Formal Semantics of a Subset of the Paderborn's BSPlib

Frdric Gava
Laboratory of Algorithms, Complexity and Logic (LACL)
University of Paris-East
gava@univ-paris12.fr

Jean Fortin
jean.fortin@ens-lyon.org

Abstract

PUB (Paderborn University BSPLib) is a C library supporting the development of Bulk-Synchronous Parallel (BSP) algorithms. The BSP model allows an estimation of the execution time, avoids deadlocks and indeterminism. This paper presents a formal operational semantics for a C+PUB subset language using the Coq proof assistant and a certified N-body computation as example of using this formal semantics .

1. Introduction

Solving a problem on a parallel machine is often a complex job. High-level tools (models, languages, *etc.*) are necessary to simplify both the design of parallel algorithms and their programming but also to ensure a better safety of the generated applications. A classical solution is to provide semantics of a language for proving correctness of programs.

A more recent approach is to use a proof assistant (*e.g.* Coq [3]) for the development of the semantics [2] and then formally prove the properties of the language and correctness of programs[1]. The use of theorem-proving systems ensures better safety (trust in the generated softwares). Even if it is longer to formally prove parallel programs than to code them, the development of certified Dwarfs[2] [1] (and tools for this) is a first step to produce less buggy parallel applications: one could program using certified libraries and really trust the results of the procedures' calls (these libraries are less buggy than normal ones).

BSP[3] is a parallel model which offers a high degree of abstraction, allows an estimation of the execution time on a wide variety of architectures, avoids deadlocks and non-determinism. The Paderborn University BSPLib (PUB [6]) is a C library of BSP communication routines (and also a Web-Java implementation [5]) initially based on the BSPlib standard [16].

The aim of this paper is the Coq development of a formal operational semantics for both BSP message passing (BSMP) and remote memory accesses (DRMA) programming styles of a kernel imperative language with PUB primitives. Using this semantics, we certified a classical parallel numerical computation: the N-body problem.

First, we briefly describe the Coq proof assistant (Section 2) and the PUB library (Section 3). Then we give our kernel language in Section 4 and in Section 5 its natural semantics that has been developed using Coq. In Section 6, our example of correctness of a BSP programs demonstrates the usefulness of this formal semantics. We end with related works (Section 7), conclusion and future work (Section 8).

2. The Coq Proof Assistant

The Coq system[4] [3] is a proof assistant based on a logic which is a variant of type theory, following the "propositions-as-types, proofs-as-terms" paradigm, enriched with built-in support for inductive and co-inductive definitions of predicates and data types.

From a user's perspective, Coq offers a rich specification language to define problems and state theorems about them. This language includes

1. a constructive logic with all the usual connectives and quantifiers;

2. inductive definitions *via* inference rules and axioms;

3. a pure functional programming language with pattern-matching and structural recursion.

Proofs are developed interactively using tactics that build incrementally the proof term behind the scene. These tactics

[1] Some examples could be find in the users' contributions web site of the Coq proof assistant.

[2] Dwarfs are high-level abstractions of parallel numerical methods (as Linear Algebra, FFT, *etc.*), which each capture a pattern of computation and communication common to a class of important applications.

[3] We refer to [4, 16] for a gentle introduction to the BSP model.

[4] Available at `http://coq.inria.fr/` with nice introductions to this theorem-proving system.

range from the trivial (intro, which adds an abstraction to the proof term) to rather complex decision procedures (omega for Presburger arithmetic).

For example, let us define in Coq a language of numerical expressions $e := n \mid e_1 + e_2 \mid \frac{e_1}{e_2}$ and its natural semantics i.e. abstract evaluation to integers (noted **Z** in Coq):

```
Inductive expr: Set :=
  num: Z→expr
| plus: expr→expr→expr
| div: expr→expr→expr

Inductive eval_expr: expr → Z → Prop :=
  eval_num: ∀ n:Z, eval (num n) n
| eval_plus: ∀ e1 e2 n1 n2,
eval e1 n1→eval e2 n2→eval (plus e1 e2) (n1+n2)
| eval_div: ∀ e1 e2 n1 n2, eval e1 n1→
eval e2 n2→n2<>0→eval (div e1 e2) (n1/n2)
```

and we noted (eval_expr e n) the evaluation of expression e to integer n.

3. The Paderborn University BSPlib

3.1. Generalities

PUB is a C-Library of communication routines to support development of parallel algorithms based on the BSP model[5]. PUB offers functions for both message passing (BSMP) and remote memory access (DRMA). Some collective communication operations like parallel prefix are also provided, but they are not interesting for our purpose because they can easily be simulated by BSMP operations.

PUB has also a number of additional features. To become more flexible, PUB allows the creation of independent BSP objects: virtual BSP computerss (with their migration to adapt to changing load on the machines) and subset synchronisation. These extensions are not modelled here because they are too complex (too architecture dependant) and not portable. We will thus use only one group of processors which is the BSP computer.

We do not model in this paper high-performance operations (unbuffered communications and oblivious synchronisation) because they are unsafe (we research here safety) and need a more complicated semantics [12] which will only be used to prove safe optimisations.

As in the standard MPI, we first need to initialise our parallel computation which is done using the function bsplib_init[6]. Now, we can query some informations about the machine: bsp_nprocs returns the number of processors p and bsp_pid returns the own processor id in the range $0, \ldots, p-1$. To terminate a BSP computation, we use bsplib_done which cleans up all PUB resources.

3.2. Message Passing and Synchronisation

According to the BSP model all messages are received during the barrier of synchronisation and cannot be read before. Barrier is done using bsp_sync which blocks the current super-step until all other nodes have called bsp_sync and all messages sent to this node in the current super-step have been received.

Sending a single message is done using **void** bsp_send (**int** dest, **void**$*$ buffer, **int** s) where buffer is a pointer to a memory address to send to processor id dest and s size of this block. After calling this function the buffer can be reused and may be overwritten or freed.

In the next super-step, each processor can access the received messages. This can be done using t_bspmsg$*$ bsp_findmsg (**int** proc_id, **int** index) where proc_id is the id of the source-node and index is the index of the message (numbered from 0 to $n-1$; if $index > n-1$ then bsp_findmsg returns NULL). t_bspmsg is the type of a message in the PUB. To access to the sending message t_bspmsg, we need bspmsg_data which returns a pointer to the sending block of data and bspmsg_size which returns its size. Also bsp_nmsgs returns the number of messages received in the last super-step.

Note that the messages of the last super-step are available until the next synchronisation call. At this point the memory used for these messages will be deallocated. So, if you want to use these messages later you have to copy them.

3.3. Remote Memory Access

Another way of communication is remote memory access: after every processor has registered a variable for direct access, all processors can read or write the value on other processors. Registering a variable or deleting it from global access is done using: **void** bsp_push_reg (t_bsp$*$ bsp, **void**$*$ ident, **int** size) and **void** bsp_pop_reg(t_bsp$*$ bsp, **void**$*$ ident) where ident is the local address of the variable and size its size (it can be different on each node). The PUB forces that if different variables have to be registered then all processors have to call the bsp_push_reg functions in the same order (same for bsp_pop_reg). DRMA operations are:

- **void** bsp_get (t_bsp$*$ bsp,**int** srcPID,**void**$*$ src,**int** offset, **void**$*$ dest, **int** nbytes) (global reading access) and

- **void** bsp_put (t_bsp$*$ bsp,**int** destPID,**void**$*$ src,**void**$*$ dest, **int** offset, **int** nbytes) (global writing access).

[5] We briefly recall that execution of a BSP program is divided into super-steps, each separated by a global synchronisation; a super-step consists of each processor doing some calculations on local data and communicating some data to other processors; the collective barrier of synchronisation event guarantee that all communications of data have completed before the commencement of the next super-step.

[6] We refer to the manual, http://wwwcs.uni-paderborn.de /~pub/documentation.html for C type and more details about other functions of the PUB.

bsp_get copies nbytes bytes from the variable src (with offset offset) on processor srcPID to the local memory address dest. bsp_put copies nbytes bytes from local memory src to variable dest (with offset offset) on node destPID. All get and put operations are executed during the synchronisation and all get operations are served before a put overwrites a value.

4. BSP Core Language

Our core language is the classical IMP with a set Exp of expressions (booleans, integers, matrix, *etc.*) with operations on them. Set X of variables is a subset of Exp with two special variables: $\mathbf{pid}$ and $\mathbf{nprocs}$. Our language is as follows (sequential control flow commands):

c	$::=$	$\mathbf{skip}$	Null command
	$\mid$	$x := e$	Assignment
	$\mid$	$c_1; c_2$	Sequence
	$\mid$	$\mathbf{if}\, e\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\, \mathbf{endif}$	Conditional
	$\mid$	$\mathbf{while}\, e\, \mathbf{do}\, c\, \mathbf{done}$	Iteration
	$\mid$	$\mathbf{declare}\, y := e\, \mathbf{begin}\, c\, \mathbf{end}$	New variable

with $x, y \in X$ and $e \in Exp$. Expressions are evaluated to values v (subset of Exp) and we write: $\mathcal{E}_i, \mathcal{R}_i \stackrel{i,p}{\models} e \Downarrow v$ with p the number of processors and i the pid. $\mathcal{E}_i$ is the store (memory as a mapping from variables to values) of processor i and $\mathcal{R}_i$ is the set of received values. We suppose that $\mathcal{E}_i, \mathcal{R}_i \stackrel{i,p}{\models} \mathbf{pid} \Downarrow i$ and $\mathcal{E}_i, \mathcal{R}_i \stackrel{i,p}{\models} \mathbf{nprocs} \Downarrow p$. Evaluation of Exp is not total (ex. evaluation of $1 + \mathbf{true}$) but for simplicity always terminates. Parallel operations are:

$\mid$	$\mathbf{sync}$	Barrier of synchronisation
$\mid$	$\mathbf{push}(x)$	Registers a variable x for global access
$\mid$	$\mathbf{pop}(x)$	Delete x from global access
$\mid$	$\mathbf{put}(e, x, y)$	Distant writing of x to y of processor e
$\mid$	$\mathbf{get}(e, x, y)$	Distant reading from x to y
$\mid$	$\mathbf{send}(x, e)$	Sending value of x to processor e

In contrast to the PUB, we use basic values instead of arbitrary buffer addresses (**char** $*$). Exp is extended with $\mathbf{findmsg}(i, e)$ that finds the eth message of processor i of the previous super-step and $\mathbf{nmsg}$ that returns the arity of $\mathcal{R}_i$ (i.e. number of received values).

5. Natural Semantics

A big-step (also call natural) semantics makes it easier to prove properties and more closely models an actual recursive abstract interpreter (proof assistant). Programs are evaluated using inference rules for building finite or infinite (rational) trees. An originality of this paper is that our semantics and proofs were done using the Coq proof assistant. We give the semantics in an human reading format but the full Coq development can be downloaded at the authors's web pages. In the following, we will write inductive rules $\frac{P}{c}$ and co-inductive ones $\begin{array}{c}P\\ \hline\hline c\end{array}$ (infinite but rational derivation trees [14]).

We recall that we do not treat high-performance routines in this semantics because they are non-deterministic, unsafe and because programs are first written using classical operations and then optimised (by hand or by the compiler): the main goal of this semantics is to prove the correctness of these programs, not to optimise them.

Our semantics is a set of inference rules. We note $\mathcal{E}[x/v]$ insertion or substitution in $\mathcal{E}$ of a new binding from x to v. We note $\mathcal{R}$ the received values of the previous super-step and $\mathcal{C}$ communications that need to be done in the current super-step. Finite evaluations are noted $\Downarrow$ and infinite ones are noted $\Downarrow^{\infty}$ (this has to be read as "program diverges").

5.1. Local Reductions of the Semantics

We note $\Downarrow_l$ for local reductions (e.g. one at each processor). Local final configurations are noted $\langle \mathcal{E}', \mathcal{C}', \mathcal{R}', \mathbf{skip} \rangle$ but we also have intermediate results due to synchronisations, that is when processors finish a super-step ($\mathbf{sync}$). We have thus to memorise the next instructions of each processor. This intermediate local configuration is noted $\langle \mathcal{E}', \mathcal{C}', \mathcal{R}', \mathbf{SYNC}(c) \rangle$ where c is the sequence of next instructions for the next super-steps. Figures 1 and 2 give finite and infinite rules of the local control flow. Figure 5 gives the rules of PUB routines.

We note $\overline{x}$ a variable that has been registered for global access (DRMA), $\underline{x}$ for the contrary and x when that is not important. Messages (communications) are used with $\mathbf{get}$, $\mathbf{put}$ or $\mathbf{send}$ with their natural arguments.

Note that to simplify our semantics and make it readable, we introduce two minor modifications to the specification from the PUB's documentation. First, we do not require that different variables have to be registered in the same order on each processor[7]. Second, to not have a confusion between new variables and those that have been registered before, one could not declare variables that have been created before[8].

5.2. Global Reductions and Communications

PUB programs are SPMD ones so a program c is started p times. We model this as a p-vector of c with its environments of execution that is store $\mathcal{E}$, communications $\mathcal{C}$ and

[7] To respect the documentation of the PUB, we can count on each processor the registering of variables and compare them at each barrier.

[8] We can introduce a dynamic change of variables's name but that is a tedious work.

$$\frac{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1 \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{SYNC}(c)\rangle}{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1; c_2 \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{SYNC}(c;c_2)\rangle}$$

$$\frac{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1 \Downarrow_l \langle \mathcal{E}_1,\mathcal{C}_1,\mathcal{R}_1,\mathbf{skip}\rangle \qquad \mathcal{E}_1,\mathcal{C}_1,\mathcal{R}_1\overset{i,p}{\models} c_2 \Downarrow_l \langle \mathcal{E}_2,\mathcal{C}_2,\mathcal{R}_2,\mathbf{Flow}\rangle}{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1; c_2 \Downarrow_l \langle \mathcal{E}_2,\mathcal{C}_2,\mathcal{R}_2,\mathbf{Flow}\rangle}$$

$$\frac{}{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{skip} \Downarrow_l \langle \mathcal{E},\mathcal{C},\mathcal{R},\mathbf{skip}\rangle}$$

$$\frac{\mathcal{E},\mathcal{R}\overset{i,p}{\models} e\Downarrow v \quad x\in\mathcal{E}}{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} x := e \Downarrow_l \langle \mathcal{E}[x/v],\mathcal{C},\mathcal{R},\mathbf{skip}\rangle}$$

$$\frac{\mathcal{E},\mathcal{R}\overset{i,p}{\models} e\Downarrow\mathbf{true} \qquad \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1 \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{Flow}\rangle}{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{if}\, e\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\, \mathbf{endif} \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{Flow}\rangle}$$

$$\frac{\mathcal{E},\mathcal{R}\overset{i,p}{\models} e\Downarrow\mathbf{false} \qquad \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_2 \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{Flow}\rangle}{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{if}\, e\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\, \mathbf{endif} \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{Flow}\rangle}$$

$$\frac{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{if}\, e\, \mathbf{then}\, (c_1; \mathbf{while}\, e\, \mathbf{do}\, c_1\, \mathbf{done})\, \mathbf{else}\, \mathbf{skip}\, \mathbf{endif} \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{Flow}\rangle}{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{while}\, e\, \mathbf{do}\, c_1\, \mathbf{done} \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{Flow}\rangle}$$

$$\frac{\mathcal{E},\mathcal{R}\overset{i,p}{\models} e\Downarrow v \text{ and } x\notin\mathcal{E} \qquad \mathcal{E}[\underline{x}/v],\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1 \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{Flow}\rangle}{\mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{declare}\, x := e\, \mathbf{begin}\, c_1\, \mathbf{end} \Downarrow_l \langle \mathcal{E}',\mathcal{C}',\mathcal{R}',\mathbf{Flow}\rangle}$$

where $\mathbf{Flow} = \mathbf{skip}$ or $\mathbf{SYNC}(c)$

Figure 1. Inductive rules for sequential control flow

$$\begin{array}{c} \mathcal{E},\mathcal{C},\mathcal{R},\overset{i,p}{\models} c_1 \Downarrow_l^\infty \\ \hline\hline \mathcal{E},\mathcal{C},\mathcal{R},\overset{i,p}{\models} c_1; c_2 \Downarrow_l^\infty \end{array}$$

$$\begin{array}{c} \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1 \Downarrow_l \langle \mathcal{E}_1,\mathcal{C}_1,\mathcal{R}_1,\mathbf{skip}\rangle \qquad \mathcal{E}_1,\mathcal{C}_1,\mathcal{R}_1\overset{i,p}{\models} c_2 \Downarrow_l^\infty \\ \hline\hline \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1; c_2 \Downarrow_l^\infty \end{array}$$

$$\begin{array}{c} \mathcal{E},\mathcal{R}\overset{i,p}{\models} e\Downarrow\mathbf{true} \qquad \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_1 \Downarrow_l^\infty \\ \hline\hline \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{if}\, e\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\, \mathbf{endif} \Downarrow_l^\infty \end{array} \qquad \begin{array}{c} \mathcal{E},\mathcal{R}\overset{i,p}{\models} e\Downarrow\mathbf{false} \qquad \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} c_2 \Downarrow_l^\infty \\ \hline\hline \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{if}\, e\, \mathbf{then}\, c_1\, \mathbf{else}\, c_2\, \mathbf{endif} \Downarrow_l^\infty \end{array}$$

$$\begin{array}{c} \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{if}\, e\, \mathbf{then}\, (c; \mathbf{while}\, e\, \mathbf{do}\, c\, \mathbf{done})\, \mathbf{else}\, \mathbf{skip}\, \mathbf{endif} \Downarrow_l^\infty \\ \hline\hline \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{while}\, e\, \mathbf{do}\, c\, \mathbf{done} \Downarrow_l^\infty \end{array}$$

$$\begin{array}{c} \mathcal{E},\mathcal{R}\overset{i,p}{\models} e\Downarrow v \text{ and } x\notin\mathcal{E} \qquad \mathcal{E}[\underline{x}/v]\overset{i,p}{\models} c \Downarrow_l^\infty \\ \hline\hline \mathcal{E},\mathcal{C},\mathcal{R}\overset{i,p}{\models} \mathbf{declare}\, x := e\, \mathbf{begin}\, c\, \mathbf{end} \Downarrow_l^\infty \end{array}$$

Figure 2. Co-inductive (infinite) rules of the control flow

received values $\mathcal{R}$. A final configuration is $\mathbf{skip}$ on all processors. We note the full evaluation:

$$\langle\!\langle \mathcal{E}_0,\mathcal{C}_0,\mathcal{R}_0 \overset{i,p}{\models} c_0 \| \cdots \| \mathcal{E}_{p-1},\mathcal{C}_{p-1},\mathcal{R}_{p-1} \overset{i,p}{\models} c_{p-1} \rangle\!\rangle$$
$$\Downarrow_g$$
$$\langle\!\langle \mathcal{E}'_0,\mathcal{C}'_0,\mathcal{R}'_0,\mathbf{skip} \| \cdots \| \mathcal{E}'_{p-1},\mathcal{C}'_{p-1},\mathcal{R}'_{p-1},\mathbf{skip} \rangle\!\rangle$$

where $\Downarrow_g$ is for global (parallel) reductions. The reductions $\Downarrow_g$ call the local (sequential) ones $\Downarrow_l$ with the two rules defined in Figure 3 that is each processor computes a final configuration or there is at least one processor that diverges.

Communications are modelled with the two inference rules of Figure 4 that is if each processor is in the $\mathbf{sync}$ case, communications are done (using the $Comm$ function that model exchanges of messages) and the current super-step is finished, then the program continues (and returns a set of environments) or diverges.

The $Comm$ function specifies the order of the messages during the communications. It modifies the environment of each processor i such that $Comm(\mathcal{C}'_i,\mathcal{R}'_i,\mathcal{E}'_i) = (\mathcal{C}''_i,\mathcal{R}''_i,\mathcal{E}''_i)$ is for BSMP as follow:

$$\mathcal{C}''_i = \emptyset$$

$$\mathcal{R}''_i = \bigcup_{j=0}^{p-1} \bigcup_{n=0}^{n_j} \{j, n + \sum_{a=0}^{j} n_a, v\} \text{ if } \{\mathbf{send}, i, v\} \in_n \mathcal{C}'_j$$

$$\frac{\forall i \quad \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i \overset{i,p}{\models} c_i \Downarrow_l \langle \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, \mathbf{skip} \rangle}{\langle\!\langle \mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0 \overset{i,p}{\models} c_0 \| \cdots \| \mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1} \rangle\!\rangle \overset{i,p}{\models} c_{p-1} \rangle\!\rangle \Downarrow_g \langle\!\langle \mathcal{E}'_0, \mathcal{C}'_0, \mathcal{R}'_0, \mathbf{skip} \| \cdots \| \mathcal{E}'_{p-1}, \mathcal{C}'_{p-1}, \mathcal{R}'_{p-1}, \mathbf{skip} \rangle\!\rangle}$$

$$\begin{array}{c} \exists i \quad \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i \overset{i,p}{\models} c_i \Downarrow_l^{\infty} \\ \hline\hline \langle\!\langle \mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0 \overset{i,p}{\models} c_0 \| \cdots \| \mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1} \rangle\!\rangle \overset{i,p}{\models} c_{p-1} \rangle\!\rangle \Downarrow_g^{\infty} \end{array}$$

Figure 3. Global rules call local ones

$$\frac{\forall i \quad \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i \overset{i,p}{\models} c_i \Downarrow_l \langle \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, \mathbf{SYNC}(c'_i) \rangle \qquad \langle\!\langle \cdots \| \text{Comm}(\mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i) \overset{i,p}{\models} c'_i \| \cdots \rangle\!\rangle \Downarrow_g \langle\!\langle \cdots \| \mathcal{E}''_i, \mathcal{C}''_i, \mathcal{R}''_i, \mathbf{skip} \| \cdots \rangle\!\rangle}{\langle\!\langle \cdots \| \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i \overset{i,p}{\models} c_i \| \cdots \rangle\!\rangle \Downarrow_g \langle\!\langle \cdots \| \mathcal{E}''_i, \mathcal{C}''_i, \mathcal{R}''_i, \mathbf{skip} \| \cdots \rangle\!\rangle}$$

$$\begin{array}{c} \forall i \quad \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i \overset{i,p}{\models} c_i \Downarrow_l \langle \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, \mathbf{SYNC}(c'_i) \rangle \qquad \langle\!\langle \cdots \| \text{Comm}(\mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i) \overset{i,p}{\models} c'_i \| \cdots \rangle\!\rangle \Downarrow_g^{\infty} \\ \hline\hline \langle\!\langle \cdots \| \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i \overset{i,p}{\models} c_i \| \cdots \rangle\!\rangle \Downarrow_g^{\infty} \end{array}$$

Figure 4. Communications rules for both finite and finite reductions

$$\frac{}{\mathcal{E}, \mathcal{C}, \mathcal{R} \overset{i,p}{\models} \mathbf{sync} \Downarrow_l \langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{SYNC}(\mathbf{skip}) \rangle}$$

$$\frac{\text{if } \{\underline{x} \mapsto v\} \in \mathcal{E} \text{ with } \mathcal{E}' = \mathcal{E}[\overline{x}/v]}{\mathcal{E}, \mathcal{C}, \mathcal{R} \overset{i,p}{\models} \mathbf{push}(x) \Downarrow_l \langle \mathcal{E}', \mathcal{C}, \mathcal{R}, \mathbf{skip} \rangle} \qquad \frac{\text{if } \{\overline{x} \mapsto v\} \in \mathcal{E} \text{ with } \mathcal{E}' = \mathcal{E}[\underline{x}/v]}{\mathcal{E}, \mathcal{C}, \mathcal{R} \overset{i,p}{\models} \mathbf{pop}(x) \Downarrow_l \langle \mathcal{E}', \mathcal{C}, \mathcal{R}, \mathbf{skip} \rangle}$$

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ and } \{\overline{y} \mapsto v'\} \in \mathcal{E} \text{ with } \mathcal{C}' = \mathcal{C} \cup \{\mathbf{put}, pid, \overline{y}, v\} \quad \text{and } 0 \le pid < p}{\mathcal{E}, \mathcal{C}, \mathcal{R} \overset{i,p}{\models} \mathbf{put}(e, x, y) \Downarrow_l \langle \mathcal{E}, \mathcal{C}', \mathcal{R}, \mathbf{skip} \rangle}$$

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ and } \{\overline{y} \mapsto v'\} \in \mathcal{E} \text{ with } \mathcal{C}' = \mathcal{C} \cup \{\mathbf{get}, pid, x, \overline{y}\} \quad \text{and } 0 \le pid < p}{\mathcal{E}, \mathcal{C}, \mathcal{R} \overset{i,p}{\models} \mathbf{get}(e, x, y) \Downarrow_l \langle \mathcal{E}, \mathcal{C}', \mathcal{R}, \mathbf{skip} \rangle}$$

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ with } \mathcal{C}' = \mathcal{C} \cup \{\mathbf{send}, pid, v\} \quad \text{and } 0 \le pid < p}{\mathcal{E}, \mathcal{C}, \mathcal{R} \overset{i,p}{\models} \mathbf{send}(x, e) \Downarrow_l \langle \mathcal{E}, \mathcal{C}', \mathcal{R}, \mathbf{skip} \rangle}$$

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e_1 \Downarrow pid \quad \mathcal{E}, \mathcal{R} \overset{i,p}{\models} e_2 \Downarrow n \quad \{pid, n, v\} \in \mathcal{R} \quad \text{and } 0 \le pid < p}{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} \mathbf{findmsg}(e_1, e_2) \Downarrow v}$$

$$\frac{n = |\mathcal{R}|}{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} \mathbf{nmsg} \Downarrow n}$$

Figure 5. Natural semantics of the PUB's routines

that is we suppose that each processor j has sent n_j messages to i and thus we take the nth message (noted $\in_n$) from this ordering set. DRMA accesses are defined as follows:

$$\mathcal{E}''_i = \mathcal{E}'_i \left[\bigcup_{j=0}^{p-1} [\overline{y}/v] \bigcup_{j=0}^{p-1} [\overline{y'}/v'] \text{ if } \begin{cases} \{\overline{y} \mapsto v\} \in \mathcal{E}'_j \text{ and } \{\mathbf{get}, j, x, \overline{y}\} \in \mathcal{C}'_i \\ \{\overline{y'} \mapsto v\} \in \mathcal{E}'_i \text{ and } \{\mathbf{put}, i, \overline{y'}, v'\} \in \mathcal{C}'_j \end{cases} \right]$$

That is, first, **get** accesses with the natural order of processors are done (list of substitutions) and then **put** accesses finish the communications (same natural order). Programs that neither evaluate nor diverge according to the rules above are said to "go wrong".

5.3. Coq Development and Lemmas

As written above, the semantics was developed using the Coq proof assistant. We give here some intuitions of this development. We note instr for a list of instructions envnmt for environments (store $\mathcal{E}$, communications $\mathcal{C}$ and received values $\mathcal{R}$).

Local reductions $\Downarrow_l$ are modelled with the inductive nat_sem_l as follows:

```
Inductive nat_sem_l (i:Z): envnmt → instr →
envnmt → ef → Prop :=
(* case x:=e*)
  nat_sem_assign: ∀ x e n env env',
  eval i env e n→update env x n env'
  → nat_sem_l i env (assign x e) env' eskip

(* case sync;c *)
| nat_sem_sync_c: ∀ env c, nat_sem_l i env
                  (sequence sync c) env (esync c)

(* case put *)
| nat_sem_put: ∀ e pid x y env env',
                    eval i env e pid
```

```
        → updateput env pid x y env'
        → nat_sem_l i env (put e x y) env' eskip

(* case send *)
| nat_sem_send: ∀ e pid x env env' v,
    eval i env e pid→lookup (fst env) x v
    → updatesend env v pid env'
    → nat_sem_l i env (send x e) env' eskip

(* Other cases *)
| ...
```

where ef is **Flow** *i.e.* eskip for **skip** or esync for **SYNC**(c). eval is the evaluation of the expressions $\overset{i,p}{\models}$. update and assign are functions for manipulating environments and lookup for searching a value binding by a variable.

p-vectors are represented as functions from **Z** (Coq's integer) to instructions or environments. The two rules of global reduction $\Downarrow_g$ (doing local calculations and communications) are modelled with the inductive nat_sem as follows:

```
Inductive nat_sem: (Z→envnmt)→(Z→instr)
                    →(Z→env)→Prop :=
(* Global rule when local reductions give skip *)
  nat_sem_par_skip:∀env instrs env',
   (∀ pid, (0<=pid<p)
        → (nat_sem_l pid (env pid) (instrs pid)
                              (env' pid) eskip))
     → nat_sem env instrs env'

(* Communication rule *)
| nat_sem_par_comm: ∀env instrs env' env_com env'',
  (∀ pid, (0<=pid<p)→
          (nat_sem_l pid (env pid) (instrs pid)
                     (env' pid) (esync (c pid))))
  → comm_g e' env_com
  → nat_sem env_com c env''
  → nat_sem env intrs env''
```

where at processor pid there is a local evaluation nat_sem_l. comm_g does the global communication. Co-inductive rules are model in the same manner.

It is easy to prove the following lemmas by (co)-induction on the derivations of programs:

Lemma 1 *$\Downarrow_g$ is deterministic.*

Lemma 2 *$\Downarrow_g$ and $\Downarrow_g^\infty$ are mutually exclusive.*

As examples of the co-inductive rules, we have proved using Coq that these programs diverge:

```
while $true$ do
  sync;
done
```

```
declare $x := 0$ begin
  declare $y := 1$ begin
    push($x$); push($y$);
    while $x <> y$ do
      get($x, y$, pid $+ 1$);
      get($y, x$, pid $- 1$);
      sync;
    done
end end
```

6. Certified N-body Computation

6.1. The N-body Problem

The classic N-body problem is to calculate the gravitational energy of N point masses:

$$E = -\sum_{\substack{i=1\\ i\neq j}}^{N}\sum_{j=1}^{N}\frac{m_i \times m_j}{r_i - r_j}$$

To compute this sum, we show a classical parallel algorithm using a systolic loop. At the beginning, each processor contains a sub-part as a list of the N point masses in its own memory. Initially, each processor calculates the interactions among its point masses. Then it sends a copy of its particles to its right-hand neighbour, while at the same time receiving the particles from its left-hand neighbour. It calculates the interactions between its own particles and those that just came in, and then it sends a copy of its particles to its right-right-hand neighbour *etc.* After $\mathbf{p} - 1$ super-steps, all pairs of particles have been treated and a parallel folding of these values can be done to finish the computation.

6.2. Certified Method

Figure 6 gives the BSP code in our core language of the direct prefixes computation (each processor sends its value to all of its right-hand neighbours and computes the final result with its received values) and the above method. We suppose a function $pair\_energy$ that computes the local interactions. This pure sequential function can be easily programmed in IMP and is thus not important for the purpose of this paper which is to formalise BSP communications in the Coq proof assistant.

For the parallel prefixes, we suppose that each processor binds a value in variable x and for the n-body that each processor binds a list of particles in $my\_particles$.

To certify this method using our semantics, we need to give the abstract intermediate environments, *i.e.* partial sums of the x and partial applications of $pair\_energy$. We also need to prove that our systolic loop just applies p times the $pair\_energy$ function on received data and then prove that this computes the intended result.

Hundreds of applications of Coq's tactics are needed to prove that environments are those intended. All this work makes the proof of this program very tedious compared to just writing it. But, to our knowledge, no work like ours have been done before and designing new tactics would certainly decrease the number of proof lines.

7. Related Works

Simplicity (yet efficiency) of the BSP model allows to prove properties and correctness of BSP programs. Different approaches for proofs of BSP programs have thus

Parallel direct prefixes:

```
declare $y := \mathbf{pid} + 1$
 begin
  while ($y <$ nprocs) do
    send($x, y$);
    $y := y + 1$;
  done
  sync;
  $y := 0$;
  while ($y <$ pid) do
    $x := x +$ findmsg($y, 0$);
    $y := y + 1$
  done;
 end
```

N-body computation:

```
declare $buffer := my\_particles$ begin
declare $energy := 0$ begin
declare $y := 0$ begin
  push($my\_particles$);
  while ($y <$ nprocs $- 1$) do
    $energy := energy + pair\_energy(buffer, my\_particles)$;
    $y := y + 1$;
    get(($y +$ pid) $mod$ nprocs, $buffer, my\_particles$);
    sync;
  done;
  $energy := energy + pair\_energy(buffer, my\_particles)$;
  $Code\_of\_prefixe\_for(energy)$;
  end
  end
```

Figure 6. Code of the direct folding and of the N-body computation

been studied such as BSP functional programming using Coq [11], the derivation of BSP imperative programs using Hoare's axiom semantics [7, 13, 17], "Refinement calculus" [15] or global state transformations [18].

The main drawback of these approaches is that they use their own languages that are in general not a subset of real programming languages. Also they neither used any proof assistant (except [11]) nor implemented dedicated tools for the proofs which is a lack of safety: users make hand proofs so they are just theoretical works. Our work simplifies and extends for BSMP routines and diverging programs the BSPLib small-steps semantics of [19]. Also, our Coq development ensures safety and allows us to certify programs that is formally prove the correctness of programs.

8. Conclusion and Future Work

In this paper, we have presented a formal deterministic operational semantics for BSP programs and used it to prove the correctness of a classical numerical computation (the N-body problem which is considered as an important Dwarf [1]) and the divergence of some programs. An originality of this paper is that all results were proved using a proof assistant (the Coq system) which ensures a better trust in the results.

The authors know that proving correctness of BSP computations only using this semantics is a too tedious work. But, it is intended to be the basis of better tools for the proof of BSP programs. We are thinking about extending the theoretical work of [9] and its C application software [10] which generates lemmas to be proved (using a proof assistant) from Hoare's assertions in C programs that ensure correctness (using a formal semantics).

We are also working on a semantics that makes appear high-performances features of the PUB [12]. This second semantics would be used to create a certified software for optimisation (a certified version of [8]): transforming buffered operations to unbuffered ones and BSP synchronisations to oblivious ones. The semantics would help to prove the equivalence of classical BSP programs transforming to high-performance ones.

The main goal of this work is an environment where programmers could prove correctness of their BSP programs and at the end automatically get high-performance versions in a certified manner. Adapting it to MPI programs would be a great challenge.

References

[1] K. Asanovic, R. Bodik, B. C. Catanzaro, J. J. Gebis, P. Husbands, K. Keutzer, D. A. Patterson, W. L. Plishker, J. Shalf, S. W. Williams, and K. A. Yelick. The Landscape of Parallel Computing Research: A View from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, 2006.

[2] B. E. Aydemir, A. Bohannon, N. Foster, B. Pierce, D. Vytiniotis, G. Washburn, S. Weirich, S. Zdancewic, M. Fairbairn, and P. Sewell. The poplmark challenge, 2005. `http://fling-l.seas.upenn.edu/plclub/ cgi-bin/poplmark/`.

[3] Y. Berthot and P. Castran. *Interactive Theorem Proving and Program Development — Coq'Art: The Calculus of Inductive Constructions*. Springer-Verlag, 2004.

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

[5] O. Bonorden, J. Gehweiler, and F. M. auf der Heide. A Web Computing Environment for Parallel Algorithms in Java.

Scalable Computing: Practice and Experience, 7(2):1–14, 2006.

[6] O. Bonorden, B. Juurlink, I. V. Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.

[7] Y. Chen and W. Sanders. Top-Down Design of Bulk-Synchronous Parallel Programs. *Parallel Processing Letters*, 13(3):389–400, 2003.

[8] A. Danalis, L. Pollock, and M. Swany. Automatic MPI application transformation with ASPhALT. In *Workshop on Performance Optimization for High-Level Languages and Libraries (POHLL 2007), in conjunction with IPDPS*, 2007.

[9] J.-C. Filliâtre. Verification of Non-Functional Programs using Interpretations in Type Theory. *Journal of Functional Programming*, 13(4):709–745, 2003.

[10] J.-C. Filliâtre and C. Marché. Multi-Prover Verification of C Programs. In *Sixth International Conference on Formal Engineering Methods (ICFEM)*, volume 3308 of *LNCS*, pages 15–29. Springer-Verlag, 2004. `http://why.lri.fr/caduceus/`.

[11] F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.

[12] F. Gava and J. Fortin. Two Formal Operational Semantics of the Paderborn University BSPlib using the Coq Proof Assistant. Technical Report 4, LACL,University of Paris-Est, 2008.

[13] H. Jifeng, Q. Miller, and L. Chen. Algebraic Laws for BSP Programming. In L. Bouge and Y. Robert, editors, *Euro-Par'96*, number 1124 in LNCS, pages 359–368. Springer, 1996.

[14] X. Leroy and H. Grall. Coinductive Big-step Operational Semantics. *Information and Computation*, 2008. to appear.

[15] D. B. Skillicorn. Building BSP Progams Using the Refinement Calculus . In *Formal Methods for Parallel Programming and Applications workshop at IPPS/SPDP'98*, 1998.

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

[17] A. Stewart and M. Clint. BSP-style Computation: a Semantic Investigation. *The Computer Journal*, 44(3):174–185, 2001.

[18] A. Stewart, M. Clint, and J. Gabarró. Axiomatic Frameworks for Developing BSP-Style Programs. *Parallel Algorithms and Applications*, 14:271–292, 2000.

[19] J. Tesson and F. Loulergue. Formal Semantics for the DRMA Programming Style Subset of the BSPlib Library. In J. Weglarz, R. Wyrzykowski, and B. Szymanski, editors, *Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM 2007)*, LNCS. Springer, 2007.