From BSP Routines to High-performance ones: Formal Verification of a Transformation Case

Jean Fortin and Frédéric Gava

Laboratory of Algorithms, Complexity and Logic, University of Paris-East
jean.fortin@ens-lyon.org and gava@univ-paris12.fr

Abstract. PUB (Paderborn University BSPLib) is a C library supporting the development of Bulk-Synchronous Parallel (BSP) algorithms. We present a formal semantics that emphasises the high-performance primitives of the PUB. This semantics is here used to formally verify (using the Coq proof assistant) a simple optimization of the source code that transforms classical BSP routines to their high-performance versions.
Key words: BSP programming, proofs, Coq, semantics, optimisation.

1 Introduction

General framework. Compilers of parallel languages 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 [10], bugs like deadlocks can occur, when the compiler silently generates an incorrect executable for a correct source.

Writing parallel programs is known to be notoriously 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 has been done [4] with intuitive explanations on their reliability. To our knowledge, no such tools have been formally verified.

BSP Framework. BSP[1] is a parallel model which allows an estimation of the execution time on a wide variety of architectures. The BSP model can also be

[1] We refer to [1,12] for a gentle introduction to the BSP model.

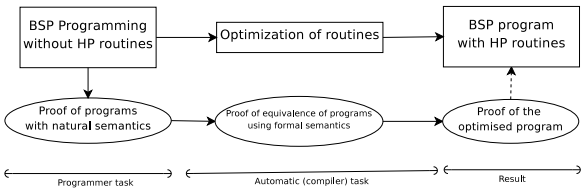


Fig. 1. Scheme of a safe environment of BSP programming

well adapted to Grid-computing and is generally presented as a bridging and emerging model for high-performance computations. The PUB [2] is a C library of BSP communication routines[2].

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. 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. A natural semantics of classical BSP routines with application to the correctness of a scientific BSP computation could be found in [6] and the Coq development is available at `http://lacl.univ-paris12.fr//gava/bsp-sem.tar`.

Moreover, BSP libraries provide high-performance (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. Replacing classical BSP routines by their high-performance pendant is thus of the responsibility of the programmer or of a non-formally verified compiler analyser as those of [4].

That is disappointing if we look for an environment where programmers can write and execute their parallel scientific programs in a safe and efficient way. Figure 1 resumes this wanted environment where first part is the subject of [6].
Outline. The aim of this paper is the Coq development of a formal operational semantics (Section 3) for both BSMP and DRMA programming styles of a kernel imperative language (Section 2) with classical and high-performance BSP primitives. As an application of this semantics, we have done a formal verification of a simple optimization of the source code that transforms some classical BSP routines by high-performance ones. That is, the original source code and the faster generated one are semantically equivalent (Section 4). Related work is discussed in Section 5, followed by conclusions and future work in Section 6.

2 BSP Core Language

Our core language is the classical IMP with a set Exp of expressions (integers, matrix, *etc.*) with operations on them. Set X of variables is a subset of Exp

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

with two special variables: **pid** and **nprocs**. The syntax is as follows:

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

with $x, y \in X$ and $e \in Exp$. Expressions are evaluated to values v (subset of Exp) and we write: $\mathcal{E}_i, \mathcal{R}_i \models^{i,p} e \Downarrow v$ 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 $\mathcal{E}_i, \mathcal{R}_i \models^{i,p} \mathbf{pid} \Downarrow i$ and $\mathcal{E}_i, \mathcal{R}_i \models^{i,p} \mathbf{nprocs} \Downarrow p$. Evaluation of Exp is not total (e.g. evaluation of $1 + \mathbf{true}$) but for simplicity always terminates. Parallel operations are[3]:

$\mid \mathbf{sync}$	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

Sending a single message is done using **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 **push** and **pop**. DRMA operations are **put** and **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 **sync** which blocks the node until all other nodes have called **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 (**char** $*$). Exp is extended with $\mathbf{findmsg}(i, e)$ that finds the eth message of processor i of the previous super-step and **nmsg** that returns the arity of $\mathcal{R}_i$.

All BSP primitives of communications have thus their high performance version (called **hpput**, **hpget** and **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 depandant). It is no surprise to say that high-performance operations improve speedup·

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

3 High Performance semantics

In [6] the dynamic semantics is specified using a big-step operational semantics. This choice simplifies greatly the semantics and the proof of programs. We used here a small-step semantics that emphasizes the high-performance (HP) routines. by specifing the operation of a program one step at a time.

There is thus a set of rules that we continue to apply to configurations until reaching a final configuration if ever. In our case, we will have two kinds of reductions: local ones (on each processor) and global ones (for the whole parallel machine). The main property is that HP routines are non-deterministic and communications can be performed at any time.

We denote $\mathcal{E}[x/v]$ the insertion or substitution in $\mathcal{E}$ of a new binding from x to v. We denote $\mathcal{R}$ the received values of the previous super-step. The communications environment $\mathcal{C}$ contains messages to be sent in the current super-step (noted with $\leftarrow$) and asynchronous messages received from other processors (noted with $\rightarrow$). 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. Note that HP routines do not put in the environment a value but a variable that is a pointer to the value: values are sent asynchronously with special rules.

Rules for the local computations are given in Figure 2 and Figure 3. In Figure 2 the rules describe the control flow, as in the classical semantics of IMP. In Figure 3, we show the semantics for the PUB-specific instructions. In the case of **send**, **put**, **get**, ... the rule just adds a message in the environment before it is actually sent by the global communication rules.

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

$$\langle \mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0, c_0 \| \cdots \| \mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1}, c_{p-1} \rangle$$

A final configuration is an empty set of instructions (with their environment) 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 **skip**. Asynchronous communications are done with these rules:

$$\begin{array}{c} \langle \cdots \| \mathcal{E}_i, \mathcal{C}_i \cup \{\mathbf{hpsend}, j, x, \leftarrow\}, \mathcal{R}_i, c_i \| \cdots \rangle \text{ where } \{x \mapsto v\} \in \mathcal{E}_i \\ \rightharpoonup \langle \cdots \| \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \| \mathcal{E}_j, \mathcal{C}_j \cup \{\mathbf{hpsend}, j, x, \rightarrow\}, \mathcal{R}_j, c_j \| \cdots \rangle \end{array}$$

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

$$\begin{array}{c} \langle \cdots \| \mathcal{E}_i, \mathcal{C}_i \cup \{\mathbf{hpget}, j, x, \overline{y}, \leftarrow\}, \mathcal{R}_i, c_i \| \cdots \| \mathcal{E}_j, \mathcal{C}_j, \mathcal{R}_j, c_j \| \cdots \rangle \text{ where } \{\overline{y} \mapsto v\} \in \mathcal{E}_j \\ \rightharpoonup \langle \cdots \| \mathcal{E}_i[x/v], \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \| \mathcal{E}_j, \mathcal{C}_j, \mathcal{R}_j, c_j \| \cdots \rangle \end{array}$$

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.

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

$$\frac{\mathcal{E}, \mathcal{R} \overset{i,p}{\models} e \Downarrow v}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, x := e\rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}[x/v], \mathcal{C}, \mathcal{R}, \textbf{skip}\rangle} \qquad \frac{}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \textbf{skip}; c_2\rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}, \mathcal{R}, c_2\rangle}$$

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

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

$$\frac{}{\langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \textbf{while } e \textbf{ do } c \textbf{ done}\rangle \underset{i,p}{\rightharpoonup} \langle \mathcal{E}, \mathcal{C}, \mathcal{R}, \textbf{if } e \textbf{ then } (c; \textbf{while } e \textbf{ do } c \textbf{ done}) \textbf{ else skip endif}\rangle}$$

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

Fig. 2. Reduction rules of sequential control flow

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

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

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

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

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

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

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

Fig. 3. Reduction rules of the PUB's routines

When all asynchronous communications have been done, synchronous communications and BSP synchronisation is done with this rule:

$$\begin{aligned}&\langle \mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0, \textbf{sync}; c_0 \| \cdots \| \mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1}, \textbf{sync}; c_{p-1}\rangle \\ &\quad \rightharpoonup \langle Comm(\mathcal{E}_0, \mathcal{C}_0, \mathcal{R}_0), c_0 \| \cdots \| Comm(\mathcal{E}_{p-1}, \mathcal{C}_{p-1}, \mathcal{R}_{p-1}), c_{p-1}\rangle \\ &\text{if } \forall i, j, x, y, v \ \{\textbf{hpsend}, j, v\} \notin \mathcal{C}_i \ \wedge \ \{\textbf{hpget}, j, x, v\} \notin \mathcal{C}_i \ \wedge \ \{\textbf{hpput}, j, y, v\} \notin \mathcal{C}_i\end{aligned}$$

That is if each processor is in the **sync** case, communications are done using the *Comm* function that exchanges the messages, which finishes the cur-

rent super-step. The $Comm$ function specifies the order of the messages during the communications. It modifies the environment of each processor i such that $Comm(\mathcal{C}_i, \mathcal{R}_i, \mathcal{E}_i) = (\mathcal{C}'_i, \mathcal{R}'_i, \mathcal{E}'_i)$ as follows: $\mathcal{C}'_i = \emptyset$ and

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

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

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

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

We denote $\Rightarrow$ for a finite derivation and $\overset{\infty}{\Rightarrow}$ for an infinite one. $\Rightarrow$ (resp. $\overset{\infty}{\Rightarrow}$) is defined by induction (resp. by co-induction):

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

$$\frac{\forall i \ \langle \cdots \| \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \rangle \rightharpoonup \langle \cdots \| \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, c'_i \| \cdots \rangle \quad \begin{array}{l} \langle \cdots \| \mathcal{E}'_i, \mathcal{C}'_i, \mathcal{R}'_i, c'_i \| \cdots \rangle \\ \Rightarrow \langle \cdots \| \mathcal{E}''_i, \mathcal{C}''_i, \mathcal{R}''_i, \mathbf{skip} \| \cdots \rangle \end{array}}{\langle \cdots \| \mathcal{E}_i, \mathcal{C}_i, \mathcal{R}_i, c_i \| \cdots \rangle \Rightarrow \langle \cdots \| \mathcal{E}''_i, \mathcal{C}''_i, \mathcal{R}''_i, \mathbf{skip} \| \cdots \rangle}$$

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

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".

As mentioned, the semantics was developed using the Coq. We give here some intuitions of this development. The full development is available at `http://lacl.univ-paris12.fr/gava/bsp-hp.tar`. 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 the inference rules for lack of space. p-vectors are represented as functions from **Z** (Coq'q integer) to instructions or environments.

Lemma 1. *$\Rightarrow$ is deterministic for programs that do not used HP routines.*

Lemma 2. *$\Rightarrow$ is not deterministic.*

Take for example, the simple following program:

```
declare x := pid begin  declare y := 1 begin
  push(x); hpput((pid + 1) mod nprocs, x, x); x := x + 1; sync; y := x
end  end
```

It is impossible to know which value (**pid**, **pid** + 1 or **pid** − 1) is affected to y.

Lemma 3. *$\Rightarrow$ and $\overset{\infty}{\Rightarrow}$ are mutually exclusive.*

4 Transformation of the source code

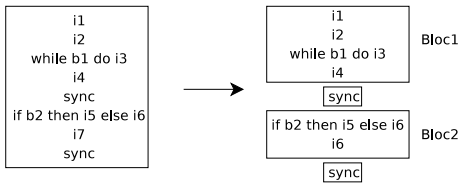
In the general case, knowing if it is possible to replace a standard operation by a high-performance one is undecidable. In this paper, we will only consider detecting some simple cases but still frequent enough in practice to be useful

(all BSP algorithms that have a constant number of super-steps, *e.g.* parallel sorting, FFT, some graph algorithms *etc.*).

Optimization Conditions. 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.

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 **put**(x_l, y_d, pid), the four conditions can be translated (within the block b_i): (1) for each processor, no communication alter 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 pid; (4) for the processor pid, there is no local modification of y_d. The optimisation function must comply with those four conditions. For (1) the check is easy by searching the block of code. For (2) it is necessary to analyse the control flow in order to know which instructions are to be executed after the call to the **put** instruction.

For (3), a problem is raised by the determination of the target pid, which might be computed by a complex method. Thus, it is practically impossible to statically check this exact condition. However, when we are in one of the following two cases, the analysis is made possible: no communication alters y_d, on any processor. and the pid are computed by simple arithmetic expressions, for which it is possible to obtain the result by a direct analysis[4]. In practice, both cases cover a significant number of programs. For (4), there is the same problem with the pid determination. We treat the programs in the same cases than previously. For other instructions **get** and **send**, the technique is identical.

Optimisation of a block. From the previous unformal analysis we can deduce the following optimisation function on a block of code (to simplify the

[4] For instance, a systolic algorithm will often have communications done by a processor to his neighbour, which is computed by the expression $(mypid + 1)\%nprocs$.

$$
\begin{aligned}
O^{pos}(c_1; c_2) &= O^{1::pos}(c_1); O^{2::pos}(c_2) \\
O^{pos}(\textbf{while}\, b\, \textbf{do}\, c\, \textbf{done}) &= \textbf{while}\, b\, \textbf{do}\, O^{1::pos}(c)\, \textbf{done} \\
O^{pos}(\textbf{if}\, b\, \textbf{then}\, c_1\, \textbf{else}\, c_2\, \textbf{endif}) &= \textbf{if}\, e\, \textbf{then}\, O^{1::pos}(c_1)\, \textbf{else}\, O^{2::pos}(c_2)\, \textbf{endif} \\
O^{pos}(c) &= c \text{ when } c \in \{\textbf{skip}, \textbf{get}, \textbf{send}, \textbf{hpput}, \textbf{hpsend}, \textbf{hpget}\} \\
O^{pos}(x := e) &= x := e \\
O^{pos}(\textbf{declare}\, y := e\, \textbf{begin}\, c\, \textbf{end}) &= \textbf{declare}\, y := e\, \textbf{begin}\, O^{pos}(c)\, \textbf{end}
\end{aligned}
$$

$$
O^{pos}(\text{put}(xl, yd, pid)) = \begin{cases} \textbf{hpput}(xl, yd, pid) \text{ if } \begin{array}{l} \texttt{no\_comm\_target}(bl, xl) \wedge \\ \texttt{no\_modify\_after}(bl, pos, xl) \wedge \\ \texttt{no\_comm\_target}(bl, yd) \wedge \\ \texttt{no\_modify}(bl, yd)) \end{array} \\ \textbf{put}(xl, yd, pid) \qquad \text{Otherwise} \end{cases}
$$

where bl is the initial whole block to be analysed

Fig. 4. Optimisation of a block

presentation we just take into account the **put** instruction). The function B that decomposes a program into a list of blocks can be written as follows:

$$
\begin{aligned}
B(\textbf{sync}) &= {}^{T}[]^{T} \\
B(i_1; \textbf{sync}) &= {}^{a}[b_1; \ldots; b_n]^{T} \text{ if } B(i_1) = {}^{a}[b_1; \ldots; b_n]^{b} \\
B(\textbf{sync}; i_2) &= {}^{T}[b_1; \ldots; b_n]^{b} \text{ if } B(i_2) = {}^{a}[b_1; \ldots; b_n]^{b} \\
B(i_1; i_2) &= {}^{a}[b_1; \ldots; b_n; b'_1; \ldots; b'_n]^{b} \\
&\quad \text{if } B(i_1) = {}^{a}[b_1; \ldots; b_n]^{b} \text{ and } B(i_2) = {}^{a'}[b'_1; \ldots; b'_m]^{b'} \text{ and } (b = T) \vee (a' = T) \\
B(c) &= {}^{F}[c]^{F} \quad \text{Otherwise}
\end{aligned}
$$

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 4 and transforms 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 $O^{[]}(bl)$ and where *pos* coutains 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 searchs 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.

Lemma 4. *For the semantics* $\Rightarrow$ *and a program c, if* $B(c) = [b_1; \cdots; b_n]$ *then* $O^{[]}(b_1); \textbf{sync}; \cdots; \textbf{sync}; O^{[]}(b_n)$ *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.

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 evaluates 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 **put** instructions when the four conditions hold.

5 Related work

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 [5] or the derivation of BSP imperative programs using Hoare's axiom semantics [3]. A small-step semantics for BSPlib programs is presented in [13] but without BSMP routines, diverging or high-performance 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 [5]).

Formally verified source-code transformations. There exists a considerable body of earlier work on machine-checked correctness proofs of parts of compilers (see [10] for surveys). Notably, there exists published work tending to focus on a special part of a compiler, such as the underlying static analyses [9] or translation of a high-level language to virtual machine code [7]. Several formal semantics of C-like languages have also been defined [11].

But all these works are for sequential programs. Also, as noticed in [10], 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.*

A work that is close to ours is that of [8]. 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 (BSP execution). But that shows that the simpler optimizations of how processors exchanged data generate hard proofs.

6 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.

In this paper, we have presented a formal operational semantics for BSP programs which also introduces high-performance primitives. We have also given

a simple transformation of the source code that generated some calls to high-performance routines in place of BSP classical ones. An originality of this paper is that the semantics of the language as well as the transformation 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. This work is our first experiment to create a certified software for optimization: transforming some buffered operations to unbuffered ones. Much work is necessary to optimized more programs and certify this translator.

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

References

1. R. H. Bisseling. *Parallel Scientific Computation. A structured approach using BSP and MPI.* Oxford University Press, 2004.
2. O. Bonorden, B. Juurlink, I. Von Otte, and O. Rieping. The Paderborn University BSP (PUB) library. *Parallel Computing*, 29(2):187–207, 2003.
3. Y. Chen and W. Sanders. Top-Down Design of Bulk-Synchronous Parallel Programs. *Parallel Processing Letters*, 13(3):389–400, 2003.
4. 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.
5. F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.
6. F. Gava and J. Fortin. Formal Semantics of a Subset of the Paderborn's BSPlib. In *PDCAT 2008*, 2008. to appear.
7. G. Klein and T. Nipkow. A machine-checked model for a java-like language, virtual machine and compiler. *ACM Transactions on Programming Languages and Systems*, 4(28):619–695, 2006.
8. D. Leinenbach, W. Paul, and E. Petrova. Towards the formal verification of a C0 compiler. In *Proc. Conf. on Software Engineering and Formal Methods (SEFM)*, pages 2–11. IEEE Computer Society Press, 2005.
9. S. Lerner, T. Millstein, E. Rice, and C. Chambers. Automated soundness proofs for dataflow analyses and transformations via local rules. In *32nd symposium Principles of Programming Languages*, pages 364–377. ACM Press, 2005.
10. Xavier Leroy. A formally verified compiler back-end. Submitted and available online at `http://gallium.inria.fr/~xleroy/publi/compcert-backend.pdf` Commented Coq development available at `http://compcert.inria.fr/`, July 2008.
11. M. Norrish. *C formalised in HOL.* PhD thesis, University of Cambridge, 1998.
12. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
13. J. Tesson and F. Loulergue. Formal Semantics for the DRMA Programming Style Subset of the BSPlib Library. In J. Weglarz, R. Wyrzykowski, and B. Szymanski, editors, *Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM 2007)*, LNCS. Springer, 2007.