Two Formal Semantics of a Subset of the Paderborn University BSPlib

Frédéric Gava
Laboratory of Algorithms, Complexity and Logic (LACL)
University of Paris-East
Créteil-Paris, France
Email: gava@univ-paris12.fr

Jean Fortin
Laboratory of Algorithms, Complexity and Logic (LACL)
University of Paris-East
Créteil-Paris, France
Email: 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 non-determinism. This paper presents two formal operational semantics for a C+PUB subset language using the Coq proof assistant, one for classical BSP operations and one that emphasises high performance primitives.

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. To ensure the lack of comprehension of the implementations or to designe tools for proofs of correctness of programs, a classical step is to provide semantics of the language.

A recent approach is to use a proof assistant (*e.g.* Coq [1]) 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] [3] (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 for Java [7]) initially based on the BSPlib standard [5].

The aim of this paper is the Coq development of two formal operational semantics for both BSP message passing (BSMP) and remote memory accesses (DRMA) programming styles of a kernel imperative language with PUB primitives. The first semantics is dedicated to classical BSP operations. The second one is to enable high-performance operations. High-performance operations improve speedup of programs using unbuffered communications but must be used wisely due to their unsafe and non-derministic nature (i.e., a program that badly uses these routines can produce different outputs whenever applied to the same input). Determinacy is a property of ordinary sequential programming languages and is a powerful aid in debugging and validating programs even if it limits the programmer's flexibility to code certain algorithms. The semantics for high-performance operations would be the basis to produce semantics certified optimisations on communications such as those of [8] (but which are unproved).

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 a small-step semantics for classical BSP operations. In Section 6 we present an extension of this semantics for high-performance primitives. We end with related works (Section 7), conclusion and future work (Section 8).

2. The Coq Proof Assistant

The Coq system[4] [1] 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. This language includes (1) constructive logic with all the usual connectives and quantifiers; (2) inductive definitions *via* in-

1. Some examples could be found 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], [5] for a gentle introduction to the BSP model.
4. Available at http://coq.inria.fr/ with nice introductions to this theorem-proving system.

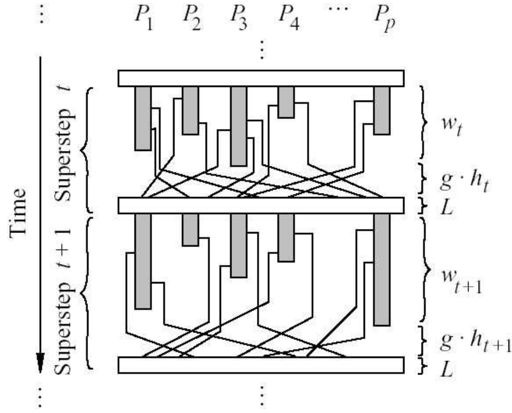

Figure 1. The BSP model of execution

ference rules and axioms; (3) a pure functional programming language with structural recursion.

Proofs are developed interactively using tactics that build incrementally the proof term behind the scene. These tactics 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 $e := n \mid e_1 + e_2$ (integers are noted $\mathbf{Z}$):

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

and its one step reduction:

```
Inductive one_step : expr → expr → Prop :=
| plus_left : ∀ e1 e'1 e2, one_step e1 e'1
           → one_step (plus e1 e2) (plus e'1 e2)
| plus_right : ∀ n1 e2 e'2, one_step e2 e'2
→ one_step (plus (num n1) e2) (plus (num n1) e'2)
| plus_sum : ∀ n1 n2,
   one_step (plus (num n1) (num n2)) (num (n1+n2))
```

The transitive and reflexive closure of one_step is:

```
Inductive step_star : expr → expr → Prop :=
 | sos_refl : ∀ e, step_star e e
 | sos_trans : ∀ e1 e2 e3, one_step e1 e2
            → step_star e2 e3 → step_star e1 e3
```

The small-step semantics i.e. abstract evaluation of expression e to integer n is noted `(step_star e (num n))`.

3. The PUB library

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 broadcast are also provided, but they are not interesting for our purpose because they can easily be simulated by primitive BSMP operations.

To become more flexible, PUB also has an additional feature which is the creation of independent BSP objects. That allows subset synchronisation and migration of BSP threads to adapt to changing load on real machines. This extension is not modelled here because too complex and too architecture dependant. We will thus use only one group of processors which is the BSP computer.

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 processor id. The processor id is 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 synchronisation barrier and cannot be read before. Barrier is done using bsp_sync which blocks the node until all other nodes have called bsp_sync and all messages sent to it 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 bytes of this block. After calling this routine the buffer may be overwritten or freed.

In the next super-step, each processor can access the received messages (type t_bspmsg). 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. To access to the message t_bspmsg, we need bspmsg_data which returns a pointer to the sending block of data and bspmsg_size 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.

3.3. Remote Memory Access

Another way of communication is through 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)

5. We briefly recall that execution of a BSP program is divided into super-steps (see left scheme in Fig. 1), 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 guarantees 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.

and **void** bsp_pop_reg(t_bsp* bsp, **void*** ident). The PUB needs 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).

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

3.4. High Performance Primitives

All communication primitives have their high-performance counterparts. The copies are done asynchronously and unbuffered, so it is finished after the next super-step and the buffer (src and dest) must not be changed before. The time when destination is written is undefined (architecture dependant).

The PUB also contains an oblivious synchronization **void** bsp_oblsync(t_bsp* bsp,**int** nmsgs) which should be used if the programmer knows the number nmsgs of messages a processor will receive in a superstep. This type of synchronization is much faster than the other one since no additional communication is needed and no barrier synchronization is done (but the user must know exactly how many messages every node will receive). Supersteps with standard synchronization can alternate with oblivious synchronizations, but within one superstep each processor has to use the same type of synchronization. The number of messages (nmsgs) that a processor waits in bsp_oblsync is:

- Each message which will be found in the message queue (sent with bsp_send, bsp_msgsend or bsp_hpsend);
- Each bsp_put (or bsp_hpput) produces "one message" at the destination. Each bsp_get (or bsp_hpget) produces "one message" at source and destination.

Why high-performance in BSP (PUB) ? Figure 2 shows the speedups for various versions of a program (the classic N-body problem[7]) that use only classical BSP operations, BSP with unbuffered communications (BSP+HP), oblivious synchronisations (BSP+OblSync) or all this optimisations (BSP+HP+OblSync). This program is based on a systolic loop algorithm [9]. In such an algorithm, data (point masses) is passed around from processor to processor in a sequence of super-steps where each processor computes the interactions of its point masses with received ones. It is no surprise to see that the use of high-performance operations results in improved speedup·

7. This computes the gravitational energy of N point masses, which is given by: $\sum_{\substack{i=1 \\ i \neq j}}^{N} \sum_{j=1}^{N} \frac{m_i m_j}{r_i - r_j}$ where m are masses and r coordinates.

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 evaluate to values v (subset of Exp) and we write: $\mathcal{E}_i, \mathcal{R}_i \models^{i,p} e \Downarrow v$ 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 $\mathcal{E}_i, \mathcal{R}_i \models^{i,p} \mathbf{pid} \Downarrow i$ and $\mathcal{E}_i, \mathcal{R}_i \models^{i,p} \mathbf{nprocs} \Downarrow p$. 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. Small-step Semantics

We give the semantics in an human reading format but the full Coq development can be downloaded at http://lacl.univ-paris12.fr/gava/pdp-coq.tar.

We recall that we do not handle 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 programs, not to optimise them.

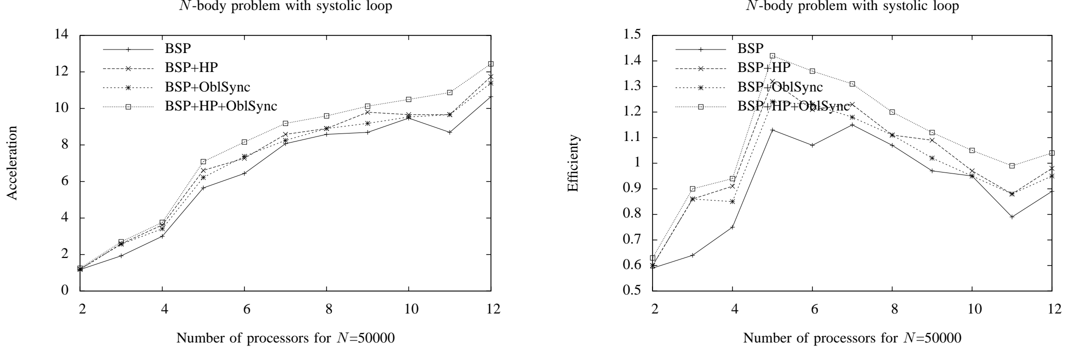

Figure 2. Performances of the N-body problem

5.1. Generalities

Small-step semantics specifies the operation of a program one step at a time. There is a set of rules that we continue to apply to configurations until reaching a final configuration if ever. In our parallel case, we will have two kinds of reductions: local ones (on each processor) and global ones (for the whole parallel machine).

Our semantics is thus a set of 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 from the previous super-step and $\mathcal{C}$ communications that need to be done in the current super-step. We also 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.

To simplify our semantics and make it readable, we introduce some minor modifications to the specification from the PUB documentation. First, we do not require that different variables have to be registered in the same order on each processor[8]. 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[9].

5.2. Local Rules

We note $\underset{i,p}{\rightharpoonup}$ for local reductions (e.g. one at each processor). Local final configurations are noted $\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, c_1 \rangle$.

Figure 3 gives rules of the local control flow and Figure 4 the rules of the PUB routines. Note that we ensure that there is always instructions after a **sync** (without introducing infinite reductions for this) and that $\mathbf{sync}; c$ and **skip** are local terminated states. The communications environment $\mathcal{C}$ now contains messages to be sent (noted with $\longleftarrow$) and asynchronous messages received from other processors (noted with $\longrightarrow$). Messages (communications) are used with **get**, **put** or **send** with their natural arguments (we note $\%$ for modulo calculus[10]).

$$\frac{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, c_1 \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}', \mathcal{C}', \mathcal{R}', c_1' \rangle \quad \text{if } c_1 \not\equiv \mathbf{sync}}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, c_1; c_2 \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}', \mathcal{C}', \mathcal{R}', c_1'; c_2 \rangle}$$

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

$$\frac{}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{skip}; c_2 \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}, \mathcal{R}, c_2 \rangle}$$

$$\frac{}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, (\mathbf{sync}; c_1); c_2 \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{sync}; (c_1; c_2) \rangle}$$

$$\frac{}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{sync} \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{sync}; \mathbf{skip} \rangle}$$

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

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

$$\frac{}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{while}\, e\, \mathbf{do}\, c\, \mathbf{done} \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{if}\, e\, \mathbf{then}\, c'\, \mathbf{else}\, \mathbf{skip}\, \mathbf{endif} \rangle}$$
$$\text{where } c' = c; \mathbf{while}\, e\, \mathbf{do}\, c\, \mathbf{done}$$

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e \Downarrow v \text{ and } x \notin \mathcal{E}}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{declare}\, x := e\, \mathbf{begin}\, c\, \mathbf{end} \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}[x/v], \mathcal{C}, \mathcal{R}, c \rangle}$$

Figure 3. Reduction of sequential control flow

8. To show regard for the PUB's documentation, we can count on each processor the registering of variables and compare them at each barrier.

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

10. Another way is to forbidden communications over $0 \cdots p-1$; we think that communications modulo p is much closer to a safe BSPlib implementation that is the PUB.

5.3. 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 contains $\mathcal{E}$, communications $\mathcal{C}$ and received values $\mathcal{R}$. A final configuration is **skip** on all processors: $\langle \mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0, \mathbf{skip} \| \cdots \| \mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1}, \mathbf{skip} \rangle$

The global reductions call the local ones with this rule:

$$\frac{\langle \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, c'_i \rangle}{\langle \cdots \| \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \rangle \rightharpoonup \langle \cdots \| \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, c'_i \| \cdots \rangle}$$

This represents a reduction by a single processor, which then introduces an interleaving of computations. Note that in the following rules, each c_i could be an empty set of instructions. Communications and BSP synchronisation are done with this rule:

$$\begin{array}{r} \langle \mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0, \mathbf{sync}; c_0 \| \cdots \| \mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1}, \mathbf{sync}; c_{p-1} \rangle \\ \rightharpoonup \langle Comm(\mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0), c_0 \| \cdots \| Comm(\mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1}), c_{p-1} \rangle \end{array}$$

That is, if all processors are in the **sync** case, communications are done (using the $Comm$ function that model exchanges of messages) and the current super-step is finished. 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 follows:

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

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

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 follow:

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

That is, first, **get** accesses with the natural order of processors are performed (list of substitutions) and then **put** accesses finish the communications (same natural order).

We note $\Rightarrow$ for a finite derivation and $\overset{\infty}{\Rightarrow}$ for an infinite one (this has to be read as "program diverges"). $\Rightarrow$ (resp. $\overset{\infty}{\Rightarrow}$) is defined by induction (resp. by co-induction i.e. infinite but rational derivation [10]) in Figure 5. Execution of a program is complete in the final configuration case or there exists a reduction step before having this final configuration or the program diverges. Programs that neither evaluate nor diverge according to the rules above are said to "go wrong".

$$\frac{\text{if } \{\underline{x} \mapsto v\} \in \mathcal{E} \text{ with } \mathcal{E}' = \mathcal{E} \oplus \{\overline{x} \mapsto v\}}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{push}(x) \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}', \mathcal{C}, \mathcal{R}, \mathbf{skip} \rangle}$$

$$\frac{\text{if } \{\overline{x} \mapsto v\} \in \mathcal{E} \text{ with } \mathcal{E}' = \mathcal{E} \oplus \{\underline{x} \mapsto v\}}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{pop}(x) \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}', \mathcal{C}, \mathcal{R}, \mathbf{skip} \rangle}$$

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ and } \{\overline{y} \mapsto v'\} \in \mathcal{E}}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{put}(e, x, y) \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}', \mathcal{R}, \mathbf{skip} \rangle}$$
where $\mathcal{C}' = \mathcal{C} \cup \{\mathbf{put}, pid\%p, \overline{y}, v, \leftarrow\}$

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ and } \{\overline{y} \mapsto v'\} \in \mathcal{E}}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{get}(e, x, y) \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}', \mathcal{R}, \mathbf{skip} \rangle}$$
where $\mathcal{C}' = \mathcal{C} \cup \{\mathbf{get}, pid\%p, x, \overline{y}, \leftarrow\}$

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ with } \mathcal{C}' = \mathcal{C} \cup \{\mathbf{send}, pid\%p, v, \leftarrow\}}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \mathbf{send}(x, e) \rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}', \mathcal{R}, \mathbf{skip} \rangle}$$

Figure 4. Reduction of the PUB's routines

5.4. Coq Development and Lemmas

As mentioned above, the semantics was developed using Coq. 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 $\mathcal{R}$). We also use `eq_envnmt` for equivalence of environments. p-vectors are represented as functions from **Z** to instructions or environments. The two rules of global reduction $\rightharpoonup$ (doing local calculations and communications) are modelled with the inductive `smallstep_sem` as follow:

```
Inductive smallstep_sem :(Z→envnmt)→(Z→instr)
          →(Z→envnmt)→(Z→instr)→Prop:=
| smallstep_sem_local : ∀
i env1 i1 env2 i2 env1i i1i env2i i2i ,
   smallstep_sem_l i env1i i1i env2i i2i →
   eq_envnmt env1i (env1 i) →
eq_envnmt env2i (env2 i) → i1 i=i1i→i2 i=i2i →
   (∀ n, n<>i → (eq_envnmt (env1 n) (env2 n)) ∧
(i1 n = i2 n)) →
   smallstep_sem env1 i1 env2 i2

| smallstep_sem_sync : ∀ env1 env2 i1 i2 ,
   comm_g env1 env2 →∀ i, 0<=i<p →
(i1 i = sequence sync (i2 i)) ∨ (i1 i = sync ∧
i2 i = skip) →smallstep_sem env1 i1 env2 i2.
```

Then, $\Rightarrow$ (resp. $\overset{\infty}{\Rightarrow}$) are defined by a inductive (resp. co-inductive) on this reduction.

The inductive `smallstep_sem_l` defines a local reduction, that is execution of a communication routine or execution of a sequential control flow. We give here an interesting part of its Coq's definition:

```
Inductive smallstep_sem_l (i:Z):envnmt→instr→envnmt
                              →instr→Prop:=
| ...
(* If then else semantics *)
| smallstep_sem_ifelse_true : ∀ e i1 i2 l1 l1',
```

$$\forall i\ \langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, \mathbf{skip} \| \cdots \rangle \Rightarrow \langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, \mathbf{skip} \| \cdots \rangle$$

$$\frac{\forall i\ \langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \rangle \rightharpoonup \langle \cdots \| n'_i, \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, c'_i \| \cdots \rangle \quad \langle \cdots \| n'_i, \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, c'_i \| \cdots \rangle \Rightarrow \langle \cdots \| n''_i, \mathcal{E}''_i, \mathcal{C}''_i, \mathcal{R}''_i, c''_i \| \cdots \rangle}{\langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \rangle \Rightarrow \langle \cdots \| n''_i, \mathcal{E}''_i, \mathcal{C}''_i, \mathcal{R}''_i, c''_i \| \cdots \rangle}$$

$$\frac{\forall i\ \langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \rangle \rightharpoonup \langle \cdots \| n'_i, \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, c'_i \| \cdots \rangle \quad \langle \cdots \| n'_i, \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, c'_i \| \cdots \rangle \stackrel{\infty}{\Rightarrow}}{\langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \rangle \stackrel{\infty}{\Rightarrow}}$$

Figure 5. Whole reduction of a PUB program

```
  b_eval i l1 e true → eq_envnmt l1 l1' →
  smallstep_sem_l i l1 (ifelse e i1 i2) l1' i1

(* put *)
| smallstep_sem_put : ∀ e pid x y l l',
    eval i l e (zvalue pid) → updateput l pid x y l' →
    smallstep_sem_l i l (put e x y) l' skip

(* send *)
| smallstep_sem_send : ∀ e pid x l l' v,
    eval i l e (zvalue pid) →
    lookup (fst l) x v  → updatesend l v pid l' →
    smallstep_sem_l i l (send x e) l' skip
| ...
```

We previously defined [11] (also in Coq) a natural semantics of the same PUB subset. This semantics has been proved (also using Coq) to be deterministic and we thus have the following lemmas by induction/co-induction:

Lemma 1: $\Rightarrow$ and $\Downarrow$ (resp. $\stackrel{\infty}{\Rightarrow}$ and $\Downarrow^{\infty}$) are equivalent.

Lemma 2: $\Rightarrow$ is deterministic.

Lemma 3: $\Rightarrow$ and $\stackrel{\infty}{\Rightarrow}$ are mutually exclusive.

6. Semantics for Hig-performances Operations

6.1. Generalities

In this semantics, we introduce high-performance features and thus, as specified in the PUB's documentation, we will have to keep track of the number of messages sent/received (noted n_i in an environment). The main property is that high-performance routines are non-deterministic and communications can be performed at any time: it does not depend on the programs but directly on external parameters such as state of the OS during execution of the program.

Note that these routines do not put in the environment a value but a variable that is a pointer to the value. In this way, values are sent asynchronously with special rules. Also, we note the high-performance semantics as the small-step one and $\Rightarrow_{hp}$ and $\rightharpoonup_{hp}$ if we need to distinguish them.

In this semantics, the rules for the sequential control flow and those of the traditional communication primitives are the same for the small-step semantics. The language is also extended with the high-performance primitives. Rules for these primitives are given in Figure 6. The only real change is for global reductions: new rules are needed to add asynchronous communications i.e. communications that can be done at "any time".

6.2. New Communications

As in the small-step semantics, we have a rule to represent a reduction by a single processor. Asynchronous communications are done with these rules:

$$\langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i \cup \{\mathbf{hpsd}, j, x, \leftarrow\}, \mathcal{R}_i, c_i \| \cdots \rangle \text{ where } \{x \mapsto v\} \in \mathcal{E}_i \rightharpoonup$$
$$\langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \| n_j{+}1, \mathcal{E}_j, \mathcal{C}_j \cup \{\mathbf{hpsd}, j, x, \rightarrow\}, \mathcal{R}_j, c_j \| \cdots \rangle$$

$$\langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i \cup \{\mathbf{hpput}, j, \overline{y}, x, \leftarrow\}, \mathcal{R}_i, c_i \| \cdots \rangle \text{ where } \{x \mapsto v\} \in \mathcal{E}_i$$
$$\rightharpoonup \langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \| n_j + 1, \mathcal{E}_j[\overline{y}/v], \mathcal{C}_j, \mathcal{R}_j, c_j \| \cdots \rangle$$

$$\langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i \cup \{\mathbf{hpgt}, j, x, \overline{y}, \leftarrow\}, \mathcal{R}_i, c_i \| \cdots \| n_j, \mathcal{E}_j, \mathcal{C}_j, \mathcal{R}_j, c_j \| \cdots \rangle$$
$$\rightharpoonup \langle \cdots \| n_i + 1, \mathcal{E}_i[x/v], \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \| n_j + 1, \mathcal{E}_j, \mathcal{C}_j, \mathcal{R}_j, c_j \| \cdots \rangle$$
$$\text{where } \{\overline{y} \mapsto v\} \in \mathcal{E}_j$$

That is, **hpsend** sends the value pointed by x to the memory $\mathcal{E}_j$ of processor j, **hpput** writes the value to the memory at destination and **hpget** takes the value at source and the two counters are increased. When all asynchronous communications have been done, synchronous communications and BSP synchronisation is done with this rule:

$$\langle n_0, \mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0, \mathbf{sync}; c_0 \| \cdots \| n_{p-1}, \mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1}, \mathbf{sync}; c_{p-1} \rangle$$
$$\rightharpoonup \langle 0, Comm(\mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0), c_0 \| \cdots \| 0, Comm(\mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1}), c_{p-1} \rangle$$
$$\text{if } \forall i, j, x, y, v\ \{\mathbf{hpsd}, j, v\} \notin \mathcal{C}_i\ \wedge \{\mathbf{hpgt}, j, x, v\} \notin \mathcal{C}_i \wedge \{\mathbf{hpput}, j, y, v\} \notin \mathcal{C}_i$$

That is if each processor is in the **sync** case, communications are done using the $Comm$ function that exchanges the messages, which finishes the current super-step. For the oblivious synchronisation we use this rule:

$$\langle \cdots \| n_i, \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, \mathbf{oblsync}(e); c_i \| \cdots \rangle \rightharpoonup \langle \cdots \| 0, Comm(\mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i), c_i \| \cdots \rangle$$
$$\text{with } \mathcal{E}_i, \mathcal{R}_i \stackrel{i,p}{\models} e \Downarrow n_i + \|Comm(\mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i)\|$$
$$\text{and if } \forall i, j, v, x, y\ \{\mathbf{hpsd}, j, v\} \notin \mathcal{C}_i\ \wedge\ \{\mathbf{hpgt}, j, x, v\} \notin \mathcal{C}_i \wedge \{\mathbf{hpput}, j, y, v\} \notin \mathcal{C}_i$$

That is it blocks the current processor i until n_i asynchronous messages have been received plus number of messages generated by the BSP synchronous communications ($\|Comm(\mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i)\|$).

Note that $\Rightarrow_{hp}$ (resp. $\stackrel{\infty}{\Rightarrow}_{hp}$) is define as $\Rightarrow$ (resp. $\stackrel{\infty}{\Rightarrow}$).

$$\frac{\mathcal{E},\mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ and } \{\overline{y} \mapsto v'\} \in \mathcal{E} \text{ with } \mathcal{C}' = \mathcal{C} \cup \{\mathbf{hpput}, pid\%p, \overline{y}, x, \leftarrow\}}{\langle n,\mathcal{E},\mathcal{C},\mathcal{R},\mathbf{hpput}(e,x,y)\rangle \underset{i,p}{\rightharpoonup} \langle n,\mathcal{E},\mathcal{C}',\mathcal{R},\mathbf{skip}\rangle}$$

$$\frac{\mathcal{E},\mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ and } \{\overline{y} \mapsto v'\} \in \mathcal{E} \text{ with } \mathcal{C}' = \mathcal{C} \cup \{\mathbf{hpgt}, pid\%p, x, \overline{y}, \leftarrow\}}{\langle n,\mathcal{E},\mathcal{C},\mathcal{R},\mathbf{hpget}(e,x,y)\rangle \underset{i,p}{\rightharpoonup} \langle n,\mathcal{E},\mathcal{C}',\mathcal{R},\mathbf{skip}\rangle}$$

$$\frac{\mathcal{E},\mathcal{R} \overset{i,p}{\models} e \Downarrow pid \text{ and } \{x \mapsto v\} \in \mathcal{E} \text{ with } \mathcal{C}' = \mathcal{C} \cup \{\mathbf{hpsd}, pid\%p, x, \leftarrow\}}{\langle n,\mathcal{E},\mathcal{C},\mathcal{R},\mathbf{hpsend}(x,e)\rangle \underset{i,p}{\rightharpoonup} \langle n,\mathcal{E},\mathcal{C}',\mathcal{R},\mathbf{skip}\rangle}$$

$$\frac{}{\langle n,\mathcal{E},\mathcal{C},\mathcal{R},(\mathbf{oblsync};c_1);c_2\rangle \underset{i,p}{\rightharpoonup} \langle n,\mathcal{E},\mathcal{C},\mathcal{R},\mathbf{oblsync};(c_1;c_2)\rangle}$$

$$\frac{}{\langle n,\mathcal{E},\mathcal{C},\mathcal{R},\mathbf{oblsync}\rangle \underset{i,p}{\rightharpoonup} \langle n,\mathcal{E},\mathcal{C},\mathcal{R},\mathbf{oblsync};\mathbf{skip}\rangle}$$

Figure 6. Reduction rules of the PUB's high-performance routines

6.3. Coq Development and Lemmas

In the Coq development of this semantics, the only real change is for the smallstep_sem inductive: we need to add the asynchronous communications. For example, we give here some modification of this inductive definition:

```
Inductive smallstep_sem :(Z→envnmt)→(Z→instr)
        →(Z→envnmt)→(Z→instr)→Prop:=
| ...
(* We read the number of messages wanted
   and go to the next superstep. *)
|smallstep_sem_oblsync_step : ∀ env1 i1 env2 i2 e v i r,
   0<=i<p → (i1 i) = sequence_HP (oblsync_HP e) r →
   e = num (zvalue v) → (env1 i).(nbmsg) = v →
   (i2 i) = r → end_comm (env1 i) (env2 i) →
   (∀ n, n<>i →(env1 i = env2 i) ∧ (i1 i = i2 i)) →
   smallstep_sem env1 i1 env2 i2

(* hpput transition, can happen anytime *)
|smallstep_sem_oblsync_hpput : ∀
env1 i1 env2 i2 i j x y v tl,
   0<=i<p → (i1 i) = (i2 i) →
   (env1 i).(envc) = (chpput_HP j x y)::tl →
   eval i (env1 i) (var x) v →
   updateput_r (env1 j) j v y (env2 j) →
   (∀ n, n<>i →(env1 i = env2 i) ∧ (i1 i = i2 i)) →
   smallstep_sem env1 i1 env2 i2.
| ...
```

Now we have following lemmas:

Lemma 4: $\Rightarrow$ and $\Rightarrow_{hp}$ (resp. $\overset{\infty}{\Rightarrow}$ and $\overset{\infty}{\Rightarrow}_{hp}$) are equivalent for programs that do not used high-performance routines.

Lemma 5: $\Downarrow$ and $\Rightarrow_{hp}$ (resp. $\Downarrow^{\infty}$ and $\overset{\infty}{\Rightarrow}_{hp}$) are equivalent for programs that do not used high-performance routines.

Programs that have been proved correct using the natural semantics [11] are also correct if we executed them in a high-performance environment.

Lemma 6: $\Rightarrow_{hp}$ is deterministic for programs that do not used high-performance routines.

Lemma 7: $\Rightarrow_{hp}$ and $\overset{\infty}{\Rightarrow}_{hp}$ are mutually exclusive for program that do not used high-performance routines.

Lemma 8: $\Rightarrow_{hp}$ is not deterministic.

Take for example, the simple following program:

$$\begin{array}{l}
\mathbf{declare}\, x := \mathbf{pid} \textbf{ and } y := 1\, \mathbf{begin} \\
\quad \mathbf{push}(x); \\
\quad \mathbf{hpput}((\mathbf{pid}+1)\, mod\, \mathbf{nprocs}, x, x); \\
\quad x := x + 1 \\
\quad \mathbf{sync}; \\
\quad y := x \\
\mathbf{end}
\end{array}$$

For each processor, it is impossible to know which value ($\mathbf{pid}$ or $\mathbf{pid}+1$ or $\mathbf{pid}-1$) is assigned to y.

Lemma 9: $\Rightarrow_{hp}$ and $\overset{\infty}{\Rightarrow}_{hp}$ are not mutually exclusive.

Infinite reductions (and deadlocks) can occur for a some execution cases of a program and the same program can terminate for other cases. For example, in the above program, if we loop on a test of equality on y and **pid**, we will have a case where it is an infinite loop for some processors.

Conjecture 1: There are some code optimisation functions **Opt** such that:

1) if $\Rightarrow$ is defined for any program c then for $\mathbf{Opt}(c)$ $\Rightarrow_{hp}$ gives the same environment
2) if $\Rightarrow^{\infty}$ is defined for any program c then it is also the case that $\mathbf{Opt}(c) \overset{\infty}{\Rightarrow}_{hp}$

A function that does not modify the code is a possible case (see Lemma 4) but not an interesting one.

7. Related work

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 been studied such as BSP functional programming using Coq [12] or the derivation of BSP imperative programs using Hoare's axiom semantics [13], [14]. The main drawback of these approaches is that they use their own languages that are in general not subsets of real programming languages. Also they neither used any proof assistant (except [12]) nor

implemented dedicated tools 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 [15]. Also, our Coq development ensures safety.

In [11], we presented a formal deterministic operational semantics (a natural semantics) for BSP programs and used it to prove the correctness of a classical numerical computation (the N-body problem which is considered to be an important Dwarf [3]) and the divergence of some programs.

8. Conclusion

In this paper, we have presented two formal deterministic operational semantics for BSP programs: one for classical BSP operations and another which introduces high-performance primitives. 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. Instead, it is intended to be the basis for better tools for the proof of BSP programs. We are thinking about extending the theoretical work of [16] and its C application software [17] which generates lemmas to be proved (using a proof assistant) from Hoare's assertions in C programs that ensure correctness (using a formal semantics).

Our second semantics could 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 transformation 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 an interesting challenge.

References

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

[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] 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," Electrical Engineering and Computer Sciences, University of California at Berkeley, Tech. Rep. UCB/EECS-2006-183, 2006.

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

[5] D. B. Skillicorn, J. M. D. Hill, and W. F. McColl, "Questions and Answers about BSP," *Scientific Programming*, vol. 6, no. 3, pp. 249–274, 1997.

[6] O. Bonorden, B. Juurlink, I. V. Otte, and O. Rieping, "The Paderborn University BSP (PUB) library," *Parallel Computing*, vol. 29, no. 2, pp. 187–207, 2003.

[7] O. Bonorden, J. Gehweiler, and F. M. auf der Heide, "A Web Computing Environment for Parallel Algorithms in Java," *Scalable Computing: Practice and Experience*, vol. 7, no. 2, pp. 1–14, 2006.

[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] K. Hinsen, "Parallel scripting with Python," *Computing in Science & Engineering*, vol. 9, no. 6, 2007.

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

[11] F. Gava and J. Fortin, "Formal Semantics of a Subset of the Paderborn's BSPlib," in *PDCAT 2008*. IEEE Press, 2008, to appear.

[12] F. Gava, "Formal Proofs of Functional BSP Programs," *Parallel Processing Letters*, vol. 13, no. 3, pp. 365–376, 2003.

[13] Y. Chen and W. Sanders, "Top-Down Design of Bulk-Synchronous Parallel Programs," *Parallel Processing Letters*, vol. 13, no. 3, pp. 389–400, 2003.

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

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

[16] J.-C. Filliâtre, "Verification of Non-Functional Programs using Interpretations in Type Theory," *Journal of Functional Programming*, vol. 13, no. 4, pp. 709–745, 2003.

[17] J.-C. Filliâtre and C. Marché, "Multi-Prover Verification of C Programs," in *Sixth International Conference on Formal Engineering Methods (ICFEM)*, ser. LNCS, vol. 3308. Springer-Verlag, 2004, pp. 15–29, http://why.lri.fr/caduceus/.