Verifying Functional Bulk Synchronous Parallel Programs Using the Coq System

Frédéric Gava and Frédéric Loulergue
Laboratory of Algorithms, Complexity and Logic
University Paris XII, Val-de-Marne
61, avenue du Général de Gaulle
94010 Créteil cedex – France
Tel: +33 (0)1 45 17 16 50
Fax: +33 (0)1 45 17 66 01
{gava, loulergue}@univ-paris12.fr
TR-2003-02

Abstract

The Bulk Synchronous Parallel ML (BSML) is a functional language for Bulk Synchronous Parallel (BSP) programming. It is based on an extension of the $\lambda$-calculus by parallel operations on a parallel data structure named parallel vector, which is given by intention. We present the formal proofs of correctness of BSML programs in the Coq proof assistant. Such development demonstrates the usefulness of higher-order logic in the process of software certification and parallel applications. They also show that proof of rather complex parallel algorithms may be made with inductive types by using the certified programs.

Keywords: Parallel Programming, Bulk Synchronous Parallelism, Functional Programming, Certification, Coq, Theorem prover

1 Introduction

Some problems require performance carried out by only massively parallel computers of which programming is still difficult. Works on functional programming and parallelism can be divided in two categories: explicit parallel extensions of functional languages — where languages are either non-deterministic [40] or non-functional [2, 15] — and parallel implementations with functional semantics [1] — where resulting languages do not express parallel algorithms directly and do not allow the prediction of execution times. Algorithmic skeleton languages [9, 10, 43, 7], in which only a finite set of operations (the skeletons) are parallel, constitute an intermediate approach. Their functional semantics is explicit but, their parallel operational semantics is implicit. The set of algorithmic skeletons has to be as complete as possible, but it is often dependent on the domain of application.
The design of parallel programming languages is, therefore, a tradeoff between:

- the possibility of expressing parallel features necessary for predictable efficiency, but which make programs more difficult to write, to prove and to port
- the abstraction of such features that are necessary to make parallel programming easier, but which must not hinder efficiency and performance prediction.

We are exploring thoroughly the intermediate position of the paradigm of algorithmic skeletons in order to obtain universal parallel languages where execution cost can be easily determined from the source code (in this context, cost means the estimate of parallel execution time). This last requirement forces the use of explicit processes corresponding to the processors of the parallel machine. Bulk Synchronous Parallel (BSP) computing [37, 45] is a parallel programming model which uses explicit processes, offers a high degree of abstraction and yet, allows portable and predictable performance on a wide variety of architectures.

An operational approach has led to a BSP $\lambda$-calculus that is confluent and universal for BSP algorithms [36], and to a library of bulk synchronous primitives for the Objective Caml [30] language which is sufficiently expressive and allows the prediction of execution times [24, 34].

This framework is a good tradeoff for parallel programming because:

- we defined a confluent calculus so:
  - we can design purely functional parallel languages from it. Without side-effects, programs are easier to prove, and to re-use (the semantics is compositional)
  - we can choose any evaluation strategy for the language. An eager language allows good performances.

- this calculus is based on BSP operations, so programs are easy to port, their costs can be predicted and are also portable because they are parametrized by the BSP parameters of the target architecture.

Bulk Synchronous Parallel ML or BSML is our extension of ML for programming direct-mode parallel BSP algorithms as functional programs. A BSP algorithm is said to be in direct mode [20] when its physical process structure is made explicit. Such algorithms offer predictable and scalable performance and BSML expresses them with a small set of primitives taken from the confluent BSA$\lambda$-calculus [36]: a parallel constructor, asynchronous parallel function application, synchronous global communications and a synchronous global conditional.

There is currently no full implementation of BSML but there is a partial implementation as a library. The BSMLlib library implements the BSML primitives using Objective Caml [30] and MPI [47]. BSMLlib can be taught to BSc. students due to the small number of basic operations (universities of Orléans and Paris Val de Marne). There are additional modules which provide several usual parallel algorithms. They constitute what is called the BSMLlib standard library. In addition, its performance follows curves predicted by the BSP cost model [3]. This environment is a safe one. Our language is deterministic and is based on a parallel abstract machine [39] which has proven correct w.r.t. the confluent BS$\lambda$P-calculus [31] using an intermediate semantics [32]. A polymorphic type system [18] has been designed, for which type inference is possible.
We are now interested in the certification of BSML programs. This paper describes our first work in this direction. Section 2 presents functional bulk synchronous parallel programming. The formalization of the BSML operators in Coq is given in section 3. It is used for verifying properties of several BSML programs which are part of the standard library of BSMLlib (section 4). We end with related work (section 5) and future work (section 6).

2 Functional Bulk Synchronous Parallelism

2.1 Bulk Synchronous Parallelism

The Bulk Synchronous Parallel (BSP) model [53, 38, 45] describes an abstract parallel computer, a model of execution and a cost model. A BSP computer has three components: a homogeneous set of processor-memory pairs, a communication network allowing inter processor delivery of messages and a global synchronization unit which executes collective requests for a synchronization barrier. A wide range of actual architectures can be seen as BSP computers.

The performance of the BSP computer is characterized by three parameters (expressed as multiples the local processing speed):

- the number of processor-memory pairs \( p \)
- the time \( I \) required for a global synchronization
- the time \( g \) for collectively delivering a 1-relation (communication phase where every processor receives/sends at most one word). The network can deliver an \( h \)-relation (communication phase where every processor receives/sends at most \( h \) words) in time \( g \times h \).

Those parameters can easily be obtained using benchmarks [27].

A BSP program is executed as a sequence of super-steps, each one divided into (at most) three successive and logically disjointed phases (Fig. 1):

1. Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes;
2. the network delivers the requested data transfers;
3. a global synchronization barrier occurs, making the transferred data available for the next super-step.

The execution time of a super-step \( s \) is, thus, the sum of the maximal local processing time, of the data delivery time and of the global synchronization time:

\[
\text{Time}(s) = \max_{i: \text{processor}} w^{(s)}_i + \max_{i: \text{processor}} h^{(s)}_i \times g + l
\]

where \( w^{(s)}_i \) = local processing time on processor \( i \) during super-step \( s \) and \( h^{(s)}_i = \max\{h^{(s)}_{i+}, h^{(s)}_{i-}\} \)
where \( h^{(s)}_{i+} \) (resp. \( h^{(s)}_{i-} \)) is the number of words transmitted (resp. received) by processor \( i \) during super-step \( s \).
The execution time $\sum_s \text{Time}(s)$ of a BSP program composed of $S$ super-steps is, therefore, a sum of 3 terms:

$$W = \sum_s \max_i w_i^{(s)}$$

$$H = \sum_s \max_i h_i^{(s)}$$

$W$, $H$ and $S$ are functions of $p$ and of the size of data $n$, or of more complex parameters like data skew. To minimize execution time, the BSP algorithm design must jointly minimize the number $S$ of super-steps, the total volume $h$ with imbalance of communication and the total volume $W$ with imbalance of local computation.

Bulk Synchronous Parallelism (and the Coarse-Grained Multicomputer, CGM, which can be seen as a special case of the BSP model) is used for a large variety of applications: scientific computing [5, 28], genetic algorithms [8] and genetic programming [11], neural networks [44], parallel databases [4], constraint solvers [21], etc. It is to notice that “A comparison of the proceedings of the eminent conference in the field, the ACM Symposium on Parallel Algorithms and Architectures, between the late eighties and the time from the mid nineties to today reveals a startling change in research focus. Today, the majority of research in parallel algorithms is within the coarse-grained, BSP style, domain” [12].

2.2 Bulk Synchronous Parallel ML

There is currently no implementation of a full Bulk Synchronous Parallel ML language but rather a partial implementation: a library for Objective Caml. The so-called BSMLlib library is based on the following elements.

It gives access to the BSP parameters of the underlying architecture. In particular, it offers the function \texttt{bsp\_p:unit->int} such that the value of \texttt{bsp\_p()} is $p$, the static number of processes of the parallel machine. The value of this variable does not change during execution (for “flat” programming, this is not true if a parallel juxtaposition is added to the language [33]).

There is also an abstract polymorphic type \texttt{\‘a\ par} which represents the type of $p$-wide parallel vectors of objects of type \texttt{\‘a}, one per process. The nesting of \texttt{par} types is prohibited. Our type system enforces this restriction [18]. This improves on the earlier design DPML/Caml Flight
in which the global parallel control structure \texttt{sync} had to be prevented \textit{dynamically} from nesting.

This is very different from SPMD programming (Single Program Multiple Data) where the programmer must use a sequential language and a communication library (like MPI [47]). A parallel program is then the multiple copies of a sequential program, which exchange messages using the communication library. In this case, messages and processes are explicit, but programs may be \textit{non deterministic} or may contain \textit{deadlocks}.

Another drawback of SPMD programming is the use of a variable containing the processor name (usually called “pid” for Process Identifier) which is bound outside the source program. A SPMD program is written using this variable. When it is executed, if the parallel machine contains \(p\) processors, \(p\) copies of the program are executed on each processor with the pid variable bound to the number of the processor on which it is run. Thus parts of the program that are specific to each processor are those which depend on the pid variable. On the contrary, parts of the program which make global decision about the algorithms are those which do not depend on the pid variable. This dynamic and \textit{undecidable} property is given the role of defining the most elementary aspect of a parallel program, namely, its local vs global parts.

The parallel constructs of BSML operate on parallel vectors. Those parallel vectors are created by:

\[
\text{mkpar: (int -> 'a) -> 'a par}
\]

so that \((\text{mkpar } f)\) stores \((f i)\) on process \(i\) for \(i\) between 0 and \((p-1)\). We usually write \(f\) as \(\text{fun pid->e}\) to show that the expression \(e\) may be different on each processor. This expression \(e\) is said to be \textit{local}. The expression \((\text{mkpar } f)\) is a parallel object and it is said to be \textit{global}.

A BSP algorithm is expressed as a combination of asynchronous local computations (first phase of a super-step) and phases of global communication (second phase of a super-step) with global synchronization (third phase of a super-step). Asynchronous phases are programmed with \texttt{mkpar} and with:

\[
\text{apply: ('a -> 'b) par -> 'a par -> 'b par}
\]

apply \((\text{mkpar } f)\) \((\text{mkpar } e)\) stores \((f i)\) \((e i)\) on process \(i\). Neither the implementation of BSMLlib, nor its semantics [32] prescribe a synchronization barrier between two successive uses of \texttt{apply}.

Readers familiar with BSPlib [45, 27] will observe that we ignore the distinction between a communication request and its realization at the barrier. The communication and synchronization phases are expressed by:

\[
\text{put:(int->'a option) par -> (int->'a option) par}
\]

Consider the expression: \(\text{put(\text{mkpar(fun i->fs}_i))}\) (*)

To send a value \(v\) from process \(j\) to process \(i\), the function \(fs_j\) at process \(j\) must be such as \((fs_j i)\) evaluates to \texttt{Some} \(v\). To send no value from process \(j\) to process \(i\), \((fs_j i)\) must evaluate to \texttt{None}.

Expression (*) evaluates to a parallel vector containing a function \(fd_i\) of delivered messages on every process. At process \(i\), \((fd_i j)\) evaluates to \texttt{None} if process \(j\) sent no message to process \(i\) or evaluates to \texttt{Some} \(v\) if process \(j\) sent the value \(v\) to the process \(i\).

The full language would also contain a synchronous conditional operation:
ifat: (bool par) * int * 'a * 'a -> 'a

such that ifat (v, i, v1, v2) will evaluate to v1 or v2 depending on the value of v at process i. But Objective Caml is an eager language and this synchronous conditional operation can not be defined as a function. That is why the core BSMLlib contains the function: at:bool par -> int -> bool to be used only in the construction: if (at vec pid) then... else... where (vec:bool par) and (pid:int). if at expresses communication and synchronization phases. Global conditional is necessary of express algorithms like:

Repeat Parallel Iteration Until Max of local errors < epsilon

Without it, the global control cannot take into account data computed locally.

2.3 Advantages of Functional BSP Programming

One important benefit of the BSP model is the ability to accurately predict the execution time requirements of parallel algorithms (communications are clearly separated from synchronization, i.e. it can be performed in any order, provided that the information is delivered at the beginning of the next super-step). This is achieved by constructing analytical formulas that are parameterized by a few values which captured the computation, communication and synchronization performance of a parallel system.

These results are based on the experimental evidence that the generic collective communication pattern generated by a super-step can be routed with predictable time [19]. The maximum amount of information sent or received by each processor during a communication time-slice should be determined at run time and used by a global communication scheduling algorithm. The super-steps also separate communication and local calculus which avoid deadlocks.

The functional approach of this parallel model allows the re-use of suitable technical for formal proof from functional languages because a few numbers of parallel operators is needed to an explicit parallel extension of a functional language. Those operators (for a static number of processes) of the BSML language are derived from a confluent calculus [36] so parallel algorithms are also confluent and keep the advantages of the BSP models: no deadlock, efficient implementation using optimized communication algorithms, static cost formulae and cost previsions. A powerful axiomatization of theses parallel operators enables to express all BSP algorithms in a natural way. Thus, we could prove and certify functional implementation of those algorithms. The extraction possibility of proof assistant can be used to generate a certified library of parallel algorithms to be used independently of the sequential or parallel implementation of BSML operators [34].

3 Formalization of BSMLlib in Coq

The Coq system [51] is a proof assistant for high-order logic. It allows to write specifications and propositions, to check mathematical proofs, and to enable synthesize computer programs from proof of their specification. One can introduce new definitions and prove facts, using an interactive prover in a natural deduction method. As a typed λ-calculus, the logic of the Coq system is naturally well-suited to prove purely functional programs [41]. We show that is also possible to certify the correctness of parallel programs.
To represent our parallel language and have an specification-extraction of functional BSP programs, we have chosen a classical approach: an axiomatization of the parallel operators. Thus, operations are given by parameters and do not depend on the implementation (sequential or parallel). The formalization in Coq is based on the BSMLlib elements: the number of processes, the parallel vectors and their operators.

The number of processes is naturally an integer upper to 0 given by the parameters. Parallel vectors are indexed over type Z (Coq’s integers), starting from 0 to the constant number of processes. They are represented in the logical world by an abstract dependent type Vector T where T is the type of its elements.

Parameters Vector: Set -> Set;
  at: (T:Set) (Vector T) -> Z -> T;
  mkpar : (T:Set) (Z -> T) -> (Vector T);
  apply: (T1,T2:Set) (Vector (T1 -> T2))
  -> (Vector T1) -> (Vector T2);
  get: (T:Set) (Vector T) -> (Vector Z) -> (Vector T);
  put:(T:Set)(Vector (Z -> (Option T)))
  -> (Vector (Z -> (Option T)));
  ifat : (T:Set) (Vector bool) -> Z -> T -> T -> T;

at is an abstract “access” function for the vectors. It gives the local value contained at a processor. The operators for local computation are the constructor mkpar of parallel vectors and the global application. They are axiomatized using the Coq’s syntax:

Axiom mkpar_def: (T:Set) (f:Z -> T)(i:Z) '0 <= i < nprocs' -> (at (mkpar f) i)=(f i).
Axiom apply_def: (T1,T2:Set) (V1:(Vector (T1 -> T2)))
(V2:(Vector T1 )) (i:Z) '0 <= i < nprocs' -> (at (apply V1 V2) i)=( (at V1 i) (at V2 i)).

For a function f, mkpar_def stores (f i) on the process i by using the access function at and in the same manner apply_def stores (V1 i) (V2 i).

The communication of values [36] has been first a get operator: each process gets a unique value from another process. Its implementation and the associated axioms are easy to write. But for some algorithms (essentially optimized BSP algorithms), the dual operator, put, is necessary. The above two functions are axiomatized as follows:

Axiom get_def: (T:Set)(V1:(Vector T))(V2: (Vector Z))
(i:Z) '0 <= i < nprocs' -> ((j:Z) '0 <= j < nprocs' -> '0=< (at V2 j) < nprocs')
-> (at (get V1 V2) i)=(at V1 (at V2 i)).
Axiom put_def: (T:Set)(Vf:(Vector (Z -> (Option T))))
(i:Z) '0 <= i < nprocs' -> (at (put Vf) i)=([j:Z]
Cases (Z_le_dec '0' j) (Z_lt_dec j nprocs) of
  (left _) (left _) => ((at Vf j) i)
  _ _ => None end).

7
put_def transforms a functional vector to another functional vector which aims communications using the functional parameter \( j \) to read values from distant processes. In real implementation, the values to communicate are first calculated and then exchanged using an optimized algorithm [34].

The full axiomatization would also contain the synchronous conditional. Its axiomatization is easy to express using the \texttt{bool} library of the \texttt{Coq} system:

\begin{verbatim}
Axiom ifat_def: (T:Set) (V:(Vector bool)) (n:Z) (R_IF,R_ELSE:T)
  '0 <= n < nprocs' -> ((ifat V n R_IF R_ELSE)=
    (Cases (sumbool_of_bool (at V n)) of
    (left _) => R_IF
    | (right _) => R_ELSE end)).
\end{verbatim}

Because, in this axiomatization, \texttt{ifat} is a function, expressions using a global conditional would be extracted by the \texttt{Coq} system programs extraction also as a function. Thus, the extracted code needs to be parsed to transform the \texttt{ifat} function to a suitable conditional. Now, with the set of axioms of our parallel operators, we will be able to verify the formal properties of classical functional BSP programs.

4 Formal Proofs of BSML Programs

We present here the proof of some parallel algorithms in the \texttt{Coq} system. In order to simplify the presentation and to ease the formal reasoning, we limit our study and formal proofs to the expressions which are very common in a BSP algorithm and the two different ways to communicate values in BSML. Those case studies have demonstrated the relevance of the use of the \texttt{Coq} system in the proof of parallel programs \textit{correctness}. In particular, we used defined predicates and libraries several times in the developments and we also used higher axioms to define a new principle and to prove correctness of programs [41] and in our case of BSML programs.

To increase the readability of this section, we give the expressions in a Objective Caml [30] syntax given by the \texttt{Coq} system program extraction [42] (with minimal hand modification). The formal developments described in this section are freely available [16]. The \texttt{Coq} system and the tactics used to prove the correctness of our programs are also freely available and described at the web page of the \texttt{Coq} System [51].

4.0.1 Replicate

The first case study we present is most simple functional BSP expression: the \textit{replication} of the same value on each process. It is often used at the beginning of BSP algorithms to replicate all the parameters:

\begin{verbatim}
let replicate param = mkpar (fun x -> param);;
\end{verbatim}

The correctness (each process has the value) of this function is easily proved with an application of the \texttt{mkpar_def} axioms. It is a property which will be always used in the following.
4.0.2 Compositionality

To validate our study, we need to give formal proof of the core library. For example, we give here a definition [33] of a weak form of parallel composition analogous to a data-parallel conditional where statement to cover the whole network.

let mask c x y = apply (apply (mkpar (fun j x0 y0 ->
   if (c j) then x0 else y0)) x ) y;;

This expression has been used in [36], to prove the capacity of the equational theory of the BS\textlambda;calculus and its functional implementation. But without a proof assistant, the authors have made some bad applications of the axioms (Lemma 4, mask commutes with apply):

Lemma mask_is_composition_apply : (T:Set) (c:(Z->bool)) (f,g:Z->T->T)
   (f',g':Z->T)(i:Z) '0<=i<nprocs' ->
   (at (mask c (apply (mkpar f) (mkpar f'))) (apply (mkpar g) (mkpar g'))) i
   =
   (at (apply (mask c (mkpar f) (mkpar g)) (mask c (mkpar f') (mkpar g'))) i).

which have been proved by the Coq system by trivial application of the axioms without forgetting cases. The necessity to certify the properties of our programs will appear in the following part of this section.

4.0.3 Broadcast of a value

Exchange of values is the critical point of parallel algorithms. BSML offers two different operators for this work: get and put which are complementary. We express this by a classical example: the broadcast of an element $v$ held by a process $n$:

\begin{center}
\begin{tikzpicture}
    \node (n) at (0,0) {$n$};
    \node (v) at (-2,-1) {$v$};
    \node (v') at (2,-1) {$v'$};
    \draw (v) -- (v');
    \draw (n) -- (v); \draw (n) -- (v');
    \foreach \x in {-2,-1,0,1,2} {
        \draw (\x,-0.5) -- (\x,-1.5);
    }
\end{tikzpicture}
\end{center}

The broadcasting can be done in BSML, using the get operator, by the following function:

let broadcast_get n vect = get vect (replicate n);;;

and the BSP cost formula for a call of this function is:

\begin{equation}
p + p \times s \times g + l \tag{1}
\end{equation}

where $s$ is the size of the value held at process $n$ and $p$ is the number of processes. Thus, in the Coq system, the totally correctness function is given as follows:

Definition broadcast_get : (T:Set)(vect:(Vector T))(n:Z)
   '0 <=n < nprocs' -> (res:(Vector T) | (i:Z) '0 <=i < nprocs' -> (at T res i)=(at T vect n)).
We prove this property by cases on $n$. Thus, to broadcast a value from a process $n$, we need that $n$ is really the $pid$ of a process. If not, we cannot prove our algorithm and like in the implementation, the program fails.

Another way to express this problem, is using the $put$ operation for broadcasting a value with the following expression:

\[
\text{let broadcast } n \text{ vect } = \text{apply (apply (mkpar (fun pid v dst ->
if pid=n then Some v else None)) vect)} \text{(replicate n)};;
\]

With this new definition of the broadcast, we can prove an interesting result, which performs and models error of the broadcast by having None on each process:

\[
\begin{align*}
\text{Definition broadcast_put : (T:Set)\langle \text{vect:(Vector T)}\rangle (n:Z)} &\text{)
\{res:(Vector (Option T)) | '0 \leq n < nprocs' -> (i:Z)
'0 \leq i < nprocs' -> (at (Option T) res i)=(Some T (at T vect n))}\}
&+ \text{\{res:(Vector (Option T)) | '0 \leq n < nprocs' -> (i:Z)
'0 \leq i < nprocs' -> (at (Option T) res i)=(None T)\}}.
\end{align*}
\]

and we prove it by cases on $n$ and by contradiction. Now we give a trivial use of the $get$ operation where the drawback of this operator is not significant.

### 4.0.4 Shift right

Shift right values on a parallel vector modulo the number of processes is used in few BSP algorithms where each process has to deal with the data of its predecessor:

\[
\begin{array}{c}
v_0 \\
v_p \end{array} \quad \begin{array}{c}
v_{i-1} \quad v_i \\
v_{i-1} \quad v_i \\
v_{i-1} \quad v_i \\
v_{i-1} \quad v_i \\
v_{i-1} \quad v_i \end{array} \quad \begin{array}{c}
v_{p-1} \\
v_{p-2} \end{array}
\]

It could be done by a unique super-step by using an application of the $get$ operator as follows:

\[
\text{let shift_right vect } = \text{get vect (mkpar (fun i ->
if i=0 then nprocs-1 else i-1))}
\]

with following the BSP cost formula:

\[
p \times n \times g + l
\]

(2)

where $n$ is the size of the value held at each process $i$. Now we can prove the desired and certified expression for any parallel vectors:

\[
\begin{align*}
\text{Definition shift_left : (Data:Set)\langle \text{vect:(Vector Data)}\rangle }
&\text{)
\{res : (Vector Data) | (i:Z) '0 \leq i < nprocs' ->
((at Data res i)=(at Data vect (Cases (Z_eq_dec i '0') of
(left _) => 'nprocs-1' 
| (right _) => 'i-1' end)))}}.
\end{align*}
\]

which could be proved by cases on $i$. The $get$ operator has proven to be sufficient to express all the BSP algorithms but not with an optimized cost formula. We give here an example, where the use of the $put$ is essential to keep the efficiency of the algorithm.
4.0.5 Total exchange

In a total exchange, each process communicates its own value to all the other processes. This could be express as:

```plaintext
let total_exchange v_data =
    put (apply (mkpar (fun a val pid -> Some val)) v_data);
```

which has the cost formula:

\[ p + p^2 \times n \times g + l \]  

(3)

where \( n \) is the size of the biggest value held at a process. The specification of this function is a parallel vector as result containing functions. The application of one of these functions, for example the function \( f_i \) at process \( i \), to \( j \) will give the value received by process \( i \) from process \( j \):

Definition total_exchange : (Data:Set; v_data:(Vector Data))
{res:(Vector (Z->(Option Data))) | (i:Z) '0 <= i < nprocs' ->
(param:Z) (Cases (Z_le_dec '0' param) (Z_lt_dec param nprocs) of
    (left _) (left _) => ( ((at (Z->(Option Data)) res i) param)
       = (Some Data (at Data v_data param)))
    | _ _ => ((((at (Z->(Option Data)) res i) param)=(None Data)) end)}. which is proved by cases on param. Like in the expression of broadcast, the specification expresses that the result is None when param is not the “pid” of a process. To end this overview of BSP expressions, we give an interesting result which has not been formally proved before: the possibility to express the get operation with the put one.

4.0.6 A new implementation of get

At a communication level, there are mainly two approaches: put and send. As far as execution is concerned, it is better to put than to get. For a simple reason, get is implemented with two puts and involves two super-steps: a process sends a request to another designated process, and it receives back a reply from it once the request is finally processed. Remark, the situation is just the opposite at the programming level (see the shift-right expression): each process controls its own communications (if it does not want to receive a new value, it just does not ask at the “get step”).

The request can be written with the put operator as follows:

```plaintext
let fa s j = if s=j then (Some true) else None;;
let request s = apply (mkpar (fun j f i -> f i))
    (put (apply (mkpar (fun i -> fa)) s));;
```

and the reply, which uses the result of the request to send its own value to a process which needs it. The implementation of the get operator is simply an application of the reply expression to the vector of process names:

```plaintext
let reply d s = put (apply (apply (mkpar (fun i x x0 x1 ->
                   match (x x1) with
                     None -> None
                   | Some b -> if b then (Some x0) else None)) (request s)) d);;
let new_get vect vect_n = apply (reply vect vect_n) vect_n;;
```
The request needs a super-step to ask to each process which of them send values or not. After, the reply expression sends the values and needs a second super-step. So the cost formula is defined as follows:

\[ p \times g \times (n + 1) + 2 \times l \]  

(4)

where \( n \) is the size of the value held at each process \( i \). The formal properties of this implementation to prove is the same as the axiom of the get operator. It could be proved by case and induction on the elements of the process names vector (\( \text{vect}_n \)):

Theorem new_get_equal_get: \((\text{vect}:(\text{Vector Data}))\) (\(\text{vect}_n:(\text{Vector Z})\)) (\(i:Z\)) '0\leq i<\text{nprocs}'

\[ \rightarrow (\{(j:Z)\} '0\leq j<\text{nprocs}' \rightarrow '0 \leq (\text{at}\ \text{vect}_n \ j)< \text{nprocs}' \)

\[ \rightarrow (\text{at} (\text{new_get} \ \text{vect} \ \text{vect}_n) \ i)=(\text{Some} \ (\text{at} (\text{get} \ \text{vect} \ \text{vect}_n) \ i)). \]

The proof obligations related to the indices are easily discharged by the useful arithmetic tactic of Coq, Omega, all then necessary inequalities being now available from the context. By contradiction, we can also prove that this function returns an empty parallel vector whereas the natural contained in the parallel vector parameter are not process names (integers between 0 and the number of processes). Thus, the get operator is a simple syntactic sugar for the call of two put.

5 Related work

The first formal semantics of BSP was an axiomatic logic and parallel explicit semantics of the BSP model for a share-memory programming implementation [26]. It is based on mapping each process to a trace sequence which may be combined to determine composite behavior. Unfortunately, no programs have been certified with this set of algebraic laws and no implementation of the mathematical model of the specifications have been made. But the idea suggested the possibility to prove parallel algorithms with the BSP models while preserving the cost model. In [46], the presented semantics allows subset synchronization. This feature is not a part of the BSP model and there are many drawbacks to add it [22]. In another approach [48], the reasoning is made using a sequence of global (parallel) state transformation. It eases the reasoning because whole parallel operations can be described as a global state transformation. This paper shows the refinement of a sequential version of the Floyd’s shortest path algorithm to a BSP version. All those approaches are based on imperative languages.

Our approach has the following advantages. It is based on a functional language so it eases the reasoning. Using the Coq proof assistant our proofs are partially automated which is not the case in other approaches. Moreover we can generate programs using Coq. In our framework the compiler and runtime system are partially certified. We also believe that the use of more complex data structures will be rather easy in the case of BSML, but would be very complex in other approaches.

6 Conclusions and Future Work

We have formalized the parallel operations of the BSML language. Using this formalization we proved properties of very often used BSML programs. This allows to validate those programs and
even to find mistakes in the hand-written proofs.

Several directions can be followed for future work:

- validation of more complex BSP programs as parallel sorts [50, 49, 52], our goal is to have a certified BSMLlib library (including its standard library)

- extension of this framework to include imperative features (like [26]) using work done on sequential programs [13] and have a software for certification of BSML programs (like in [14])

- studying of the possibility to prove cost formulas.

We also continue our work on the certification of the environment. In particular, the parallel abstract machine proved with respect to the BSL-calculus is a SECD based parallel machine, which is of course not the one used in the implementation of the BSMLlib. For this reason, we designed a parallel abstract machine based on the Zinc Abstract Machine [17] used in the current implementation of Objective Caml [29]. This machine has to be proved correct. We may use a version of the BSL-calculus with explicit substitution in order to follow the methodology presented in [25] for the validation of several abstract machines with respect to a calculus of explicit substitution.

Future work will also consider extensions of the BSML framework with parallel compositions: the parallel juxtaposition [33] which allows to divide the network in subnetworks while preserving the BSP cost model and the parallel superposition [35] which allows to have in a pure functional setting a constructor which can be used to run to “BSP threads”. This new construction is particularly interesting for multiprogramming and thus is a first step towards the use of BSML for Grid computing.

Acknowledgments This work is supported by the ACI Grid program from the French Ministry of Research, under the project CARAMEL (www.caraml.org).

References


[41] C. Parent. Developing certified programs in the system Coq: The program tactic. BRA Workshop Types for Proofs and Programs, May 93.


