10th International Symposium on
High-Level Parallel Programming and Applications (HLPP 2017)
Valladolid, Spain
July 10-11th, 2017

Imperative Characterization of BSP Algorithms

Y. Marquer · F. Gava

Abstract What is a bulk-synchronous parallel (BSP) algorithm? Behind this naive question, we can find a complex formalism. First, defining what is formally an algorithm. Second extending this definition to parallel (BSP) computing. And finally, characterizing BSP algorithms by using an imperative language and prove, using an operational semantics and a simulation, the equivalence between the language and the algorithms. For this work, we follow the Gurevich's axiomatic model of sequential algorithms. This model comes with what is call abstract state machines (ASMs) that capture sequential algorithms. We have extended both to capture and characterize BSP algorithms.

Keywords BSP · ASM · Semantics · Algorithm · Completeness · Simulation.

1 Introduction

1.1 Context of the work

Nowadays, parallel programming is the norm in many areas but it remains *hard* to have well defined paradigms and a common vocabulary as in the traditional sequential world. One of the problems comes from the difficulty to get a taxonomy of computer architectures and frameworks: there is a zoo of (informal) definitions of the systems, languages (paradigms) and programming models. Indeed, in the HPC community, several terms could be used to designate the same thing and can lead to misunderstandings. Furthermore, because there are many paradigms, it could be difficult, even for experts, to distinguish them. For example, parallel patterns [8,17] versus algorithmic skeletons [16]; shared memory (PRAM) versus thread concurrency and direct memory access (DRMA, **d**irec **rem**ote **a**ccess); asynchronous send/received routines (MPI) versus communicating processes (π-calculus).

In fact, in the sequential world, it is easier to classify programming languages with their paradigm (functional, object oriented, *etc.*) or 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

LACL, University of Paris-East, Créteil, France
E-mail: marquer.yoann@hotmail.fr · E-mail: frederic.gava@univ-paris-est.fr

(and formal) consensus of what are sequential programming languages. For them, formal semantics have been very often studied and there are now many tools for testing, verifying (functional correctness), debugging, cost analyzing, software engineering, *etc.*. In this way, programmers can implement sequential algorithms using sequential languages which characterize well these algorithms.

By the way, this consensus is fair only because everyone agrees to what is informally a sequential algorithm. And from now half a century, there is a growing interest in defining *formally* the notion of algorithms [19]. An *axiomatic* definition [19] of the algorithms ($\texttt{Algo}_{\texttt{seq}}$) is mapped to the notion of Abstract State Machine (ASM) [19] with a *strict lock-step* simulation [33]. According to [19], every sequential algorithm can be computed step-by-step by an ASM. This allows a common vocabulary about sequential algorithms and gives a notion of "algorithmic completeness".

Furthermore, common imperative languages ($\texttt{Imp}_{\texttt{seq}}$) such as JAVA or C, are Turing-complete, which means that they can simulate the input-output relation of a Turing machine and thus compute (up to unbounded memory) every calculable functions. Algorithmic completeness is a stronger notion than Turing-completeness. It focuses not only on the input-output behavior of the computation but more importantly on the step-by-step behavior (a fair simulation) that exhibit the cost to simulate the computations. Moreover, the issue is not limited to partial recursive functions, it applies to any set of functions. A model could compute all the desired functions, but some algorithms (ways to compute these functions) could be missing. It has been proved in [33] that common imperative languages are not only Turing-complete but also algorithmically complete, by using the axiomatic definition of [19]. That proves what informally everyone knows: every sequential algorithm could be programmed with an imperative language [19] and with the appropriate cost [33]. This can be summarized by: $\texttt{Algo}_{\texttt{seq}} \simeq \text{ASM} \simeq \texttt{Imp}_{\texttt{seq}}$

Nevertheless, to our knowledge, there is not such of work for parallel/distributed (HPC) frameworks. First, due to the zoo of (informal) definitions and second, due to a lack of realistic cost models (of common HPC architectures) that give the cost of parallel algorithms. It is a shame because the community is failing to have rigorous definitions of what are parallel algorithms and study their algorithmic completeness. The algorithmic completeness is the basis to specify what can be programmed effectively *or not*. Moreover, wanted to take into account all the features of all parallel/distributed paradigms is a daunting task that is unlikely to materialize. Instead, a bottom up strategy, from the simplest models to the most complex, is more likely to materialize.

1.2 Content of the work

A first step to the solution is the used of a *bridging model*, such as BSP here, because it allows to simplify the task of the algorithm design, their programming and ease the reasoning on *cost* and to ensure a better *portability* from one system to another [2,31]. We conscientiously limit our work to BSP because it has also the advantage to be endowed with a simple model of execution closer to the sequential one and we let more complex models to future works.

There are many different libraries and languages for programming BSP algorithms. The most known are the BSPLIB for C [21] or JAVA [27], the functional BSP programming language BSML [15], NESTSTEP [22], BSP++ [20], *etc.*

As stands in [23] about BSML: *"An operational approach has led to a* BSP *λ-calculus that is confluent and universal for* BSP *algorithms"*. The argument is that these primitives can simulate any BSPLIB program, if the BSPLIB is algorithmic complete for BSP. But none of these languages and libraries has been proved to be BSP algorithmic complete. Informally, it is the case but we want to guarantee it. Another advantage is the ability to prove that a programming language is *not* BSP: it can be too expressive (MPI) or, on the contrary, it does not allow programming a BSP algorithm with the right complexity.

To do so, as has been done in [19,33] but for BSP, we give an axiomatic definition of BSP algorithms. Four postulates will be necessary, basically: three postulates for sequential computations (as in [19]) and one more for the communications. With such postulates, we can extend the ASMs of [19] to take into account the BSP algorithms. And finally, by extending for BSP the work of [33], we construct an imperative programming language and we prove that this language computes exactly BSP algorithms in lock step (one step of the BSP-ASM is simulated by using a constant number of steps).

We finally answer previous criticisms by defining a convincing set of parallel algorithms running in a predictable time (using the step-timer principle), and by constructing a programming language computing those algorithms. This can be summarized by: $\mathtt{Algo}_{\mathtt{bsp}} \simeq \text{BSP}-\text{ASM} \simeq \mathtt{Imp}_{\mathtt{bsp}}$

2 General definitions

In this section, we recall some past definitions. We will then introduce the definition of a fair simulation between computation models and finally a core imperative language (adjunct with its operational semantics).

2.1 Characterizing sequential algorithms

In [19], the author introduced an axiomatic presentation of the sequential algorithms. The main idea is that there is no language that truly represent all sequential algorithm. In fact, every algorithmic book or paper present the algorithm in its own way. Depending of the reader to understand and having its own idea of how the algorithm works, especially if we want to adapt it for its own purpose or implemented it to get an executable code. Indeed, programming languages gives too much details and are largely machine dependant (or at least compiler dependant): for example, the programmer has to manage the allocation of memory using pointers (C) or references (ADA, OCAML), *etc.*

The main idea of [19] is to consider algorithms in axiomatic way in the sense that they are in a purely "mathematical world" (hence an axiomatic definition using four postulates in place of a definition using the semantics of a programming language). This formal definition of what is a sequential algorithm has been studied by the ASM community from several years [7].

By lack of space, we cannot recall this axiomatic characterization of sequential algorithms and ASMs. We put then in appendix A.

2.2 Fair Simulation and algorithmic equivalence

A *model of computation* can be defined in general as a set of programs given with their operational semantics. In our study we only study sequential and BSP algorithms, which have both a step-by-step execution determined by their transition function. So, these operational semantics can be defined by a set of transition rules (defined later).

Sometimes, not only the simulation between two models of computation can be proven, but also their identity [18]. But generally it is only possible to prove a simulation between two models of computation. In our framework, a computation model M_1 can simulate another computation model M_2 if for every program P_2 of M_2 there exists a program P_1 of M_1 producing in a "reasonable way" the "same" executions as those produced by P_2.

$X|_{\mathcal{L}_2}$ denotes the **restriction** of the $\mathcal{L}_1$-structure X to the signature $\mathcal{L}_2$. The signature of $X|_{\mathcal{L}_2}$ is $\mathcal{L}_2$, its universe is the same than X, and every symbol $s \in \mathcal{L}_2$ has the same interpretation in $X|_{\mathcal{L}_2}$ than in X. This notation is extended to a set of updates:

$$\Delta|_{\mathcal{L}} =_{def} \{(f, \mathbf{a}, b) \in \Delta \mid f \in \mathcal{L}\}$$

But fresh function symbols could be "too powerful", for example a dynamical unary symbol *env* alone would be able to store an unbounded amount of information. In order to obtain a fair simulation, we assume that the difference $\mathcal{L}_1 \setminus \mathcal{L}_2$ between both signatures is a set containing only a bounded number of variables (0-ary dynamical symbols).

The initial values of these **fresh variables** could be a problem if they depend on the inputs. For example, the empty program could compute any $f(\mathbf{n})$ if we assume that an output variable contains in the initial state the result of the function f on the inputs $\mathbf{n}$. So, in this paper we use an initialization which depends only on the constructors. Because this initialization is independent (up to isomorphism) from the initial states, we call it a **uniform initialization**.

Also, in the following a (constant) **temporal dilation** d is allowed. We will say that the simulation is step-by-step, and strictly step-by-step if $d = 1$. Unfortunately, contrary to the previous example this constant may depend on the simulated program. But this temporal dilation is not sufficient to ensure the termination of the simulation. For example, a simulated execution $Y_0, \ldots, Y_t, Y_t, \ldots$ could have finished, but the simulating execution $X_0, \ldots, X_{d\times t}, X_{d\times t+1}, \ldots, X_{d\times t+(d-1)}, X_{d\times t}, X_{d\times t+1}, \ldots$ may continue forever. So, an ending condition like $\mathit{time}(A, X) = d \times \mathit{time}(B, X) + e$ is necessary. It corresponds to the usual consideration for asymptotic time complexity.

Definition 1 (Fair Simulation)
Let M_1, M_2 be two models of computation.
M_1 simulates M_2 if for every program P_2 of M_2 there exists a program P_1 of M_1 such that:

1. $\mathcal{L}(P_1) \supseteq \mathcal{L}(P_2)$, and $\mathcal{L}(P_1) \backslash \mathcal{L}(P_2)$ is a finite set of variables (with a uniform initialization)

and there exists $d \in \mathbb{N} \setminus \{0\}$ and $e \in \mathbb{N}$ (depending only on P_2) such that, for every execution $\mathbf{Y}$ of P_2 there exists an execution $\mathbf{X}$ of P_1 satisfying:

2. for every $t \in \mathbb{N}$, $X_{d \times t}|_{\mathcal{L}(P_2)} = Y_t$
3. $time(P_1, X_0) = d \times time(P_2, Y_0) + e$

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

Remark 1 *The second condition $X_{d \times t}|_{\mathcal{L}(P_2)} = Y_t$ implies for $t = 0$ that the initial states are the same, up to temporary variables.*

2.3 Imperative language

After defining the axiomatic definition of algorithms, we have defined ASMs. But this model of computation is obviously not suitable for programming. ASMs will serve us as a bridging model between programming languages and algorithms. For this purpose, we first define a core programming language, then its operational semantics as a set of rewriting rules and finally having a fair simulation between ASMs and this core language.

2.3.1 Formal definition of the sequential core language

$$\begin{aligned} c =_{def}\ & f(t_1, \ldots, t_\alpha) := t_0 \\ & \mid \texttt{if } F\ \{P_1\} \texttt{ else } \{P_2\} \\ & \mid \texttt{while } F\ \{P\} \\ P =_{def}\ & \epsilon \mid c; P \end{aligned}$$

Fig. 1 The core language.

The syntax P_{seq} of our language is a standard imperative core language (see Fig. 1) for C and JAVA. As usual, we choose a subset of realistic imperative languages (such as C and JAVA) to ease the understanding and analysis. A program is a sequence of commands c with traditional constructions: conditionals, `if`, and loops, `while`. f is a dynamic α-ary function symbol, $t_0, t_1, \ldots, t_\alpha$ are closed terms and F is a (boolean) formula. The composition of commands $c; P$ can be generalized by induction to **composition of programs** $P_1 ⨟ P_2$ by $\epsilon ⨟ P_2 =_{def} P_2$ and $(c; P_1) ⨟ P_2 =_{def} c; (P_1 ⨟ P_2)$.

Notation 1 *For the sake of clarity, we will omit the ϵ inside curly brackets in the rest of the paper. For example, as is the case for* `ASM` *programs, we write only* `if` *F $\{P\}$ for the command* `if` *F $\{P\}$* `else` *$\{\epsilon\}$.*

The operational semantics of P_{seq} is formalized by a state transition system. That specifies the execution of the program, one step at a time. A set of rules is repeatedly applied on program states, until a final state is reached. If rules can be applied infinitely, it means the program diverges. If at one point in the execution there is no rule to apply, it is a faulty program. A state of the system is a pair $P \star X$ of a program and a structure. Its transitions $\succ$ are determined only by the head command and the current structure. The rules are defined in Fig 5. For sake of conciseness, we merge these rules with the ones of sequential computations of the BSP language; so the rules for managing the environment of communications is no need (and no sense) here.

The successors are unique, so this transition system is deterministic. We denote by $\succ_t$ a succession of t transition steps. Only the states $\epsilon \star X$ have no successor, so they are the terminating states.

Notation 2 *P **terminates** on X if there exists t and X' such that:*

$$P \star X \succ_t \epsilon \star X'$$

Because the transition system is deterministic, t and X' are unique. So X' is denoted $P(X)$ and t is denoted $time(P, X)$.

The difference with common models of computation is that the data structures are not fixed. As is the case for the ASMs, the equality and the booleans are needed, and the unary integers will also be necessary for the number of processors (and their ids), but the other data structures are seen as oracular. If they can be implemented in a sequential algorithm then they are implemented using the same language, universe and interpretation in this programming language. So, the fair simulation between $\texttt{ASM}_{\mathcal{P}}$ and P_{seq} is proven for control structures, up to data structures. Finally, it has been proved in [33]:

Theorem 1 *(1) `ASM` fairly simulates P_{seq} and (2) P_{seq} fairly simulates `ASM`*

The proof is done by syntactical induction [33] on both `ASM` and P_{seq}.

3 Characterizing BSP algorithms

The BSP model (**B**ulk **S**ynchronous **P**arallelism) [31,2] is a bridging model that is it eases the way of programming various parallel architectures using a certain level of abstraction. That allows to reduce the gap between an abstract execution (programming an algorithm) and concrete parallel systems (using a compiler). The assumptions of the BSP model is thus having portable and scalable performance prediction on HPC systems. Without dealing with low-level details of parallel architectures, the programmer can focus on algorithm design — complexity, correctness, *etc.* A nice introduction for its "philosophy" can be found in [28] and a complete book of numerical algorithms is [2].

In this section, we present the BSP model and try to formally characterizing BSP algorithms. For this, we will extend the previously presented postulates in order to capture asynchronous computations, and add a fourth postulate for the synchronization barrier and communication.

3.1 The BSP bridging model of computation

The BSP bridging model describes a parallel architecture, an execution model and a cost model which allows to predict the performances of a BSP algorithm on a given architecture. We recall each of them in 3 the following.

3.1.1 The BSP architecture

A BSP computer is form by 3 main components: (1) A set of homogeneous pairs of processors-memories; (2) A communication network to exchange messages between pairs; (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, *etc.* can be thus considered as BSP computers. For example share memory machines could be used in a way such as 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.

3.1.2 The execution model

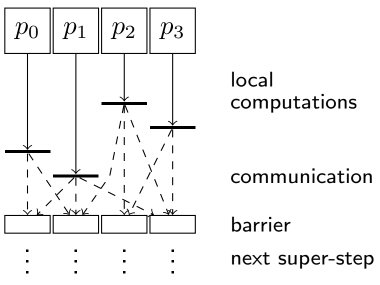

Fig. 2 A BSP super-step.

The execution of the BSP program is a sequence of *super-steps* (Fig. 2), each one divided into three successive disjoint phases: (1) Each processor only uses its local data to perform sequential computations and to request data transfers to other nodes; (2) The network delivers the requested data; (3) A global synchronization 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 but only transfer requests; only at the synchronization *barrier* information is actually exchanged. The BSP model supports communication *en masse*. This is less flexible than asynchronous messages, but easier to debug. Note that a BSP library can send messages during the computation phase of a super-step, but this is hidden to programmers. There exist different BSP programming libraries. The most known are BSPLIB [21,35], PUB [5] for the C language and HAMA [27] for JAVA. A MPI program only using *collective operations* can also be viewed as a BSP program.

3.1.3 The cost model

The *performance* of a BSP computer is characterized by 4 *parameters*: (1) The local processing speed $\mathbf{r}$; (2) The number of processors $\mathbf{p}$; (3) The time $\mathbf{L}$ required for a barrier; (4) The time $\mathbf{g}$ for collectively delivering a 1-relation.

A 1-relation is a collective exchange where every processor receives/sends at most one word. The network can deliver an h-relation in time $\mathbf{g} \times h$. To accurately *estimate* the execution time of a BSP program, these 4 parameters can be easily benchmarked [2]. 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 program is the sum of its super-steps' costs.

3.1.4 Other presentations of BSP

In some previous works, the BSP model is presented in a slightly different way. We still prefer the presentation of [28] that is more common. For example, In [31,29], there is a barrier each $\mathbf{L}$ computation or communication steps. This born of computation/communication is hard to introduces in libraries and programming languages. Furthermore, when the algorithm has unknown (at compile time) steps of computations (*e.g.* model-checking algorithms), there is no need of these additional barriers.

Another example is in [2]. To simplify the cost analysis of the presented algorithms, there are two kinds of super-steps: those that perform only computations and on contrary, those that only performs communications. Both are terminated by a barrier. That do not truly change the model rather only introduces more barriers.

3.2 Axiomatic characterization of BSP algorithms

The BSP model defines the machine with multiple computing units (processors, cores, *etc.*), which have their own memory. Therefore, a state S_t of the algorithm must be a $\mathbf{p}$-tuple $\left(X_t^1, \ldots, X_t^{\mathbf{p}}\right)$[1] where $\mathbf{p}$ is the number of computing units. Each units needs to know its own process identification and the total number of units, so the language of the algorithm will contain two more symbols **nproc** and **pid**. We extend the sequential postulates to capture BSP algorithms. An execution of a BSP algorithm stay a sequence of states and the duration the number of steps done before reaching a final state.

BSP-Postulate 1 (Sequential Time) *A* BSP *algorithm A_{bsp} is given by:*

1. *a set of states $S(A_{bsp})$*
2. *a set of initial states $I(A_{bsp}) \subseteq S(A_{bsp})$*
3. *a transition function $\tau_{A_{bsp}} : S(A_{bsp}) \to S(A_{bsp})$*

An **execution** of A_{bsp} is a sequence of states $\mathbf{S} = S_0, S_1, S_2, \ldots$ such that:

1. S_0 is an initial state of $S(A_{\text{bsp}})$
2. For every $t \in \mathbb{N}$, $S_{t+1} = \tau_{A_{\text{bsp}}}(S_t)$

BSP-Postulate 2 (Abstract States) *For every* BSP *algorithm A_{BSP}:*

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

1. *The states of A are* $\mathbf{p}$*-tuples*[2] *of (first-order) structures with the same signature $\mathcal{L}(A)$, which contain the symbols* **nproc** *and* **pid**
2. *$S(A)$ and $I(A)$ are closed by* $\mathbf{p}$*-isomorphism*
3. *The transition function τ_A preserves the universes and the number* $\mathbf{p}$ *of processors, and commutes with the* $\mathbf{p}$*-isomorphisms*

A $\mathbf{p}$-isomorphism is a isomorphism that can into account $\mathbf{p}$-tuples. If $\left(X^1, \ldots, X^{\mathbf{p}}\right)$ is a state of the algorithm A, the structures $X^1, \ldots, X^{\mathbf{p}}$ will be called processor memories or **local memories**. The set of the local memories of A will be denoted by $M(A)$. 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 $\overline{t}^X \overset{\text{def}}{=} \overline{c}^X$
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 $\overline{t}^X \overset{\text{def}}{=} \overline{f}^X(\overline{t_1}^X, \ldots, \overline{t_\alpha}^X)$

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

BSP-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 $\left(X^1, \ldots, X^{\mathbf{p}}\right)$ and $\left(Y^1, \ldots, Y^{\mathbf{p}}\right)$, if they coincide over $T(A)$ then for every $1 \leq i \leq \mathbf{p}$, we have $\Delta(A, X^i) = \Delta(A, Y^i)$.*

This $T(A)$ is called the **exploration witness** of A. If a set of terms T is finite then its closure by subterms is finite too, so as in [19] we will assume that $T(A)$ is closed by subterms.

Lemma 1 (BSP structures as sequential ones) *If $(f, a_1, \ldots, a_\alpha, b) \in \Delta(A, X)$ then $a_1, \ldots, a_\alpha, b$ are interpretations in X of terms in T.*

The proof is as in [19] but using $\mathbf{p}$-tuples.

A purely asynchronous computation as a **parallel algorithm** is an object verifying these three postulates.

Lemma 2 *A parallel algorithm running with only one processor ($\mathbf{p} = 1$) is a sequential algorithm.*

For a BSP algorithm, the sequence of states is organized using super-steps. Notably, the communication between the processor memories occurs only during a phase. To do so, a BSP algorithm A will use two functions $\mathbf{comp}_A$ and $\mathbf{sync}_A$ indicating if the algorithm runs computations or runs communications (following by a synchronization). A BSP algorithm is an object verifying these four postulates, and we denote by $\texttt{Algo}_{\texttt{bsp}}$ the set of the BSP algorithms.

[2] $\mathbf{p}$ is fixed for the algorithm, so A can have states using different number of units.

BSP-Postulate 4 (Synchronization Barrier) *For every* BSP *algorithm A there exists two applications $\mathbf{comp}_A : M(A) \to M(A)$ and $\mathbf{sync}_A : S(A) \to S(A)$ such that :*

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

This requires three remarks. First, not only one processor performs the local computations but all those who can. Second, in case of a communication from i to j, $\mathcal{L}(X^j)$ must be interpreted in $\mathcal{L}(X^i)$. Third, we do not specified function $\mathbf{sync}_A$ in order to be generic about which BSP library of communications is used. For example, the BSPLIB [21] and the PUB [5] do not used the same kind of receiving routines:

```
BSPLib:
  void bsp_qsize(int *packets, int *accum_nbytes)
  void bsp_move(void *payload,int reception_bytes)
PUB:
  t_bspmsg* bsp_findmsg(int proc_id, int index)
  void* bspmsg_data(t_bspmsg* msg)
```

where, using the BSPLIB, `bsp_qsize` checks to see how many packets arrived and `bsp_move` moves a packet from the system queue independently of the emitter. Whereas, using the PUB, `bsp_findmsg` finds the nth message in the queue of a particular emitter and `bspmsg_data` returns the data of this message. We thus prefer to be generic of how data are communicated and just saying that data are moved.

A final states $(X^1, \ldots, X^{\mathbf{p}})$ is now where $\tau_A(X^1, \ldots, X^{\mathbf{p}}) = (X^1, \ldots, X^{\mathbf{p}}) = \mathbf{sync}_A(X^1, \ldots, X^{\mathbf{p}})$ that is there exists not $\leq i \leq \mathbf{p}$ such that $\mathbf{comp}_A(X^i) \neq X^i$ and the $\mathbf{sync}_A$ is the identical function[3].

4 Operational semantics of BSP and transformations

We now give the operational semantics of two kinds of BSP objects. First, a certain class of ASM which captures the above 4 BSP postulates. Second, a core language for imperative programming. We then give transformations of these two objets to each other. We finally prove their equivalence using a fair simulation and conclude that the core language *à la* BSPLIB is *universel* for BSP: all BSP algorithms could be programming with this language with the expected cost.

4.1 Semantics of BSP-ASM

The operational semantics of the BSP-ASMs is based on the following idea: one machine operates on $\mathbf{p}$ distinct memories until nothing happens; Then

[3] It is like a final synchronization phase which terminates the BSP algorithm; as we can found in MPI.

the communication function performs the exchanges of data and the machine backs to work; But if after the communication, the states remain the same, the BSP-ASM stops. BSP programs are SPMD (**S**ingle **P**rogram **M**ultiple **D**ata) so a BSP-ASM $\mathit{\Pi}$ is started $\mathbf{p}$ times.

Definition 2 (A BSP-ASM) A BSP-ASM is a pair of an ASM $\mathit{\Pi}$ with a $\mathbf{p}$-uplet of memories $(X^1, \cdots, X^{\mathbf{p}})$.

Because there exists subtle differences between the primitives of the BSP libraries, for both **B**ulk **M**essage **P**assing (BSMP) and **R**emote **M**emory **A**ccess (DRMA), we abstract and separate them into three kinds of routines [14] and thus the syntax of $\mathit{\Pi}$ is completed with: (1) the asynchronous sending $f_{send}(t_1, \ldots, t_\alpha) := t_0^{send}$ (that are only adding request of communication); (2) asynchronous reading of received messages $t_0^{rcv} := f_{rcv}(t_1, \ldots, t_\alpha)$ (reading the environment of communications); (3) a synchronous one f_{sync}, that performs the barrier and cause the communications of the whole BSP machine. Note that we can also simulate a collective routine such a broadcast using appropriate asynchronous sending and a synchronization. In this way, we abstract how performing communication between processors and synchronizing them. This allows to focus the work on the semantics study and not to the order (scheduling) of sending and reception of messages.

Remark 1 If these two $\mathbf{p}$-uplets $\left(X^1, \ldots, X^{\mathbf{p}}\right)$ and $\left(Y^1, \ldots, Y^{\mathbf{p}}\right)$ coincide over T then they coincide over $T \cup \{\mathbf{nproc}, \mathbf{pid}\}$. Indeed, we have $\overline{\mathbf{nproc}}^{X^i} = \mathbf{p} = \overline{\mathbf{nproc}}^{Y^i}$ for every X^i and Y^i, and moreover we have $\overline{\mathbf{pid}}^{X^i} = i = \overline{\mathbf{pid}}^{Y^i}$.

Remark 2 *There is a separation between the computation functions and the sending routines. And both definitions of are close. A naif simplification would to merge them. But that is not possible because when sending a value, it cannot contains symbols that the receiving processor is unable to interpret.*

The semantics $\overrightarrow{\mathit{\Delta}}$ of a BSP-ASM is as follow:

$$\overrightarrow{\mathit{\Delta}}(\mathit{\Pi}, X^1, \cdots, X^{\mathbf{p}}) =_{def} \begin{cases} \mathit{\Delta}(\mathit{\Pi}, X^1), \cdots \mathit{\Delta}(\mathit{\Pi}, X^{\mathbf{p}}) & \text{if } \exists 1 \leq i \leq \mathbf{p} \text{ such that } \mathit{\Delta}(\mathit{\Pi}, X^i) \neq \emptyset \\ \mathbf{sync}_{\mathit{\Pi}}(X^1, \cdots, X^{\mathbf{p}}) & \text{otherwise} \end{cases}$$

where $\mathit{\Delta}$ is the local semantics of ASMs (a recall is defined in appendix A). The *Read* (terms read) and *Write* (updates) are modified accordingly to the f_{send} and f_{rcv} functions. Fig 3 gives also the modification of $\mathit{\Delta}$ for these functions. For communicating routines, we supposes a specific symbol *env* (environment of communication) that can store the values to be communicated. This symbol comes with the function **newEnv** that can add (resp. delete) values when a request of communication (resp. reading values). This function uses the interpretation in X of its parameters to update the symbol *env*. Note that there is no rules for synchronizing routines. They will be used for the global reduction.

Definition 3 (BSP Terminal) A BSP-ASM is in a terminal state if $\forall 1 \leq i \leq \mathbf{p}, \mathit{\Delta}(\mathit{\Pi}, X^i) = \emptyset$ and $\mathbf{sync}_{\mathit{\Pi}}(X^1, \cdots, X^{\mathbf{p}}) = (X^1, \cdots, X^{\mathbf{p}})$

$$\begin{aligned}
\Delta(f_{send}(t_1,\ldots,t_\alpha) := t_0, X \sqcup \{env\}) &=_{def} \{X \sqcup \overline{\mathbf{newEnv}(env, f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)}^X\} \\
\Delta(t_0^{rcv} := f_{rcv}(t_1,\ldots,t_\alpha), X \sqcup \{env\}) &=_{def} \{X \sqcup \overline{(env \oplus (f_{rcv}, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0^{rcv}}^X)}^X\}
\end{aligned}$$

Fig. 3 Adding the the manage of the environment of communications in BSP-ASMs.

that is not computation nor communications are possible. It now is easy to see that the BSP-ASMs correspond to the previous postulate of BSP algorithms:

Proposition 1 $\texttt{Algo}_{\texttt{bsp}} \simeq$ BSP-ASM*s*

The full proofs could be find in [34] and appendix A.

Corollary 1 *Every* BSP-ASM *has a normal form.*

Proof According to the previous proposition, an BSP-ASM is a BSP algorithm, and according to the previous proof every BSP algorithm can be captured with a BSP-ASM in normal form. Thus, our initial BSP-ASM is equivalent to a BSP-ASM in normal form.

We can also adapt to BSP the definition of a fair simulation. Because the BSP-ASM continues to work (but by doing nothing) for processors that have not finish the current super-step, the time dilation d could be used as for sequential algorithms. Indeed, we count the number of steps in the super-steps and thus also the number of of super-steps. The only need difference is the dilatation of the communication. We thus introduce a new factor c that born the communications. The BSP fair simulation of P_1 by P_2 is now $time(P_1, X_0) = d \times time(P_2, Y_0) + e + c \times Allcomm(P_2, Y_0)$ where $Allcomm$ counts all the communications of the BSP algorithm and is defined as follow: $Allcomm(A_{\mathrm{bsp}}, S_0) \overset{\mathrm{def}}{=} \sum\{t \in \mathbb{N} \mid \text{if } \tau_{A_{\mathrm{bsp}}}(S_t) = S_{t+1} \text{ then}$

$$\begin{cases} \max\{i \in \{1 \cdots, \mathbf{p}\}, \max(\|X^i\|_{send}, \|X'^i\|_{rcv}) & \text{if } \tau_{A_{\mathrm{bsp}}} = \mathbf{sync}_A \\ 0 & \text{otherwise} \end{cases} \}$$

where $S_t = X^1, \ldots, X^{\mathbf{p}}$ and $S_{t+1} = X'^1, \ldots, X'^{\mathbf{p}}$. $\|Y^i\|_{send}$ denotes the size of the sending (resp. receiving) of Y^i.

4.2 Semantics of a core BSP language

$$\begin{aligned}
c =_{def} &\ \text{Sequential commands} \\
&\mid f_{send}(t_1,\ldots,t_\alpha) := t_0^{send} \\
&\mid f_{sync}() \\
&\mid t_0^{rcv} := f_{rcv}(t_1,\ldots,t_\alpha) \\
P =_{def} &\ \epsilon \mid c; P
\end{aligned}$$

Fig. 4 The core language.

The syntax of our language Imp_{bsp} is as for sequential computations with an additional syntax for BSP instructions. Fig. 4 gives the grammar. As explain before, we have the asynchronous sending $f_{send}(t_1,\ldots,t_\alpha) := t_0^{send}$, the asynchronous reading of received messages $t_0^{rcv} := f_{rcv}(t_1,\ldots,t_\alpha)$ and the barrier f_{sync}.

Sequential rules:

$$f(t_1,\dots,t_\alpha) := t_0; P \star X \succ^i P \star X \oplus (f, \overline{t_1}^X, \dots, \overline{t_\alpha}^X, \overline{t_0}^X)$$

$$\begin{array}{rl}
\texttt{if } F\ \{P_1\}\ \texttt{else}\ \{P_2\}; P_3 \star X \succ^i P_1 \fatsemi P_3 \star X & \text{if } F \text{ is true in } X \\
\texttt{if } F\ \{P_1\}\ \texttt{else}\ \{P_2\}; P_3 \star X \succ^i P_2 \fatsemi P_3 \star X & \text{if } F \text{ is false in } X \\
\texttt{while } F\ \{P_1\}; P_2 \star X \succ^i P_1; \texttt{while } F\ \{P_1\} \fatsemi P_2 \star X & \\
 & \text{if } F \text{ is true in } X \\
\texttt{while } F\ \{P_1\}; P_2 \star X \succ^i P_2 \star X & \text{if } F \text{ is false in } X
\end{array}$$

Managing the environment of communications :

$$f_{send}(t_1,\dots,t_\alpha) := t_0; P \star X \sqcup \{env\} \succ^i P \star X \sqcup \{\mathbf{newEnv}(env, f, \overline{t_1}^X, \dots, \overline{t_\alpha}^X, \overline{t_0}^X)\}$$
$$t_0^{rcv} := f_{rcv}(t_1,\dots,t_\alpha); P \star X \sqcup \{env\} \succ^i P \star X \oplus (f_{rcv}, \overline{t_1}^X, \dots, \overline{t_\alpha}^X, \overline{t_0^{rcv}}^X) \sqcup \{env\}$$

Fig. 5 Operational semantics; Local (sequential) rules of a computation unit i.

To define the operational semantics $\succ$, we need local rules to execute asynchronous computations and global rules for managing the whole distributed machine. BSP programs are SPMD so a program P is started $\mathbf{p}$ times. For this, we use $\mathbf{p}$-state that is a program (a sequence of commands) with its own $\mathcal{L}(X)$: $\langle P^1 \star X^1, \dots, P^{\mathbf{p}} \star X^{\mathbf{p}}\rangle$. The evaluation terminates when reaching a $\mathbf{p}$-state of the form $\langle \epsilon \star X^1, \dots, \epsilon \star X^{\mathbf{p}}\rangle$. If not, it is an invalid program or a diverging one if always we can apply the two following rules.

We first the rules for the local execution $\succ^i$, *i.e.* on a single processor i in Fig 5. They are close to the ones of the BSP-ASMs.

Compared to a semantics of a sequential language, the major change is located within the distributed rules $\succ$ and we note $\succ^*$ the transitive and reflexive closure of $\succ$. These two rules are defined in Fig 6. The first rule is for performing the local computations on all processors until reaching a synchronization routine. The second rule is for performing the whole synchronization/communication. For this, we suppose a function **Comm** that update all the $\mathcal{L}(X^i)$ and all the env^i. The **Comm** function models the exchanges of messages and thus specifies the order of the received messages depending on the parameter: it modifies the environment of each processor i; it is "just" a reordering of the $\mathbf{p}$ signatures. As usual, this function depending of you own BSP library.

Proposition 2 (Determinism) *The operational semantics is deterministic considering all external function $f_?$ deterministic. That is if $\langle P \star X^1, \dots, P \star X^{\mathbf{p}}\rangle \succ^* \langle \epsilon \star X'^1, \dots, \epsilon \star X'^{\mathbf{p}}\rangle$ and $\langle P \star X^1, \dots, P \star X^{\mathbf{p}}\rangle \succ^* \langle \epsilon \star X''^1, \dots, \epsilon \star X''^{\mathbf{p}}\rangle$ then $\langle X'^1, \dots, X'^{\mathbf{p}}\rangle = \langle X''^1, \dots, X''^{\mathbf{p}}\rangle$.*

The determinacy is easy to prove since all rules are locally deterministic and the $\succ$ reduction is locally confluent: if from one state two steps are possible, they can only be local executions on two different processors.

$$\frac{\exists 1 \leq i \leq \mathbf{p} \quad P^i \star X^i \succ^i P'^i \star X'^i}{\langle P^1 \star X^1, \ldots, P^{\mathbf{p}} \star X^{\mathbf{p}} \rangle \succ \langle P'^1 \star X'^1, \ldots, P'^{\mathbf{p}} \star X'^{\mathbf{p}} \rangle}$$

$$\frac{\langle P'^1 \star X'^1, \ldots, P'^{\mathbf{p}} \star X'^{\mathbf{p}} \rangle = \mathbf{Comm}(\langle P^1 \star X^1, \ldots, P^{\mathbf{p}} \star X^{\mathbf{p}} \rangle)}{\langle P^1 \star X^1, \ldots, P^{\mathbf{p}} \star X^{\mathbf{p}} \rangle \succ \langle P'^1 \star X'^1, \ldots, P'^{\mathbf{p}} \star X'^{\mathbf{p}} \rangle}$$
where $\forall i,\ P^i \equiv f_{sync}(); P'^i$

Fig. 6 Operational semantics; global rules of the whole machine.

$$\begin{aligned}
\mathcal{G}(\epsilon) &\overset{\text{def}}{=} \{\epsilon\} \\
\mathcal{G}(c; P) &\overset{\text{def}}{=} \mathcal{G}(c); P \cup \mathcal{G}(P) \\
\mathcal{G}(f_?(t_1, \ldots, t_\alpha) := t_?) &\overset{\text{def}}{=} \{f_?(t_1, \ldots, t_\alpha) := t_?\} \quad \text{where } ? = \_, sync, send, rcv \\
\mathcal{G}(\texttt{if } F\ \{P_1\} \texttt{ else } \{P_2\}) &\overset{\text{def}}{=} \{\texttt{if } F\ \{P_1\} \texttt{ else } \{P_2\}\ \} \cup \mathcal{G}(P_1) \cup \mathcal{G}(P_2) \\
\mathcal{G}(\texttt{while } F\ \{P\}) &\overset{\text{def}}{=} \{\texttt{while } F\ \{P\}\ \} \cup \mathcal{G}(P); \texttt{while } F\ \{P\}
\end{aligned}$$

Fig. 7 The control flow graph of a program P.

4.3 Transformations of the objets

4.3.1 From Imp_{bsp} to BSP-ASMs

The compilation of a program P into a BSP-ASM Π_P is done first by the simulation of each step of the evaluation of the program by the ASM and second, to find these steps, generating a code (defined in Fig 8) for each instruction of P using a control flow graph[4] (defined in Fig 7).

In order to define the control flow graphs we need to introduce the notation:

$$\mathcal{G}; P =_{def} \{P_j; P \mid P_j \in \mathcal{G}\}$$

where $\mathcal{G}$ is a set of imperative programs and P is an imperative program. Let P be an imperative program. $\mathcal{G}(P)$, in Fig 7, is the set of every possible $\tau_X^t(P)$ programs, which does not depend on an initial state X.

The fresh boolean variables will be denoted b_{P_j}, where $P_j \in \mathcal{G}(P)$. One and only one b_{P_j} will be true for each step of an execution, so in the following we will write $X[b_{P_j}]$ if b_{P_j} is true and the other booleans b_{P_k} are false, where X denotes a $\mathcal{L}(P)$-structure. Notice that $X[b_{P_j}]|_{\mathcal{L}(P)} = X$.

Now, the translation of an imperative program P into an ASM is:

$$\Pi_P \overset{\text{def}}{=} \texttt{if } \neg b_{\text{wait}} \texttt{ then } \underset{P_j \in \mathcal{G}(P)}{\texttt{par}} \texttt{ if } b_{P_j} \texttt{ then } [\![P_j]\!]_{asm} \texttt{ endpar endif}$$

where $[\![P_j]\!]_{asm}$ is defined in Fig 8. The b_{wait} is another fresh boolean that is initiated and set to false by the **Comm** function (of communication). Then the synchronous primitive set it to true making the BSP-ASM nothing for the processor until the end of the current super-step. Note that the functions

[4] The possible executions of a program will be represented by a graph where the edges are the possible transitions, and the vertices are the possible programs.

```
$[\![\epsilon]\!]_{asm} \overset{\text{def}}{=}$ par endpar

$[\![f_?(t_1, \ldots, t_\alpha) := t_0; Q]\!]_{asm} \overset{\text{def}}{=}$ par
    $b_{f_?(t_1,\ldots,t_\alpha):=t_0;Q} :=$ false
    $\| f_?(t_1, \ldots, t_\alpha) := t_0$
    $\| b_Q :=$ true
endpar      where $? = \_, send, rcv$

$[\![$if $F$ then $\{P_1\}$ else $\{P_2\}; Q]\!]_{asm} \overset{\text{def}}{=}$ par
    $b_{\texttt{if } F \texttt{ then } \{P_1\} \texttt{ else } \{P_2\};Q} :=$ false
    $\|$if $F$ then
      $b_{P_1;Q} :=$ true
      else
      $b_{P_2;Q} :=$ true
      endif
endpar

$[\![$while $F$ $\{P\}; Q]\!]_{asm} \overset{\text{def}}{=}$ par
    $b_{\texttt{while } F \ \{P\};Q} :=$ false
    $\|$if $F$ then
      $b_{P;\texttt{while } F \ \{P\};Q} :=$ true
      else
      $b_Q :=$ true
      endif
endpar

$[\![f_{sync}(); Q]\!]_{asm} \overset{\text{def}}{=}$ par
    $b_{\mathsf{sync};Q} :=$ false
    $\| b_{\mathsf{wait}} :=$ true
    $\| b_Q :=$ true
endpar
```

Fig. 8 Generation of ASM code from the program.

(f), asynchronous send/receive (f_{send} and f_{rcv}) are transformed to their ASM counterparts. A special case is the loop with an empty code ϵ. A simple solution is available in [33] but is not presented here since it is irrelevant.

Notice that for every $P_j \in \mathcal{G}(P)$, $\Delta(\Pi_P, X[b_{P_j}]) = \Delta(P_j^{tr}, X[b_{P_j}])$. We use this fact in [34] to prove by exhaustion on $\tau_X^t(P)$ that the translation of the imperative program P behaves as intended:

Proposition 3 *(Step-by-Step Simulation)*

$$\textit{For every } t < \textit{time}(P, X), \tau_{\Pi_P}(\tau_P^t(X)[b_{\tau_X^t(P)}]) = \tau_P^{t+1}(X)[b_{\tau_X^{t+1}(P)}]$$

Proposition 4 BSP-ASM *fairly simulates* Imp_{bsp} *with (ending)* $e = 0$ *and (temporal dilatation)* $d = 1$ *and* $c = 1$ *(without communication dilatation).*

The full proofs could be find in [34].

4.3.2 From BSP-ASM*s to* Imp_{bsp}

Let Π be a BSP-ASM program. The purpose of this sub-section is to find a BSP program P_{step} simulating the same executions as Π. We construct it in three steps: (1) Translate Π into an imperative program P_{step} simulating one step of the BSP-ASM; (2) Repeat P_{step} a sufficient number of times, depending on φ_Π, the complexity of Π; (3) Ensure that the final program stops at the same time as the BSP-ASM, up to temporal dilation.

According to [19], every ASM is equivalent to an ASM in normal form, so we can assume that Π is in the following normal form:

```
par if $F_1$ then $\Pi_1$
    || if $F_2$ then $\Pi_2$
        ⋮
    || if $F_c$ then $\Pi_c$
endpar
```

So that every F_i is a guard, which means that only one F_i is true for the current state X. And, each Π_i contains only updates inside a `par` command.

(1) Translation of one Step. The solution is to translate the commands directly by adding temporal variables since we must paying attention to the ASM's parallelism of updates: the simultaneous commands of `ASM` must be sequentialized; To do so, we need to store the values of the variables read in the ASM program into fresh variables otherwise such variables may be overwrite by the parallelism of updates. For example, the following ASM:

```
par $x := y \| y := x$ endpar
```

must be transformed into a Imp_{bsp} code of the form:

$$v_y := y; v_x := x; x := v_y; y := v_x;$$

Such a transformation could be find in [33]. Mainly, by using the set of read variables, on step of Π is translated as following:

```
$P_{step} =_{def}$
    $v_{t_1} := t_1; \cdots ; v_{t_r} := t_r;$
    if $v_{F_1}$ then $\{ f_1^1(\mathbf{v}_{t_1^1}) := v_{t_1^1}; \cdots ; f_{m_1}^1(\mathbf{v}_{t_{m_1}^1}) := v_{t_{m_1}^1;\texttt{skip}^{(m-m_1)};}; \};$
⋮
    if $v_{F_c}$ then $\{ f_1^c(\mathbf{v}_{t_1^c}) := v_{t_1^c}; \cdots f_{m_c}^c(\mathbf{v}_{t_{m_c}^c}) := v_{t_{m_c}^c;\texttt{skip}^{(m-m_c)};}; \};$
end
```

where v_{t_i} are the fresh variables. Because these variables must have a uniform initialization, we have to update them explicitly at the beginning of the program by using a sequence of updates. Moreover, the execution time depends on the current initial state; This is an issue because, according to our definition of the fair simulation, every step of the ASM Π must be simulated by d steps, where d depends only on Π. In order to obtain a uniform temporal dilation, we will add `skip` commands[5] to the program. As intended, we prove that P_{step} simulates one step of Π in a constant time t_Π:

Proposition 5 (Semantical Translation of the BSP-ASM programs) *There exists t_Π, depending only on Π, such that for every state X of P_{step}:*
- $(P_{step}(X) \ominus X)|_{\mathcal{L}(\Pi)} = \Delta(\Pi, X|_{\mathcal{L}(\Pi)})$
- $time(P_{step}, X) = t_\Pi$

The proof is done by case analysis and could be find in [34].

(2) Translation for the whole machine.

```
$b_{stop} := false$;
while $\neg b_{stop}$
{
    $P_{step}$;
    while $\neg F_\Pi$ {$P_{step;}$}
    $f_{sync}()$;
}
```

(3) Termination of the program.

Proposition 6 *Imp_{bsp} fairly simulates $\texttt{ASM}_\mathcal{P}$ with (ending) $e = t_\Pi + 6 \times P_\Pi + 1$ and (temporal dilatation) $d = t_\Pi + 2$ and $c = 1$ (without communication dilatation).*

Theorem 2 $\texttt{Algo}_{\texttt{bsp}} \simeq \text{BSP}-\text{ASM} \simeq \texttt{Imp}_{\texttt{bsp}}$

5 Conclusion and Future Work

5.1 Summary of the Contribution

The following equation summarizes the result: $\texttt{Algo}_{\texttt{bsp}} \simeq \text{BSP}-\text{ASM} \simeq \texttt{Imp}_{\texttt{bsp}}$. More precisely, we have give an axiomatic definition of BSP algorithms by adding only one postulate to the sequential definition. Mainly this postulate is the call of a global, abstract and synchronous function of communications. This function is also used to synchronize both a parallel extension of ASMs and a imperative programming language *à la* BSPLIB.

We have give two functions of compilation, one from BSP-ASM to this core language and another for the opposite path. And we prove a fair simulation of these objects. We can conclude that the core language is BSP algorithmic complete and thus that the BSPLIB is.

[5] It may seem strange in an algorithmic purpose to lose time, but these `skip` commands do not change the asymptotic behavior and are only necessary for our strict definition of the fair simulation. It is possible to weaken the definition of the simulation to simulate one step with $\leq d$ steps and not $= d$ steps, but that makes the proof unnecessarily more complicated.

5.2 Questions and answers about this work

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

For sequential computing, it is known that Turing machines could simulate every algorithms 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 BSP-ASM *instead of using* ASM*s? Indeed, each processor can be seen as a sequential* ASM *"à la Gurevich". So, in order to simulate one step of a* BSP *algorithm using several processors, we could use pid to compute sequentially the next step for each processor by using an* ASM.

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

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

Indeed, the total number of steps depends on the size of the inputs, which includes *nprocs*. But the execution time should not be confused with the cost for simulating one step. If the BSP algorithm requires $f(n)$ steps, where n is the size of the inputs, then our simulation will require $d * f(n) + e$ steps (and the same communications). Our simulation respects the $O(f(n))$ in time, but this is not the main point. Our simulation is algorithmic because it respects the step-by-step behavior of the BSP algorithm, by simulating each step by d steps, where d does not depend of the size of the inputs or the number of processors. Also, it is not possible to use $\mathbf{p}$ ASM's parallel updates since it breaks the bound postulate. Simulating the execution of a BSP program with a sequential one has been studied by the second author in [14] but in the context of proving the correctness of algorithms not the algorithmically completeness.

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) 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 commands?*

The sequential parts can be model 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 with random reading of messages such as in the BSPLIB*?*

Random algorithms has also been studied in the ASM framework [7]. We can say that such structures are typically heaps of messages. And thus, two executions, within the same environments, will lead to identical results. For determinacy, adding the environments of execution is a tricky (but not elegant) solution. Otherwise, truly formally studying randomized algorithms is a hard task, even for sequential computing, which is only at its beginning and which is irrelevant in this article.

What happen in case of runtime error during communications?

Typically, when one processor has a bigger number of super-steps than other processors, or when there is an out-of-bound sending or reading, it leads to a runtime error. The BSP function of communication can return a $\perp$ value and nothing can be do stopping the semantics for both BSP-ASM and $\texttt{Imp}_{\texttt{bsp}}$.

Why uses this plethoric number of definitions to just prove an oblious result?

Well, It is the main drawback of formal proofs: everything needs to be defined even the most intuitive concepts. But keep in mind that this algorithmic equivalence has never been proved before. And having given these definitions would surely simplify an adaptation of this work to other bridging models.

What about related work?

To our knowledge, some works exist to model parallel or distributed programs using ASMs but none to characterize BSP algorithms and, using ASMs, to prove that a language is algorithmic complete. One example is the work of [10] to models the P3L's skeletons. That allows the analyses of P3L programs using standard ASM tools but not a formal characterization what is P3L and what is not (in an algorithm point of view).

The first work to extends ASMs for concurrent, distributed, agent-mobile algorithms is [4]. Too many postulates are used making the comprehension hard to follow and worst, loses confidence in these postulates. A first attend to simplify this work has been done in [26] and again simplified in [13] by the the use of multiset comprehension terms to maintain the bounded exploration. Then, the authors prove that ASMs captures these postulates. We believe that their postulates are more general than ours. But the authors are not concerned about the problems of algorithm completeness using a cost model which is the heart of our work (and the main advantage of the BSP model).

Extends the ASMs for distributed computing is not new [6]. For example the works of [25, 9] about multi-agents and data-flow programs. We think that our extension still remains simple and natural for BSP computing.

5.3 Future Work

This work leads to many possible work. First, to gain in confidence, we are working on a mechanical proof using the COQ system. Second, how adapting our work to an hierarchical extension of BSP [32] (which is more close to modern HPC architectures)? And for other models such as logp [11], dBSP [12] (for subgroups synchronizations), *etc.*? Can we thus imagine a method that "automatically" characterizes parallel model since there is (or will be) a plethoric number of bridging models.

More concretely, there is many languages having a BSP model of execution, for example Pregel [24] for writing large-graph algorithms. An interesting work is proving which are BSP algorithmic complete and which are not. Pregel is a good candidate to be *not* BSP. Indeed, a short-path computation using Pregel needs n super-steps (where n is the shorter path) since a node could only communicating with its neighborhood, whereas $\log(\mathbf{p})$ super-steps could be done [30]. Similarly, once can imaging proving which languages are too expressive to be restricted to BSP.

References

1. Biedl, T.C., Buss, J.F., Demaine, E.D., Demaine, M.L., Hajiaghayi, M.T., Vinar, T.: Palindrome recognition using a multidimensional tape. Theor. Comput. Sci. **302**(1-3), 475–480 (2003)
2. Bisseling, R.H.: Parallel Scientific Computation. A structured approach using BSP and MPI. Oxford University Press (2004)
3. Blass, A., Dershowitz, N., Gurevich, Y.: When are two algorithms the same? Bulletin of Symbolic Logic **15**(2), 145–168 (2009)
4. Blass, A., Gurevich, Y.: Abstract state machines capture parallel algorithms. ACM Trans. Comput. Log. **4**(4), 578–651 (2003)
5. Bonorden, O., Judoiink, B., von Otte, I., Rieping, O.: The Paderborn University BSP (PUB) library. Parallel Computing **29**(2), 187–207 (2003)
6. Börger, E., Schewe, K.: Concurrent abstract state machines. Acta Inf. **53**(5), 469–492 (2016)
7. Börger, E., Stärk, R.F.: Abstract State Machines. A Method for High-Level System Design and Analysis. Springer (2003)
8. Cappello, F., Guermouche, A., Snir, M.: On Communication Determinism in Parallel HPC Applications. In: Computer Communications and Networks (ICCCN), pp. 1–8. IEEE (2010)
9. Cavarra, A.: A data-flow approach to test multi-agent asms. Formal Asp. Comput. **23**(1), 21–41 (2011)
10. Cavarra, A., Riccobene, E., Zavanella, A.: A formal model for the parallel semantics of p3l. In: ACM Symposium on Applied Computing (SAC), pp. 804–812 (2000)
11. Culler, D., Karp, R., Patterson, D., Sahay, A., Schauser, K., Santos, E., Subramonian, R., von Eicken, T.: Logp : Toward a realistic model of parallel computation. ACM SIGPLAN Symposium on Principles and Practises of Parallel Programming pp. 1–12 (1993)
12. Fantozzi, C.: A computational model for parallel and hierarchical machines. Ph.D. thesis, University degli Studi di Padova (2003)
13. Ferrarotti, F., Schewe, K.D., Tec, L., Wang, Q.: A new thesis concerning synchronised parallel computing simplified parallel {ASM} thesis. Theoretical Computer Science **649**, 25–53 (2016)
14. Fortin, J., Gava, F.: BSP-WHY: An Intermediate Language for Deductive Verification of BSP Programs. In: High-Level Parallel Programming and Applications (HLPP), pp. 35–44. ACM (2010)
15. Gesbert, L., Gava, F., Loulergue, F., Dabrowski, F.: Bulk Synchronous Parallel ML with Exceptions. Future Generation Computer Systems **26**, 486–490 (2010)
16. González-Vélez, H., Leyton, M.: A Survey of Algorithmic Skeleton Frameworks: High-level Structured Parallel Programming Enablers. Software, Practrice & Experience **40**(12), 1135–1160 (2010)
17. Gorlatch, S.: Send-receive considered harmful: Myths and realities of message passing. ACM TOPLAS **26**(1), 47–56 (2004). DOI http://doi.acm.org/10.1145/963778.963780

18. Grigorieff, S., Valarcher, P.: Evolving multialgebras unify all usual sequential computation models. In: J. Marion, T. Schwentick (eds.) Theoretical Aspects of Computer Science (STACS), *LIPIcs*, vol. 5, pp. 417–428. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik (2010)
19. Gurevich, Y.: Sequential abstract-state machines capture sequential algorithms. ACM Trans. Comput. Log. **1**(1), 77–111 (2000)
20. Hamidouche, K., Falcou, J., Etiemble, D.: A Framework for an Automatic Hybrid MPI +OPEN-MP Code Generation. In: L.T. Watson, G.W. Howell, W.I. Thacker, S. Seidel (eds.) Simulation Multi-conference (SpringSim) on High Performance Computing Symposia (HPC), pp. 48–55. SCS/ACM (2011)
21. Hill, J.M.D., McColl, B., Stefanescu, D.C., Goudreau, M.W., Lang, K., Rao, S.B., Suel, T., Tsantilas, T., Bisseling, R.: BSPLIB: The BSP Programming Library. Parallel Computing **24**, 1947–1980 (1998)
22. Keßler, C.W.: NestStep: Nested Parallelism and Virtual Shared Memory for the BSP Model. The Journal of Supercomputing **17**(3), 245–262 (2000)
23. Loulergue, F.: BSλ_p: Functional BSP Programs on Enumerated Vectors. In: J. Kazuki (ed.) International Symposium on High Performance Computing, no. 1940 in Lecture Notes in Computer Science, pp. 355–363. Springer (2000). DOI 10.1007/3-540-39999-2_34
24. Malewicz, G., Austern, M.H., Bik, A.J.C., Dehnert, J.C., Horn, I., Leiser, N.: Pregel: A System for Large-scale Graph Processing. In: Management of data, pp. 135–146. ACM (2010)
25. Schewe, K.D., Ferrarotti, F., Tec, L., Wang, Q., An, W.: Evolving concurrent systems: Behavioural theory and logic. In: Australasian Computer Science Week Multiconference (ACSW), pp. 1–10 (2017)
26. Schewe, K.D., Wang, Q.: A simplified parallel asm thesis. In: J. Derrick, J. Fitzgerald, S. Gnesi, S. Khurshid, M. Leuschel, S. Reeves, E. Riccobene (eds.) Abstract State Machines, Alloy, B, VDM (ABZ), pp. 341–344. Springer (2012)
27. Seo, S., Yoon, E.J., Kim, J.H., Jin, S., Kim, J.S., Maeng, S.: HAMA: An Efficient Matrix Computation with the MapReduce Framework. In: Cloud Computing (CloudCom), pp. 721–726. IEEE (2010)
28. Skillicorn, D.B., Hill, J.M.D., McColl, W.F.: Questions and Answers about BSP. Scientific Programming **6**(3), 249–274 (1997)
29. Tiskin, A.: The bulk-synchronous parallel random access machine. Theoretical Computer Science **196**(1–2), 109–130 (1998)
30. Tiskin, A.: The Design and Analysis of Bulk-Synchronous Parallel Algorithms. Ph.D. thesis, Oxford University Computing Laboratory (1998)
31. Valiant, L.G.: A Bridging Model for Parallel Computation. Comm. of the ACM **33**(8), 103–111 (1990)
32. Valiant, L.G.: A bridging model for multi-core computing. J. Comput. Syst. Sci. **77**(1), 154–166 (2011)
33. Yoann Marquer: Caractérisation impérative des algorithmes séquentiels en temps quelconque, primitif récursif ou polynomial (Imperative Characterization of Sequential Algorithms in general, primitive recursive or polynomial time). Ph.D. thesis, Univerity of Paris-Est (LACL) (2016). URL `\url{https://tel.archives-ouvertes.fr/tel-01280467/document}`
34. Yoann Marquier, F.G.: Imperative characterization of BSP algorithms: definitions and proofs. Tech. Rep. TR-2017-1, Univerity of Paris-Est (LACL) (2017; to appear.)
35. Yzelman, A.N., Bisseling, R.H.: An Object-oriented Bulk Synchronous Parallel Library for Multicore Programming. Concurrency and Computation: Practice and Experience **24**(5), 533–553 (2012)

A Characterizing sequential algorithms

In this section, we recall the axiomatic definition of sequential algorithms. We will introduce the postulates briefly in the first subsection, as well as other notions from ASMs such as execution, time, structure and update. In the second subsection, we recall the use of ASM to capture in an operational manner the axiomatic definition of sequential algorithms. In the third subsection, we finish with the algorithmic equivalence between ASM and the core imperative language.

A.1 Axiomatic definition of sequential algorithms

A common definition of algorithms is the set of functions computable by a determinist Turing Machine in a finite time. That is to say a set of instructions, a memory and a finite and determinist execution of these instructions. This definition uses a step-timer principle (bounding the running time), which can be criticized in two ways:

1. According to this definition, one-tape Turing Machines and two-tape Turing Machines should be equivalent. But the palindrome recognition can be done in $O(n)$ steps with a two-tape Turing machine while requiring (see
citePalyndrome at least $O(n^2/log(n))$ steps with a one-tape Turing machine. So, the concrete implementation and its associated complexity (more specifically, the degree of the time complexity) depend on the considered model. Therefore, we need a more precise definition of time algorithms.
2. The second issue comes from the step-timer principle itself. It is not convenient for programmers to use an external function attached to the algorithm. The answer comes from *implicit complexity computation* frameworks. But in general, these external functions are unknown and can break the complexity. Their thus a need to explain what is acceptable or not.

In [19], sequential algorithms are giving by the three following postulates [19]: (1) *Sequential Time*; (2) *Abstract States* and (3) *Bounded Exploration.* In the rest of the paper, the (infinite) set of "objects" satisfying these three postulates is denoted by $\texttt{Algo}_{\texttt{seq}}$. It is to notice that by using only 4 postulates to trust, we can have a good confidence to the results.

Postulate 1 (Sequential Time) *A sequential algorithm A_{seq} is given by:*

1. *a set of states $S(A_{seq})$*
2. *a set of initial states $I(A_{seq}) \subseteq S(A_{seq})$*
3. *a transition function $\tau_{A_{seq}} : S(A_{seq}) \to S(A_{seq})$*

An **execution** of A_{seq} is a sequence of states $\mathbf{S} = S_0, S_1, S_2, \ldots$ such that:

1. S_0 is an initial state of $S(A_{\mathrm{seq}})$
2. For every $t \in \mathbb{N}$, $S_{t+1} = \tau_{A_{\mathrm{seq}}}(S_t)$

Remark 3 *Two sequential algorithms A_{seq} and B_{seq} of $\texttt{Algo}_{\texttt{seq}}$ have the same set of executions if they have the same set of initial states $I(A_{seq}) = I(B_{seq})$ and the same transition function $\tau_{A_{seq}} = \tau_{B_{seq}}$. In that case, they can only be different on the states which cannot be reached by an execution.*

There is a debate [3] about this equality but it is not the subject of this paper.

A state S_t of an execution is final if $S_{t+1} = S_t$. An execution is **terminal** if it contains a final state. The duration of an execution is defined by the number of steps[6] done before reaching a final state:

$$\mathrm{time}(A_{\mathrm{seq}}, S_0) \overset{\mathrm{def}}{=} \begin{cases} \min\{t \in \mathbb{N} \mid \tau^t_{A_{\mathrm{seq}}}(S_0) = \tau^{t+1}_{A_{\mathrm{seq}}}(S_0)\} & \text{if } \mathbf{S} \text{ is terminal} \\ \infty & \text{otherwise} \end{cases}$$

Notice that if the execution $\mathbf{X}$ is not terminal then $time(A_{\mathrm{seq}}, X_0) = \infty$.

To state the second postulate, we need to introduce the notion of structure. States of a sequential algorithm are formalized with first-order structures [19]. A (first-order) **structure** X is given by:

[6] In the definition of time, f^i is the iteration of f defined by $f^0 = id$ and $f^{i+1} = f(f^i)$.

1. An infinite[7] set $\mathcal{U}(X)$ called the **universe** (or domain) of X
2. A finite set of function symbols $\mathcal{L}(X)$ called the **signature** (or language) of X
3. For every symbol $s \in \mathcal{L}(X)$ an **interpretation** $\overline{s}^X$ such that:
   (a) If c has arity 0 then $\overline{c}^X$ is an element of $\mathcal{U}(X)$
   (b) If f has an arity $\alpha > 0$ then $\overline{f}^X$ is an application: $\mathcal{U}(X)^\alpha \to \mathcal{U}(X)$

In order to have a uniform presentation, as in [19], 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. Moreover, partial functions can be implemented with a special value **undef**. This notion of interpretation allows to go from the world of language to the platonic world of mathematics.

The second postulate can be seen as a claim assuming that every data structure can be formalized as a first-order structure[8]. Moreover, since the representation of states should be independent from their concrete implementation (for example the name of objects), isomorphic states will be considered as equivalent.

Postulate 2 (Abstract States) *For every sequential algorithm A_{seq},*

1. *The states of A_{seq} are (first-order) structures with the same signature $\mathcal{L}(A_{seq})$*
2. *$S(A_{seq})$ and $I(A_{seq})$ are closed by isomorphism*
3. *The transition function $\tau_{A_{seq}}$ preserves the universes and commutes with the isomorphisms*

The symbols of the signature $\mathcal{L}(A)$ are distinguished between:

1. **Dyn**(A) the set of **dynamic symbols**, whose interpretation can change during an execution (as an example, a variable x)
2. **Stat**(A) the set of **static symbols**, which have a fixed interpretation during an execution. They are also distinguished between:
   (a) *Init*(A), the set of **parameters**, whose interpretation depends only on the initial state (as an example, two given integers m and n).
   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), and they are also distinguished between:
   (b) *Cons*(A) the set of **constructors** (*true* and *false* for the booleans, 0 and S for the unary integers, ...)
   (c) *Oper*(A) the set of **operations** ($\neg$ and $\wedge$ for the booleans, $+$ and $\times$ for the unary integers, ...)

The **size** of an element of the universe is the length of its representation [33], in other words the number of constructors necessary to write it. For examples, $|\overline{\neg\neg true}^X| = |\overline{true}^X| = 1$ and $|\overline{1+2}^X| = |\overline{S(S(S(0)))}^X| = 4$ when using an unary numeral representation. The logical variables are not used in this paper: every term and every formula is closed, and every formula is without quantifier. In this framework the **variables** are the 0-ary dynamic function symbols.

The size of a state is the maximum of the size of its inputs:

$$|X| =_{def} \underset{f \in Dyn(A) \sqcup Init(A)}{max} \{|f|_X\}, \text{ where } |f|_X =_{def} \underset{a_i \in \mathcal{U}(A)}{sup} |\overline{f}^X(\mathbf{a})|$$

For a sequential algorithm A, let X be a state of A, $f \in \mathcal{L}(A)$ be a dynamic α-ary function symbol, and $a_1, \ldots, a_\alpha, b \in \mathcal{U}(X)$. $(f, a_1, \ldots, a_\alpha)$ denotes a location of X and $(f, a_1, \ldots, a_\alpha, b)$ denotes an **update** on X at the location $(f, a_1, \ldots, a_\alpha)$.

[7] Usually the universe is only required to be non-empty, we need the universe to be at least countable in order to define unary integers.

[8] We tried to characterize common data types (such as integers, words, lists, arrays, and graphs) in [33]. But we will not go into details, because this is not the point of this article.

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:

$$\overline{f}^{X \oplus u}(\mathbf{a}) =_{def} \begin{cases} b & \text{if } u = (f, \mathbf{a}, b) \\ \overline{f}^{X}(\mathbf{a}) & \text{else} \end{cases}$$

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

If Δ is a set of updates then Δ is **consistent** on X if it does not contain two distinct updates with the same location. If Δ is inconsistent, there exists $(f, \mathbf{a}, b), (f, \mathbf{a}, b') \in \Delta$ with $b \neq b'$, so the entire set of updates clashes:

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

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

Let $\Delta(A, X) = \tau_A(X) \ominus X$ be the set of updates done by a sequential algorithm A on the state X.

During an execution, if the number of updates is not *statically* bounded then the algorithm will be said massively parallel, not sequential. The two first postulates cannot ensure that only local and bounded explorations/changes are done at every step. The third postulate states that only a bounded number of terms must be read or updated during a step of the execution:

Postulate 3 (Bounded Exploration) *For every algorithm A there exists a finite set T of terms (closed by subterms) such that for every state X and Y, if the elements of T have the same interpretations on X and Y then $\Delta(A, X) = \Delta(A, Y)$.*

This T is called the **exploration witness** of A.

In [19], it has been proved that if $(f, a_1, \dots, a_\alpha, b) \in \Delta(A, X)$ then $a_1, \dots, a_\alpha, b$ are interpretations in X of terms in T. So, since T is finite there exists a bounded number of $a_1, \dots, a_\alpha, b$ such that the update $(f, a_1, \dots, a_\alpha, b)$ belongs to $\Delta(A, X)$. Moreover, since $\mathcal{L}(A)$ is finite there exists a bounded number of dynamic symbols f. Therefore, $\Delta(A, X)$ has a bounded number of elements, and for every step of the algorithm only a bounded amount of work is done.

A.2 ASM

We now recall the definition of Abstract State Machines (ASM) [19] and their operational semantics. We also how to get a constructive (from an operational point of view) occurrence of sequential algorithms. The model of ASM is a kind of super Turing machine that works not on simple tapes (with finite alphabets) but on multi–sorted algebras. A program is a finite set of rules that updates terms. It is shown in [33], that the expressive power of ASM lies not in control structures but in data structures, which are modeled within $\mathcal{L}$-structures. Without going into details, ASM require only in the $\mathcal{L}$-structures the equality $=$, the constants *true* and $false$, the unary operation $\neg$ and the binary operations $\wedge$. They are defined as follow:

Definition 4 (`ASM` programs)

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

where f is a dynamic α-ary function symbol, $t_0, t_1, \dots, t_\alpha$ are closed terms, and F is a formula.

Notation 3 *For $n = 0$ a* `par` *command is an empty program, so let* `skip` *be the command* `par endpar`*. If the* `else` *part of an* `if` *is a* `skip` *we only write* `if` F `then` Π `endif`*.*

The sets $Read(\Pi)$ of terms read by Π and $Write(\Pi)$ of terms written by Π can be used to define the exploration witness of Π. But we will also use them in the rest of the article, especially to define the μ-formula F_Π p.

$Read(\Pi)$ is defined by induction on Π:

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

$Write(\Pi)$ is defined by induction on Π:

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

Remark 4 *The exploration witness of Π is the closure by subterms of $Read(\Pi) \cup Write(\Pi)$ and not only $Read(\Pi)$ because the updates of a command could be trivial.*

An `ASM` program Π determines a transition function $\tau_\Pi(X) =_{def} X \oplus \Delta(\Pi, X)$, where the set of updates $\Delta(\Pi, X)$ done by Π on X is defined by induction:

Definition 5 (Operational Semantics of ASMs)

$$\Delta(f(t_1, \ldots, t_\alpha) := t_0, X) =_{def} \{(f, \overline{t_1}^X, \ldots, \overline{t_\alpha}^X, \overline{t_0}^X)\}$$
$$\Delta(\texttt{if } F \texttt{ then } \Pi_1 \texttt{ else } \Pi_2 \texttt{ endif}, X) =_{def} \Delta(\Pi_i, X)$$

$$\text{where } i = \begin{cases} 1 \text{ if } F \text{ is true on } X \\ 2 \text{ else} \end{cases}$$

$$\Delta(\texttt{par } \Pi_1 \| \ldots \| \Pi_n \texttt{ endpar}, X) =_{def} \Delta(\Pi_1, X) \cup \ldots \cup \Delta(\Pi_n, X)$$

Notice that the semantics of the `par` is a set of updates done simultaneously, contrary to the imperative language defined in the next subsection, which is strictly sequential. Furthermore, the `par` construction has a constant arity which does not depend of the input so could not be the number of processors of the machine (during execution). This construction is only used for update the $\mathcal{L}$-structures.

Remark 5 *For every states X and Y, if the terms of $Read(\Pi)$ have the same interpretation on X and Y then $\Delta(\Pi, X) = \Delta(\Pi, Y)$.*

Definition 6 An ASM M with signature $\mathcal{L}$ is given by:

- an `ASM` program Π on $\mathcal{L}$
- a set $S(M)$ of $\mathcal{L}$-structures closed by isomorphisms and τ_Π
- a subset $I(M) \subseteq S(M)$ closed by isomorphisms
- an application τ_M, which is the restriction of τ_Π to $S(M)$

For every sequential algorithm A, the finiteness of the exploration witness in the third postulate allows us (see [19] to write a finite `ASM` program Π_A, which has the same set of updates than A for every state. Every program Π_A obtained in this way has the same form, which we call the **normal form**:

```
par if $F_1$ then $\Pi_1$
  || if $F_2$ then $\Pi_2$
       ⋮
  || if $F_c$ then $\Pi_c$
endpar
```

where F_i are "guards", which means that for every state X one and only one F_i is *true*, and the programs Π_i have the form

$$\texttt{par } u_1 \| \ldots \| u_{m_i} \texttt{ endpar}$$

where $u_1, \ldots, u_{m_i}$ are update commands.

Remark 6 $\Delta(\Pi_A, X) = \Delta(A, X) = \tau_A(X) \ominus X$*, so* $\Delta(\Pi_A, X)$ *is consistent without trivial updates.*

Theorem 3

$$\texttt{Algo} = \texttt{ASM}$$

The proof [19] that the set of sequential algorithms is identical to the set of ASMs uses mainly the fact that every ASM has a finite exploration witness. Reciprocally, for every sequential algorithm we can define an ASM with the same transition function. Thus, we get that the axiomatic presentation of sequential algorithms defines the same objects than the operational presentation of ASMs: every ASM is a sequential algorithm and every sequential algorithm can be simulated by an ASM in normal form. So, for every ASM there exists an equivalent ASM in normal form.

A.3 Axiomatic definition of BSP algorithms and ASM

Proposition 7 $\texttt{Algo}_{\texttt{bsp}} \simeq$ BSP-ASM

Proof We prove this proposition by double inclusion:

BSP-ASM $\subseteq \texttt{Algo}_{\texttt{bsp}}$ Let M be an BSP-ASM. M has an ASM program Π, a synchronization function $\mathbf{sync}_M$, a set of states $S(M)$ and a set of initial states $I(M)$ with the attended properties. We prove that M verifies the four postulates of the BSP algorithms :

1. M has states, initial states, and according to the operational semantics the ASM program Π and the synchronization function $\mathbf{sync}_M$ determine a transition function τ_M. Thus, M verifies the first postulate.
2. These states are $\mathbf{p}$-tuples of first-order structures containing the symbols **nproc** and **pid**. We assumed that these structures are closed by isomorphism. The transition function τ_Π preserves the universes of the local memories and the number $\mathbf{p}$ of processors. τ_M commutes with every isomorphism. Thus, M verifies the second postulate.
3. By using the operational semantics, we can prove that the set of terms $T(\Pi) = \mathit{Read}(\Pi) \cup \mathit{Write}(\Pi)$ is an exploration witness for M. Thus, M verifies the third postulate.
4. Let $\mathbf{comp}_M = \tau_\Pi$ restricted to 1-tuples. By using the operational semantics, we have:
$$\tau_M(X^1, \ldots, X^{\mathbf{p}}) = (\mathbf{comp}_M(X^1), \ldots, \mathbf{comp}_M(X^{\mathbf{p}}))$$
as long as there exists a $1 \leq i \leq \mathbf{p}$ such that $\mathbf{comp}_M(X^i) \neq X^i$, and we have:
$$\tau_M(X^1, \ldots, X^{\mathbf{p}}) = \mathbf{sync}_M(X^1, \ldots, X^{\mathbf{p}})$$
if for every $1 \leq i \leq \mathbf{p}$ we have $\mathbf{comp}_M(X^i) = X^i$. Thus, M verifies the fourth postulate.

$\texttt{Algo}_{\texttt{bsp}} \subseteq$ BSP-ASM Let A be a BSP algorithm. A has a set of states $S(A)$, a set of initial states $I(A)$, a transition function τ_A, a computation function $\mathbf{comp}_A$ and a synchronization function $\mathbf{sync}_A$ verifying the four postulates.
We prove the existence of a BSP-ASM program Π such that for every state $(X^1, \ldots, X^{\mathbf{p}}) \in S(A)$ and for every $1 \leq i \leq \mathbf{p}$ we have $\tau_\Pi(X^i) = \mathbf{comp}_A(X^i)$.
Let A' be the triplet defined by:

1. $S(A') \overset{\text{def}}{=} \bigcup\{X^i \mid (X^1, \ldots, X^{\mathbf{p}}) \in S(A)$ and $1 \leq i \leq \mathbf{p}$ and there exists $1 \leq j \leq \mathbf{p}$ such that $\mathbf{comp}_A(X^j) \neq X^j\} \subseteq M(A)$
2. $I(A') \overset{\text{def}}{=} \bigcup\{X^i \mid (X^1, \ldots, X^{\mathbf{p}}) \in I(A)$ and $1 \leq i \leq \mathbf{p}$ and there exists $1 \leq j \leq \mathbf{p}$ such that $\mathbf{comp}_A(X^j) \neq X^j\}$
3. $\tau_{A'}(X^i) \overset{\text{def}}{=} \mathbf{comp}_A(X^i)$

By definition, A' verifies the first postulate of Gurevich.
Because $S(A)$, $I(A)$ and τ_A verify the BSP postulates, $S(A')$, $I(A')$ and $\tau_{A'}$ verify the second postulate of Gurevich.
It remains to prove that $T(A)$ is an exploration witness for A'. According to the previous remark, we can assume[9] that **nproc** and **pid** are in $T(A)$.
Let X^i and Y^j two states of $S(A')$ which coincide over $T(A)$. Because **pid** is in $T(A)$ we have $i = j$. X^i comes from a state $(X^1, \ldots, X^p) \in S(A)$ and Y^i comes from a state $(Y^1, \ldots, Y^q) \in S(A)$. Because **nproc** is in $T(A)$ we have $p = q$.
By hypothesis on X^i there exists $1 \leq j \leq \mathbf{p}$ such that $\mathbf{comp}_A(X^j) \neq X^j$, and by hypothesis on Y^i there exists $1 \leq k \leq \mathbf{p}$ such that $\mathbf{comp}_A(Y^k) \neq Y^k$. So, by using the fourth BSP postulate :

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

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

Therefore $\Delta(A, X^i) = \mathbf{comp}_A(X^1) \ominus X^i = \Delta(A', X^i)$ and $\Delta(A, Y^i) = \mathbf{comp}_A(Y^1) \ominus X^i = \Delta(A', Y^i)$.
Moreover[10], because X^i and Y^i coincide over $T(A)$ we have that $\Delta(A, X^i) = \Delta(A, Y^i)$. So $\Delta(A', X^i) = \Delta(A, X^i) = \Delta(A, Y^i) = \Delta(A', Y^i)$. Therefore A' verify the third postulate of Gurevich.
A' verify the three postulates of Gurevich, so by using [19] there exists an ASM program Π such that for every state X^i we have $\tau_\Pi(X^i) = \tau_{A'}(X^i)$. Therefore, for every state X_i during the computation steps, we have $\mathbf{comp}_A(X_i) = \tau_\Pi(X_i)$.
During the synchronization steps, when for every $1 \leq i \leq \mathbf{p}$ we have $\mathbf{comp}_A(X^i) = X^i$, we only have to take $\mathbf{sync}_A$ as synchronization function.
Therefore, the BSP-ASM using the program Π and the synchronization function $\mathbf{sync}_A$ is equivalent to A.
Notice that the program Π obtained before is in normal form.

[9] Alternatively we could prove that $T(A') = T(A) \cup \{\mathbf{nproc}, \mathbf{pid}\}$ is the exploration witness, without the previous remark.

[10] To prove by using the third BSP postulate with the fact that the processor (respectively X^i or Y^i) depends only of its previous state and not the previous states of the other processors.