Formal Operational Semantics
of a Core Imperative BSP language
Application to the proof of a numerical program and a certified optimization

Jean Fortin & Frédéric Gava

October 2008
TR–LACL–2008–6

Laboratoire d’Algorithmique, Complexité et Logique (LACL)
University of Paris–East

Université Paris 12 – Val de Marne, Faculté des Science et Technologie
61, Avenue du Général de Gaulle,
94010 Créteil cedex, France
Tel.: (33)(1) 45 17 16 47, Fax: (33)(1) 45 17 66 01
Formal Operational Semantics
of a Core Imperative BSP language
Application to the proof of a numerical program and a certified optimization

Jean Fortin Frédéric Gava

Laboratory of Algorithmics, Complexity and Logic (LACL)
University of Paris–East (Paris 12),
61 avenue du Général de Gaulle, P2 du CMC
94010 Créteil cedex, France

gava@univ-paris12.fr & 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 three formal operational semantics for a C+PUB subset language using the Coq proof assistant. A first semantics, for classical BSP operations, has been used to certified a classical numerical computation, the N-body problem. A second semantics is used as an intermediary interpretation of the programs. The last semantics emphasises high performance primitives of the PUB and has been used to certified a function of optimisation of the source code. Some meta-lemmas of the semantics are also given and have been prove in Coq.
## Contents

<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Introduction</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Coq and BSP Programming</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>The Coq Proof Assistant</td>
<td>4</td>
</tr>
<tr>
<td>2.2</td>
<td>The Paderborn University BSPlib</td>
<td>5</td>
</tr>
<tr>
<td>2.2.1</td>
<td>BSP model</td>
<td>5</td>
</tr>
<tr>
<td>2.2.2</td>
<td>Generalities</td>
<td>5</td>
</tr>
<tr>
<td>2.2.3</td>
<td>Message Passing and Synchronisation</td>
<td>6</td>
</tr>
<tr>
<td>2.2.4</td>
<td>Remote Memory Access</td>
<td>6</td>
</tr>
<tr>
<td>2.2.5</td>
<td>High Performance primitives</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Core language</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>Big-step operational semantics</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Local Reductions of the Semantics</td>
<td>9</td>
</tr>
<tr>
<td>3.2</td>
<td>Global Reductions and Communications</td>
<td>9</td>
</tr>
<tr>
<td>3.3</td>
<td>Coq Development and Lemmas</td>
<td>12</td>
</tr>
<tr>
<td>3.4</td>
<td>A Certified BSP program for the ( N )-body Problem</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>Small-step operational semantics</td>
<td>15</td>
</tr>
<tr>
<td>4.1</td>
<td>Local rules</td>
<td>15</td>
</tr>
<tr>
<td>4.2</td>
<td>Global Reductions and Communications</td>
<td>15</td>
</tr>
<tr>
<td>4.3</td>
<td>Coq Development and Lemmas</td>
<td>17</td>
</tr>
<tr>
<td>5</td>
<td>High-performance operational semantics</td>
<td>19</td>
</tr>
<tr>
<td>5.1</td>
<td>Local rules</td>
<td>19</td>
</tr>
<tr>
<td>5.2</td>
<td>Global Reductions and Communications</td>
<td>19</td>
</tr>
<tr>
<td>5.3</td>
<td>Coq Development and Lemmas</td>
<td>21</td>
</tr>
<tr>
<td>5.4</td>
<td>Transformation of the source code</td>
<td>23</td>
</tr>
<tr>
<td>5.4.1</td>
<td>Generalities</td>
<td>23</td>
</tr>
<tr>
<td>5.4.2</td>
<td>Optimization Conditions</td>
<td>23</td>
</tr>
<tr>
<td>5.4.3</td>
<td>Overview of the Translation</td>
<td>23</td>
</tr>
<tr>
<td>5.4.4</td>
<td>Function of optimisation of a block</td>
<td>24</td>
</tr>
<tr>
<td>5.4.5</td>
<td>Correctness of the Translation and Coq development</td>
<td>25</td>
</tr>
<tr>
<td>6</td>
<td>Related work</td>
<td>26</td>
</tr>
<tr>
<td>6.1</td>
<td>Proof of BSP Programs</td>
<td>26</td>
</tr>
<tr>
<td>6.2</td>
<td>Formally verified source-code transformations</td>
<td>26</td>
</tr>
<tr>
<td>7</td>
<td>Conclusion and future work</td>
<td>28</td>
</tr>
<tr>
<td>7.1</td>
<td>Conclusion</td>
<td>28</td>
</tr>
<tr>
<td>7.2</td>
<td>Future work</td>
<td>29</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

General framework. Can you trust your parallel compiler? More precisely, can you trust the optimizations on communication routines that could be automatically done by a compiler to your source code. Compilers are generally assumed to generate semantically equivalent machine’s instructions from the source program. Despite intensive testing and assurance that the sequential part of your program source is well generated [22], bugs like deadlocks can occur, when the compiler silently generates an incorrect executable for a correct source.

Writing parallel programs is notoriously known to be difficult, as well as tracking down compiler-introduced bugs. Debugging both is usually a nightmare.

To cope with the first difficulty, structured approaches to parallel programming using high-level tools (models, languages, etc.) are classical solutions. They are necessary to simplify both the design of parallel algorithms and their programming but also to ensure a better safety of the generated applications.

The second difficulty needs formal methods (rigorous testing is in general not sufficient). The verification of the optimizations of communications introduced by a compiler guarantees that the safety properties proved on the source code hold for the executable compiled code as well.

For sequential programming, this kind of work has begun to be well studied and would clearly become the norm. But for parallel (high-performance) computing, this kind of research is stammering. There are two main reasons. First, parallel computing is complex. Second, most of researchers on parallel computing are usually not accustomed to formal methods. Research on tools that optimize communication routines have been done [8] with intuitive explanations on their reliability. To our knowledge, no such tools have been formally verified.

Solving a problem on a parallel machine is thus 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 [4]. 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] (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 Framework. BSP 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 [25]. The BSP model can also be well adapted to Grid-computing [5] and is generally presented as a bridging and emerging model for high-performance computations.

---

1Some examples could be find in the users’ contributions web site of the Coq proof assistant.
2Dwarfs 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.
3We refer to [4][25] for a gentle introduction to the BSP model.
An interesting advantage of BSP libraries is that they use a very small number of safe primitives compared to the hundreds of the standard MPI which are generally unsafe. Furthermore, the model of execution is structured, making the analysis of BSP programs easier, yet programs are still efficient. BSP libraries are thus good candidates for formal semantics investigations.

Moreover, BSP libraries provide high-performances (HP) versions of the BSP routines for both BSP message passing (BSMP) and remote memory accesses (DRMA). These routines are proposed to programmers for improved speedup of their program even if they are unsafe: they are unbuffered and do not really follow the safe BSP model of execution [6]. Replacing classical BSP routines by their high-performance pendant is thus of the responsibility of the programmer (some conditions for this transformation are provided in the reference manual) or of a non-formally verified compiler analyzer as those of [8].

This is disappointing if we look for an environment where programmers can write and execute their parallel programs in a safe and efficient way. Figure 1.1 resumes this wanted environment (first part is the subject of [13]): it is easier to code a program using only safe operations, to prove this program using a natural and deterministic semantics [13] (or dedicated tools using this semantics), and then to optimize it automatically, in order to convert standard operations into high performance operations when it is possible. Formally verifying that this transformation is semantically correct needs the design of a formal operational semantics that emphasizes the high-performances routines.

**Aim of the report.** The aim of this paper is the Coq development of three 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 semantics is a transition one of the two others. The third semantics is for enable high performance operations. High-performance operations improve speedup of programs using unbuffered communications but must be used wisely due to their unsafe and undermistic nature (i.e., to not produce the same output 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).

The first semantics has been used to certified a classical parallel numerical computation: the N-body problem. As an application of the third semantics (high-performance ), we have done a formal verification of a simple function of optimization of the source code that transforms some classical BSP routines by high-performances ones. That is, the original source code and the faster generated one are semantically equivalent.

The work presented in this paper is part of an ongoing project that aims at developing a realistic environment where programmers could prove BSP programs (scientific computations) and formally verifying that parallel execution preserves the semantics of the programs being compiled.

As written above, all operational semantics were developed using the Coq proof assistant. The full development is available at [http://lacl.univ-paris12.fr/gava/bsp-sem.tar](http://lacl.univ-paris12.fr/gava/bsp-sem.tar).

---

4Determinism is a property that is a powerful aid in debugging and validating programs even if it limits the programmer’s flexibility to code certain algorithms.
Outline. The remainder of this report is organized as follows. First, we briefly describe the Coq proof assistant, the PUB library and we give our kernel language in Chapter 2.

Then, in Section 3 we give an overview of its big-step operational semantics (natural semantics), its Coq development and our example of correctness of a BSP programs demonstrates the usefulness of this formal semantics. We give in Chapter 4 a small-step semantics for classical BSP operations that has been developed using Coq. This semantics would be used as intermediate result.

In Chapter 5 we give an overview of a small-step semantics that emphasises the high-performance primitives of the PUB and the formal development using the Coq proof assistant. We also present the usefulness of this formal semantics by showing a function of optimization of the source code and describing the formal verification of semantics correctness of this transformation.

Related work is discussed in Chapter 6 followed by conclusions and future work in Chapter 7.
Chapter 2

Coq and BSP Programming

2.1 The Coq Proof Assistant

The Coq system\footnote{Available at \url{http://coq.inria.fr/} with nice introductions to this theorem-proving system.} 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) constructive logic with all the usual connectives and quantifiers; (2) inductive definitions \textit{via} inference 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 (\texttt{intro}, which adds an abstraction to the proof term) to rather complex decision procedures (\texttt{omega} for Presburger arithmetic).

For example, let us define in Coq a language of numerical expressions $e := n | e_1 + e_2 | \frac{e_1}{e_2}$:

\begin{verbatim}
Inductive expr : Set := num : Z → expr
| plus : expr → expr → expr
| div : expr → expr → expr
\end{verbatim}

(integers are noted $\mathbb{Z}$). Now, its natural semantics i.e. abstract evaluation to integers would be defined by the following inductive:

\begin{verbatim}
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)
\end{verbatim}

and we noted $(eval\_expr\ e\ n)$ the evaluation of expression $e$ to integer $n$. For a small-step semantics, we will first defined a one step reduction:

\begin{verbatim}
Inductive one_step : expr → expr → Prop :=
(* plus *)
| 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))
\end{verbatim}

\begin{verbatim}
(* div *)
| div_left : ∀ e1 e’1 e2, one_step e1 e’1 → one_step (div e1 e2) (div e’1 e2)
| div_right : ∀ n1 e2 e’2, one_step e2 e’2 → one_step (plus (num n1) e2) (plus (num n1) e’2)
\end{verbatim}
\[ \text{div} \div : \forall n_1 \, n_2, \, n_2 \neq 0 \rightarrow \text{one_step} (\text{div} (\text{num} n_1) (\text{num} n_2)) \, (\text{num} (n_1/n_2)) \]

and then the transitive and reflexive closure of \text{one_step}:

\[
\text{Inductive } \text{step} \_ \text{star} : \text{expr} \rightarrow \text{expr} \rightarrow \text{Prop} ::= \\
| \text{sos_refl} : \forall e, \text{step} \_ \text{star} e e \\
| \text{sos_trans} : \forall e_1 e_2 e_3, \text{one_step} e_1 e_2 \\
\quad \rightarrow \text{step} \_ \text{star} e_2 e_3 \rightarrow \text{step} \_ \text{star} e_1 e_3
\]

The small-step semantics i.e. abstract evaluation of expression \( e \) to integer \( n \) is noted \((\text{step} \_ \text{star} e (\text{num} n))\).

### 2.2 The Paderborn University BSPlib

#### 2.2.1 BSP model

A BSP machine is a set of pairs CPU-memory distributed across a communication network. The execution of a BSP program is divided into super-steps (see Fig. 2.1), 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 start of the next super-step; this ensures the determinism of the parallel program.

A BSP computer is characterized by three parameters, which are given in function of the processor speed \( s \):

- The number of processors \( p \);
- The time taken by a global synchronisation \( l \);
- The time taken for a collective exchange where each processor to send and/or receive at most one word \( g \). This exchange is called a 1-relation. Thus, a communication phase where each processor sends and/or receives \( h \) words is bounded by \( h \times g \).

Any BSP program’s complexity is given in function of these parameters.

#### 2.2.2 Generalities

PUB is a C-Library of communication routines to support development of parallel algorithms based on the BSP model. 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 computers (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 \cite{14} 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 bspplib_init\footnote{We refer to the manual, \url{http://wwwcs.uni-paderborn.de/~pub/documentation.html} for C type and more details about other functions of the PUB.}. 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 bspplib_done which cleans up all PUB resources.

### 2.2.3 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 \texttt{void bsp_send (int dest, void* buffer, int s)} where \texttt{buffer} is a pointer to a memory address to send to processor id \texttt{dest} and \texttt{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 \texttt{t_bspmsg* bsp_findmsg (int proc_id, int index)} where \texttt{proc_id} is the id of the source-node and \texttt{index} is the index of the message (numbered from 0 to \( n − 1 \); if \texttt{index} > \( n − 1 \) then \texttt{bsp_findmsg} returns NULL). \texttt{t_bspmsg} is the type of a message in the PUB. To access to the sending message \texttt{t_bspmsg}, we need \texttt{bspmsg_data} which returns a pointer to the sending block of data and \texttt{bspmsg_size} which returns its size. Also \texttt{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.

### 2.2.4 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: \texttt{void bsp_push_reg (t_bsp* bsp, void* ident, int size)} and \texttt{void bsp_pop_reg (t_bsp* bsp, void* ident)} where \texttt{ident} is the local address of the variable and \texttt{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 \texttt{bsp_push_reg} functions in the same order (same for \texttt{bsp_pop_reg}). DRMA operations are:

- \texttt{void bsp_get (t_bsp* bsp, int srcPID, void* src, int offset, void* dest, int nbytes)} (global reading access) and
- \texttt{void bsp_put (t_bsp* bsp, int destPID, void* src, void* dest, int offset, int nbytes)} (global writing access).

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

All primitives of communications have their high performance version. 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 destination is written is undefined (architecture dependant).

Why high-performance in BSP? Figure 2.2 shows the speedups for various versions of a program (the classic $N$-body problem) that uses only classical BSP operations, BSP with unbuffered communications (BSP+HP), oblivious synchronisations (BSP+OblSync) or all these optimisations (BSP+HP+OblSync). This program is based on a systolic loop [15] followed by a parallel prefixe. In such an algorithm, data (point masses) are 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 high-performance operations improve speedup.

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 ($nmsgs$) each 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.

2.3 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: $pid$ and $nprocs$. The abstract syntax of our language is as follows (sequential control flow commands):

$$
c ::= \quad \text{skip} \quad \text{Null command}
\quad \mid \quad x ::= e \quad \text{Assignment}
\quad \mid \quad c_1 ; c_2 \quad \text{Sequence}
\quad \mid \quad \text{if } e \text{ then } c_1 \text{ else } c_2 \text{ endif} \quad \text{Conditional}
\quad \mid \quad \text{while } e \text{ do } c \text{ done} \quad \text{Iteration}
\quad \mid \quad \text{declare } y ::= e \text{ begin } c \text{ end} \quad \text{New variable}
$$
with \(x, y \in X\) and \(e \in \text{Exp}\). Expressions are evaluated to values \(v\) (subset of \(\text{Exp}\)) and we write: \(\mathcal{E}_i, \mathcal{R}_i \models^{p} e \Downarrow v\) with \(p\) the number of processors and \(i\) the pid. In the Coq formalization, this abstract syntax is presented as inductive data types. \(\mathcal{E}_i\) 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 \(\mathcal{E}_i, \mathcal{R}_i \models^{p} \text{pid} \Downarrow i\) and \(\mathcal{E}_i, \mathcal{R}_i \models^{p} \text{nprocs} \Downarrow p\). Evaluation of \(\text{Exp}\) is not total (e.g. evaluation of \(1 + \text{true}\)) but for simplicity always terminates. Parallel operations are:

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

Sending a single message is done using \(\text{send}\). In the next super-step, each processor can access the received messages.

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 \(\text{push}\) and \(\text{pop}\). DRMA operations are \(\text{put}\) and \(\text{get}\). All get and put operations are executed during the synchronisation and all get operations are served before a put overwrites a value.

According to the BSP model all messages are received during the barrier of synchronisation and cannot be read before. Barrier is done using \(\text{sync}\) which blocks the node until all other nodes have called \(\text{sync}\) and all messages sent to it in the current super-step have been received.

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

All BSP primitives of communications have thus their high performance version (called \(\text{hpput}, \text{hpget}\) and \(\text{hpsend}\) with the same parameters). The copies are done asynchronously and unbuffered. They are finished after the next super-step and the buffer (src and dest) must not be changed before. Time the destination is written is undefined (architecture dependent).

The PUB also contains an oblivious synchronization written \(\text{oblsync}(n)\) which should be only used if the programmer knows the number \(n\) of messages a processor will receive in a superstep. This kind of synchronization is much faster (but unsafe) than the other one since no additional communication is needed and no barrier synchronization is done. 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 each a processor waits in \(\text{oblsync}\) is: (1) Each message send sent with \(\text{send}\) (or \(\text{hpsend}\)); (2) Each \(\text{put}\) (or \(\text{hpput}\)) produces “one message” at the destination; (3) Each \(\text{get}\) (or \(\text{hpget}\)) produces “one message” at source and destination.
Chapter 3

Big-step operational 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 $P^c$ and co-inductive ones $\rho^c$ (infinite but rational derivation trees [21]).

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 $E[x/v]$ insertion or substitution in $E$ of a new binding from $x$ to $v$. We note $R$ the received values of the previous super-step and $C$ 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”).

3.1 Local Reductions of the Semantics

We note $\downarrow_l$ for local reductions (e.g. one at each processor). Local final configurations are noted $\langle E', C', R', \text{skip} \rangle$ but we also have intermediate results due to synchronisations, that is when processors finish a super-step (sync). We have thus to memorise the next instructions of each processor. This intermediate local configuration is noted $\langle E', C', R', \text{SYNC}(c) \rangle$ where $c$ is the sequence of next instructions for the next super-steps. Figures 3.1 and 3.5 give finite and infinite rules of the local control flow. Figure 3.5 gives the rules of PUB routines.

We note $\tau$ a variable that has been registered for global access (DRMA), $x$ for the contrary and $x$ when that is not important. Messages (communications) are used with get, put or 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. 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.

3.2 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 $E$, communications $C$ and received values $R$. A final configuration is $\text{skip}$ on all processors. We note the full evaluation:

\footnotesize
\begin{itemize}
    \item $^1$To respect the documentation of the PUB, we can count on each processor the registering of variables and compare them at each barrier.
    \item $^2$We can introduce a dynamic change of variables’s name but that is a tedious work.
\end{itemize}
the program continues (and returns a set of environments) or diverges.

Communications are modelled with the two inference rules of Figure 3.4 that is if each processor is in the local (sequential) ones with the two rules defined in Figure 3.3 that is each processor computes a final configuration or there is at least one processor that diverges.

The *Comm* function specifies the order of the messages during the communications. It modifies the environ-
\[ \forall i \quad E_i, C_i, R_i \xRightarrow{\text{c_i \in \{E_i', C_i', R_i', \text{skip}\}}} (E_i, C_i, R_i \xRightarrow{c_0} \cdots \xRightarrow{E_{p-1}, C_{p-1}, R_{p-1}} \xRightarrow{c_{p-1}} \vdash \forall \phi (E_0', C_0', R_0, \text{skip}) \cdots \xRightarrow{E_{p-1}', C_{p-1}', R_{p-1}', \text{skip}}) \]

\[ \forall \phi (E_0, C_0, R_0) \xRightarrow{\text{c_i \in \{E_i', C_i', R_i', \text{skip}\}}} (E_0, C_0, R_0 \xRightarrow{c_0} \cdots \xRightarrow{E_{p-1}, C_{p-1}, R_{p-1}} \xRightarrow{c_{p-1}} \vdash \forall \phi \]

**Figure 3.3:** Global rules call local ones

\[ \forall i \quad E_i, C_i, R_i \xRightarrow{\text{c_i \in \{E_i', C_i', R_i', \text{SYNC}(c_i')\}}} (\cdots \vdash \text{Comm}(E_i', C_i', R_i') \xRightarrow{c_i'} \cdots \vdash \cdots \vdash \forall \phi (E_i', C_i', R_i') \xRightarrow{c_i'} \vdash \cdots \vdash \forall \phi) \]

**Figure 3.4:** Communications rules for both finite and finite reductions

\[ \begin{align*}
\text{E, C, R \xRightarrow{c \in E}} \quad & \text{sync} \vdash \phi \quad (E, C, R, \text{SYNC}(\text{skip})) \\
\text{E, C, R \xRightarrow{c \in E}} \quad & \text{push}(x) \vdash \phi \quad (E, C, R, \text{skip}) \\
\text{E, R \xRightarrow{e \in E}} \quad & \text{pop}(x) \vdash \phi \quad (E, C, R, \text{skip}) \\
\text{E, R \xRightarrow{e \in E}} \quad & \text{pid} \quad (E, C, R, \text{skip}) \\
\text{E, R \xRightarrow{e \in E}} \quad & \text{send}(x, e) \vdash \phi \quad (E, C, R, \text{skip}) \\
\text{E, R \xRightarrow{e \in E}} \quad & \text{findmsg}(e_1, e_2) \vdash v \\
\end{align*} \]

**Figure 3.5:** Natural semantics of the PUB’s routines

The access 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".
3.3 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 \texttt{instr} for a list of instructions \texttt{envnmt} for environments (store \(E\), communications \(C\) and received values \(R\)).

Local reductions \(\downarrow_l\) are modelled with the inductive \texttt{nat\_sem\_l} as follows:

\textbf{Inductive} \texttt{nat\_sem\_l} (\(i:\mathbb{Z}\)) : \texttt{envnmt} \to \texttt{instr} \to \texttt{envnmt} \to \texttt{ef} \to \text{Prop} :=

\begin{align*}
\text{(case } x:=e*) & \quad \text{nat\_sem\_assign} : \forall x e n \text{ env env}',
\quad \text{eval i env e n} \Rightarrow \text{update env x n env} '
\quad \Rightarrow \text{nat\_sem\_l i env (assign x e) env'} eskip \\
\text{(case sync; c*)} & \quad \text{nat\_sem\_sync\_c} : \forall \text{ env c, nat\_sem\_l i env}
\quad \text{(sequence sync c) env (esync c)} \\
\text{(case put*)} & \quad \text{nat\_sem\_put} : \forall e \text{ pid x y env env}',
\quad \text{eval i env e pid}
\quad \Rightarrow \text{updateput env pid x y env'}
\quad \Rightarrow \text{nat\_sem\_l i env (put e x y) env'} eskip \\
\text{(case send*)} & \quad \text{nat\_sem\_send} : \forall e \text{ pid x env env'} \text{ v,}
\quad \text{eval i env e pid} \Rightarrow \text{lookup (fst env) x v}
\quad \Rightarrow \text{updatesend env v pid env'}
\quad \Rightarrow \text{nat\_sem\_l i env (send x e) env'} eskip \\
\text{(Other cases*)} & \quad \ldots
\end{align*}

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

\(p\)-vectors are represented as functions from \(\mathbb{Z}\) (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 \texttt{nat\_sem} as follows:

\textbf{Inductive} \texttt{nat\_sem} : (\(\mathbb{Z}\to\text{envnmt}\)) \to (\(\mathbb{Z}\to\text{instr}\)) \to (\(\mathbb{Z}\to\text{env}\)) \to \text{Prop} :=

\begin{align*}
\text{(Global rule when local reductions give skip*)} & \quad \text{nat\_sem\_par\_skip} : \forall \text{ env instrs env}',
\quad (\forall \text{ pid, } (0 \leq \text{pid} < p))
\quad \Rightarrow \text{nat\_sem\_l pid (env pid) (instrs pid)}
\quad \text{(env' pid eskip))}
\quad \Rightarrow \text{nat\_sem\_l instrs env'} \\
\text{(Communication rule*)} & \quad \text{nat\_sem\_par\_comm} : \forall \text{ env instrs env'} \text{ env\_com env''},
\quad (\forall \text{ pid, } (0 \leq \text{pid} < p))
\quad \Rightarrow \text{nat\_sem\_l pid (env pid) (instrs pid)}
\quad \text{(env' pid (esync (c pid))))}
\quad \Rightarrow \text{comm\_g c' env\_com}
\quad \Rightarrow \text{nat\_sem\_env\_com c env''}
\quad \Rightarrow \text{nat\_sem\_env\_intrs env''}
\end{align*}

where at processor \texttt{pid} there is a local evaluation \texttt{nat\_sem\_l}. \texttt{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:

\textbf{Lemma 3.3.1} \(\downarrow_g\) is deterministic.

\textbf{Lemma 3.3.2} \(\downarrow_g\) 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
```

### 3.4 A Certified BSP program for the $N$-body Problem

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

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

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 $p-1$ super-steps, all pairs of particles have been treated and a parallel folding of these values can be done to finish the computation.

Figure 3.6 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 \textit{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 \textit{pair\_energy}. We also need to prove that our systolic loop just applies $p$ times the \textit{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.
Parallel direct prefixes:

```plaintext
declare y := 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:

```plaintext
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_prefix_for(energy);
end
end
```

Figure 3.6: Code of the direct folding and of the N-body computation
Chapter 4

Small-step operational semantics

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 be an intermediate semantics between the natural one and the semantics that emphasizes high-performances operations.

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).

4.1 Local rules

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

We have thus to memorise the next instructions of each processor. This intermediate local configuration is noted \( \langle E', C', R', \text{SYNC}(c) \rangle \) where \( c \) is the sequence of next instructions for the next super-steps.

Figure 4.1 gives rules of the local control flow and Figure 4.2 the rules of the PUB routines. The communications environment \( C \) now contains messages to be sent (noted with \( \leftarrow \)) and asynchronous messages received from other processors (noted with \( \rightarrow \)). Messages (communications) are used with \( \text{get}, \text{put} \) or \( \text{send} \) with their natural arguments (we note \( \% \) for modulo calculus).

4.2 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 \( E \), communications \( C \) and received values \( R \). A final configuration is \( \text{skip} \) on all processors: \( \langle E_0, C_0, R_0, \text{skip} \parallel \cdots \parallel E_{p-1}, C_{p-1}, R_{p-1}, \text{skip} \rangle \)

The global reductions call the local ones with this rule:

\[
\begin{align*}
\langle E_i, C_i, R_i, c_i \rangle \rightarrow_{i,p} \langle E'_i, C'_i, R'_i, c'_i \rangle \\
\langle \cdots \parallel n_i, E_i, C_i, R_i, c_i \parallel \cdots \rangle \rightarrow \langle \cdots \parallel n_i, E'_i, C'_i, R'_i, c'_i \parallel \cdots \rangle
\end{align*}
\]

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:

\[
\begin{align*}
\langle E_0, C_0, R_0, \text{sync}; c_0 \parallel \cdots \parallel E_{p-1}, C_{p-1}, R_{p-1}, \text{sync}; c_{p-1} \rangle \\
\rightarrow \langle \text{Comm}(E_0, C_0, R_0), c_0 \parallel \cdots \parallel \text{Comm}(E_{p-1}, C_{p-1}, R_{p-1}), c_{p-1} \rangle
\end{align*}
\]

that is if each processor is in the \text{sync} case, communications are done (using the \text{Comm} function that model exchanges of messages) and the current super-step is finished. The \text{Comm} function specifies the order of the messages during the communications. It modifies the environment of each processor \( i \) such that \( \text{Comm}(C'_i, R'_i, E'_i) = (C''_i, R''_i, E''_i) \) is for BSMP as follows:
(E, C, R, c_1) \rightarrow (E', C', R', c'_1)
(E, C, R, c_1, c_2) \rightarrow (E', C', R', c'_1, c'_2)
E, R \xrightarrow{I_p} e \| v
\langle E, C, R, x := e \rangle \rightarrow_{I_p} \langle E[x/v], C, R, \text{skip} \rangle
\langle E, C, R, \text{skip}; c_2 \rangle \rightarrow_{I_p} \langle E, C, R, c_2 \rangle
E, R \xrightarrow{I_p} e \| \text{true}
\langle E, C, R, \text{if } e \text{ then } c_1 \text{ else } c_2 \text{ endif} \rangle \rightarrow_{I_p} \langle E, C, R, c_1 \rangle
E, R \xrightarrow{I_p} e \| \text{false}
\langle E, C, R, \text{if } e \text{ then } c_1 \text{ else } c_2 \text{ endif} \rangle \rightarrow_{I_p} \langle E, C, R, c_2 \rangle
\langle E, C, R, \text{while } e \text{ do } c \text{ done} \rangle \rightarrow \langle E, C, R, \text{if } e \text{ then } c' \text{ else } \text{skip} \text{ endif} \rangle
\langle E, C, R, \text{Declare } x := e \text{ begin } c \text{ end} \rangle \rightarrow_{I_p} \langle E[x/v], C, R, c \rangle

Figure 4.1: Reduction of sequential control flow

if \{x \mapsto v\} \in E \text{ with } E' = E \oplus \{x \mapsto v\}
\langle E, C, R, \text{push}(x) \rangle \rightarrow_{I_p} \langle E', C, R, \text{skip} \rangle

if \{x \mapsto v\} \in E \text{ with } E' = E \oplus \{x \mapsto v\}
\langle E, C, R, \text{pop}(x) \rangle \rightarrow_{I_p} \langle E', C, R, \text{skip} \rangle

E, R \xrightarrow{I_p} e \| \text{pid and } \{x \mapsto v\} \in E \text{ and } \{x \mapsto v'\} \in E
\langle E, C, R, \text{put}(e, x, y) \rangle \rightarrow_{I_p} \langle E', C', R, \text{skip} \rangle
\text{where } C' = C \cup \{\text{put, pid}(p, y, v, \_\_)\}

E, R \xrightarrow{I_p} e \| \text{pid and } \{x \mapsto v\} \in E \text{ and } \{x \mapsto v'\} \in E
\langle E, C, R, \text{get}(e, x, y) \rangle \rightarrow_{I_p} \langle E', C', R, \text{skip} \rangle
\text{where } C' = C \cup \{\text{get, pid}(p, x, y, v, \_\_)\}

E, R \xrightarrow{I_p} e \| \text{pid and } \{x \mapsto v\} \in E \text{ with } C' = C \cup \{\text{send, pid}(p, v, \_\_)\}
\langle E, C, R, \text{send}(x, e) \rangle \rightarrow_{I_p} \langle E', C', R, \text{skip} \rangle

Figure 4.2: Reduction of the PUB’s routines

C''_i = \emptyset
R''_i = \bigcup_{j=0}^{p-1} \bigcup_{n=0}^{n_j} \{j, n + \sum_{a=0}^{j} n_a, v\} \text{ if } \{\text{send, i, v}\} \in C_j

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

E''_i = E'_i \bigcup_{j=0}^{p-1} \bigcup_{k=0}^{n-1} \{y \mapsto v\} \text{ if } E'_{i} \text{ and } \{\text{get, j, x, y}\} \in C'_i
\text{and } \{y \mapsto v\} \in E'_{i} \text{ and } \{\text{put, i, y, v'}\} \in C'_i

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).
We note $\Rightarrow$ for a finite derivation and $\Rightarrow^\infty$ for an infinite one (this has to be read as "program diverges"). $\Rightarrow$ (resp. $\Rightarrow^\infty$) is defined by induction (resp. by co-induction i.e. infinite but rational derivation [21]) in Figure 4.3 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".

### 4.3 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 $\texttt{instr}$ for a list of instructions $\texttt{envnmt}$ for environments (store $\mathcal{E}$, communications $\mathcal{C}$ and received values $\mathcal{R}$). We also use $\texttt{eq\_envnmt}$ for equivalence of environments. $p$-vectors are represented as functions from $\mathbb{Z}$ to instructions or environments. The two rules of global reduction $\rightarrow$ (doing local calculations and communications) are modelled with the inductive $\texttt{smallstep\_sem}$ as follow:

\begin{align*}
\textbf{Inductive} \quad & \texttt{smallstep\_sem} : (\mathbb{Z} \rightarrow \texttt{envnmt}) \rightarrow (\mathbb{Z} \rightarrow \texttt{instr}) \\
& \rightarrow (\mathbb{Z} \rightarrow \texttt{envnmt}) \rightarrow (\mathbb{Z} \rightarrow \texttt{instr}) \rightarrow \texttt{Prop} := \\
\texttt{smallstep\_sem\_local} : & \forall i. \texttt{env1 i i} \texttt{env2 i2} \texttt{envli i3 i2i} \\
& \texttt{smallstep\_sem\_l i envli i3 env2i i2i} \rightarrow \\
& \texttt{eq\_envnmt envli i} \rightarrow \texttt{eq\_envnmt env2i i} \\
& \forall n. \forall i. \forall n. (\texttt{eq\_envnmt (env1 n) (env2 n)}) \land (\texttt{skip}) \\
& \texttt{smallstep\_sem\_l i env1 i1 env2 i2} \\
\texttt{smallstep\_sem\_sync} : & \forall i. \forall n. \forall i. \forall n. (\texttt{seq\_sync (i2 i)}) \lor (i i = \texttt{sys\_cmd} \land \texttt{skip}) \\
& \rightarrow \texttt{smallstep\_sem\_env1 i env2 i2}.
\end{align*}

Then $\Rightarrow$ (resp. $\Rightarrow^\infty$) are then defined by a inductive (resp. co-inductive) on this reduction.

The inductive $\texttt{smallstep\_sem\_l}$ 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:

\begin{align*}
\textbf{Inductive} \quad & \texttt{smallstep\_sem\_l} (i : \mathbb{Z}) : \texttt{envnmt} \rightarrow \texttt{instr} \rightarrow \texttt{envnmt} \rightarrow \texttt{instr} \rightarrow \texttt{Prop} := \\
\texttt{...}
\end{align*}

(* If then else semantics *)

\begin{align*}
\text{\texttt{smallstep\_sem\_ifelse\_true}} : & \forall e. \text{\texttt{b\_eval e i1 i2 l1 l1'}} \\
& \text{\texttt{eval e i1 e true}} \\
& \text{\texttt{eq\_envnmt l1 l1'}} \\
& \text{\texttt{smallstep\_sem\_l i1 l1' (ifelse e i1 i2) l1' l1'}}
\end{align*}

(* put *)

\begin{align*}
\text{\texttt{smallstep\_sem\_put}} : & \forall e. \text{\texttt{pid x y l1 l1'}} \\
& \text{\texttt{eval e i1 e (zvalue pid)}} \\
& \text{\texttt{update\_put l pid x y l1' l1'}} \\
& \text{\texttt{smallstep\_sem\_l i1 l1' (put e x y) l1' l1'}}
\end{align*}

(* send *)

\begin{align*}
\text{\texttt{smallstep\_sem\_send}} : & \forall e. \text{\texttt{pid x y l1' v}}
\end{align*}
In previous chapter we have defined (also in Coq) a natural semantics of the same subset of the PUB. This semantics has been proved (also using Coq) to be deterministic and we have following lemmas by induction/co-induction:

**Lemma 4.3.1** $\Rightarrow$ and $\Downarrow$ (resp. $\Rightarrow_{\infty}$ and $\Downarrow_{\infty}$) are equivalent.

Thus:

**Lemma 4.3.2** $\Rightarrow$ is deterministic.

**Lemma 4.3.3** $\Rightarrow_{\infty}$ and $\Downarrow_{\infty}$ are mutually exclusive.
High-performance operational semantics

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. 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 as state of the network and OS during execution of the program.

Note that high-performance routines do not put in the environment a value but a variable that is a pointer to the value. In this way, values sent asynchronously with special rules.

Also, we note the high-performance semantics as the small-step one and $\Rightarrow_{hp}$ and $\rightarrow_{hp}$ if we need to distinguish them.

5.1 Local rules

Rules for the local computations are given in Figure 5.1 and Figure 5.2. In Figure 5.1 the rules describe the control flow, as in the classical semantics of IMP. In Figure 5.2 we show the semantics for the PUB-specific instructions. In the case of $\text{send}$, $\text{put}$, $\text{get}$, . . . the rule just adds a message in the environment before it is actually sent by the global communication rules.

5.2 Global Reductions and Communications

PUB programs are SPMD so a configuration of the parallel machine is represented by a $p$-vector of instructions, stores, communications, received values, and numbers of messages received:

$$(n_0, E_0, C_0, R_0, c_0, \cdots | n_{p-1}, E_{p-1}, C_{p-1}, R_{p-1}, c_{p-1})$$

A final configuration is an empty set of instructions (with their environment) on all processors:

$$(n_0, E_0, C_0, R_0, \text{skip} | n_{p-1}, E_{p-1}, C_{p-1}, R_{p-1}, \text{skip})$$

The global reductions call the local ones with this rule:

$$(\cdots | n_i, E_i, C_i, R_i, c_i) \xrightarrow{hp} (\cdots | n_i, E_i', C_i', R_i', c_i')$$

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 $\text{skip}$. Asynchronous communications are done with these rules:

$$(\cdots | n_i, E_i, C_i, R_i, c_i | \text{hpsend}, j, x, \leftarrow, R_j, c_j, \cdots) \xrightarrow{hp} (\cdots | n_i + 1, E_i, C_i, R_i, c_i | \text{hpsend}, j, x, \leftarrow, R_j, c_j, \cdots)$$

$$(\cdots | n_i, E_i, C_i, R_i, c_i | \text{hpput}, j, y, \leftarrow, R_j, c_j, \cdots) \xrightarrow{hp} (\cdots | n_i + 1, E_i, C_i, R_i, c_i | \text{hpput}, j, y, \leftarrow, R_j, c_j, \cdots)$$

$$(\cdots | n_i, E_i, C_i, R_i, c_i | \text{hpget}, j, x, \rightarrow, \rightarrow, R_j, c_j, \cdots) \xrightarrow{hp} (\cdots | n_i + 1, E_i, C_i, R_i, c_i | \text{hpget}, j, x, \rightarrow, \rightarrow, R_j, c_j, \cdots)$$
That is \texttt{hpsend} sends the value pointed by \texttt{x} to the memory \texttt{E}_j of processor \texttt{j}, \texttt{hpput} writes the value to the memory at destination and \texttt{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:

\begin{align*}
\langle \langle E, C, R, c_1 \rangle \rightarrow \langle E', C', R', c'_1 \rangle \\
\langle E, C, R, c_1; c_2 \rangle \rightarrow \langle E', C', R', c'_1; c'_2 \rangle
\end{align*}

\texttt{E}, \texttt{R} \xrightarrow{\text{true}} e \parallel v
\texttt{E}, \texttt{R} \xrightarrow{\text{false}} (\texttt{E}, \texttt{C}, \texttt{R}, \text{if } e \text{ then } c_1 \text{ else } c_2 \text{ endif}) \rightarrow (\texttt{E}, \texttt{C}, \texttt{R}, c_1)
\texttt{E}, \texttt{R} \xrightarrow{\text{true}} (\texttt{E}, \texttt{C}, \texttt{R}, \text{while } e \text{ do } \texttt{c} \text{ done}) \rightarrow (\texttt{E}, \texttt{C}, \texttt{R}, \text{if } e \text{ then } (\texttt{c}; \text{ while } e \text{ do } \texttt{c} \text{ done}) \text{ else } \texttt{skip} \text{ endif})
\texttt{E}, \texttt{R} \xrightarrow{\text{true}} \texttt{E}, \texttt{C}, \texttt{R}, \text{while } \texttt{c} \text{ do } \texttt{done} \rightarrow (\texttt{E}, \texttt{C}, \texttt{R}, \text{if } e \text{ then } (\texttt{c}; \text{ while } e \text{ do } \texttt{c} \text{ done}) \text{ else } \texttt{skip} \text{ endif})
\texttt{E}, \texttt{R} \xrightarrow{\text{true}} (\texttt{E}, \texttt{C}, \texttt{R}, \text{while } e \text{ do } \texttt{c} \text{ done}) \rightarrow \texttt{E}, \texttt{C}, \texttt{R}, \text{if } e \text{ then } (\texttt{c}; \text{ while } e \text{ do } \texttt{c} \text{ done}) \text{ else } \texttt{skip} \text{ endif})
\texttt{E}, \texttt{R} \xrightarrow{\text{true}} (\texttt{E}, \texttt{C}, \texttt{R}, \text{while } e \text{ do } \texttt{c} \text{ done}) \rightarrow (\texttt{E}, \texttt{C}, \texttt{R}, \text{if } e \text{ then } (\texttt{c}; \text{ while } e \text{ do } \texttt{c} \text{ done}) \text{ else } \texttt{skip} \text{ endif})
\texttt{E}, \texttt{R} \xrightarrow{\text{true}} (\texttt{E}, \texttt{C}, \texttt{R}, \text{while } e \text{ do } \texttt{c} \text{ done}) \rightarrow (\texttt{E}, \texttt{C}, \texttt{R}, \text{if } e \text{ then } (\texttt{c}; \text{ while } e \text{ do } \texttt{c} \text{ done}) \text{ else } \texttt{skip} \text{ endif})
during the communications. It modifies the environment of each processor \( i \) such that \( \text{Comm}(\mathcal{C}_i, \mathcal{R}_i, \mathcal{E}_i) = (\mathcal{C}'_i, \mathcal{R}'_i, \mathcal{E}'_i) \) as follows: \( \mathcal{C}'_i = \emptyset \) and

\[
\mathcal{R}'_i = \bigcup_{j=0}^{p-1} \bigcup_{n=0}^{n_j} \{ j, n + \sum_{a=0}^{j} n_{a, v} \} \text{ if } \{ \text{send}, i, v \} \in \mathcal{C}_j
\]

for BSMP, that is we suppose that each processor \( j \) has sent \( n_j \) messages to \( i \) and thus we take the \( n \)th message from this ordering set. DRMA accesses are defined as follow:

\[
\mathcal{E}'_i = E_i \left[ \begin{array}{c}
\{ x \mapsto v & \in \mathcal{E}_i \text{ if } \{ \text{get}, j, x, v \} \in \mathcal{C}_j \\
\{ y \mapsto v' & \in \mathcal{E}_i \text{ if } \{ \text{put}, i, y, v' \} \in \mathcal{C}_j \\
\{ \text{hpget}, j, x, v \} & \notin \mathcal{E}_i \\
\{ \text{hpsend}, j, v \} & \notin \mathcal{E}_i
\end{array} \right]
\]

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

For the oblivious synchronisation we use this rule:

\[
\{ \cdots \| n_i, E_i, C_i, R_i, \text{oblsync}(v); c_i \| \cdots \} \Rightarrow \{ \cdots \| 0, \text{Comm}(E_i, C_i, R_i), c_i \| \cdots \} \text{ with } E_i, R_i \models \mathcal{E}_i \Rightarrow \mathcal{E}_i + \| \text{Comm}(E_i, C_i, R_i) \|
\]

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 \(|\text{Comm}(E_i, C_i, R_i)|\). We note \( \Rightarrow \) for a finite derivation and \( \Rightarrow \) for an infinite one. \( \Rightarrow \) (resp. \( \Rightarrow \)) is defined by induction (resp. by co-induction):

\[
\begin{align*}
\forall i \{ \cdots \| n_i, E_i, C_i, R_i, \text{skip} \| \cdots \} & \Rightarrow \{ \cdots \| n_i, E_i, C_i, R_i, \text{skip} \| \cdots \} \\
\forall i \{ \cdots \| n_i, E_i, C_i, R_i, c_i \| \cdots \} & \Rightarrow \{ \cdots \| n_i, E'_i, C'_i, R'_i, c'_i \| \cdots \} \Rightarrow \{ \cdots \| n_i, E''_i, C''_i, R''_i, \text{skip} \| \cdots \}
\end{align*}
\]

That is, execution of a program is complete in the final configuration case or there exists a reduction step or the program diverges. 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. We give here some intuitions of this development. In the Coq specification, the dynamic semantics are encoded as inductive predicates. Each defining case of each predicate corresponds exactly to an inference rule in the conventional, on-paper presentation of our semantics. For example, we have one inference rule for each kind of expression and statement. We do not list all the inference rules by lack of space, but show some representative examples.

We note \( \text{instr} \) for a list of instructions \( \text{envmt} \) for environments. We also use \( \text{eq} \_ \text{envmt} \) for equivalence of environments. \( p \)-vectors are represented as functions from \( Z \) to instructions or environments. The two rules of global reduction \( \Rightarrow \) (doing local calculations and communications) are modelled with the inductive \( \text{smallstep} \_ \text{sem} \) as follow:

\[
\begin{align*}
\textbf{Inductive} & \quad \text{smallstep} \_ \text{sem} : (Z \rightarrow \text{envmt}) \rightarrow (Z \rightarrow \text{instr}) \\
& \quad \Rightarrow (Z \rightarrow \text{envmt}) \rightarrow (Z \rightarrow \text{instr}) \rightarrow \text{Prop} := \\
& \quad \text{smallstep} \_ \text{sem} \_ \text{local} : \forall i \text{ envl } i \text{ env2 } i 2 \text{ envl } i \text{ i2i }, \\
& \quad \text{smallstep} \_ \text{sem} \_ \text{local} i \text{ envl } i \text{ env2 } i 2 i \rightarrow \\
& \quad \text{eq} \_ \text{envmt} \text{ envl } i (\text{env1 } i) \rightarrow \text{eq} \_ \text{envmt} \text{ env2 } i (\text{env2 } i) \rightarrow \text{eq} \_ \text{envmt} \text{ envl } i (\text{env1 } n) (\text{env2 } n) \rightarrow (i \text{ i = i2 }) \rightarrow \\
& \quad \text{smallstep} \_ \text{sem} \_ \text{local} \text{ envl } i \text{ env2 } i 2 \rightarrow \forall i \text{ i } \rightarrow (i \text{ i = skip} \rightarrow \text{smallstep} \_ \text{sem} \_ \text{local} \text{ envl } i 2 \text{ env2 } i 2 ) \rightarrow \\
\end{align*}
\]
is then defined by a inductive on this reduction. The inductive smallstep_sem_l defines a local reduction, that is execution of a communication routine or of a sequential control flow. We give here an interesting part of its Coq’s definition:

\[
\text{Inductive smallstep_sem_l (i:Z): envnmnt_HP \rightarrow instr_HP \rightarrow envnmnt_HP \rightarrow instr_HP \rightarrow Prop :=}
\]

\[
(\text{ Sequence, with } i1 = \text{ skip } =)
\]
\[
| \text{ smallstep_sem_sequence1 : } \forall i2 \text{ env} ,
| \text{ smallstep_sem_l i env (sequence_HP skip_HP i2) env i2}
\]

\[
(\text{ Sequence, with } i1 \rightarrow r1 =)
\]
\[
| \text{ smallstep_sem_sequence2 : } \forall i1 i2 r1 \text{ env1 env2} ,
| \text{ smallstep_sem_l i env1 i1 env2 r1} \rightarrow
| \text{ smallstep_sem_l i env1 (sequence_HP i1 i2) env2 (sequence_HP r1 i2)}
\]

\[
(\text{ hpput instruction =})
\]
\[
| \text{ smallstep_sem_hpput : } \forall e \text{ pid x y env env’},
| \text{ eval i env e (xvalue pid) } \rightarrow
| \text{ updatehpput env pid x y env’ } \rightarrow
| \text{ smallstep_sem_l i env (put_HP e x y) env’ skip_HP}
\]

Now we have following lemmas:

**Lemma 5.3.1** \( \Rightarrow \) and \( \Rightarrow_{hp} \) (resp. \( \Rightarrow_{\infty} \) and \( \Rightarrow_{hp} \)) are equivalent for any program that do not used high performance routines.

Thus:

**Lemma 5.3.2** \( \Downarrow \) and \( \Rightarrow_{hp} \) (resp. \( \Downarrow_{\infty} \) and \( \Rightarrow_{hp} \)) are equivalent for any program that do not used high performance routines.

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

**Lemma 5.3.3** \( \Rightarrow_{hp} \) is deterministic for any program that do not used high performance routines.

**Lemma 5.3.4** \( \Rightarrow_{hp} \) and \( \Rightarrow_{\infty} \) are mutually exclusive for any program that do not used high performance routines.

**Lemma 5.3.5** \( \Rightarrow_{hp} \) is not deterministic in general.

Take for example, the simple following programs:

\[
\begin{align*}
&\text{declare } x := \text{ pid begin} \\
&\text{declare } y := 1 \text{ begin} \\
&\text{push}(x); \\
&\text{hpput}((\text{pid} + 1) \mod nprocs, x, x); \\
&x := x + 1 \\
&\text{sync}; \\
&y := x \\
&\text{end} \\
&\text{end}
\end{align*}
\]

For each processir, it is impossible to know which value (\( \text{pid}, \text{pid} + 1 \) or \( \text{pid} - 1 \)) is affected to \( y \). Thus, deadlocks can occur in we have a number of super-step that depends of \( y \). Also:

**Lemma 5.3.6** \( \Rightarrow_{hp} \) and \( \Rightarrow_{\infty} \) are not mutually exclusive in general.

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 \( \text{pid} \), we will have a case where it is an infinite loop for some processors.
5.4 Transformation of the source code

5.4.1 Generalities

Proving a program using only basic operations of the PUB library can be done with the natural semantics, and thus is much simpler than the proofs of programs using high performance functions.

It appears then that it would be easier to build a program using only safe operations, to prove this program, and then to optimize it automatically, in order to convert standard operations into high performance operations when it’s possible.

In the general case, knowing if it is possible to replace a standard operation by a high-performance one is undecidable. In this report, we will only consider detecting some simple cases but still frequently enough in practice to be useful (all BSP algorithms that have a constant number of super-steps, e.g. parallel sorting, FFT, some graph algorithms etc.).

A standard operation in a program to be optimized is one of those :

- The operation is optimizable, and detected by the optimization function. It’s the ideal case.
- The operation is optimizable, yet not detected. We try to write an optimization function so that this case is kept as rare as possible. Such a situation does not change the validity of the optimization function.
- The operation is not optimizable, and not detected. There is nothing we can do to improve the program in such a case.
- The operation is not optimizable, yet detected as optimizable. This case must never happen, or else the optimization function is not correct.

5.4.2 Optimization Conditions

The main difference between the put instruction and the hpput instruction is that, for the put instruction, there is no buffering, neither at source nor at destination.

When using the hpput instruction, correct data delivery is only guaranteed if: (1) no communications alter the source area; (2) no subsequent local computations alter the source area; (3) no other communications alter the destination area (4) no computation on the remote process alters the destination area during the entire superstep.

Conditions to replace safely the routine get by hpget are close to these ones. For the send instruction, we suppress only one buffer on the source area, so the conditions (1) and (2) are enough.

5.4.3 Overview of the Translation

In order to simplify the detection of optimizable operations, we will only consider the programs in which the instruction sync are located outside conditional instructions (while and if). The program is then composed of an alternation of sequential blocks and sync instructions:

The advantage is that during the execution, every processor does the same synchronisation at the same point of the program. This way, if in the block \( b_i \) we have an instance of \( \text{put}(x_l, y_d, \text{pid}) \), the four conditions can be translated (within the block \( b_i \)):

1. For each processor, no communication alters \( x_l \);
2. The processor does not modify \( x_l \) after the call to put;
3. For each processor, no other communication alters \( y_d \) on processor \( \text{pid} \);
4. For the processor \( \text{pid} \), there is no local modification of \( y_d \).

\[ \text{It is possible to write programs that execute different blocks of code, and account the same number of synchronization for each processor. But such programs are relatively rare, and far more difficult to analyze.} \]
From the previous unformal analysis we can deduce the following optimisation function on a block of code (to simplify the presentation we just take into account the put instruction). The function $B$ that decomposes a program into a list of blocks is can be written as follows:

\[
B(\text{sync}) = \begin{cases} T \end{cases}
\]

\[
B(i_1; \text{sync}) = a'[b_1; \ldots; b_n]^T \text{ if } B(i_1) = a[b_1; \ldots; b_n]^b
\]

\[
B(\text{sync}; i_2) = T[b_1; \ldots; b_n]^T \text{ if } B(i_2) = a[b_1; \ldots; b_n]^b
\]

\[
B(i_1; i_2) = a'[b_1; \ldots; b_n; b'_1; \ldots; b'_m]^b \text{ if } B(i_1) = a[b_1; \ldots; b_n]^b \text{ and } B(i_2) = a'[b'_1; \ldots; b'_m]^b \text{ and } (b = T) \lor (a' = T)
\]

\[
B(c) = T \quad \text{ Otherwise}
\]

where $a[b_1; \ldots; b_n]^b$ are blocks where $a$ (resp. $b$) is a boolean ($T$ or $F$) that indicates if a synchronization occurs before (resp. after) the first (resp. last) block.

This inductive function $O^{pos}$ is defined in Figure 5.3 and transform a block (as a command) to another one. It is apply to a whole block $bl$ to be analysed (search for a put instructions to be optimized) as follow $O^{pos}(bl)$ and where $pos$ contains the position in the instruction block tree of the current instruction, encoded by a list of directions from the root (1 for the first sub-tree, 2 for the second sub-tree if it exists). The functions no_modify and no_comm_target are simple in-depth searches of the instruction, to check that the matching instructions are not called.

$pos$ is useful in the call to no_modify_after, which searches only in the instructions that are executed after the put instruction. no_modify_after is in the same way defined by an in-depth recursive search.

\[\text{where } bl \text{ is the initial whole block to be analysed}\]
5.4.5 Correctness of the Translation and Coq development

Lemma 5.4.1 For the semantics \( \Rightarrow_{hp} \) and a program \( c \), if \( B(c) = [b_1; \cdots; b_n] \) then \( O^\| (b_1); \text{sync}; \cdots; \text{sync}; O^\| (b_n) \) and \( c \) holds to an equivalent result.

The proof of semantic preservation for the translation proceeds by induction over the evaluation derivation and case analysis on the last evaluation rule used. The proof shows that, assuming suitable consistency conditions over the BSP routines, the generated high-performance ones evaluate in ways that simulate the evaluation of the corresponding BSP programs.

The function that decomposes a program into blocks and the optimisation function are written recursively, according to the definitions given above, with the Fixpoint constructor. To prove the correctness of the translation, we proceed by equivalence between the different states of the formalisation.

The function that decomposes a program into blocks and the optimisation function are written recursively, according to the definitions given above, with the Fixpoint constructor. To prove the correctness of the translation, we proceed by equivalence between the different states of the formalisation.

The fact that two programs give the same results is defined by:

\[
\text{Definition } \text{sem}_{eq} \quad (i_1 : \text{instr}_{HP}) \quad (i_2 : \text{instr}_{HP}) := \forall \text{env}_1 \text{ env}_2.
\]

\[
\text{smallstep}_{\text{sem}_{star}} \quad \text{env}_1 \quad (\text{fun } _ \Rightarrow i_1) \quad \text{env}_2 \quad (\text{fun } _ \Rightarrow \text{skip}_{HP}) \longleftrightarrow
\]

\[
\text{smallstep}_{\text{sem}_{star}} \quad \text{env}_1 \quad (\text{fun } _ \Rightarrow i_2) \quad \text{env}_2 \quad (\text{fun } _ \Rightarrow \text{skip}_{HP}).
\]

First, we prove that the decomposition into blocks is correct, that is to say, the execution with the small-step semantics of the sequence of blocks gives the same results that the execution of the original source code.

Then, for a given block, we prove that if the four conditions listed above are true, the optimized code executes to the same values that the initial code would give. To conclude the proof, we show that the optimization function given above only changes the \texttt{put} instructions when the four conditions hold.
Chapter 6

Related work

6.1 Proof of BSP Programs

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], the derivation of BSP imperative programs using Hoare’s axiom semantics [7, 27, 17, 26] or “Refinement calculus” [24]. A small-steps semantics for BSPlib programs is presented in [28] but without BSMP routines, diverging or high-performances programs.

The main drawback of these approaches is that they use their own languages that are not a subset of real programming languages. Also they neither used any proof assistant (except [12]) nor implemented dedicated tools for the proofs which is a lack of safety: users make hand proofs so they are just theoretical works. No formal work has been done on optimization of communications routines.

we have presented a formal deterministic operational semantics (a natural semantics) for BSP programs and used it to prove the correctness a classical numerical computation (the N-body problem which is considered as an important Dwarf [1]) and the divergence of some programs.

6.2 Formally verified source-code transformations

There exists a considerable body of earlier work on machine-checked correctness proofs of parts of compilers (see [9] or [22] for surveys). Notably, there exists published work tending to focus on a special part of a compiler, such as optimizations (sequential) and the underlying static analyses [20] or translation of a high-level language to virtual machine code: a verified non-optimizing byte-code compilers from a subset of Java to a subset of the JVM using Isabelle/HOL is presented in [18]. Several formal semantics of C-like languages have also been defined [23].

Most of these works did not address compiler optimizations nor generation of actual machine code. The first full formal verification of an optimized compiler back-end is described in [22].

But all these works are for sequential programs. Also, as noticed in [22], shared-memory concurrency is raising serious difficulties both with the verification of concurrent programs and with the reuse, in a concurrent setting, of languages and compilers designed for sequential execution.

Another approach is to formally characterize concurrent optimizations and programs using a concurrent separation logic: an operational semantics for an extension with threads and locks of a core imperative language as been developed in [16]. This semantics is pseudo-sequential in that threads run sequentially between two operations over locks (close to the BSP model of execution). It is conceivable that, for a fixed but arbitrary oracle, the proofs of semantics preservation would still hold: the concurrent separation logic would guarantee that this pseudo-sequential semantics for the generated code captures all possible actual executions of the source code. But, to our knowledge, this work has not be done and is limited to shared-memory concurrency. A BSP program is more portable and could be executed both to share or distributed memories.

1But how optimized bulk-sending phases of communication on specific architectures is the main work of BSPlib implemenations.
A work that is close to ours is that of [19]. Using Isabelle/HOL, they formalize the semantics of C0 (a subset of the C language, close to Pascal) and a compiler from C0 down to DLX assembly code. They provide both a big step semantics and a small step semantics for C0, the latter enabling reasoning about non-terminating and concurrent executions.

Our approach has the advantage to be simpler than concurrent programming: we used a structured parallelism that is the BSP model of execution. But that shows that the simpler optimizations of how processors exchanged data generate hard proofs\(^2\)(proof of parallel programs is also hard as show in [13]).

\(^2\)Note that the formal proof of data exchange protocols on specific hardwares is also a serious and very hard work to be done
Chapter 7

Conclusion and future work

7.1 Conclusion

Formal methods in general and program proof in particular are increasingly being applied to software. These applications create a strong need for on-machine formalization and verification of programming language semantics and tools such as compilers and static analyzers. In particular, formal operational semantics are required to validate the logic of programs (e.g. axiomatic semantics) used to reason about programs. As for tools, the formal certification of compilers (that is, a proof that the generated executable code behaves as prescribed by the semantics of the source program) is needed to ensure that the guarantees obtained by verification of the source program carry over to the executable code.

In this report, we have presented formal operational semantics for BSP programs which also introduces high-performances primitives. For programs that do not used high-performances routines, the semantics are deterministic. 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.

We have used the natural semantics to prove the correctness of a classical numerical computation (the N-body problem which is considered as an important Dwarf [3]) and the divergence of some programs.

We have also give a simple function of optimization of the source code that generated some calls to high-performances routines in place of BSP classical ones. An originality of this paper is that the semantics of the language as well as the function of optimization have been written in the specification language of the Coq proof assistant. The proof of observational semantic equivalence between the source and generated code has been machine-checked using Coq which ensures a better trust in the results. An executable compiler can be obtained by automatic extraction of executable Caml code from Coq.

From a formal methods standpoint, this work is interesting in two respects. First, parallel optimizers are complex programs that perform sophisticated symbolic computations. Their formal verification is challenging, requiring difficult proofs by induction that are beyond the reach of many program provers. Second, Coq development is an indirect but original way to validate the semantics of the source language: It is relatively easy to formalize an operational semantics, but much harder to make sure that this semantics is correct and captures the intended meaning of programs. In our experience, proving the correctness of this translator, detects many small errors in the semantics and therefore generates additional confidence in both.

This work is our first experiment to create a certified software for optimization: transforming some buffered operations to unbuffered ones. Many work is necessary to optimized more programs (as BSP synchronisations to oblivious ones) and certify this translator. The formal operational semantics would helped to prove the needed equivalence.

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.
7.2 Future work

The authors know that proving correctness of BSP computations only using semantics in Coq is a too tedious work. But, it is intended to be the basis of better tools for the proof of BSP programs. In particular, it would be interesting to develop axiomatic semantics for our language and validate them against our operational semantics. We are thinking about extending the theoretical work of [10] and its C application software [11] which generates lemmas to be proved (using a proof assistant) from Hoare’s assertions in C programs that ensure correctness (using a formal semantics).

In final, adapting all these works to MPI would be a great challenge.
Bibliography


