Formal Semantics of DRMA-Style Programming in BSPlib

Julien Tesson and Frédéric Loulergue

LIFO – University of Orléans, France
{julien.tesson,frederic.loulergue}@univ-orleans.fr

Abstract. BSPlib is a programming library for C and Fortran which supports bulk synchronous parallelism (BSP). This paper is about a formal semantics for the DRMA programming style of the BSPlib library. The aim is to study the behavior of BSPlib programs and to propose some syntactic characterizations used to provide guarantees on semantic properties. This work is the basis for future tools dedicated to the validation of BSPlib programs.
Keywords : B.S.P., formal Semantics, Parallel Programming.

1 Introduction

In the range of possibilities to program parallel architectures, from concurrent programming with an imperative language and a message passing library such as MPI [12] to sequential programming and parallelizing compilers, bulk synchronous parallelism or BSP [11] is an intermediate approach. It aims at maximizing the portability of performances by adding a notion of explicit processes to data parallelism.

There are several libraries and languages which support bulk synchronous parallel programming : libraries to be used with imperative languages such as C and Fortran [6], or to be used with object oriented languages [5], or to be used with functional languages [9, 10].

If in parallel programming the execution should be fast, other aspects such as the ease of programs development or the ease of programs validation are also important. In the case of concurrent programming, the difficulty of these two tasks are confirmed by the high complexity of related validation problems [1]. Moreover the semantics of a concurrent program being in general very complex, the time required to run it (related to its operational semantics) is also difficult to determine, which hinders the portability of performances. The structured parallelism of the BSP model eases both programming and validation. Performance prediction has been validated by experiments.

For pure functional bulk synchronous parallel programming, the complexity is the same than the proof of pure functional sequential programs. It is possible to use the Coq proof assistant to extract functional BSP programs from constructive proofs [4]. Other theories of the proof of BSP programs [7, 13, 3, 8] are close in complexity to the sequential case.

In this paper we focus on the semantics of imperative BSP programs in SPMD style. The proposed semantics models the BSPlib library subset which allows direct remote memory access (DRMA) communications. From this semantics we want to find properties on the syntax of programs which could guarantee some properties on the semantics of the programs. Our aim was not to set a priori constraints on the syntax to guarantee semantic properties such as done in [2] for data-parallelism. We aimed at modeling a widely used and practical library for BSP programming (BSPlib), to exhibit some undesirable behaviors and some ways to avoid them. In the next section we give a quick overview of the BSPlib and the model we designed, called BSP-IMP. In section 3 we present the rules of the formal semantics. Section 4 relates syntactic properties of BSP-IMP programs to semantic properties and gives an example. We end by conclusion and future work in section 5. Omitted proofs and complete semantics can be found in [14].

2 An Overview of BSPlib and BSP-IMP

BSPlib [6] is a library for bulk synchronous parallel (BSP) programming. In the BSP model, a computer is a set of uniform *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 (for the sake of conciseness, we refer to [11] for more details). A BSP program is executed as a sequence of *super-steps*, each one divided into (at most) three successive and logically disjoint phases: (a) Each processor uses its local data (only) to perform sequential computations and to request data transfers to/from other nodes; (b) the network delivers the requested data transfers; (c) a global synchronization barrier occurs, making the transferred data available for the next super-step.

BSPlib contains 20 basic operations and follows the SPMD paradigm. These operations are distributed into two parts: One for direct remote memory access (DRMA) and one for bulk synchronous message passing (BSMP). The BSPlib offers functions to start and to stop the parallel execution as well as functions to access the process identifier and the number of processes. The synchronization barrier is called with the `bsp_sync` function.

In DRMA style, communications are performed by the `bsp_put` and `bsp_get` functions:

- `bsp_put(dest, src, tgt, offset, nbytes)` sends data to a remote memory location. `dest` is the identifier of the process where data are to be stored, `src` and `tgt` are the locations where the data are to be read / stored , `offset` is a displacement in byte from `tgt` where data will be copied and `nbytes` is the amount of data to transfer.
- `bsp_get(dest, rloc, offset, tgt, nbytes)` requests data from a remote memory location. `dest` is the identifier of the process where requested data are. `rloc` and `tgt` are the locations where the data are to be remotely

read / locally stored, `offset` is a displacement in byte from `src` from where data will be copied and `nbytes` is the amount of data to transfer.

DRMA access are allowed only on registered memory locations: registration and unregistration are done using the `bsp_push_reg`, `bsp_pop_reg` functions.

In our model, called BSP-IMP, the programs instructions consist of a small imperative subset and two DRMA communication instructions: **put**$(dest, src, tgt)$ and **get**$(dest, rloc, tgt)$ where *dest* and *src* are arithmetic expressions as we only use integer values and *tgt* and *rloc* are variables. Memory locations are not registered in BSP-IMP but this could be easily added to the semantics. The following grammars define respectively the set of arithmetic expressions **aexp**, the set of boolean expressions **Bexp** and the set of programs or commands **Com**:

$$
\begin{array}{lll}
\textbf{aexp} & : & a ::= n \mid X \mid a + a \mid a - a \mid a \times a \mid \textbf{This} \mid \textbf{Nproc} \\
\textbf{bexp} & : & b ::= \textbf{True} \mid \textbf{False} \mid a = a \mid a \leq a \mid \neg b \mid b_0 \wedge b_1 \mid b_0 \vee b_1 \\
\textbf{com} & : & c ::= c; c \mid X := a \mid \textbf{if } b \textbf{ then } c \textbf{ end} \mid \textbf{while } b \textbf{ do } c \textbf{ end} \mid \textbf{skip} \\
 & & \quad \mid \textbf{put}(a, a, X) \mid \textbf{get}(a, X, X) \mid \textbf{sync}
\end{array}
$$

where X is a variable (memory location) and n is an integer constant.

3 Formal Operational Semantics

The operational semantics specifies, by means of a set of rules, how a program will be executed. In the BSP model the execution is a sequence of super-steps. In each super-step, the first phase of asynchronous computations is performed independently on each processor. These computations are described by a first set of rules which are called *local* rules because these rules describe the computation at a specific processor of the parallel machine. The communications and the synchronization barrier need the cooperation of all the processors. These phases of the super-steps are described by a second set of rules called *global* rules.

The first set of rules defines a relation $\longrightarrow_p^i$ between:
- A triple $\langle c,\ \sigma,\ r \rangle$ consisting of a program c (an element of the set **Com**), an environment σ which describes the memory state as a function from variables to values, and a communication requests queue r;
- A triple $\langle s,\ \sigma',\ r' \rangle$ consisting of an execution state s being either Ok, Err or Wait(c), an environment and a communication requests queue.

Ok refers to the final state of a process that ended well, Err to the state of a process ending with an error. Wait(c') means that the local process is waiting for a global synchronization, c' is a sequence of commands that have to be executed after the synchronization.

This relation means "starting from an initial memory state σ and a communication requests queue r, the program c will evaluate at processor i in a parallel machine with p processors to the execution state s with final memory state σ' and final communication requests queue r'".

The second set of rules defines a relation $\longrightarrow_p$ between:
- A triple $\langle\ C,\ \Sigma,\ R\ \rangle$ of vectors of width p. C is the vector of programs

$[c_0, \ldots, c_{p-1}]^p$, as BSP-IMP follows the SPMD paradigm, initially we have the same program c everywhere. Σ is the vector of environments (one per processor) and R is the vector of communication requests queues (one per processor). The environment (resp. queue) at processor i is written $\Sigma[i]$ (resp. $R[i]$).
- A triple $\langle\ S,\ \Sigma',\ R'\ \rangle$ where S is the final global execution state which can be either Ok or Err (it is not a vector). Σ' and R' are the final vectors of environments and queues.

3.1 Local Rules

We omit here the rules for the evaluation of boolean and arithmetic expressions. They are similar to the ones in [15] and can be found in [14]. There are two special arithmetic expressions: **This** which evaluates to the processor identifier and **Nproc** which evaluates to the number of processors. These two values are the ones given on the relation $\longrightarrow_p^i$. We focus here on the evaluation of commands.

Idle Command. The **skip** command does nothing. Its main purpose is to indicate that there is nothing to do after a synchronization.

$$\langle \textbf{skip},\ \sigma,\ r\rangle \longrightarrow_p^i \langle \mathsf{Ok},\ \sigma,\ r\rangle \tag{1}$$

Sequence of Commands. For a sequence of command $c_0; c_1$ if c_0 ends well then c_1 is evaluated in the new environment (rule 2), if c_0 raises an error c_1 is not evaluated and the error is re-raised (rule 3), finally if c_0 leads to a waiting state then c_1 is added in this state as remaining work (rule 4).

$$\frac{\langle c_0,\ \sigma,\ r\rangle \longrightarrow_p^i \langle \mathsf{Ok},\ \sigma'',\ r''\rangle \qquad \langle c_1,\ \sigma'',\ r''\rangle \longrightarrow_p^i \langle s,\ \sigma',\ r'\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \longrightarrow_p^i \langle s,\ \sigma',\ r'\rangle} \tag{2}$$

$$\frac{\langle c_0,\ \sigma,\ r\rangle \longrightarrow_p^i \langle \mathsf{Err},\ \sigma',\ r'\rangle}{\langle c_0; c_1,\ \sigma,\ r\rangle \longrightarrow_p^i \langle \mathsf{Err},\ \sigma',\ r'\rangle} \tag{3}$$

$$\frac{\langle c_0,\ \sigma,\ m\rangle \longrightarrow_p^i \langle \mathsf{Wait}(c_0'),\ \sigma',\ m'\rangle}{\langle c_0; c_1,\ \sigma,\ m\rangle \longrightarrow_p^i \langle \mathsf{Wait}(c_0'; c_1),\ \sigma',\ m'\rangle} \tag{4}$$

Conditional Execution. In the evaluation of **if** b **then** c **end** if the condition b evaluates to true then c is evaluated (rule 5) else there is nothing to do (rule 6).

$$\frac{\langle b,\ \sigma,\ m\rangle \longrightarrow_p^i \textbf{True} \qquad \langle c,\ \sigma,\ m\rangle \longrightarrow_p^i \langle s,\ \sigma',\ m'\rangle}{\langle \textbf{if}\ b\ \textbf{then}\ c\ \textbf{end},\ \sigma,\ m\rangle \longrightarrow_p^i \langle s,\ \sigma',\ m'\rangle} \tag{5}$$

$$\frac{\langle b,\ \sigma,\ m\rangle \longrightarrow_p^i \textbf{False}}{\langle \textbf{if}\ b\ \textbf{then}\ c\ \textbf{end},\ \sigma,\ m\rangle \longrightarrow_p^i \langle \mathsf{Ok},\ \sigma,\ m\rangle} \tag{6}$$

While Loop. In the evaluation of **while** b **do** c **end** if the condition b evaluates to false there is nothing to do (rule 7), else the body c of the loop is evaluated. If it evaluates to Err then the evaluation of the **while** loop is stopped (rule 9) otherwise **while** b **do** c **end** is evaluated in the new environment obtained after the evaluation of the body of the loop. This recursive evaluation could lead either to the request for a synchronization barrier (rule 10) or not (rule 8).

$$\frac{\langle b,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \textbf{False}}{\langle \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end},\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Ok},\ \sigma,\ m\rangle} \tag{7}$$

$$\frac{\langle b,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \textbf{True} \qquad \langle c,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Ok},\ \sigma'',\ m''\rangle \quad \langle \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end},\ \sigma'',\ m''\rangle \ \longrightarrow_p^i\ \langle s,\ \sigma',\ m'\rangle}{\langle \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end},\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle s,\ \sigma',\ m'\rangle} \tag{8}$$

$$\frac{\langle b,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \textbf{True} \qquad \langle c,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Err},\ \sigma',\ m'\rangle}{\langle \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end},\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Err},\ \sigma',\ m'\rangle} \tag{9}$$

$$\frac{\langle b,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \textbf{True} \qquad \langle c,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Wait}(c'),\ \sigma',\ m'\rangle}{\langle \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end},\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Wait}(c'; \textbf{while}\ b\ \textbf{do}\ c\ \textbf{end}),\ \sigma',\ m'\rangle} \tag{10}$$

Remote Memory Write. $\textbf{put}(a_1, a_2, X)$ is a command which aims at writing the value of the expression a_2 in the memory location X at processor given by expression a_1. If the arithmetic expression a_1 evaluates to a value in the range $[0, p-1]$ then a communication request is added to the local queue (rule 11). The communication request $\langle X_{@j} \leftarrow n\rangle$ means that value n should be written into memory location X at processor j.

If a_1 is not a valid processor identifier an error is raised (rule 12).

$$\frac{\langle a_1,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ j,\ j \in [0,p-1] \qquad \langle a_2,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ n}{\langle \textbf{put}(a_1, a_2, X),\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Ok},\ \sigma,\ m.\langle X_{@j} \leftarrow n\rangle\rangle} \tag{11}$$

$$\frac{\langle a_1,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ j,\ j \notin [0,p-1]}{\langle \textbf{put}(a_1, a_2, X),\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Err}_{\text{PUT}},\ \sigma,\ m\rangle} \tag{12}$$

Remote Memory Read. Similar to remote memory write.

$$\frac{\langle a_1,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ j,\ j \in [0,p-1]}{\langle \textbf{get}(a_1, Y, X),\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Ok},\ \sigma,\ m.\langle X_{@i} \leftarrow Y_{@j}\rangle\rangle} \tag{13}$$

$$\frac{\langle a_1,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ j,\ j \notin [0,p-1]}{\langle \textbf{get}(a_1, Y, X),\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Err}_{\text{GET}},\ \sigma,\ m\rangle} \tag{14}$$

Local Affectation. The local environment is modified by changing the value $\sigma(X)$ to the value of the arithmetic expression a.

$$\frac{\langle a,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ n}{\langle X := a,\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Ok},\ \sigma[X \mapsto n],\ m\rangle} \tag{15}$$

Synchronization Awaiting. The command **sync** requests a global synchronization. The synchronization barrier can only be global so this request can only be performed at the global level. Thus at the local level the **sync** command leads to a waiting state $\mathsf{Wait}(\textbf{skip})$.

$$\langle \textbf{sync},\ \sigma,\ m\rangle \ \longrightarrow_p^i\ \langle \mathsf{Wait}(\textbf{skip}),\ \sigma,\ m\rangle \tag{16}$$

3.2 Global Rules

The global rules are used to perform the communication requests and the global synchronization barrier or to end globally the computation. In the following rules, $\mathcal{P}$ denotes the range of processor identifiers. There are four different cases.
Rule (17)**:** All processes are in a waiting state. In this case data are exchanged which is modeled by the $\mathscr{C}$ operation between the vector of memory states and the vector of communication requests queues. $\mathscr{C}$ could be either:

(a) A relation to model the behavior of the BSPlib: in this case the semantics is non-deterministic because two processors could write different values of the same memory location of a third processor and the behavior is not specified.

(b) A function to determinise the semantics. This could be done for example by giving a priority to each processor for remote memory write, or by giving a binary commutative operator to combine the different values written on the same memory location by remote processors. It is also possible to add a rule to raise an error when two processors try to write different values to the same memory location. This various options are described in more details in [14].
Rule (18)**:** if at least one process ends ($\downarrow$) either in the Ok state or erroneously while at least one other is requesting a global synchronization then a global error $\mathsf{Err}_{\text{SYNC}}$ is raised.
Rule (19)**:** If all processes end well, the final global execution state is Ok.
Rule (20)**:** If at least one local process ends with an error $\mathsf{Err}_L \in \{\mathsf{Err}_{\text{GET}}; \mathsf{Err}_{\text{PUT}}\}$ and no other requests a global synchronization then the Err_G error is raised at the global level.

$$\frac{\begin{array}{c}\forall i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i]\rangle \ \longrightarrow_p^i\ \langle \mathsf{Wait}(c_i'),\ \Sigma'[i],\ R'[i]\rangle \\ \mathscr{C}(\Sigma', R', \Sigma'') \qquad \langle\ [c_0', \ldots, c_{p-1}']^p,\ \Sigma'',\ \emptyset\ \rangle \ \longrightarrow_p\ \langle\ \downarrow,\ \Sigma''',\ R''\ \rangle\end{array}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \ \longrightarrow_p\ \langle\ \downarrow,\ \Sigma''',\ R''\ \rangle} \tag{17}$$

$$\frac{\begin{array}{c}\exists i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i]\rangle \ \longrightarrow_p^i\ \langle \mathsf{Wait}(c_i'),\ \Sigma'[i],\ R'[i]\rangle \\ \exists j \in \mathcal{P}, \langle c_j,\ \Sigma[j],\ R[j]\rangle \ \longrightarrow_p^j\ \langle\downarrow,\ \Sigma'[j],\ R'[j]\rangle\end{array}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \ \longrightarrow_p\ \langle\ \mathsf{Err}_{\text{SYNC}},\ \Sigma',\ \emptyset\ \rangle} \tag{18}$$

$$\frac{\forall i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i]\rangle \ \longrightarrow_p^i\ \langle \mathsf{Ok},\ \Sigma'[i],\ R'[i]\rangle}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \ \longrightarrow_p\ \langle\ \mathsf{Ok},\ \Sigma',\ \emptyset\ \rangle} \tag{19}$$

$$\frac{\begin{array}{c}\exists i \in \mathcal{P}, \langle c_i,\ \Sigma[i],\ R[i]\rangle \ \longrightarrow_p^i\ \langle \mathsf{Err}_L,\ \Sigma'[i],\ R'[i]\rangle \\ \forall j \in \mathcal{P}, \langle c_i,\ \Sigma[j],\ R[j]\rangle \ \longrightarrow_p^j\ \langle\downarrow,\ \Sigma'[j],\ R'[j]\rangle\end{array}}{\langle\ [c_0, \ldots, c_{p-1}]^p,\ \Sigma,\ R\ \rangle \ \longrightarrow_p\ \langle\ \mathsf{Err}_G,\ \Sigma',\ \emptyset\ \rangle} \tag{20}$$

4 Synchronization Error Free Programs

An interesting property to check for a BSP-IMP program is the absence of synchronisation errors. A program is free of such an error if each process reaches the same number of **sync** during the program evaluation. Due to possible presence of **sync** in or after a loop the problem is undecidable in general. Nevertheless we can decide it for a subset of BSP-IMP programs. We characterize those who have the *replicate synchronization* property .

A program $c \in \mathbf{Com}$ is said to have the *replicate synchronization* property if for all "**if** b **then** c' **end**" and "**while** b **do** c' **end**" in which c' contains **sync**, b evaluates to the same value at each processor in $[0, \mathbf{Nproc} - 1]$. Of course, to evaluate each **sync** needed at global level, each process has to be free of local errors that could break the normal program evaluation flow.

Theorem 1. *A program Pr without local error, wich terminates and for which the replicate synchronization property hold, is synchronization error free.*

A variable which has the same value at all processors is called a replicated variable. It can be seen as a shared variable. A boolean expression will evaluate to the same value at all processors if all the variable occurences are replicated.

A subset $\mathcal{R}ep(Pr)$ of replicated variables in a program Pr can be build from variables not modified by a communication, and which are affected to expression that contains only constants and replicated variables. Those affectations cannot be made inside **while** or **if** statements for which the condition does not evaluate identically over all the processors. Furthermore a value has to be previously assigned to the variable at least one time in the program. Indeed initial local environment are not in general identical over all processor, so uninitialized variables are not replicated occurrences.

We have here mutually dependent definitions of replicated variables and replicated boolean expressions, but the $\mathcal{R}ep(Pr)$ can be build as the greatest fixed point of variables having the previous property.

The following scan algorithm computes the parallel prefix sums:

Algorithm 1 (Scan)
i:=1;
while ($2^{i-1} \leq$ **Nproc**) **do**
  if (**This** $\geq 2^{i-1}$) **then get**(**This** $- 2^{i-1}, X, X_{in}$) **end**;
  sync;
  if (**This** $\geq 2^{i-1}$) **then** $X = X_{in} + X$ **end**;
  $i = i + 1$
end

It is an example of program that can be shown synchronization error free using the previous characterization.

We can easily prove that there is no error at local level.

Furthermore the only conditional component of the program which contains a **sync** is the main **while** loop and its condition ($2^{i-1} \leq$ **Nproc**) contains only replicated variables. **Nproc** has clearly the same value over all processors and i satisfies the conditions previously described.

5 Conclusion and Future Work

We proposed an operational semantics for a small bulk synchronous parallel imperative language. This BSP-IMP syntax and semantics models very closely the behavior of the BSPlib programming library. With some additional conditions BSP-IMP programs are deterministic. It is to notice that the BSPlib could be easily modified to follow the BSP-IMP semantics which raises an error when non deterministic remote memory writes occur. We used this semantics to show how a subclass of BSP programs can be shown to be free of synchronization errors.

The presented work is limited to the DRMA part of the BSPlib library. Future work includes the extension to the bulk synchronous message passing part (BSMP) of BSPlib. Other classes of programs will be studied. We also plan to develop tools for the analysis of BSPlib programs.

References

1. K. R. Apt and E.-R. Olderog. *Verification of sequential and concurrent programs*. Springer-Verlag, 2nd ed. edition, 1997.
2. L. Bougé. Le modèle de programmation à parallélisme de données: une perspective sémantique. *RAIRO Technique et Science Informatiques*, 12(5), 1993.
3. Y. Chen and W. Sanders. Top-Down Design of Bulk-Synchronous Parallel Programs. *Parallel Processing Letters*, 13(3):389–400, 2003.
4. F. Gava. Formal Proofs of Functional BSP Programs. *Parallel Processing Letters*, 13(3):365–376, 2003.
5. Yan Gu, Bu-Sung Lee, and Wentong Cai. JBSP: A BSP programming library in Java. *Journal of Parallel and Distributed Computing*, 61(8):1126–1142, 2001.
6. J.M.D. Hill and W.F. et al. McColl. BSPlib: The BSP Programming Library. *Parallel Computing*, 24:1947–1980, 1998.
7. H. Jifeng, Q. Miller, and L. Chen. Algebraic laws for BSP programming. In *Euro-Par'96*, LNCS 1123-1124, pages 359–368. Springer, 1996.
8. D. S. Lecomber. *Methods of BSP Programming*. PhD thesis, Oxford University Computing Laboratory, July 1998.
9. F. Loulergue, F. Gava, and D. Billiet. Bulk Synchronous Parallel ML: Modular Implementation and Performance Prediction. In *Proc. of ICCS*, LNCS 3515, pages 1046–1054. Springer, 2005.
10. Q. Miller. BSP in a Lazy Functional Context. In *Trends in Functional Programming*, volume 3. Intellect Books, may 2002.
11. D. B. Skillicorn, J. M. D. Hill, and W. F. McColl. Questions and Answers about BSP. *Scientific Programming*, 6(3):249–274, 1997.
12. M. Snir and W. Gropp. *MPI the Complete Reference*. MIT Press, 1998.
13. A. Stewart, M. Clint, and J. Gabarró. Axiomatic Frameworks for Developing BSP-Style Programs. *Parallel Algorithms and Applications*, 14:271–292, 2000.
14. J. Tesson and F. Loulergue. Formal Semantics for the DRMA programming style subset of the BSPlib library, May 2007. to appear.
15. G. Winskel. *The Formal Semantics of Programming Languages*. Foundations of Computing Series. MIT Press, 1993.