Semantics of a data-parallel logic language using the BSP execution model

Arnaud Lallouet

Abstract

We present a parallel logic language which uses the bulk synchronous parallelism (BSP) execution model [13]. The BSP model provides a simple way to program parallel machines by restricting the SPMD style: processes are limited to a bounded asynchronism during local computations and communication is a global operation followed by a global synchronization. This in conjunction with explicit location provides a simple cost model that allow performance prediction. We adapt the data-parallel logic language DPLOG [7] to fit with theses characteristics and we present both a declarative and an operational semantics. The resulting language, we call BS-DPLOG, offers a great expressive power without any compromise to the advantages of BSP.

Keywords: Parallel Logic Programming, Semantics, BSP model

1 Introduction

We propose a parallel logic language called BS-DPLOG which uses the Bulk Synchronous Parallelism (BSP) execution model. It aims to inherits from logic programming a declarative semantics and a high level of abstraction and from BSP an efficient execution model to reason about parallel computations and a simple model of performances prediction.

Context of this work Distributed memory and MIMD is now a well-established architecture for parallel machines but they are notoriously difficult to program. When using concurrent (CSP-like) languages, one has to deal with features like indeterminism or deadlock. Although they provide additional power of expression suitable for certain problems, they are superfluous when one wishes only to speed up the resolution of a problem. Collection-oriented languages [11] manipulate

Université l'Orléans - LIFO, 45067 Orléans Cedex 2
E-mail: `Arnaud.Lallouet@lifo.univ-orleans.fr`

vectors instead of scalars and are easier to use because they provide a single thread of control to the programmer : they act like a processor of arrays. We call this synchronous and centralized point of view *macroscopic* [2]. They are now associated to the SPMD model of programmation (single program, multiple data) and their implementation synchronize the different threads (at least) when needed. Following [2], this asynchronous and distributed point of view can be called *microscopic*. Declarative languages belonging to this category are mostly functional : Caml-Flight [4], NESL [1] or $8\frac{1}{2}$ [8]. In Logic Programming, most of the work has been done for concurrent languages and automatic parallelization [3].

The BSP model The data-parallel execution model BSP [13] has now a wide recognition as a useful paradigm for parallel programming and provides a simple way to write programs (SPMD) associated to good performance and cost prediction. However, the integration with declarative programming, and especially logic programming has not yet been done.

A BSP program is a data-parallel SPMD program composed of a sequence of *superstep*, each one divided in three phases. First, each processor executes a local and independent computation in which it may request transfers of data from/to other nodes. The second phase performs the requested communications and the third is a global synchronization. Locations are explicit (*direct mode*) and there is no virtualization mechanism (see figure 1). Performances of this

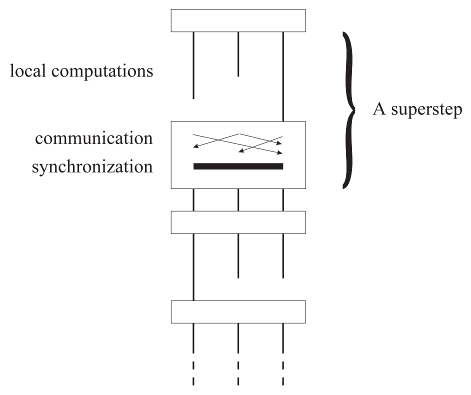

Figure 1: The BSP execution model

model are expressed by three parameter : the number of processors p, the time l required for a global synchronization and the time g needed for all the processors to communicate one word to another (1-relation). For a relation of arity h, the time is gh. The execution time of a superstep s is the sum of the longest local processing time, of the data transfers time and the global synchronization time :

$$Time(s) = \max_{0 \leq i < p} W_i^{(s)} + \max_{0 \leq i < p} h_i^{(s)} * g + l$$

where $W_i^{(s)}$ = local processing time on processor i, $h_i^{(s)}$ = the number of words transfered by processor i.

Data-parallel logic programming In [7] and [6], we present a data-parallel logic language called DPLOG. We take this language as a basis for the language proposed in this paper. Here follows an intuitive presentation of the language.

A DPLOG program is a set of definite Horn clauses (i.e. without negation) of the form $h \leftarrow b_1, \dots b_n$ where h, b_1, ..., b_n are vectorial atoms. A DPLOG program handle vectorial (multi-dimensional) objects indexed by locations and reductions on the vector's elements are done in parallel. The workspace of locations is described by particular objects called *indexes*. The most usual index domain is a finite subset of $I\!N^n$ (n-dimensional arrays). The same program is attributed to each index, and each computation occurs locally, starting with its own query. For instance, let's consider the program P_1 in figure 2, and the set of indexes $\{0, 1, 2, 3\}$. Then, we use a vectorial notation $[\![a, b, a, a]\!]_{\{0,1,2,3\}}$ to express the query a on the set of indexes $\{0, 2, 3\}$ and the query b on the index 1 at the same time. This query succeeds because each goal can be locally deduced from the program. The fact that programs are identical does not mean that all compu-

```
P1:            P2:                 P3:
  a ← b          a ← b               a ← get b from This - 1
  b ←            b ← This = 3        b ←
```

Figure 2: Three small DP-LOG programs

tations have to be the same. The behavior of the program may also depend on the value of the actual index. This is achieved by a special vectorial constant, `This`, whose value at an index is precisely the index of the computation. Considering the same query and the program P_2 of figure 2, the query fails because the goal `b` can only succeed at index 3. A computation at a given index may also depend on results computed at other indexes. This is the purpose of the general communication primitive "`get` p `from` j" where p is an atom to be proven at index j. This is illustrated by the program P_3 of figure 2 : the atom a is true at index i if the atom b is true at index $i-1$. This means that every proof of `a` at index i will use an auxiliary proof of `b` at index $i-1$. A communication can ask to get a *relocated proof* of an atom or may use any other mean to ensure that the communicated atom belongs to the model of the called index. Operationally, when asked for a "get", an index launches a new proof process to get a proof of this atom. In this language, communications are thus non-atomic. Practically, they necessitate the use of tables to memorize past computations in order to avoid infinite loops through communications and to limit the cost of such an operation [6]. The operational semantics we propose for this language is synchronous (one could say "SIMD-like") but the BSP execution model provides a simple and powerful way to express asynchronous parts. The choice of the independent computations is left to the programmer as in every language with explicit parallelism. The solution we present here can also be viewed as a particular case of the general model in which tables are limited to a single element (a more general execution model is left for future work).

The truth value of a vector is simply the conjunction of the truth value of its components. According to this point of view, data-parallel computations are nothing more than a restricted form of AND-parallelism [5]. But we present a vector semantics, in order to fit better with the operational view, in particular wrt cost prediction which is not easy without explicit locations.

Adaptation to the BSP model To fit with the BSP model, we need four important features that are not present in DPlog :

1. Explicit processor location : an index corresponds to a true processor location.

2. Global synchronization : there is no context of activity like in most data-parallel languages. Hence all indexes perform every synchronization.

3. Local computation part : we split the set of predicates into two subsets. One is devoted to local computations while the second is concerned with global ones. This is a two-level language construct : from a high level point of view, global predicates operate like ordinary predicates in logic programming (except the fact they act on vectors). Their execution is synchronized (from the programmer's point of view) and when they perform backtracking, this yields to express several BSP calculus. This latter feature can be used to model the traversal of a BSP program space (see figure 3). On the other side, local predicates act differently. They are just tools used

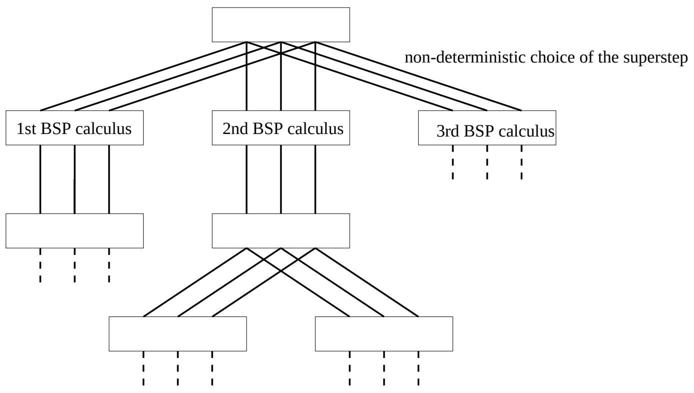

Figure 3: Describing several BSP computations as a search space

inside of a superstep. For them we choose to forbid backtracking across synchronization by pruning the end of the search tree. Backtracking over potentially different predicates would lead to a complex behavior and should be difficult to implement to preserve efficiency. As a consequence, we loose completeness wrt the declarative semantics, but we do not make concessions on the expressivity of the language because non-determinism with search space traversal is still available at the higher level.

4. Atomic communication primitive: in the BSP model, communication is a global and atomic operation, i.e. it concerns all indexes and the cost of such an operation is predictable. In BS-DPLOG, we choose to limit the table used for communication in DPLOG to one item we call the public memory. Every time a global atom is reduced and a synchronization occur, the atom is copied into the processor's public memory. When a `get` is performed, it must unify with the atom in this memory and the result is immediately returned. It induces a particular style of programmation (i.e. produce an atom and transfer it somewhere else) that fits well with the BSP programming style.

All these features provide a declarative way to construct BSP programs in the logic programming style. The benefits we expect come from these two fields: a declarative style for which there exists many formal techniques to specify and validate programs, to perform semantics-based transformations, ... and simple programmation of complex parallel machines with performance prediction.

Plan of the paper In this paper, we start from the synchronous data-parallel language DPLOG and we add step by step different features to desynchronize parts of the program. First we give a declarative semantics to state what is computed. Then we explore different operational semantics. The first one we give is ground and top-down [7]. Then we precise the non-deterministic use of the rules by giving the DP-SLD strategy [6]. Then we carefully provide asynchronous local computations by using don't care non-determinism and a commit operator. Because these asynchronous parts are forced to be independent by a syntactical condition, we preserve the expressiveness of the language as well as the interesting features of BSP like the cost model. Finally we finalize our approach with the language BS-DPLOG which provides a concrete syntax for bulk synchronous logic programming.

2 Least model declarative semantics

We propose to define the declarative semantics. We only define here what is computed with no relation to bulk synchronous parallelism, which is a highly operational notion. Nevertheless, we do so with operations whose BSP interpretation is possible. A fixpoint semantics has already been presented to define the declarative semantics of the (operationally different) language DPlog [7]. Let's first set some basic definitions.

Let $VAR, FUNC$ and $PRED$ be denumerable sets of variables, function and predicate symbols. Let $PRED_{loc} \subseteq PRED$ be a set of predicate symbol which will have a special operational meaning. On top of these sets, we build $TERM$ the set of terms (including $TERM_G$ the set of ground terms) and $ATOM$ the set of atomic formulae (including HB the set of ground atoms or Herbrand base). We call a normal atomic formula a *simple atom*. For the purpose of distribution, we consider $LOC \subseteq TERM_G$ a finite set of special ground terms we call *index* or *locations*. Let *NProcs* be the cardinality of LOC. A *context* is a subset of LOC.

A *vector* or *collection* on a set E is a family of elements of E indexed by a context. Conversely, if A is a vector, we denote its context by $\widetilde{A}$. For every index $i \in \widetilde{A}$, we denote by $A|_i$ the projection of A on the index i, and more generally, for a context $c \subseteq \widetilde{A}$, let $A|_c$ be the projection of A on the indices of c. We usually denote vectors by ordinary letters — in order to keep the formalism as light as possible — but sometimes it is necessary to have a more precise notation, especially when communications are involved. Then we use the notation $[\![a_i]\!]_{i \in c}$ to denote the vector (a_i) indexed by $i \in c$.

Parallel programs need to communicate and hence we introduce a special goal construct in order to tackle this : "`get` p `from` l" where p is a simple atom and l a location. We call it a *relocated atom*. The intended meaning of this is that the relocated atom "`get` p `from` l" is true at a given index i if p is true at index l. In this context, we define a *litteral* to be a simple or relocated atom. Note that we do not allow imbrication of `get`s for evident reasons.

We define as usual a goal to be a finite sequence of litterals. A definite clause is a formula of the form $h \leftarrow B$ where h is a simple atom and B a goal. Note that relocated atoms cannot be heads of clauses. A program is a set of definite clauses. A BS-DPlog goal is a vector of goals indexed by a context. A BS-DPlog program is a vector indexed by LOC of the same program at each index.

We follow a classical way of defining the declarative semantics : it consists in considering only ground instances of the program clauses : $Inst(P)$. An interpretation is thus a set of ground atoms. Here we say that an interpretation is a set of vectors of ground litterals, but we place a restriction upon this : we impose that the set is the cartesian product of its projection on each index ($A = \times_{i \in LOC} A|_i$). We do this because logical consequences are handled independently at each index. Thus, the macroscopic and microscopic point of view commute : an interpretation can be considered as a set of vectors as well as a vector of sets.

But having the same program on each index does not mean the same consequences hold everywhere, or the benefit of this language would not be important. In order to make computations dependent of the location, we introduce the special vector constant `This` whose value at a location is the location itself (the actual index of computation). However, for the double purpose of readability and optimisation, this constant will be implemented differently in BS-DPLOG.

An interpretation I is a model of a vector of litterals a if, for every index $i \in LOC$, either $a|_i$ is a simple atom and $a|_i \in I|_i$ or $a|_i$ is a relocated atom `get` p `from` l and $p \in I|_l$. In the last case, we say that the relocated atom `get` p `from` l belongs to $I|_i$.

Interpretations are structured into a complete lattice ordered by the product ordering of the set inclusion at each index. An interpretation is a model of a clause $h \leftarrow B$ if $h \in I$ whenever $B \subseteq I$. It is a model of the program if it is a model of every clause. The least model following this ordering is equal to the intersection of all models, as in classical logic programming. The existence of the model $[\![\mathcal{HB}]\!]_{LOC}$ states that this model exists and is not empty. As usual, we call this model M_P and we take it as the declarative semantics of the program.

3 Operational semantics

In this section, we describe several operational semantics, in increasing order of determinism and precision.

3.1 Top-down semantics

This first semantics allows to build a proof tree for a given ground goal. It consists of the three following rules :

- Partitioning. Reducing a goal at each index often involves more than one clause. We choose to split the context into different parts (that could possibly use the same clause), and continue the computation independently :

  (Partitioning [TDG-P])

  $$\frac{A|_{c_1} \ldots A|_{c_n}}{A} \quad \text{if } (c_i)_{i \in 1..n} \text{ is a partition of } \widetilde{A}$$

- Reduction. This rule reduces an atom from a goal according to a program clause :

  (Reduction [TDG-R])

  $$\frac{\{b \mid b \in B\}}{h} \quad \text{if } h \leftarrow B \in Inst(P)$$

- Communication. We handle general communications, i.e. any location can communicate with any other. The following rule simply relocates atoms to be remotely proved to their respective indices. Let $v = [\![\texttt{get } p_i \texttt{ from } j_i]\!]_{i \in c}$ be a vector of relocated atoms. We define two functions: CI (for called indices) associates to v the set of indices (context) involved in the communication and $Reloc_k$ associates to v the set of simple atoms called at the index k :

  $$CI(v) = \{j \mid \exists i \in c, v|_i = \texttt{get } p \texttt{ from } j\}$$

$$Reloc_k(v) = \{p \mid \exists i \in c, v|_i = \texttt{get } p \texttt{ from } k\}$$

The cartesian product of the sets $Reloc_k$ is the set of vector obtained after the expansion of all relocations.

(Communication [TDG-C])

$$\frac{\times_{k \in CI(v)} Reloc_k(v)}{v = [\![\texttt{get } p_i \texttt{ from } j_i]\!]_{i \in c}}$$

The top-down ground semantics ($\mathcal{S}_{TDG}$) is the least set of vectors of litterals closed by the rules. The following theorem states the equivalence between the declarative and operational semantics :

Theorem 3.1

$$M_P = \mathcal{S}_{TDG}$$

3.2 Synchronous Data-Parallel SLD

Here we present an extension of SLD-resolution suitable to reduce multiple goals synchronously. Three levels of non-determinism are left in the above rules : choice of the selected atom and choice of the clause as in logic programming, but moreover choice of the context split. We give here a computation rule close to the idea of SLD-resolution to go through the search tree.

As in SLD-resolution, we choose to select the leftmost atom and reduce it with the first possible clause given by the definition of the predicate. Here we give a rule for any LD-resolution, i.e. any choice of the selected atom. Let's take a vector of goals, say $[\![a_1, \ldots, a_k, \ldots, a_m]\!]_c$ on a context c and let's suppose the chosen atom is a_k and the chosen clause is $h \leftarrow b_1, \ldots, b_n$. After a partition of the context $c_1 \cup c_2 = c$, we get the following result, where mgu denotes the most general unifier of its arguments :

$$[\![a_1, \ldots, a_k, \ldots, a_m]\!]_c \rightsquigarrow [\![a_1, \ldots, b_1, \ldots, b_n, \ldots, a_m]\!]_{c_1}([\![\theta]\!]_{c_1}) \cup [\![a_1, \ldots, a_k, \ldots, a_m]\!]_{c_2}$$

with $\theta = mgu(a_k, h)$, at each index. In this rule, the context c_2 may be empty and the vector substitution mgu is applied pointwise to each element of the vector.

The main problem is to find a suitable partition of the context to minimize independent computation. This is desirable in this context because the semantics is synchronous. If the context was splitted too many times, it would yields more applications of the rule. On the other hand, a single reduction step on a large context performs reductions on many vector elements at the same time. The programmer has this rule in mind as an operational reference and the model is virtually synchronous, whatever the actual execution. The context split is done only when necessary when only a subcontext of the current context can be reduced with the chosen clause. In our example, we choose to let $c_1 = \{i \in c \mid mgu(a_k, h) \text{ exists }\}$. However, if computation of part of this subcontext fails, we must backtrack with a remaining $c'_1 \subseteq c_1$. In this case, the remaining context is simply merged with c_2 and the computation proceeds.

The following rule implements this concept, for the general case of multiple subcontexts to be reduced independently :

Definition 3.2 (DP-SLD rule)

$$\begin{aligned}\bigcup_{c\in C}[\![a_1,\ldots,a_{k_c},\ldots,a_{m_c}]\!]_c \quad \leadsto \quad & \bigcup_{c\in C, c\neq c'} [\![a_1,\ldots,a_{k_c},\ldots,a_{m_c}]\!]_c \\ & \cup\ [\![a_1,\ldots,b_1,\ldots,b_n,\ldots,a_{m'_c}]\!]_{c'_1}\ ([\![\theta]\!]_{c'_1}) \\ & \cup\ [\![a_1,\ldots,a_{k_c},\ldots,a_{m'_c}]\!]_{c'_2}\end{aligned}$$

where : c is a partition of LOC, c' is the selected context, a_k is the selected atom, $h \leftarrow b_1,\ldots,b_n$ is the selected clause, $c' = c'_1 \cup c'_2$, and $\theta = mgu(a_k, h)$

The model is virtually synchronous, hence applications of this rule are supposed to occur sequentially. In the next section concerning the BSP model, we relax this constraint. Because of the global synchronisation implicit in this rule, some dirty — but useful — features of Prolog can be freely added : cut, I/O, assert, ... with the same advantages and drawbacks than in classical logic programming.

4 Local computations and the BSP model

4.1 An operational semantics for BSP logic programming

One main characteristics of the BSP model is that it allows a limited form of asynchronism during the local computation parts. Hence, since our aim is to model BSP programs, we want to be able to describe these local steps. However, due to the balance needed for BSP, we put a simple and reasonable limitation upon these computations for implementation conveniences : we refuse any backtracking across synchronizations. Completeness is lost, of course, but without limiting the expression power of the language. The programmer only has to express the search space at top level, using the local computations as tools to achieve his algorithm.

Here comes the use of the set of predicates symbols $PRED_{loc} \subseteq PRED$. They are the local predicates and their execution is asynchronous. Moreover, during a local computation phase, no communication can occur nor the use of a global predicate, in order to ensure the independence of every local calculus. Hence we place the following restriction on clause bodies : a definition for $p \in PRED_{loc}$ can only use simple atoms whose predicate symbols is in $PRED_{loc}$. Our language BS-DPLOG reflects this limitation in its two levels construct.

As a counterpart, we force global predicate to execute on the full location range (LOC). Hence we have two alternatives : either the predicate is global and every index proceeds, or the predicate is local and computation depends on the actual index. This has a major impact on the resolution rule: either *one* synchronous step involving every index is performed, or *NProc* asynchronous and independent steps are performed.

A global atom p subject to communication is memorized in its public memory $mem(p)$. Hence a communication just has to take the value. This memorization is performed whenever a new value has been encountered for this relation and a global synchronization occur. It can of course yield a partially instanciated atom.

An interesting point is that we have preserved the opportunity of adding extra-logical features, exactly as in the synchronous version. For example, the cut would have the same effect as usual for the global synchronous part: it prunes the right side of the search tree at each index.

Here are the BSDP-SLD rules. The three rules are mutually exclusive since they do not operate on the same kind of atom. The first rule defines the reduction of a global predicate ($\overset{G}{\leadsto}$), the second models communications ($\overset{C}{\leadsto}$) and the last implements a local computation step ($\overset{L}{\leadsto}$) that abstracts the details of the local execution. The choice of the rule for a local derivation step is syntactically determined by the membership of the selected atom to $PRED_{LOC}$:

Definition 4.1 (BSDP-SLD rule)
(Global atom [BSDP-SLD G])

$$[\![a_1,\ldots,a_k,\ldots,a_m]\!]_{LOC} \overset{G}{\leadsto} [\![a_1,\ldots,b_1,\ldots,b_n,\ldots,a_m]\!]_{LOC} \;([\![\theta]\!]_{LOC})$$

with: a_k *= selected atom (global),* $h \leftarrow b_1,\ldots,b_n$ *= selected clause,* $\theta = mgu(a_k,h)$

(Relocated atom [BSDP-SLD C])

$$[\![a_1,\ldots,a_k,\ldots,a_m]\!]_{LOC} \overset{C}{\leadsto} [\![a_1,\ldots,a_{k-1},a_{k+1},\ldots,a_m]\!]_{LOC} \;([\![\theta]\!]_{LOC})$$

with: a_k = `get` p `from` j *= selected atom (relocated),* $\theta = mgu(p, mem(p))$

(Local atom [BSDP-SLD L])

$$[\![a_1,\ldots,a_k,\ldots,a_m]\!]_{LOC} \overset{L}{\leadsto} [\![a_1,\ldots,a_{k-1},a_{k+1},\ldots,a_m]\!]_{LOC} \;([\![\theta]\!]_{LOC})$$

with: a_k *= selected atom (local),* θ *= the substitution obtained by the full local reduction of* a_k *at each index.*

4.2 Description of the language and examples

Here we make a further step towards a real language : we provide a simple syntactic way to express whether a predicate belongs to $PRED_{loc}$ or not.

Global predicates are written as ordinary Prolog predicates. They can call local predicates or other global predicates, or perform communications with the `get` construction. This latter structure only ranges on global predicates. Moreover, global atoms are copied into a public memory whenever they occur as argument in a `get`. For example, the following program computes at an index the average of its successor and predecessor :

```
average(X,Y) :-
        dec(I1),
        inc(I2),
        get average(X1,_) from I1,
        get average(X2,_) from I2,
        Y is (X1+X2)/2.
```

Local predicates are indexed by a term enclosed by brackets. This term is often a variable and denotes their actual index :

```
dec[0](I) :- I is maxproc.
dec[X](I) :- I is X-1.

inc[maxproc](0).
inc[X](I) :- I is X+1.
```

Let assume that the index space is $\{0,1,2\}$, thus the constant `nproc` (number of processors) is 3 and `maxproc` (number of last processor) is 2. A typical query for this program is the vector $[\![$ `average(3,Y), average(7,Y), average (1,Y)` $]\!]$. The predicate `average` occurs in a communication, thus its initial value is memorized at every index. Let's see what happens at index 1. Once the index of communication is known, `get average(X1,_) from I1` unifies the atom `average (X1,_)` with the content of the private memory at index 0, i.e. `average(3,_)`, yielding the binding `X1 = 3`. Similarly, the latter `get` yields `X2 = 1` and we get `Y = 3+1/2 = 2` at this index. The same behavior occurs at the other indexes at the same time.

The bracket notation for local computations is far more convenient than the vector constant `This` we used for the semantics. Moreover, it allows the following optimisation : clauses labeled by a constant index should not be copied on the other locations. This static repartition is suitable if the local programs are truly different :

```
local[0] :- long_computation 1 ...
local[1] :- long_computation 2 ...
```

As a summary, here is a formal specification of the syntax of BS-DPLOG. Basic objects are simple atoms (s-atoms) and locations. Simple atoms are splitted in two categories : local atoms (l-atoms) that act upon scalars and global atoms (g-atoms) that act upon vectors.

```
l-goal ::= sequence of l-atom
l-clause ::= l-atom[location] :- l-goal
t-location ::= location | variable
```

```
g-litteral ::= g-atom |
               l-atom |
               get g-atom from location |
               par (l-atom -> g-litteral ; g-litteral)
g-goal ::= sequence of g-litteral
g-clause ::= g-atom :- g-goal
clause ::= g-clause | l-clause
program ::= set of clause
```

5 Conclusion

Communications act upon goals and not only variables. This choice allows to define a clear declarative semantics. Another choice would have been to use something like "`get` X `from` l `into` Y" whose meaning is yet declaratively unclear. The counterpart of our choice is that when you are only interested in values of variables at a different index, a special operational mechanism has to short-cut the whole evaluation of the goal. This is implemented by the public memory.

Data-parallel execution of logic programs has also been investigated — among others — in the Reform project [9] and [12]. Both approaches are relevant to automatic parallelization. A more complete survey of these approaches and a few more can be found in [6]. An other interesting approach is multi-dimensional logic programming [10], although it does not yet provide a parallel execution model. In all these approaches, there is no performance model. Comparisons can also be made with non-logic languages like NESL [1] or Caml-Flight (formerly DPML) [4], both in functional programming. We do not exactly tacke the same problems (BSP vs general data-parallelism).

In summary, we propose a logic language adapted to the BSP execution model. It has been given all the important features : explicit locations, independent local computation phases and atomic general communication. Our aim is to take the benefits of both worlds : declarative specification of a problem and simple parallel execution model with performance prediction.

Acknowledgments The author whishes to thank Gaétan Hains for many fruitful discussions and useful comments on an early version of the document.

References

[1] Guy E. Blelloch. Programming parallel algorithms. *Communications of the ACM*, 39(3):85–97, 1996.

[2] Luc Bougé. Le modèle de programmation à parallélisme de données : une perspective sémantique. *Technique et Science Informatiques*, 12(5):541–562, 1993. also in english RR92-45, IMAG, ENS Lyon.

[3] J. Chassin de Kergommeaux and P. Codognet. Parallel logic programming systems. *ACM Computing surveys*, July 1994.

[4] C. Foisy and E. Chailloux. Caml Flight : a portable SPMD extension of ML for distributed memory multiprocessors. In A.W. Böhm and J.T. Feo, editors, *Workshop on high performance functional computing*, Denver, Colorado, April 1995. Lawrence Livermore National Laboratory.

[5] Manuel V. Hermenegildo and Manuel Carro. Relating data-parallelism and (And-) parallelism in logic programs. In *EUROPAR'95*, volume 966 of *LNCS*, pages 27–42. Springer, August 1995.

[6] Arnaud Lallouet. Expressivité d'un langage de programmation logique data-parallèle. *Technique et Science Informatiques*, accepted for publication, 1998.

[7] Arnaud Lallouet and Yann Le Guyadec. Contribution to the semantics of a data-parallel logic programming language. In Fernando Silva Vítor Santos Costa and Inês de Castro Dutra, editors, *Post International Logic Programming Symposium Workshop on Parallel Logic Programming Systems*, pages 32–41. Portland, Oregon, December 8 1995.

[8] Olivier Michel and Jean-Louis Giavitto. Design and implementation of a declarative data-parallel language. In J. Barklund, B. Jayaraman, and J. Tanaka, editors, *ICLP post-conference workshop on parallel and data-parallel execution of declarative languages*, S. Margherita Ligure, Italy, June 17 1994. UPMAIL, Uppsala University, ftp://ftp.csd.uu.se/pub/papers/reports/0078/.

[9] H. Millroth. Reforming compilation of logic programs. In *International Logic Programming Symposium*, San Diego, CA, 1991.

[10] Mehmet A. Orgun and Weichang Du. Multi-dimensional logic programming : theoretical foundations. *Theoretical Computer Science*, 185:319–345, 1997.

[11] Jay M. Sipelstein and Guy E Blelloch. Collection-oriented languages. *Proceedings of the IEEE*, 79(4):504–523, April 1991.

[12] Donald A. Smith and Timothy Hickey. Multi-SLD resolution. In *Logic Programming and Automated Reasoning*. Springer-Verlag, 1994.

[13] L. Valiant. A bridging model for parallel computation. *Communications of the ACM*, pages 103–111, August 1990.