www.multicorebsp.com/
[6] http://www.kunzhou.net/2008/BSGP.pdf
[7] http://www.lume.ufrgs.br/bitstream/handle/10183/18662/000731056.pdf;sequence=1
[8] https://hama.apache.org/
[9] http://www.ida.liu.se/~chrke55/neststep/index.html

PREGEL[10] [14] is a powerful (proprietary) language dedicated to graph algorithms. The approach centers around computations on the vertices of the graph. Each vertex of the graph has a unique "id", an associated value and a list of outgoing weighted edges. On each superstep, each worker node invokes a procedure `Compute` for each active vertex that is under its control. his procedure is responsible for the execution of the algorithm and is allowed, among other actions, to invoke other methods, compute new values for the vertex, add or remove vertices and edges, and send messages to other vertices. These messages are exchanged directly among the vertices, even if the vertices are being executed on different machines of the platform. The messages are sent asynchronously in order to allow the overlapping of computation and communication, but are delivered to the destination vertex only on the beginning of the next superstep. If a vertex declares that all its processing was done, it sends a message informing all the other nodes and deactivates itself. Any change on the topology is only performed on the next superstep, before the invocation of the `Compute` procedure.

Last is BSML[11] [13], a functional (OCAML extension) language for BSP. It uses a small set of primitives which are working over parallel data structures called a *parallel vectors*. A vector expresses that each of the $\mathbf{p}$ processors *embeds* a value of any type. There is two primitives for the asynchronous manipulations of vectors and also two for the communications (barrier). These primitives and the use of vectors has been used (and adapted) to PYTHON[12] and C++[13].

[10] The free version: http://giraph.apache.org/
[11] http://traclifo.univ-orleans.fr/BSML/
[12] http://dirac.cnrs-orleans.fr/ScientificPython/
[13] https://github.com/jfalcou/BSPPP

C Proofs of technical lemmas

C.1 Replacement in a Structure

Definition 9 (Replacement in a Structure). *Let X be a structure, and let $U_1 \subseteq \mathcal{U}(X)$ and U_2 be two sets such that there exists a bijection φ from U_1 to U_2. The structure Y obtained by replacing in X the elements of U_1 by the elements of U_2 is defined by:*

1. $\mathcal{L}(Y) = \mathcal{L}(X)$
2. $\mathcal{U}(Y) = (\mathcal{U}(X) \setminus U_1) \cup U_2$
3. *If $c \in \mathcal{L}(X)$ is a 0-ary symbol then:*

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

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

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

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

A replacement in a structure is a structure. Moreover, we proved in [15] that the replacement is an isomorphism:

Lemma 5 (A Replacement is an Isomorphism). *If X is a structure, $U_1 \subseteq \mathcal{U}(X)$ and $U_2 \cap \mathcal{U}(X) = \emptyset$, then the structure Y obtained by replacing in X the elements of U_1 by the elements of U_2 is isomorphic to X. Moreover, if T is a set of $\mathcal{L}(X)$ terms closed by subterms and such that for every $t \in T$, $\overline{t}^X \notin U_1$, then for every $t \in T$, $\overline{t}^X = \overline{t}^Y$.*

Proof. See [15] □

Therefore, according to the second postulate, a replacement with fresh values in one local memory of a state produced a state. We prove here the Lemma p.6 stating that every value in an update is a critical element:

Lemma 6 (Critical Elements). *For every state $\left(X^1, \ldots, X^p\right)$ of the parallel algorithm A, for every $1 \leq i \leq p$, if $(f, a_1, \ldots, a_\alpha, b) \in \Delta^i(A, \overrightarrow{X})$ then $a_1, \ldots, a_\alpha, b$ are interpretations in X^i of terms in $T(A)$.*

Proof. The proof is made by contradiction.

We assume that there exists an update $(f, a_1, \ldots, a_\alpha, a_0) \in \Delta^i(A, \overrightarrow{X})$ such that at least one a_i is not an interpretation in X^i of a term in $T(A)$.

Let v be a fresh value, and Y^i be the structure obtained by replacing a_i by v in X^i. According to the lemma 5, X^i and Y^i are isomorphic.

So, according to the second postulate, because $\overrightarrow{X} = \left(X^1, \dots, X^i, \dots, X^p\right)$ is a state of A, $\overrightarrow{Y} = \left(X^1, \dots, Y^i, \dots, X^p\right)$ is also a state of A.

Because a_i is not an interpretation in X^i of a term in $T(A)$, the states $\left(X^1, \dots, X^i, \dots, X^p\right)$ and $\left(X^1, \dots, Y^i, \dots, X^p\right)$ coincide over $T(A)$.

So, according to the third postulate, $\Delta^i(A, \overrightarrow{X}) = \Delta^i(A, \overrightarrow{Y})$.

But $a_i \notin \mathcal{U}(Y^i)$, so $(f, a_1, \dots, a_\alpha, a_0)$ cannot appear in $\Delta^i(A, \overrightarrow{Y})$, which contradicts $(f, a_1, \dots, a_\alpha, a_0) \in \Delta^i(A, \overrightarrow{X})$. □

We prove here the Lemma p.7 stating that for each step of the algorithm only a bounded amount of work is done:

Lemma 7 (Bounded Set of Updates).
For every state $\left(X^1, \dots, X^p\right)$ of the parallel algorithm A, for every $1 \leq i \leq p$, $\#\Delta^i(A, \overrightarrow{X})$ is bounded.

Proof. According to the Lemma 6, if $(f, a_1, \dots, a_\alpha, b) \in \Delta^i(A, \overrightarrow{X})$ then $a_1, \dots, a_\alpha, b$ are interpretations in X^i of terms in $T(A)$.

But, according to the third postulate, $T(A)$ is finite. So there exists a bounded number of possible $a_1, \dots, a_\alpha, b$ in $\Delta^i(A, \overrightarrow{X})$.

Moreover, because $\mathcal{L}(A)$ is finite there exists a bounded number of dynamic symbols f. Therefore $\Delta^i(A, \overrightarrow{X})$ has a bounded number of updates. □

C.2 Computation and Communication Phases

Lemma 8 (Computing States are Closed by Multi-Isomophism).
If the state $\left(X^1, \dots, X^p\right)$ is in a computing phase and multi-isomorphic to the state $\left(Y^1, \dots, Y^p\right)$, then $\left(Y^1, \dots, Y^p\right)$ is in a computing phase too.

Proof. The proof is made by contradiction. We assume that $\left(Y^1, \dots, Y^p\right)$ is not in a computing phase, so for every $1 \leq i \leq p$, we have $\mathrm{comp}_A(Y^i) = Y^i$.

Because $\left(X^1, \dots, X^p\right)$ is in a computing phase, according to the fourth postulate we have:

$$\tau_A\left(X^1, \dots, X^p\right) = \left(\mathrm{comp}_A(X^1), \dots, \mathrm{comp}_A(X^p)\right)$$

Let $\overrightarrow{\zeta} = (\zeta_1, \dots, \zeta_p)$ be a multi-isomorphism from $\left(X^1, \dots, X^p\right)$ to $\left(Y^1, \dots, Y^p\right)$.

According to the second postulate, the transition function commutes with multi-isomorphisms, so we have:

$$\begin{aligned}\tau_A\left(Y^1, \dots, Y^p\right) &= \tau_A\left(\zeta_1(X^1), \dots, \zeta_p(X^p)\right) \\ &= \overrightarrow{\zeta}\left(\tau_A\left(X^1, \dots, X^p\right)\right)\end{aligned}$$

According to the fourth postulate, the computation function commutes with multi-isomorphisms, so we have:

$$\begin{aligned}\overrightarrow{\zeta}\left(\tau_A\left(X^1, \dots, X^p\right)\right) &= \overrightarrow{\zeta}\left(\mathrm{comp}_A(X^1), \dots, \mathrm{comp}_A(X^p)\right) \\ &= \left(\mathrm{comp}_A\left(\zeta_1(X^1)\right), \dots, \mathrm{comp}_A\left(\zeta_p(X^p)\right)\right) \\ &= \left(\mathrm{comp}_A(Y^1), \dots, \mathrm{comp}_A(Y^p)\right)\end{aligned}$$

We assumed that for every $1 \leq i \leq p$, we have $\text{comp}_A(Y^i) = Y^i$, so:

$$\left(\text{comp}_A(Y^1), \ldots, \text{comp}_A(Y^p)\right) = \left(Y^1, \ldots, Y^p\right)$$

Therefore $\tau_A\left(Y^1, \ldots, Y^p\right) = \left(Y^1, \ldots, Y^p\right)$. So we have:

$$\overrightarrow{\zeta}\left(\tau_A\left(X^1, \ldots, X^p\right)\right) = \overrightarrow{\zeta}\left(X^1, \ldots, X^p\right)$$

By applying $\overrightarrow{\zeta}^{-1}$ on both sides we have:

$$\tau_A\left(X^1, \ldots, X^p\right) = \left(X^1, \ldots, X^p\right)$$

But $\tau_A\left(X^1, \ldots, X^p\right) = \left(\text{comp}_A(X^1), \ldots, \text{comp}_A(X^p)\right)$, so for every $1 \leq i \leq p$, we have $\text{comp}_A(X^i) = X^i$, which contradicts that $\left(X^1, \ldots, X^p\right)$ is in a computing phase. □

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

Lemma 9 (Properties of the Communication Function). *For every* BSP *algorithm A and for every state in communication phase, comm_A preserves the universes, the number of processors, and commutes with multi-isomorphisms.*

Proof. Let $\overrightarrow{X}$ be a state in a communication phase.

Because $\overrightarrow{X}$ is in a communication phase, according to the fourth postulate, $\text{comm}_A(\overrightarrow{X}) = \tau_A(\overrightarrow{X})$. Therefore, according to the second postulate, comm_A preserves the universes and the number of processors.

Let $\overrightarrow{\zeta}$ be a multi-isomorphism between $\overrightarrow{X}$ and another state.

Because $\overrightarrow{X}$ is in a communication phase, according to the previous lemma, $\overrightarrow{\zeta}(\overrightarrow{X})$ is in a communication phase too. So, according to the fourth postulate, $\tau_A\left(\overrightarrow{\zeta}(\overrightarrow{X})\right) = \text{comm}_A\left(\overrightarrow{\zeta}(\overrightarrow{X})\right)$.

Because $\tau_A(\overrightarrow{X}) = \text{comm}_A(\overrightarrow{X})$, we have that $\overrightarrow{\zeta}\left(\tau_A(\overrightarrow{X})\right) = \overrightarrow{\zeta}\left(\text{comm}_A(\overrightarrow{X})\right)$.

But, according to the second postulate, $\tau_A\left(\overrightarrow{\zeta}(\overrightarrow{X})\right) = \overrightarrow{\zeta}\left(\tau_A(\overrightarrow{X})\right)$.

Therefore $\text{comm}_A\left(\overrightarrow{\zeta}(\overrightarrow{X})\right) = \overrightarrow{\zeta}\left(\text{comm}_A(\overrightarrow{X})\right)$. □

Because the states of an algorithm may not be accessible, the set of states $S(A)$ may be extended with new tuples of local memories which are compatible with the execution. According to the third postulate, only the critical elements of the local memories matter to compute a step of the execution.

So, we will assume[14] that $S(A)$ is **closed with respect to the exploration witness**[15], which means that if $\left(X^1, \ldots, X^i, \ldots, X^p\right)$ is a state and

[14] This assumption is without cost, because we can construct the algorithm B with $S(B) \supseteq S(A)$ and prove results like the Lemma p.31 for B, then apply the result for the restricted set of states $S(A)$.

[15] And similarly, in the Definition p.8 of ASM_{BSP}, $S(M)$ should be closed with respect of $T(M) = T(\Pi) \cup T(\text{comm}_M)$, where $T(\Pi) = \{true\} \cup \text{Read}(\Pi) \cup \text{Write}(\Pi)$.

the structure Y^i has the same language and critical elements than X^i, then $\big(X^1, \dots, Y^i, \dots, X^p\big)$ is a state too.

Notice that two states may coincide over $T(A)$ without being isomorphic, so this is truely an extension of the set of states. The point is to apply the third postulate on a bigger set of states in the proof of the lemma 10 p.28.

Lemma 10 (No Communication during Computation Phases).
For every states $\big(X^1, \dots, X^p\big)$ and $\big(Y^1, \dots, Y^q\big)$ in a computing phase, if X^i and Y^j have the same critical elements then $\Delta^i(A, \overrightarrow{X}) = \Delta^j(A, \overrightarrow{Y})$.

Proof. $\big(X^1, \dots, X^p\big)$ and $\big(Y^1, \dots, Y^q\big)$ are in a computing phase, so according to the fourth postulate:

$$\begin{aligned} \tau_A\big(X^1, \dots, X^p\big) &= \big(\mathrm{comp}_A(X^1), \dots, \mathrm{comp}_A(X^p)\big) \\ \tau_A\big(Y^1, \dots, Y^q\big) &= \big(\mathrm{comp}_A(Y^1), \dots, \mathrm{comp}_A(Y^q)\big) \end{aligned}$$

So $\Delta^i(A, \overrightarrow{X}) = \mathrm{comp}_A(X^i) \ominus X^i$ and $\Delta^j(A, \overrightarrow{Y}) = \mathrm{comp}_A(Y^j) \ominus Y^j$.
The proof is made by case:

1. If $\mathrm{comp}_A(X^i) = X^i$ and $\mathrm{comp}_A(Y^j) = Y^j$, then $\Delta^i(A, \overrightarrow{X}) = \emptyset = \Delta^j(A, \overrightarrow{Y})$.
2. Otherwise $\mathrm{comp}_A(X^i) \neq X^i$ or $\mathrm{comp}_A(Y^j) \neq Y^j$. We assume $\mathrm{comp}_A(Y^j) \neq Y^j$, the other case $\mathrm{comp}_A(X^i) \neq X^i$ being similar.
   Let $\overrightarrow{Z}$ be the p-tuple $\big(X^1, \dots, Y^j, \dots, X^p\big)$, which is the state $\big(X^1, \dots, X^i, \dots, X^p\big)$ where X^i has been replaced by Y^j.
   X^i and Y^j have the same critical elements, and we assumed that $S(A)$ is closed with respect to the exploration witness, so because $\overrightarrow{X}$ is a state $\overrightarrow{Z}$ is a state too.
   X^i and Y^j have the same critical elements, so the states $\overrightarrow{X}$ and $\overrightarrow{Z}$ coincide over $T(A)$. Therefore, according to the third postulate, $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \overrightarrow{\Delta}(A, \overrightarrow{Z})$. In particular $\Delta^i(A, \overrightarrow{X}) = \Delta^i(A, \overrightarrow{Z})$.
   We assumed that $\mathrm{comp}_A(Y^j) \neq Y^j$, so $\overrightarrow{Z}$ is in a computing phase. So, according to the third postulate, we have:

$$\tau_A(\overrightarrow{Z}) = \big(\mathrm{comp}_A(X^1), \dots, \mathrm{comp}_A(Y^j), \dots, \mathrm{comp}_A(X^p)\big)$$

   So $\Delta^i(A, \overrightarrow{Z}) = \mathrm{comp}_A(Y^j) \ominus Y^j = \Delta^j(A, \overrightarrow{Y})$.
   Therefore $\Delta^i(A, \overrightarrow{X}) = \Delta^i(A, \overrightarrow{Z}) = \Delta^j(A, \overrightarrow{Y})$.

□

$T(\mathrm{comm}_M)$ is defined in the Definition p.8. Read (Π) and Write (Π) are defined in Definition p.30.

C.3 Detailed Proofs for ASM-BSP

Let ζ be an isomorphism from the structure X to the structure Y, and let $(f, a_1, \ldots, a_\alpha, b)$ be an update of X. We will denote by $\zeta(f, a_1, \ldots, a_\alpha, b)$ the update $(f, \zeta(a_1), \ldots, \zeta(a_\alpha), \zeta(b))$ of Y. We generalize this notation to set of updates:

$$\zeta(\{u_1, \ldots, u_k\}) \stackrel{\text{def}}{=} \{\zeta(u_1), \ldots, \zeta(u_k)\}$$

Lemma 11 (Isomorphic ASM).
For every ASM *program Π with signature $\mathcal{L}(X)$ and every isomorphism ζ from X:*

$$\zeta\left(X \oplus \Delta(\Pi, X)\right) = \zeta(X) \oplus \Delta(\Pi, \zeta(X))$$

Proof. We remind the definition of the updates:

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

where $X \oplus \Delta$ has the same universe and signature than X.

So $\zeta\left(X \oplus \Delta(\Pi, X)\right)$ and $\zeta(X) \oplus \Delta(\Pi, \zeta(X))$ has the same universe $\mathcal{U}(\zeta(X))$ and the same language $\mathcal{L}(X)$.

Therefore, to prove that $\zeta\left(X \oplus \Delta(\Pi, X)\right) = \zeta(X) \oplus \Delta(\Pi, \zeta(X))$, we have to prove that they have the same interpretation for every symbol $f \in \mathcal{L}(X)$.

According to the definition of the updates:

$$\overline{f}^{\zeta(X \oplus \Delta)}(\overrightarrow{\zeta(a)}) = \begin{cases} \zeta(b) & \text{if } (f, \overrightarrow{a}, b) \in \Delta \text{ and } \Delta \text{ is consistent} \\ \zeta\left(\overline{f}^{X}(\overrightarrow{a})\right) & \text{else} \end{cases}$$

$$\overline{f}^{\zeta(X) \oplus \zeta(\Delta)}(\overrightarrow{\zeta(a)}) = \begin{cases} \zeta(b) & \text{if } (f, \overrightarrow{\zeta(a)}, \zeta(b)) \in \zeta(\Delta) \text{ and } \zeta(\Delta) \text{ is consistent} \\ \zeta\left(\overline{f}^{X}(\overrightarrow{a})\right) & \text{else} \end{cases}$$

By definition of $\zeta(\Delta)$, $(f, \overrightarrow{a}, b) \in \Delta$ if and only if $(f, \overrightarrow{\zeta(a)}, \zeta(b)) \in \zeta(\Delta)$. Moreover, we remind that Δ is inconsistent means that there exists $(f, \overrightarrow{a}, b), (f, \overrightarrow{a}, b') \in \Delta$ with $b \neq b'$. Because ζ is an isomorphism from X, according to the remark p.17, $a = b$ in X if and only if $\zeta(a) = \zeta(b)$ in $\zeta(X)$. So Δ is consistent if and only if $\zeta(\Delta)$ is consistent.

Therefore, for every set of updates Δ, $\zeta(X \oplus \Delta) = \zeta(X) \oplus \zeta(\Delta)$. Moreover, according to the operational semantics of the ASM, we have $\zeta\left(\Delta(\Pi, X)\right) = \Delta(\Pi, \zeta(X))$. Therefore $\zeta\left(X \oplus \Delta(\Pi, X)\right) = \zeta(X) \oplus \zeta\left(\Delta(\Pi, X)\right) = \zeta(X) \oplus \Delta\left(\Pi, \zeta(X)\right)$. □

We recall: Let $\overrightarrow{X} = (X^1, \ldots, X^p)$ be a state of A. According to the transition function τ_A, the next state is $\tau_A(\overrightarrow{X})$, which will be denoted by $(\tau_A(\overrightarrow{X})^1, \ldots, \tau_A(\overrightarrow{X})^p)$. 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 $\overrightarrow{\Delta}(A, \overrightarrow{X}) \stackrel{\text{def}}{=} (\Delta^1(A, \overrightarrow{X}), \ldots, \Delta^p(A, \overrightarrow{X}))$ the "multiset" of updates done by A on the state $\overrightarrow{X}$. In particular, if a state $\overrightarrow{X}$ is final, then $\tau_A(\overrightarrow{X}) = \overrightarrow{X}$, so $\overrightarrow{\Delta}(A, \overrightarrow{X}) = \overrightarrow{\emptyset}$.

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

Lemma 12 (Multi-Isomorphic Set of Updates).
If $\overrightarrow{\zeta}$ is a multi-isomorphism from the state $\overrightarrow{X}$ to the state $\overrightarrow{Y}$ then:

$$\overrightarrow{\zeta}\left(\overrightarrow{\Delta}(A, \overrightarrow{X})\right) = \overrightarrow{\Delta}(A, \overrightarrow{Y})$$

Proof. $\overrightarrow{X}$ is a p-tuple. Let $1 \leq i \leq p$.

$$\begin{aligned}
\zeta_i\left(\Delta^i(A, \overrightarrow{X})\right) &= \zeta_i\left(\tau_A(\overrightarrow{X})^i \ominus X^i\right) \\
&\quad \text{by Definition p.29 of } \Delta^i(A, \overrightarrow{X}) \\
&= \zeta_i\left(\left\{(f, \overrightarrow{a}, b) \mid \overline{f}^{\tau_A(\overrightarrow{X})^i}(\overrightarrow{a}) = b \text{ and } \overline{f}^{X^i}(\overrightarrow{a}) \neq b\right\}\right) \\
&\quad \text{by Definition p.30 of the difference} \\
&= \left\{\zeta_i(f, \overrightarrow{a}, b) \mid \overline{f}^{\tau_A(\overrightarrow{X})^i}(\overrightarrow{a}) = b \text{ and } \overline{f}^{X^i}(\overrightarrow{a}) \neq b\right\} \\
&\quad \text{by the previous definition} \\
&= \left\{\zeta_i(f, \overrightarrow{a}, b) \mid \overline{f}^{\zeta_i\left(\tau_A(\overrightarrow{X})^i\right)}(\overrightarrow{\zeta_i(a)}) = \zeta_i(b) \text{ and } \overline{f}^{\zeta_i(X^i)}(\overrightarrow{\zeta_i(a)}) \neq \zeta_i(b)\right\} \\
&\quad \text{according to the Remark p.17 on formulas} \\
&= \zeta_i\left(\tau_A(\overrightarrow{X})^i\right) \ominus \zeta_i\left(X^i\right) \\
&\quad \text{by Definition p.30 of the difference} \\
&= \tau_A\left(\overrightarrow{\zeta}(\overrightarrow{X})\right)^i \ominus \zeta_i\left(X^i\right) \\
&\quad \text{according to the second postulate} \\
&= \tau_A(\overrightarrow{Y})^i \ominus Y^i \\
&\quad \text{because } \overrightarrow{\zeta} \text{ is a multi-isomorphism from } \overrightarrow{X} \text{ to } \overrightarrow{Y} \\
&= \Delta^i(A, \overrightarrow{Y}) \\
&\quad \text{by Definition p.29 of } \Delta^i(A, \overrightarrow{X})
\end{aligned}$$

□

Definition 10 (Updates (Terms Read/Write) of ASM programs).

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

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

Lemma 13 (Each Transition is Captured by an ASM).
For every state $\left(X^1, \ldots, X^p\right)$ *of the parallel algorithm* A*, and for every* $1 \leq i \leq p$*, there exists an* ASM *program* $\Pi_i^{\overrightarrow{X}}$ *such that Read* $\left(\Pi_i^{\overrightarrow{X}}\right) \subseteq T(A)$ *and* $\Delta(\Pi_i^{\overrightarrow{X}}, X^i) = \Delta^i(A, \overrightarrow{X})$.

Proof. According to the Lemma p.7, $\Delta^i(A, \overrightarrow{X})$ contains a bounded number m of updates $(f^1, a_1^1, \ldots, a_{\alpha^1}^1, a_0^1), \ldots, (f^m, a_1^m, \ldots, a_{\alpha^m}^m, a_0^m)$.

According to the Lemma p.6, for every update $(f, a_1, \ldots, a_\alpha, a_0) \in \Delta^i(A, \overrightarrow{X})$ there exists $t_1, \ldots, t_\alpha, t_0 \in T(A)$ such that for every $0 \leq j \leq \alpha$ we have $\overline{t_j}^{X^i} = a_j$. So, the command $f(t_1, \ldots, t_\alpha) := t_0$ is interpreted by $(f, a_1, \ldots, a_\alpha, a_0)$ in X^i.

Therefore, there exists m ASM commands $f^1(t_1^1, \ldots, t_{\alpha^1}^1) := t_0^1, \ldots, f^m(t_1^m, \ldots, t_{\alpha^m}^m) := t_0^m$ which are respectively interpreted by $(f^1, a_1^1, \ldots, a_{\alpha^1}^1, a_0^1), \ldots, (f^m, a_1^m, \ldots, a_{\alpha^m}^m, a_0^m)$ in X^i, such that every term t_j^k read in these commands is in $T(A)$.

Let $\Pi_i^{\overrightarrow{X}}$ be the following program:

$$\begin{array}{l}\texttt{par } f^1(t_1^1, \ldots, t_{\alpha^1}^1) := t_0^1 \\ \quad \| \ f^2(t_1^2, \ldots, t_{\alpha^2}^2) := t_0^2 \\ \quad \vdots \\ \quad \| \ f^m(t_1^m, \ldots, t_{\alpha^m}^m) := t_0^m \\ \texttt{endpar}\end{array}$$

By using the ASM operational semantics, we proved that:

$$\begin{aligned}\Delta(\Pi_i^{\overrightarrow{X}}, X^i) &= \{(f^1, a_1^1, \ldots, a_{\alpha^1}^1, a_0^1), \ldots, (f^m, a_1^m, \ldots, a_{\alpha^m}^m, a_0^m)\} \\ &= \Delta^i(A, \overrightarrow{X})\end{aligned}$$

□

The problem is there can be an infinite number of such ASM program $\Pi_i^{\overrightarrow{X}}$, so we need to narrow the number of relevant states. In order to do that, we will use the finiteness of the exploration witness $T(A)$. For each processor memory X, we denote by E_X the equivalence relation on pairs (t_1, t_2) of terms in $T(A)$ defined by:

$$E_X(t_1, t_2) \stackrel{\text{def}}{=} \begin{cases} \textit{true} \text{ if } \overline{t_1}^X = \overline{t_2}^X \\ \textit{false} \text{ otherwise} \end{cases}$$

We prove that the number of relevant states can be reduced to the number of the relations E_X:

Lemma 14 (Syntactically Equivalent Memories).
For every BSP *algorithm* A *and for every state* $\left(X^1, \ldots, X^p\right)$ *and* $\left(Y^1, \ldots, Y^q\right)$ *in a computing phase, if* $E_{X^i} = E_{Y^j}$ *then* $\Delta(\Pi_i^{\overrightarrow{X}}, Y^j) = \Delta^j(A, \overrightarrow{Y})$.

Proof. Let $\tilde{Y}^j$ be the structure obtained by replacing in $\mathcal{U}(Y^j)$ the elements appearing both in $\mathcal{U}(X^i)$ and $\mathcal{U}(Y^j)$ by fresh values. According to the Lemma p.25, Y^j and $\tilde{Y}^j$ are isomorphic.

Let Z^j be the structure obtained by replacing in $\mathcal{U}(\tilde{Y}^j)$ the critical elements of $\tilde{Y}^j$ by the critical elements of X^i. Because $E_{X^i} = E_{Y^j}$, this operation is well-defined[16]. Because $\mathcal{U}(X^i) \cap \mathcal{U}(\tilde{Y}^j) = \emptyset$, according to the Lemma p.25, $\tilde{Y}^j$ and Z^j are isomorphic.

Therefore, Y^j and Z^j are isomorphic. So, because $\overrightarrow{Y} = \left(Y^1, \dots, Y^j, \dots, Y^q\right)$ is a state, according to the second postulate, the q-tuple $\overrightarrow{Z} = \left(Y^1, \dots, Z^j, \dots, Y^q\right)$ obtained by replacing Y^j by Z^j in $\overrightarrow{Y}$ is a state too.

Because $\overrightarrow{Y}$ is in a computing phase and is multi-isomorphic to the state $\overrightarrow{Z}$, according the Lemma p.26, $\overrightarrow{Z}$ is in a computation phase too.

The states $\overrightarrow{X} = \left(X^1, \dots, X^i, \dots, X^p\right)$ and $\overrightarrow{Z}$ are in a computing phase, and X^i and Z^j have the same critical elements. Therefore, according to the Lemma p.7 we have $\Delta^i(A, \overrightarrow{X}) = \Delta^j(A, \overrightarrow{Z})$.

According to the Lemma p.31, $\Delta(\Pi_i^{\overrightarrow{X}}, X^i) = \Delta^i(A, \overrightarrow{X})$. Therefore $\Delta(\Pi_i^{\overrightarrow{X}}, X^i) = \Delta^j(A, \overrightarrow{Z})$. According to the Lemma p.31, $\text{Read}\left(\Pi_i^{\overrightarrow{X}}\right) \subseteq T(A)$. So, because X^i and Z^j have the same critical elements they coincide over $\text{Read}\left(\Pi_i^{\overrightarrow{X}}\right)$, so we have $\Delta(\Pi_i^{\overrightarrow{X}}, X^i) = \Delta(\Pi_i^{\overrightarrow{X}}, Z^j)$. Therefore $\Delta(\Pi_i^{\overrightarrow{X}}, Z^j) = \Delta^j(A, \overrightarrow{Z})$.

Let ζ be an isomorphism between Y^j and Z^j. According to the Lemma p.30, $\zeta(\Delta^j(A, \overrightarrow{Y})) = \Delta^j(A, \overrightarrow{Z})$. Moreover, because for every $t \in T(A)$ we have $\zeta(\overline{t}^{Y^j}) = \overline{t}^{Z^j}$, we have $\zeta(\Delta(\Pi_i^{\overrightarrow{X}}, Y^j)) = \Delta(\Pi_i^{\overrightarrow{X}}, Z^j)$.

Therefore $\zeta(\Delta(\Pi_i^{\overrightarrow{X}}, Y^j)) = \zeta(\Delta^j(A, \overrightarrow{Y}))$, and by applying ζ^{-1} on both sides we have the result $\Delta(\Pi_i^{\overrightarrow{X}}, Y^j) = \Delta^j(A, \overrightarrow{Y})$. □

Proposition 3 (BSP-ASMs capture Computations of BSP Algorithms).

For every BSP *algorithm A, there exists an* ASM *program Π_A such that for every state $\overrightarrow{X}$ in a computation phase:*

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

Proof. Let c be the number of relations E_X, where X is a local memory from a state in a computing phase.

According to the third postulate, $T(A)$ is finite, so there exists n such that $T(A) = \{t_1, \dots, t_n\}$. For every local memory X, E_X is an equivalence relation for the elements of $T(A)$. So, c is bounded by the n-th Bell number B_n, which is the number of equivalence relations on a set that has exactly n elements.

Therefore, there exists a bounded number c of local memories $Y_1^{i_1}, \dots, Y_c^{i_c}$ from states $\overrightarrow{Y_1}, \dots, \overrightarrow{Y_c}$ in a computing phase, such that for every local memory X from a state in a computing phase there exists one and only one $1 \leq j \leq c$ such that $E_X = E_{Y_j^{i_j}}$.

[16] If $\overline{t_1}^{\tilde{Y}^j} = \overline{t_2}^{\tilde{Y}^j}$ then, by using the isomorphism between Y^j and $\tilde{Y}^j$, we have $\overline{t_1}^{Y^j} = \overline{t_2}^{Y^j}$. So, because $E_{X^i} = E_{Y^j}$, we have $\overline{t_1}^{X^i} = \overline{t_2}^{X^i}$.

Let $\Pi_1, \ldots, \Pi_c$ be the ASM programs obtained at the Lemma p.31 such that for every $1 \leq j \leq c$, $\Delta(\Pi_j, Y_j^{i_j}) = \Delta^{i_j}(A, \overrightarrow{Y_j})$.

For every relation $E_{Y_j^{i_j}}$, let F_j be the formula[17] defined by:

$$F_j \stackrel{\text{def}}{=} \bigwedge_{1 \leq k,\ell \leq n} E_{k\ell} \text{ where } E_{k\ell} = \begin{cases} t_k = t_\ell \text{ if } E_{Y_j^{i_j}}(t_k, t_\ell) \text{ is true} \\ t_k \neq t_\ell \text{ otherwise} \end{cases}$$

Notice that according to the definition of $E_{Y_j^{i_j}}$ and F_j, we have that F_j is true in a local memory X if and only if $E_X = E_{Y_j^{i_j}}$. Therefore, 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. These formulas are sometimes called the **guards** of the program Π_A, which is defined as the following ASM program:

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

Such an ASM program with guards and non-trivial updates will be called a **program in normal form**.

Let $\overrightarrow{X} = \left(X^1, \ldots, X^p\right)$ be a state in a computation phase, and $1 \leq i \leq p$. We prove now that $\Delta(\Pi_A, X^i) = \Delta^i(A, \overrightarrow{X})$.

We proved that there exists one and only one $1 \leq j \leq c$ such that $E_{X_i} = E_{Y_j^{i_j}}$. So, according to the definition of the formulas $F_1, \ldots, F_c$, we have that F_j is true in X_i and the other formulas are false in X_i. Therefore $\Delta(\Pi_A, X^i) = \Delta(\Pi_j, X^i)$.

Because $\overrightarrow{X}$ and $\overrightarrow{Y_j}$ are in a computation phase, and because $E_{X_i} = E_{Y_j^{i_j}}$, according to the Lemma p.31 we have: $\Delta(\Pi_j, X^i) = \Delta^i(A, \overrightarrow{X})$.

Therefore, we proved for every $1 \leq i \leq p$ that $\Delta(\Pi_A, X^i) = \Delta^i(A, \overrightarrow{X})$. □

Theorem 2. $\text{ALGO}_{\text{BSP}} \subseteq \text{ASM}_{\text{BSP}}$

Proof. Let A be the BSP algorithm $(S(A), I(A), \tau_A)$. According to the fourth postulate, there exists comp_A and comm_A such that for every state $\overrightarrow{X}$:

$$\tau_A(\overrightarrow{X}) = \begin{cases} \overrightarrow{\text{comp}_A}(\overrightarrow{X}) \text{ if } \overrightarrow{X} \text{ is in a computation phase} \\ \text{comm}_A(\overrightarrow{X}) \text{ otherwise} \end{cases}$$

where $\overrightarrow{\text{comp}_A}\left(X^1, \ldots, X^p\right) = \left(\text{comp}_A(X^1), \ldots, \text{comp}_A(X^p)\right)$. According to the proposition 1, there exists an ASM program Π_A such that for every state $\overrightarrow{X}$ in a computation phase, we have $\overrightarrow{\Delta}(\Pi_A, \overrightarrow{X}) = \overrightarrow{\Delta}(A, \overrightarrow{X})$.

[17] This is the reason why we assumed that every signature in this paper contains the booleans and the equality.

So, for every $1 \leq i \leq p$, we have:

$$\begin{aligned} X^i \oplus \Delta(\Pi_A, X^i) &= X^i \oplus \Delta^i(A, \overrightarrow{X}) \\ &= X^i \oplus \left(\tau_A(\overrightarrow{X})^i \ominus X^i\right) \\ &= \tau_A(\overrightarrow{X})^i \\ &= \mathrm{comp}_A(X^i) \end{aligned}$$

Therefore, we have for every state $\overrightarrow{X}$ in a computation phase:

$$\begin{aligned} \tau_{\Pi_A}\left(X^1, \dots, X^p\right) &= \left(X^1 \oplus \Delta(\Pi_A, X^1), \dots, X^p \oplus \Delta(\Pi_A, X^p)\right) \\ &= \left(\mathrm{comp}_A(X^1), \dots, \mathrm{comp}_A(X^p)\right) \end{aligned}$$

By definition, $\overrightarrow{X}$ is in a computation phase means that $\overrightarrow{\mathrm{comp}_A}(\overrightarrow{X}) \neq \overrightarrow{X}$, so we proved that $\overrightarrow{X}$ is in a computation phase if and only if $\tau_{\Pi_A}(\overrightarrow{X}) \neq \overrightarrow{X}$.

So, we have:

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

Moreover, for every state $\overrightarrow{X}$ such that $\tau_{\Pi_A}(\overrightarrow{X}) = \overrightarrow{X}$, we proved that $\overrightarrow{X}$ is in communication phase. So, according to the Lemma p.27, comm_A preserves the universes, the number of processors, and commutes with multi-isomorphisms. The other properties are immediatly true according to the three postulates. Therefore A is a $\mathrm{ASM}_{\mathrm{BSP}}$ machine. □

Theorem 3. $\mathrm{ALGO}_{\mathrm{BSP}} \supseteq \mathrm{ASM}_{\mathrm{BSP}}$

Proof. Let M be the $\mathrm{ASM}_{\mathrm{BSP}}$ machine $(S(M), I(M), \tau_M)$. By definition, there exists an ASM program Π and an application comm_M such that:

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

In order to prove that M is a BSP algorithm, we have to prove that it verifies the four postulates:

1. M verifies the first postulate because $I(M) \subseteq S(M)$ and τ_M is a transition function.
2. According to the operational semantics of the ASM, τ_Π preserves the universes and the number of processors. We prove by using the Lemma p.29 that τ_Π commutes with any multi-isomorphism $\overrightarrow{\zeta}$:

$$\begin{aligned} \overrightarrow{\zeta}\left(\tau_\Pi\left(X^1, \dots, X^p\right)\right) &= \overrightarrow{\zeta}\left(X^1 \oplus \Delta(\Pi, X^1), \dots, X^p \oplus \Delta(\Pi, X^p)\right) \\ &= \left(\zeta_1\left(X^1 \oplus \Delta(\Pi, X^1)\right), \dots, \zeta_p\left(X^p \oplus \Delta(\Pi, X^p)\right)\right) \\ &= \left(\zeta_1(X^1) \oplus \Delta(\Pi, \zeta_1(X^1)), \dots, \zeta_p(X^p) \oplus \Delta(\Pi, \zeta_p(X^p))\right) \\ &= \tau_\Pi\left(\zeta_1(X^1), \dots, \zeta_p(X^p)\right) \\ &= \tau_\Pi\left(\overrightarrow{\zeta}\left(X^1, \dots, X^p\right)\right) \end{aligned}$$

   Therefore, by using the properties of comm_M in the Definition p.8 of $\mathrm{ASM}_{\mathrm{BSP}}$, M verifies the second postulate.

3. $\Delta^i(\tau_\Pi, \overrightarrow{X}) = \big(X^i \oplus \Delta(\Pi, X^i)\big) \ominus X^i$ so, if $\Delta(\Pi, X^i)$ is consistent then $\Delta^i(\tau_\Pi, \overrightarrow{X})$ is the set of updates $\Delta(\Pi, X^i)$ less the trivial updates, and otherwise $\Delta^i(\tau_\Pi, \overrightarrow{X}) = \varnothing$.
   The sets $\mathrm{Read}\,(\Pi)$ and $\mathrm{Write}\,(\Pi)$ are from Definition p.30. By definition of the operational semantics of the ASM programs, if X and Y coincide over $\{true\} \cup \mathrm{Read}\,(\Pi)$ then $\Delta(\Pi, X) = \Delta(\Pi, Y)$. Moreover, if X and Y coincide over $\mathrm{Write}\,(\Pi)$ too, then an update of $\Delta(\Pi, X)$ is trivial on X if and only if the corresponding update of $\Delta(\Pi, Y)$ is trivial on Y.
   Therefore, $T(\Pi) = \{true\} \cup \mathrm{Read}\,(\Pi) \cup \mathrm{Write}\,(\Pi)$ is an exploration witness for τ_Π. So, by using the properties of comm_M in the Definition p.8 of $\mathrm{ASM_{BSP}}$, $T(M) = T(\Pi) \cup T(\mathrm{comm}_M)$ is an exploration witness for M.
   Therefore, M verifies the third postulate.
4. For every local memory X, let $\mathrm{comp}_M(X) = X \oplus \Delta(\Pi, X)$.
   So $\tau_\Pi(\overrightarrow{X}) = \overrightarrow{\mathrm{comp}_M}(\overrightarrow{X})$, and we have:

$$\tau_M(\overrightarrow{X}) = \begin{cases} \overrightarrow{\mathrm{comp}_M}(\overrightarrow{X}) \text{ if } \overrightarrow{\mathrm{comp}_M}(\overrightarrow{X}) \neq \overrightarrow{X} \\ \mathrm{comm}_M(\overrightarrow{X}) \text{ otherwise} \end{cases}$$

   Therefore, M verifies the fourth postulate.

Therefore M is a BSP algorithm. □

Corollary 1. *Every* $\mathrm{ASM_{BSP}}$ *program has a normal form.*

Proof. According to the previous theorem, an $\mathrm{ASM_{BSP}}$ is a BSP algorithm and, according to the proof of the Proposition p.9, every BSP algorithm is captured by an $\mathrm{ASM_{BSP}}$ with a program in normal form. Thus, our initial $\mathrm{ASM_{BSP}}$ is equivalent to an $\mathrm{ASM_{BSP}}$ with a program in normal form. □

D A function of communication

D.1 Generalities

There is no shared memory, and point-to-point communication is considered. Since in the BSP model, a parallel machine consists of a set of processors, each with its own private memory, and an interconnection network that can route packets between processors, we may reason about BSP algorithms as a set of parameterized processes that communicate via message-passing.

The presented function of communication is naive since only a point-to-point of two processors exchanging a single data is considered. If the network can send blocks of data or can involve more than 2 processors, a more efficient (but much more complicated) function can be give. But our function is sufficient because it involves the right BSP cost and the traditional BSP programming primitives.

To simplify the work, we assume that every term can be shared with other processors and thus we do not take into account the BSPLIB's `push`/`pop` primitives. Adding them complex the exploration witness without fundamentally changing the solution.

In this section, we use the standard BSPLIB's primitives `bsp_get`, `bsp_put`, `bsp_send`and `bsp_move`. We rename them read, write, send and rcv. Reading and writing are now not on buffers of bytes (memory aera as in the C programming language) but on variables. Similarly, the send primitives sends terms and not buffers. For example, in an ASM_{BSP} we get the command $\text{write}(x, j, y)$ (sending the value of x to processor j in the variable y) of Π will be interpreted in the processor X^i by $(i, x \mathbin{@} j, y)$ and added to the buffer $\overline{\text{IdMsg}}^{X^i}$. A command $x := \text{read}(j, y)$ of Π will be interpreted in the processor X^i by $(j, y \mathbin{@} i, x)$ and added to the buffer $\overline{\text{IdMsg}}^{X^i}$.

The conditions about the communication function in the Definition p.8 of ASM_{BSP} are a version of the second and third postulate:

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

The first condition is not particularly restricting, especially if the communication function only manages the communications between processors with a syntactical process, as we will see in the following. So, the main restriction is to assign an exploration witness to the communication function.

Notice that we may want to ensure that the local memories are only updated at the end of the communication phase. Indeed, in a state $\overrightarrow{X}$ during the communication phase, if a local memory is updated we may have $\tau_\Pi(\overrightarrow{X}) \neq \overrightarrow{X}$, so the computation phase begins even if the communication phase is not completed. If necessary, we could impose a boolean b_{comm} which is true during the communication phase, and such that the computation phase can begin only if b_{comm}

becomes false. In that case, b_{comm} should be added to the exploration witness of the communication function.

In this section we we give how to built the function of communication in the ASM context. It is a bit technical. The main ideas are:

- The messages are sent letter by letter; The representation gives the formal size of the communicated values;
- The communications use two buffers of sending IdMsg and Msg; One for sending the order of communication for DRMA routines and another one for the reception of messages of other processors
- If we assume that every processor can only send at most one message and receive at most one message during an "exchange", then the h-relation (communication phase) requires exactly h exchanges;

D.2 Representation, Size and Characterization of Terms

The **representation** t_a of an element a in the local memory X is the unique term formed only by constructors such that $\overline{t_a}^X = a$. For example, the representation of 314 as a decimal number is $\underline{314}_{10}$, and its representation as a binary number is $\underline{100111010}_2$.

If t is a term, let $\overrightarrow{t}$ be the sequence of the letters used to write it in the prefix notation. For example, if f is a binary symbol, g is a unary symbol, and a and b are constants, then the sequence of letters of the term $t = f(f(b, g(a)), g(f(a, b)))$ is $\overrightarrow{t} = (f, f, b, g, a, g, f, a, b)$. Because a letter is not itself a term, for convenience we will add to the language $\mathcal{L}(M)$ a constant symbol $\underline{f}$ for every symbol $f \in \mathcal{L}(M)$, interpreted by $\overline{\underline{f}}^{X^i} = f$. So, the sequentialization of this term will be $\overrightarrow{\underline{t}} = (\underline{f}, \underline{f}, \underline{b}, \underline{g}, \underline{a}, \underline{g}, \underline{f}, \underline{a}, \underline{b})$.

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

The **weight** $\mathrm{w}(f)$ of a symbol f with arity $\alpha(f)$ is defined by $\mathrm{w}(f) \stackrel{\text{def}}{=} 1 - \alpha(f)$. Because for every symbol f we have $\alpha(f) \geq 0$, notice that $\mathrm{w}(f) \leq 1$. The weight of a word $f_1 \dots f_\ell$ of length ℓ is defined by:

$$\mathrm{w}(f_1 \dots f_\ell) \stackrel{\text{def}}{=} \sum_{i=1}^{\ell} \mathrm{w}(f_i)$$

If $1 \leq i \leq \ell$, the word $f_i \dots f_\ell$ is called a suffix of the word $f_1 \dots f_\ell$. The weight of a word is the sum of the weight of its letters:

Lemma 15 (Characterization of a Term). *The word $f_1 \dots f_\ell$ is a term if and only if $w(f_1 \dots f_\ell) = 1$ and for every suffix $f_i \dots f_\ell$ we have $w(f_i \dots f_\ell) \geq 1$.*

Proof. Firstly, we prove the implication by induction on the term t:

- If t is a constant c, then $\mathrm{w}(c) = 1 - 0 = 1$.
- If $t = ft_1 \dots t_\alpha$ where $\alpha(f) = \alpha$ and $t_1, \dots, t_k$ are terms, then:

$$\begin{aligned}\mathrm{w}(ft_1 \dots t_\alpha) &= \mathrm{w}(f) + \textstyle\sum_{i=1}^{\alpha} \mathrm{w}(t_i) \\ &= (1 - \alpha) + \alpha \times 1 \\ &= 1\end{aligned}$$

  A (proper) suffix of $ft_1 \dots t_\alpha$ has the form $s_i t_{i+1} \dots t_\alpha$ where $1 \leq i \leq \alpha$, and s_i is a suffix of the term t_i. So, by induction hypothesis $\mathrm{w}(s_i) \geq 1$, and the terms (if there is any) $t_{i+1}, \dots, t_\alpha$ have a weight of 1. Therefore $\mathrm{w}(s_i t_{i+1} \dots t_\alpha) \geq 1 + (\alpha - i) \geq 1$.

Secondly, we prove the converse by induction on the length of the word $f_1 \dots f_\ell$:

- If $\ell = 1$ then by hypothesis $\mathrm{w}(f_1) = 1$, so $\alpha(f_1) = 0$. Therefore, the word f_1 is a constant, which is a term. It has no other suffix.
- Let $\ell \geq 2$. We assume by induction that for every $1 \leq \ell' < \ell$ we have the converse, which means that for every word $f_1 \dots f_{\ell'}$, if $\mathrm{w}(f_1 \dots f_{\ell'}) = 1$ and for every $1 \leq i \leq \ell'$ we have $\mathrm{w}(f_i \dots f_{\ell'}) \geq 1$, then $f_1 \dots f_{\ell'}$ is a term.
  Let $f_1 \dots f_\ell$ be a word such that $\mathrm{w}(f_1 \dots f_\ell) = 1$ and for every $1 \leq i \leq \ell$ we have $\mathrm{w}(f_i \dots f_\ell) \geq 1$. We prove that $f_1 \dots f_\ell$ is a term.
  f_ℓ is a suffix, so $\mathrm{w}(f_\ell) \geq 1$. But, by definition, we have $\mathrm{w}(f_\ell) \leq 1$. So we have $\mathrm{w}\big(f_{(\ell-1)+1}\big) = \mathrm{w}(f_\ell) = 1$. Therefore, there exists a $1 \leq i < \ell$ such that $\mathrm{w}(f_{i+1} \dots f_\ell) = 1$. Let i_1 be the smallest. By hypothesis we have for every $1 \leq i < i_1$ that $\mathrm{w}(f_{i+1} \dots f_{i_1} f_{i_1+1} \dots f_\ell) \geq 1$.
  But $\mathrm{w}(f_{i+1} \dots f_{i_1} f_{i_1+1} \dots f_\ell) = \mathrm{w}(f_{i+1} \dots f_{i_1}) + \mathrm{w}(f_{i_1+1} \dots f_\ell)$, and by definition of i_1 we have $\mathrm{w}(f_{i_1+1} \dots f_\ell) = 1$.
  So $\mathrm{w}(f_{i+1} \dots f_{i_1}) \geq 0$. The case $\mathrm{w}(f_{i+1} \dots f_{i_1}) = 0$ can be excluded, because it implies that $\mathrm{w}(f_{i+1} \dots f_{i_1} f_{i_1+1} \dots f_\ell) = 0 + 1 = 1$, which contradicts the minimality of i_1.
  Therefore $\mathrm{w}(f_{i+1} \dots f_{i_1}) \geq 1$. In particular $\mathrm{w}(f_{i_1}) \geq 1$, so $\mathrm{w}(f_{i_1}) = 1$.
  So, we obtain $i_2 < i_1$ in the same way that we obtained $i_1 < i_0 = \ell$, and so on. We obtain a strictly decreasing sequence of elements $i_j \geq 1$, until $i_k = 1$.
  Therefore we have:

$$f_1 \dots f_\ell = f_{i_k}\big(f_{i_k+1} \dots f_{i_{k-1}}\big) \dots \big(f_{i_1+1} \dots f_{i_0}\big), \text{ such that:}$$

$$\begin{aligned}\mathrm{w}\big(f_{i_k+1} \dots f_{i_{k-1}}\big) = 1, &\text{ and for every suffix: } \mathrm{w}\big(f_i \dots f_{i_{k-1}}\big) \geq 1 \\ &\vdots \\ \mathrm{w}(f_{i_1+1} \dots f_{i_0}) = 1, &\text{ and for every suffix: } \mathrm{w}(f_i \dots f_{i_0}) \quad \geq 1\end{aligned}$$

  By induction hypothesis, $f_{i_k+1} \dots f_{i_{k-1}}$ is a term t_k, …, and $f_{i_1+1} \dots f_{i_0}$ is a term t_1.

By hypothesis $\mathrm{w}(f_1 \dots f_\ell) = 1$, so:

$$\begin{aligned}\alpha(f_1) &= 1 - \mathrm{w}(f_1)\\ &= \mathrm{w}(f_1 \dots f_\ell) - \mathrm{w}(f_1)\\ &= \sum_{j=1}^{k} \mathrm{w}\left(f_{i_j+1} \dots f_{i_{j-1}}\right)\\ &= \sum_{j=1}^{k} 1\\ &= k\end{aligned}$$

$t_1, \dots, t_k$ are terms, and $\alpha(f_1) = k$, so $f_1 \dots f_\ell = f_1 t_1 \dots t_k$ is also a term.

□

Notice that if $f_1 \dots f_\ell$ is the empty word, then $\ell = 0$ and $\mathrm{w}(f_1 \dots f_\ell) = \sum_{i=1}^{\ell} \mathrm{w}(f_i) = 0$. So, because the empty word is not a term, the equivalence holds in that case too.

D.3 Buffers of communications

So, a message from a processor i to a processor j will be sent letter by letter. The processors have two buffers IdMsg and Msg, which does not appear in Π. IdMsg contains the identifiers $(i, x \mathbin{@} j, y)$ of the messages, where i is the identifier of the sending processor, x the read location, j the identifier of the receiving processor, and y the written location.

For example, a command $\mathrm{write}(x, j, y)$ of Π will be interpreted in the processor X^i by $(i, x \mathbin{@} j, y)$ and added to the buffer $\overline{\mathrm{IdMsg}}^{X^i}$. A command $x := \mathrm{read}(j, y)$ of Π will be interpreted in the processor X^i by $(j, y \mathbin{@} i, x)$ and added to the buffer $\overline{\mathrm{IdMsg}}^{X^i}$.

At the end of the computation phase[18] theses identifiers are mixed together, and for every processor X^i its buffer $\overline{\mathrm{Msg}}^{X^i}$ is filled with the messages $(\overrightarrow{\underline{t}_{\overline{x}^{X^i}}} \mathbin{@} j, y)$ for every identifier $(i, x \mathbin{@} j, y)$, and $\overline{\mathrm{IdMsg}}^{X^i}$ is emptied.

Then the processor X^i sends the message letter by letter to the processor X^j. If the message were reconstructed on X^j only at the end of the communication phase, then the term t itself would be necessary, and the exploration witness could not be bounded. Instead, the value is reconstructed step by step each time a letter is received.

For example, to reconstruct the term $t = f(f(b, g(a)), g(f(a, b)))$, the processor X^i sends $\overrightarrow{\underline{t}} = (\underline{f}, \underline{f}, \underline{b}, \underline{g}, \underline{a}, \underline{g}, \underline{f}, \underline{a}, \underline{b})$ one letter at a time from the end to the

[18] We can always build a formula which becomes true at the end of the computation phase. For example, for every $t \in \mathrm{Read}(\Pi)$ we add a fresh variable v initialized at $\bot$. The number of fresh variables is bounded because $\mathrm{Read}(\Pi) \subseteq T(M)$ which is bounded according to the third postulate. We build a new program $\Pi' = \texttt{par}\ v_1 := t_1 \parallel \dots \parallel v_k := t_k \parallel \Pi\ \texttt{endpar}$ which has the same execution than Π if the fresh variables are ignored. The desired formula is $v_1 = t_1 \wedge \dots \wedge v_k = t_k$. Indeed, it becomes true only if the old values $\overrightarrow{v}$ of the terms read by Π are the same than the new values $\overrightarrow{t}$, which means that nothing has changed.

beginning, and the processor X^j reconstructs t by using the following updates:

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

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

Firstly, notice that the number of variables used that way cannot be bounded. For example, the term $f(\dots f(a_{n+1}, a_n) \dots, a_1)$ requires $n + 1$ variables. So, instead of a variable x_{nArg} we will use a unary symbol RcvVal(nArg).

Secondly, we need to specify how the index nArg evolves. We set $\overline{\mathrm{nArg}}^{X^i} = 0$ for every local memory at the beginning of a communication phase. Then, each time a symbol $\underline{f}$ is received, nArg is updated by $\mathrm{nArg} := \mathrm{nArg} + \mathrm{w}(f)$, where $\mathrm{w}(f)$ is the **weight** of the symbol f (see previously).

The weight of a word is the sum of the weight of its letters. We prove in Lemma p.37 that $\mathrm{nArg} \geq 1$ during the communication phase, and that for every term t we have $\mathrm{w}(t) = 1$. Therefore, at the end of the communication phase, nArg is the number of terms received.

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

D.4 Read and Write

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

$$\begin{array}{l}
\texttt{par} \\
\qquad \mathrm{nArg}(\underline{y}) := \mathrm{nArg}(\underline{y}) + 1 \\
\quad \| \ \mathrm{RcvVal}\left(\underline{y}, \mathrm{nArg}(\underline{y}) + 1\right) := c \\
\texttt{endpar}
\end{array}$$

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

$$\begin{array}{l}\texttt{par}\\ \qquad \mathrm{nArg}(\underline{y}) := \mathrm{nArg}(\underline{y}) + 1 - \alpha\left(\underline{f}\right)\\ \quad \| \ \mathrm{RcvVal}\left(\underline{y}, \mathrm{nArg}(\underline{y}) + 1 - \alpha\left(\underline{f}\right)\right) := f\left(\begin{array}{l}\mathrm{RcvVal}\left(\underline{y}, \mathrm{nArg}(\underline{y})\right),\\ \ldots,\\ \mathrm{RcvVal}\left(\underline{y}, \mathrm{nArg}(\underline{y}) + 1 - \alpha\left(\underline{f}\right)\right)\end{array}\right)\\ \texttt{endpar}\end{array}$$

Therefore, at the end of the communication phase, $\mathrm{nArg}(\underline{y})$ is the number of terms received at the location $\underline{y}$. if $\mathrm{nArg}(\underline{y}) = 0$ the variable y does not require to be updated. If $\mathrm{nArg}(\underline{y}) \geq 2$, more than one term have been received, so there might be a conflict. In that case, we assume that the variable y should not be updated[19]. So, at the end of the communication phase (the synchronization), the following updates are made for the variables $y_1, \ldots, y_k \in \mathrm{WriteComm}(\Pi)$:

$$\begin{array}{l}\texttt{par}\\ \quad \texttt{if} \ \ \mathrm{nArg}(\underline{y_1}) = 1 \ \texttt{then} \ y_1 := \mathrm{RcvVal}\left(\underline{y_1}, 1\right)\\ \| \ \vdots\\ \| \ \texttt{if} \ \ \mathrm{nArg}(\underline{y_k}) = 1 \ \texttt{then} \ y_k := \mathrm{RcvVal}\left(\underline{y_k}, 1\right)\\ \texttt{endpar}\end{array}$$

Notice that because in our example the symbols read by Π are only updated at the end of the communication phase, the computation phase can begin at the next step, so we do not need a boolean b_{comm} to prevent the computations to happen during the communication phase.

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

$$\bigcup_{y \in \mathrm{WriteComm}(\Pi)} \ \bigcup_{f \in \mathcal{L}(M)} \ \bigcup_{\alpha(f)=0} \left\{f, \mathrm{RcvVal}\left(\underline{y}, \mathrm{nArg}(\underline{y}) + 1\right)\right\}$$

$$\bigcup_{\alpha(f) \geq 1} \left\{f\left(\mathrm{RcvVal}\left(\underline{y}, \mathrm{nArg}(\underline{y})\right), \ldots, \mathrm{RcvVal}\left(\underline{y}, \mathrm{nArg}(\underline{y}) + 1 - \alpha\left(\underline{f}\right)\right)\right)\right\}$$

$$\cup \left\{\mathit{true}, \mathrm{nArg}(\underline{y}) = 1, y, \mathrm{RcvVal}\left(\underline{y}, 1\right)\right\}$$

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

[19] This decision may be problematic if the same value has been sent several times, but we will not consider this problem in this paper.

D.5 Send and Receive

But WriteComm(Π) may not be restricted to variables. In fact, terms are required for communication primitives[20] like send or rcv.

Indeed, in that case we cannot use an identifier $(i, x \ @ \ j, y)$ during the communication phase because the written location is not a variable y but will be determined during the next computation phase by a command $t_2 :=$ rcv. So, we will use instead a symbol $\_$ to indicate that the message should be stored in a buffer of the receiving processor.

Therefore, during the computation phase a command send(x, j) will be interpreted in the processor X^i by an identifier $(i, x \ @ \ j, \_)$ and added to the buffer $\overline{\text{IdMsg}}^{X^i}$, and at the end of the computation phase, these identifiers are mixed together (with the identifiers for the read and write primitives).

As for the read and write primitives, the message should be sent letter by letter. The received value cannot be constructed by using $\text{RcvVal}\left(\underline{y}, \text{nArg}(\underline{y})\right)$ because the final location y is not known. Instead, we assume that the number h^{in} of messages received by a processor must be bounded for every execution of the algorithm[21].

So, the communication function can assign a location $\text{RcvLoc}(n)$, where $1 \leq n \leq h^{\text{in}}$ and RcvLoc is an injection, to each received message in a processor j. Therefore, for every sending processor X^i and for every identifier $(i, x \ @ \ j, \_)$, its buffer $\overline{\text{Msg}}^{X^i}$ is filled with the messages $(\underline{\overrightarrow{t_{\_x}}^{X^i}} \ @ \ j, \text{RcvLoc}(n))$ and $\overline{\text{IdMsg}}^{X^i}$ is emptied.

Then, the messages are sent letter by letter, and each value is reconstructed in $\text{RcvVal}\left(\text{RcvLoc}(n), 1\right)$ in the same way as before. If the primitives read and write are used at the same time as the primitives send and rcv, we can assume that the language contains a variable $1 \leq \text{nLoc} \leq h^{\text{in}}$ such that if $1 \leq n < \text{nLoc}$ then $\text{RcvLoc}(n) = \underline{y}$ is the location of a symbol y, and if $\text{nLoc} \leq n \leq h^{\text{in}}$ then $\text{RcvLoc}(n)$ is a location for a rcv command.

At the end of the communication phase, for every $y \in \text{WriteComm}(\Pi)$ such that $\text{RcvLoc}(n) = \underline{y}$, the program `if` $\text{nArg}(\text{RcvLoc}(n)) = 1$ `then` $y := \text{RcvVal}\left(\text{RcvLoc}(n), 1\right)$ is applied[22] . Then, during the next computation phase, a

[20] Notice that in our presentation, it does not matter if the location x read by a command is a variable or a term t_1, because this term may be added to the exploration witness $T(M)$, but in this paper we assume the most restrictive case. And indeed, if we want to prevent computations during the communication phase, we should restrict the location to be a variable.

[21] In other words, in our framework we can notice an algorithmic difference between the primitives $\{\text{read}, \text{write}\}$ and the primitives $\{\text{send}, \text{rcv}\}$: the first have automatically a bounded number of received messages (or a clash), but the second may not. The restriction of h^{in} is physically sound because it states that there must exists a bound to the quantity of information stored in a given volume (here, the processor), which can be estimated by the Bekenstein bound, https://en.wikipedia.org/wiki/Bekenstein_bound.

[22] If they are applied sequentially, they required $\text{nLoc} - 1 \leq h$ steps, and we have to add a counter to the language and to the exploration witness $T(\text{comm}_M)$.

command $t_2 :=$ rcv is interpreted (for example[23]) by `par` $t_2 :=$ RcvVal (RcvLoc(nLoc), 1) $\parallel$ nLoc := nLoc +1 `endpar` if nLoc $\leq h$.

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

$$\bigcup_{1\leq n\leq h^{\text{in}}} \bigcup_{f\in\mathcal{L}(M)} \bigcup_{\alpha(f)=0} \left\{ f,\ \text{RcvVal}\left(\text{RcvLoc}(n), \text{nArg}(\text{RcvLoc}(n)) + 1\right) \right\}$$
$$\bigcup_{\alpha(f)\geq 1} \left\{ f\Big(\text{RcvVal}\left(\text{RcvLoc}(n), \text{nArg}(\text{RcvLoc}(n))\right) \right.$$
$$\left. \ldots, \text{RcvVal}\left(\text{RcvLoc}(n), \text{nArg}(\text{RcvLoc}(n)) + 1 - \alpha\left(\underline{f}\right)\right)\Big) \right\}$$
$$\cup\, \text{WriteComm}(\Pi) \cup \left\{ \mathit{true},\quad \text{nArg}(\text{RcvLoc}(n)) = 1, \text{RcvVal}\left(\text{RcvLoc}(n), 1\right) \right\}$$

Therefore $T(\text{comm}_M)$ is finite, so we gave an example of communication function verifying the conditions given at the Definition 6 p.8 of ASM_{BSP}:

Proposition 4 (Communication "à la BSPlib"). *A function of communication with primitives for distant readings/writings and point-to-point sending of data can be design using* ASM.

Notice that we did not specify how the index RcvLoc(n) is constructed, nor whether or not RcvVal is emptied at the beginning of the next communication phase. These are details which are left to the implementation and depend on the chosen algorithm.

In our example the latency **L** of the barrier of synchronization includes, at the end of the communication phase, the assurance that every message has been sent and received. The bandwidth **g** includes, at the beginning of the communication phase, the exchange of the message identifiers in order to prepare the communication relation between the processors; The preparation of the messages themselves by a kind of serialization and, at the end of the communication phase, the update of the written variables.

D.6 Number of Exchanges

During the communication phase of a superstep, every processor X^i must send h_i^{out} messages[24] and receive h_i^{in} messages. Such a communication pattern is called

[23] There exists many variants of the rcv command. Notice that in our variant, we should add RcvVal (RcvLoc(nLoc), 1), nLoc +1 and nLoc $\leq h$ to the exploration witness $T(M)$.

[24] Or letter of a message, as in our example of communication function.

a h-relation, where:

$$\begin{aligned} h &= \max(h^{\text{in}}, h^{\text{out}}) \\ \text{with } h^{\text{in}} &= \max_{1 \leq i \leq p} \left(h_i^{\text{in}}\right) \\ \text{and } h^{\text{out}} &= \max_{1 \leq i \leq p} \left(h_i^{\text{out}}\right) \end{aligned}$$

We assume in that section that every processor can only send at most one message and receive at most one message during an "exchange", and we prove that the communication phase requires exactly h exchanges.

Notice that if $h = h^{\text{out}}$ then there exists a processor X^i that requires at least h exchanges to send its h_i^{out} messages, and if $h = h^{\text{in}}$ then there exists a processor X^i that requires at least h exchanges to receive its h_i^{in} messages. So, we only have to prove that there exists a sequence of exchanges requiring at most h exchanges:

Lemma 16 (Order of Exchanges).
For every h-relation there exists a sequence of exchanges requiring at most h exchanges.

Proof. The proof is made by induction on the number of messages:

1. If there is no communication, then $h = 0$ and the relation does not require any exchange.
2. We assume that there is a h-relation which can be realized with at most h exchanges, and we add a new message sent by the processor X^i to the processor X^j.
   The proof is made by case:
   - If $h_i^{\text{out}} = h$ or $h_j^{\text{in}} = h$, then the new pattern of communication is a $h+1$ relation.
     So we can do the exchanges of the previous relation then the new exchange in $h + 1$ exchanges.
   - Otherwise, the new pattern of communication is a h relation.
     Moreover, in the previous relation X^i sends $h_i^{\text{out}} \leq h - 1$ messages and X^j receives $h_j^{\text{in}} \leq h - 1$ messages. So, during the sequence of the h exchanges of the previous relation, there exists an exchange e_i when X^i does not send, and there exists an exchange e_j when X^j does not receive.
     We construct a sequence $X^{i_0}, X^{i_1}, X^{i_2}, \dots$ of sending processors, and a sequence $X^{j_0}, X^{j_1}, X^{j_2}, \dots$ of receiving processors, while we change the communications in the following way:
     (a) We begin with $X^{i_0} = X^i$ and $X^{j_0} = X^j$.
     (b) We assume that X^{i_n} does not send at e_i and X^{j_n} does not receive at e_j, and that it remains a communication to be done from X^{i_n} to X^{j_n}.
     These properties are true for $n = 0$.
     We prove that either we can construct the following processors $X^{i_{n+1}}$ and $X^{j_{n+1}}$ with the same properties, or the sequences of processors end.

(c) If at the exchange e_j, X^{i_n} sends no message, then we can add the communication from X^{i_n} to X^{j_n} at e_j without increasing the number of exchanges, because X^{j_n} does not receive at e_j. In that case, there is no more communication left, so we can end the sequences of sending and receiving processors.

(d) Otherwise, if at the exchange e_j, X^{i_n} sends a message to a processor X^k, then we set $X^{j_{n+1}} = X^k$, we remove the communication from X^{i_n} to $X^{j_{n+1}}$, and we add the communication from X^{i_n} to X^{j_n}. This communication is done without increasing the number of exchanges. But now it remains the communication from X^{i_n} to $X^{j_{n+1}}$ to be done.

Notice that before $X^{j_{n+1}}$ received a message from X^{i_n} at the exchange e_j, but we removed this communication without adding a new communication to $X^{j_{n+1}}$. So, because a processor can only receive one message per exchange, we have that $X^{j_{n+1}}$ now does not receive at e_j.

(e) If at the exchange e_i, $X^{j_{n+1}}$ receives no message, then we can add the communication from X^{i_n} to $X^{j_{n+1}}$ at e_i without increasing the number of exchanges, because X^{i_n} does not send at e_i. In that case, there is no more communication left, so we can end the sequences of sending and receiving processors.

(f) Otherwise, if at the exchange e_i, $X^{j_{n+1}}$ receives a message from a processor X^k, then we set $X^{i_{n+1}} = X^k$, we remove the communication from $X^{i_{n+1}}$ to $X^{j_{n+1}}$, and we add the communication from X^{i_n} to $X^{j_{n+1}}$. This communication is done without increasing the number of exchanges.

But now it remains the communication from $X^{i_{n+1}}$ to $X^{j_{n+1}}$ to be done.

Notice that before $X^{i_{n+1}}$ sent a message to $X^{j_{n+1}}$ at the exchange e_i, but we removed this communication without adding a new communication from $X^{i_{n+1}}$. So, because a processor can only send one message per exchange, we have that $X^{i_{n+1}}$ now does not send at e_i.

(g) Now $X^{i_{n+1}}$ does not send at e_i, $X^{j_{n+1}}$ does not receive at e_j, and it remains the communication from $X^{i_{n+1}}$ to $X^{j_{n+1}}$ to be done. So, by induction we can continue until the sequences end.

Notice that, after N repetitions of this process from the initial processors $X^{i_0} = X^i$ and $X^{j_0} = X^j$, we have that:

- at the exchange e_j, for every $0 \leq n < N$, we have a communication from X^{i_n} to X^{j_n}
- at the exchange e_j, X^{j_N} receives no message
- at the exchange e_i, for every $0 \leq n < N$, we have a communication from X^{i_n} to $X^{j_{n+1}}$
- at the exchange e_i, X^{i_N} sends no message

So X^{i_N} is different from every previous X^{i_n}, and X^{j_N} is different from every previous X^{j_n}.

Therefore, the processors of the sequence $X^{i_0}, X^{i_1}, X^{i_2}, \dots$ are distinct, and the processors of the sequence $X^{j_0}, X^{j_1}, X^{j_2}, \dots$ are distinct too. But the number of processors is finite, so these sequences must end. So, after a finite sequence of sending and receiving processors, the communications (including the new one) can be done without increasing the number of exchanges. Therefore, the h-relation requires at most h exchanges.

□

Notice that we proved that such a sequence of exchanges exists, but this sequence is probably costly to compute. By the way, our purpose is proving the achievability of such a function of communication, not finding an efficient one (using hardware optimisations, routing in the network, *etc.* that is not our work).

Proposition 5 (A function of communication). *A function of communication performing h-relation requiring at most h exchanges with primitives for distant readings/writings and point-to-point sending of data can be design using* ASM*.*

Proof. By construction of the previous function of communication and application of the previous lemma. □