Static Analysis of

a Parallel Logic Language

based on the Blackboard Model

T.Castagnetti
Dipartimento di Informatica
Università di Pisa - Italy

P.Ciancarini
Yale University and Università di Pisa
E-mail ciancarini@cs.yale.edu
Suggested running head: Static Analysis of a Parallel Logic Language

Author responsible for proofs:
Paolo Ciancarini
Dipartimento di Informatica
Corso Italia, 40
56100 Pisa Italy
E-mail: ciancarini@cs.yale.edu
Tel.: +39 50 510264
Fax: +39 50 510226

ABSTRACT

Shared Prolog is a parallel logic language based on the blackboard interpretation of logic programming. In such an interpretation a logic program is seen as a set of rules executed by a set of agents cooperating via a shared working memory called blackboard. A distributed interpreter for Shared Prolog was implemented and described in another paper, where the blackboard was a centralized data structure. In this paper we show how the blackboard can be distributed using some static analysis techniques. The basic idea is to perform an abstract interpretation starting from the Shared Prolog operational semantics to generate data structures which represent possible interactions and links among agents. The resulting data structures are used to reduce the number of run time communication operations in an implementation distributed over a network of workstations.
LIST OF SYMBOLS

∃ exists
∀ for all
∧ and
∨ or
≈ union
∈ belongs to
λ lambda
∅ right arrow
| vertical bar
_ underscore
1. INTRODUCTION

Most of the recent proposals concerning new parallel logic programming languages - e.g. Janus [14] and Strand [11] - are based on the so called process interpretation of logic programming, in which shared logic variables are seen as dataflow-like communication channels of networks of agents [15]. These languages abandon Prolog as a programming tool, requiring different styles of programming. This means also that Prolog programs are difficult to rephrase in these new frameworks [4].

A different parallel procedural interpretation for logic programming has been introduced by one of the authors of this paper, based on the so called blackboard model [5]. In the blackboard interpretation no sharing of logic variables is allowed; the computing units are logic agents that can read and write logic tuples using associative operations on a shared data structure called blackboard, that is close to the Linda’s Tuple Space [12]. The blackboard model introduces naturally in logic programming a notion of rule-based parallelism totally different from dataflow parallelism exploited in other parallel logic languages. This kind of parallelism allows to express both large grained sequential computations as Prolog programs, and fine-grained dataflow-like computations as set of rules to be executed in parallel.

Shared Prolog (SP) is a parallel logic language based on the blackboard interpretation of logic programming. SP is not the first logic language based on the blackboard concept: it has been preceded by Polka [7]. However, Polka was defined as an extension of the dataflow-based logic language Parlog; contrariwise, SP directly embeds the blackboard model. The initial design of Shared Prolog has been studied in [2]; that paper contains a formal operational semantics and some programming examples. A prototype distributed implementation was built using the specification offered by the operational semantics, but the blackboard data structure was centralized [1]. In this paper we show that the blackboard can be distributed; moreover, we will develop some static analysis and compilation techniques able to improve the performance of SP programs. A formal discussion of these techniques is included as
well.

The paper has the following structure: section 2 contains a description of the main concepts of the SP language; Section 3 shows a programming example. Section 4 introduces some techniques for static semantic analysis of SP programs that are useful for some compilation-time optimizations. Section 5 describes the SP abstract machine and its distributed implementation, giving some performance evaluations. Finally, the last section lists some future works that will be done in the Shared Prolog project.

2. Shared Prolog

In the following discussion we will assume a familiarity with Edinburgh Prolog syntax, especially with the notions of logic atom, variable, and goal. The readers not familiar with logic programming will find a good reference in the book by Sterling and Shapiro [16].

2.1 Theories

Syntactically, a Shared Prolog program consists of a number of theories. Each theory is composed of a heading, that is a term containing the theory name and a number of parameters, the keyword “: -”, a set of clauses called patterns (separated by “#”), the keyword “with”, and a Prolog program called knowledge base.

\[
\text{theory_name}(V_1,...,V_j) : - \; \% \; \text{theory heading} \\
\quad \text{pattern}_1 \; \# \; \ldots \; \# \; \text{pattern}_n \; \% \; \text{patterns} \\
\quad \textbf{with} \; \text{Knowledge_Base.} \; \% \; \text{Prolog program}
\]

The number of variables $V_i$ in the heading is called arity of the theory. Their scope is limited to the patterns; they allow parametric definitions of theories.

Intuitively, a theory consists of two set of logic rules: the patterns, that specify the interface of the theory with respect to the external world, and the Prolog rules, that describe local computations invisible from outside the agent.

2.2 Blackboard

A blackboard is a multiset of atoms. Syntactically a blackboard is a goal between braces
that contains (eventually not ground) atoms.

% blackboard
\{a_1, ..., a_n\}.

A key issue is that variables in the blackboard have no global scope; every atom is a tuple independent from other atoms in the blackboard. Given a blackboard

\{a(X), b(X)\}.

its declarative reading is given in (1):

(\forall X:a(X)) \land (\forall X:b(X)) \tag{1}

so that the above goal is logically equivalent to the following:

\{a(Y), b(Z)\}.

where variables have been renamed. In this way all atoms in the blackboard are completely independent: no variables are shared. A Shared Prolog blackboard is similar to the Linda' Tuple Space; atoms can be associatively written, read, or deleted.

2.3 Agents

Atoms in the blackboard can be either passive or active. An atom is defined passive if it is not active. An atom is defined active if the program includes a theory that has the same name and arity. Computation in the blackboard is carried over by active atoms. An active atom is also called an agent (while passive atoms are messages). Each agent executes a theory, i.e., it loops evaluating the patterns trying to satisfy some guard using the blackboard contents (we note that different agents can execute the same theory). Agents can read, write, and delete atoms: in fact, they communicate only by writing and reading messages in the blackboard.

2.4 Patterns

Agents execute (a set of) patterns. The syntax of a pattern is the following:

Read \{ In \} \emptyset Body \{ Out \}

This is a logic clause where braces denote actions that modify the blackboard, whereas the right arrow is a symbol of commitment. The four components of a pattern have the
following forms:

Read. A goal including both messages to be found in the blackboard and predefined predicates called *built-ins*. Most of the built-ins are constraint predicates inherited from Prolog, like `not/1`, `is/2`, `var/1`, `nonvar/1`; some built-ins are SP-specific: e.g. the built-in `all/2` is a specialization of `bagof/3` (i.e., `all(X,List) = bagof(X,X,List)` where `X` is a message).

In. A goal composed of a set of messages that have to be found in the blackboard and consumed before the agent can be activated.

Body. A Prolog goal. Its evaluation is done locally by the agent using the logic program (KB) of the theory. No side effects on the blackboard are allowed.

Out. A set of messages that have to be written on the blackboard at the end of the derivation of Body.

Read and In guards, that together form the *preactivation guard*, are a synchronization mechanism between an agent and the blackboard. Their are similar to Linda’s asynchronous `read` and `in` operations, the main differences being that 1) in Shared Prolog we use the non-blocking variants `readp` and `inp`, and 2) the evaluation of a preactivation guard is an atomic action.

If a theory includes several patterns, all their guards are evaluated in OR-parallel by the agent. If an agent finds two or more preactivation guards satisfied in different patterns, one is arbitrarily chosen. In Shared Prolog there is also another kind of local non deterministic choice when a guard is looking for an atom `a(X)` and the blackboard contains say `a(1)`, `a(2)`, and `a(3)`; in this case the agent can bind `X` to either of 1, 2, or 3.

When an agent finds that the preactivation guard of a pattern is satisfied, it commits and executes the related Body; when the local computation ends, the Out set is written in the blackboard. This operation is very close to Linda’s `out` operation. Note that in the Out set only messages are allowed. After outputing the Out set the agent loops trying to activate one of its patterns.
2.5 Goals

The blackboard interpretation supports two kinds of user goals. The first one is called starting goal; it has the form of an Out set, and is written as a pattern without preactivation guard:

\[ ?- \emptyset \{ t_{h1}, \ldots, t_{hn} \}. \]

This goal inserts in the blackboard a number of agents, and starts their parallel execution. Only one starting goal can be issued in SP: no dynamic creation of agents is allowed.

The second kind of goal is called final goal. It has the form of a Read guard without Postactivation part:

\[ ?- m_{1}, \ldots, m_{n} \emptyset. \]

When such a Read guard succeeds, activities in the blackboard stop (agents cannot continue their execution). This goal is useful to define terminating SP systems. So a program either terminates successfully if a final goal is satisfied, or deadlocks because no agent can be activated, or loops indefinitely.

2.6 Evaluation of a goal

Formal semantics for Shared Prolog is studied in [2]. To convey a feeling of the SP programming style, we present a very simple program that shows the interaction of two agents.

**Example:** we define two theories that communicate simulating the interaction of two tabletennis players.
% theory for player1
player1:-
    {ping} % In guard
    ∅ % Out set
    {pong}.
% theory for player2
player2:-
    {pong} % In guard
    ∅ % Out set
    {ping}.
% starting goal
?- ∅ {player1, player2, ping}.

The starting goal specifies two agents, player1 and player2, and a message, ping. Executing their theories, agents player1 and player2 look in the blackboard for atom ping and atom pong, respectively. Since only ping can be found, only player1 fires: it consumes such an atom, then outputs atom pong, and loops on its guard. Now player2 can fire, and this entire process repeats itself indefinitely.

If we use variables, we can write a different version of the same program with one theory only.

**Example:** tabletennis players (version with variables)
% generic theory for players
player(X):-
    p(X,M1), p(Y,M2), Y≠X % Read guard
    {M1} % In guard
    ∅ % Out set
    {M2}.
% starting goal
?- ∅ {player(1), player(2),p(1,ping), p(2,pong), ping}.

This program is slightly more difficult to read because different agents share the same theory, so we must specify all different cases, i.e., distinctive messages that fire an agent. The theory includes a pattern whose Read guard has the purpose of identifying the messages that must be respectively input and output by the agent. So player(1) will use message ping to fire, outputing message pong. The converse will happen with player(2).
3. Programming in Shared Prolog

In this section we study a longer and more meaningful example: a parallel program that implements a system able to play Mastermind™. We have chosen such an example because its (sequential) Prolog version has been used by other authors in the field of logic programming [16] and moreover it is well suited for performance evaluation and comparisons, as will be shown in the subsequent sections. For instance, it is simple to change the size of the problem to solve, simplifying the study of the performance of the program.

Given an alphabet \( A \) of \( k \) symbols, a coder secretly builds a code \( \text{Code} \) that is a string over \( A \) of length \( c \). In our version of the game there are \( N \) decoders that cooperate to discover the code \( \text{Code} \) by issuing guesses that are answered by the coder. The following SP program is composed of:

- a theory \( \text{coder} \) executed by an agent that randomly generates a code \( \text{Code} \) of length \( c \); such an agent then has to answer to the guesses tried by decoders;
- a theory \( \text{decoder(I)} \) executed by \( N \) agents that try to discover the code \( \text{Code} \) offering guesses to the coder.

The \( \text{coder} \) theory handles the random generation of the secret code and answers to guesses coming from decoders.

```prolog
% coder theory
coder :-
  length(C), { start }
  ∅                % pattern to generate the code
  generate_random(C,Code)
  { }
  #
  { try(Guess) }
  ∅                % pattern to reply to query
  answer(Guess,[Bulls,Cows])
  {tried(Guess,[Bulls,Cows])}. 
with
...                % Knowledge base for coder
```

The \( \text{coder} \) theory has two patterns: the first one generates the code to start the game; the second one answers to a guess, previously written on the blackboard by some
Each decoder, independently from other decoders and using all old tries stored in the blackboard, generates a new try that writes on the blackboard.

```
% decoder theory
decoder(I) :- % I is an index
    length(C),
    not try(_) % no try without answer
    all(tried(Guess,Answer),Guesslist) % set constructor Ø
    compute(C,I,Guesslist,NewGuess)
    {try(NewGuess)}
```

Even if apparently all decoders execute the same program, the index \( I \) allows for different tries to be generated. New tries are generated by the predicate `compute` in the Knowledge Base simply scanning every possible permutation of the alphabet; it is not difficult to partition the set of all the permutations to have different guesses from different decoders.

A possible starting goal is the following, where a system is started with one coder and two decoders. Atom `start` is a message that initializes the system, whereas atom `length(4)` tells how many symbols have to be included in the code.

```
% starting goal
?- Ø { coder,decoder(1),decoder(2), start, length(4) }
```

The `coder` starts generating a code of length 4; simultaneously, the `decoders` generate their first guess. The coder then has to answer to both guesses, and when the answers are ready the decoders build another guess, using all available tried guesses, and so on.

The game terminates when a guess gets an answer including \( c \) bulls and no cows. Thus, the final goal corresponding to the above starting goal is the following:

```
% final goal
?- tried(G/[4,0]) Ø.
```

The program will stop when a guess \( G \) receives 4 bulls and 0 cows. When the `coder`
answers to a guess giving 4 bulls, the atom containing the answer fires the final goal, and the computation terminates.

4. Static Analysis of Shared Prolog Programs

Shared Prolog has been firstly implemented building a parallel interpreter that supports communications among different instances of sequential Prolog interpreters distributed over a network of workstations [1]. However, in this prototype the blackboard was a central database containing messages accessed by processes implementing agents. Since communication in such an architecture is the most expensive run time operation, we decided to introduce a static analysis phase aiming at reducing and speeding actions done by the SP runtime support. This is a standard approach for both Prolog [8] and parallel logic languages [18]. It has been suggested also for non logic languages, e.g. for languages belonging to the Linda's family [3].

Time spent by every agent during the execution of a SP program can be roughly divided in three phases:

– evaluation of the guards of the patterns;
– local computation of the body of the chosen pattern;
– communication and synchronization events.

A more efficient guard evaluation is obtained reducing the size of the guard itself, either eliminating redundant code or using data derived from the starting goal (i.e., from theory parameters). The local computation of an agent can be improved using static analysis and partially evaluating the knowledge base with respect to the body. Finally, we can reduce the number of communication and synchronizing events eliminating some useless messages used by the run time support. We built a parallel compiler (written in Shared Prolog) that operates such code improvements.

4.1 Partial Evaluation of the Guards and the Knowledge Base

We have to transform a guard to obtain a new reduced guard equivalent to the original one. The reductions are obtained by partial evaluation [19]. Compared to partial
evaluation of Prolog programs, partial evaluation of SP guards has to neither unfold
procedure calls, nor deal with problems related to cut or metalevel predicates, like
assert and retract, that are allowed neither in the guard nor in the body (so
actually knowledge bases inside theories are written in an extended restriction of
Prolog). Such a partial evaluation must anticipate, if possible, variables substitutions
and built-in predicates evaluation.

Example: the pattern

\[
a(X), \ X=3, \ b(X) \ \{ \ f(X) \ \} \ \emptyset \ \text{body}(_,X) \ \{ \ \text{out}(_,X) \ \}
\]

should be changed in the new pattern

\[
a(3), \ b(3) \ \{ \ f(3) \ \} \ \emptyset \ \text{body}(_,3) \ \{ \ \text{out}(_,3) \ \}
\]

A theory is composed of several patterns and one knowledge base (KB for short) that is
a Prolog program. The KB is used to evaluate the Body of a successful pattern. All the
patterns are statically known, so it is possible to partially evaluate the KB obtaining, for
each pattern, a specialized and more efficient KB.

The Body of a pattern is in general a partially ground Prolog goal. It can contain
variables used also in the guard. Apart from the parameters in the head of the theory,
the values of variables shared with the guard will be known only after guard evaluation.
However, it is possible to statically bind the Body local variables (not shared with the
guard) by partial evaluating the Knowledge Base with respect to the Body itself. After
this transformation some Body local variables could be bound. During such a partial
evaluation as many as possible variable unifications are propagated. Binding is done
both by forward and backward propagation of values. The compiler attempts to
evaluate all built-in predicates, and moreover it tries to expand the body of a clause
executing the required substitutions of predicates. With all the limits of Prolog partial
evaluation [17,19], the resulting Knowledge Base, that is a specialized version of the
original one, contains a lesser number of procedure calls and a greater number of bound
variables.
4.2 Abstract Interpretation of Communication Flows

An SP starting goal can be globally analyzed to improve its performance reducing the communication overhead. The basic idea is to perform a global analysis to collect data about the possible runtime interactions and dependencies among agents. Given a set of theories, an initial blackboard state $BB_{start}$, and a starting goal $G$, we statically study the atom flows across the blackboard to find out which are the possible runtime communications and which are the messages to be exchanged. Moreover, we analyze AND-parallelism studying possible conflicts among patterns and agents. All data returned by this analysis are collected in three graphs. These graphs will be used to distribute the blackboard contents among agents, for faster communications and improved AND-parallelism.

We use the general theoretic framework of abstract interpretation to collect data on communications conflicts among agents. To simplify the analysis we abstract by the local behavior of the theory and consider only the communication dependencies among the agents. Since each theory interfaces itself with the blackboard only by means of its pattern, in our analysis we consider an abstract program as composed only of a set of abstract patterns.

Given a program $Prog$, let $Th_1,\ldots, Th_n$ be all the theories in $Prog$. Let $P_{1,1},\ldots, P_{n,k}$ all patterns in $Prog$, and let $P_{0,start}$ be the pattern representing the starting goal. Moreover, given a pattern $P$, let us define the following multisets of atoms:

\[
\begin{align*}
Read(P) &= \{ A \mid (A \in \text{Read of } P) \lor (\neg (A) \in \text{Read of } P) \lor (\forall (A, \_ ) \in \text{Read of } P) \} \\
In(P) &= \{ A \mid (A \in \text{In of } P) \lor (\forall (A, \_ ) \in \text{In of } P) \} \\
Out(P) &= \{ A \mid A \in \text{Out_Set of } P \}.
\end{align*}
\]

Now we introduce the some functions of abstraction in order to map a concrete pattern $P_{i,j}$ into its correspondent abstract representation $\text{AbsP}_{i,j}$. We start defining what is an abstract atom.

**Def 4.1 Abstract atom**
We define an abstraction function \( \text{AbsAtom} \) over atoms

\[
\text{AbsAtom}(A) = (\text{functor}(A), \text{arity}(A))
\]

\( i.e., \text{AbsAtom}(A) \) is a pair whose first component is the functor of \( A \), whereas the second component is a non-negative integer representing the number of components of \( A \).

**Def 4.2 Functions of abstraction for patterns**

Now we can define the notion of abstract pattern defining a function \( \text{AbsPatt} \).

\[
\text{AbsPatt}(P) = <\text{AbsRead}(P), \text{AbsIn}(P), \text{AbsOut}(P)>
\]

where

\[
\text{AbsRead}(P) = \{ \text{AbsAtom}(A) \mid A \in \text{Read}(P) \}
\]

\[
\text{AbsIn}(P) = \{ \text{AbsAtom}(A) \mid A \in \text{In}(P) \}
\]

\[
\text{AbsOut}(P) = \{ \text{AbsAtom}(A) \mid A \in \text{Out}(P) \}
\]

**Def 4.3 Abstract program**

An abstract program \( \text{AbsProg} \) is a multiset of abstract patterns \( \text{AbsP}_{i,j} \)

\[
\text{AbsProg} = \{ \text{AbsP}_{1,1}, \ldots, \text{AbsP}_{n,k} \} \approx \{ \text{AbsP}_{0,\text{start}} \}
\]

This is a set of abstract patterns, one for each pattern included in the program, plus one special pattern for the starting goal.

**Def 4.4 Function of abstraction for the program**

Let \( \text{AbsProg}(\text{Prog}) \) the function of abstraction that, given a set of theories \( \text{Prog} \) and a starting goal \( P_{0,\text{start}} \) outputs the correspondent abstract representation \( \text{AbsProg} \).

\[
\text{AbsProg}(\text{Prog}) = \{ \text{AbsP}_{i,j} \mid P_{i,j} \in \text{Prog} \land \text{AbsPatt}(P_{i,j}) = \text{AbsP}_{i,j} \} \approx \{ \text{AbsP}_{0,\text{start}} \}
\]

This function is implemented by a compiler that inputs a program composed of a set of theories and the initial blackboard state, and generates the corresponding abstract representation \( \text{AbsProg} \). In the following sections we will show how the compiler manipulates this abstract representation. The compiler analyzes communication flows and conflicts building and using three graphs: Out Flow, Conflict Flow, and In Flow.
4.2.1 Abstract Interpretation of Communications

The blackboard can be seen as a communication channel shared between any pair of patterns. We can interpret the generation of an atom in the Out set of a pattern $P_s$ and the evaluation of the same atom inside the guard of a pattern $P_r$ as a direct communication between the two patterns, where $P_s$ is the sender and $P_r$ is the receiver.

Given an abstract program $\text{AbsProg}$ we build the *Out Flow Graph* $\text{OFG}$ for the program. The graph represents possible runtime communications, i.e., it shows the relationship sender/receiver for each pair of patterns. The Out Flow Graph is an oriented labeled graph $(N,E)$ where $N$ is the set of the nodes correspondent to the patterns of the abstract program and $E$ is a set of labeled edges.

**Def 4.5 Out Flow Graph (OFG)**

Given an abstract program $\text{AbsProg}$ its associated Out Flow Graph $\text{OFG}=(N,E)$ is an oriented and labeled graph such as:

1) $\forall \text{Abs}P_{i,j} \in \text{AbsProg} \exists P_{i,j} \in N$

2) there is an edge $<P_{i,j}-P_{k,l}>$ labeled $\text{AbsAtom}(A) \in E$ from $P_{i,j}$ to $P_{k,l}$ iff

$$(A \in \text{Out}(P_{i,j}) \land (\exists B \in (\text{Read}(P_{k,l}) \approx \text{In}(P_{k,l}))) : \text{AbsAtom}(A) = \text{AbsAtom}(B)$$

The OFG includes an edge from $P_{i,j}$ to $P_{k,l}$ if and only if $P_{i,j}$ can generate an atom $A$ that will be used by the pattern $P_{k,l}$. We say that $P_{i,j}$ and $P_{k,l}$ are in the relation $P_{i,j} \text{ com } P_{k,l}$. The label on the edge represents the message.

![Out Flow Graph for the Mastermind program](image_url)
Considering the program Mastermind with two decoders, the corresponding OFG is shown in fig 1. The picture clearly shows that patterns for decoders are clients of pattern 2 of theory `coder`. Pattern 1 of theory `coder` is used only for initialization purposes. The Out_Flow graph will be used at runtime when an agent produces an Out set.

### 4.2.2 Discovering Possible Conflicts

Now we gather data useful to improve AND-parallel execution of an SP program. According to the semantics of Shared Prolog [2], during a computation two patterns can be activated in parallel if and only if they consume different atoms from the blackboard. If this is not the case, the patterns are in conflict and only one of them can be activated.

**Def 4.6 Conflict relation conf**

Two patterns $P_{i,j}$ and $P_{k,l}$ are in the conflict relation $conf$ if and only if they share an atom in their In guards.

$$P_{i,j} \text{ conf } P_{r,k} \iff ( \exists A \in \text{In}(P_{i,j}) ) \land ( \exists B \in \text{In}(P_{k,l}) ) : \text{AbsAtom}(A) = \text{AbsAtom}(B)$$

The compiler checks for each abstract pattern the In component and returns the conflict relation in the form of a *Conflict Flow Graph* (CFG).

**Def 4.7 Conflict Flow Graph CFG**

Given an abstract program $\text{AbsProg}$ its Conflict Flow Graph $\text{CFG}=(N,E)$ is a not oriented and not labeled graph such that:

1) $\forall \text{Abs}P_{i,j} \in \text{AbsProg} \exists P_{i,j} \in N$;

2) there is an edge $<P_{i,j}-P_{k,l}> \in E$ from $P_{i,j}$ to $P_{k,l}$ iff $((P_{i,j} \text{ conf } P_{k,l}) \lor i=k)$.

The CFG represents the possible conflicts among patterns. An agent can be activated only by exactly one of the patterns of theory it is associated to, so all such patterns are in conflict among themselves. Patterns not directly connected in this graph can be activated in parallel.
Figure 2 Conflict Flow Graph for the Mastermind program

Figure 2 depicts the CFG for the Mastermind program. Pattern 1 of theory code is mutually exclusive with respect to other patterns.

4.2.3 Mutual exclusion conflicts

The compiler generates a third graph, the In Flow Graph IFG, that shows mutual exclusion conflicts on data.

**Def 4.8 In Flow Graph IFG**

Given an abstract program AbsProg the In Flow Graph IFG=(N,E) is an oriented labeled graph such that:

1) ∀ AbsP_{i,j} ∈ AbsProg ∃ P_{i,j} ∈ N,

2) there is an edge <P_{i,j}-P_{k,l}> labelled A∈ E from P_{i,j} to P_{k,l} iff

\[(A \in \text{In}(P_{i,j})) \land (\exists B \in (\text{Read}(P_{k,l}) = \text{In}(P_{k,l}))) : \text{AbsAtom}(A) = \text{AbsAtom}(B)\]

The In Flow Graph shows logical dependencies between patterns that use in their guards the same atoms (at least one of them consumes a common atom in its In guard).
Figure 3 In Flow Graph for the Mastermind program

Figure 3 depicts the IFG for the Mastermind program. The In Flow Graph is used to speed up the commitment of the agent to one of the patterns it is associated to. Starting from the activated pattern and following the edges going out, is possible to locate the agents that have to update their state.

5. DISTRIBUTED EXECUTION OF SHARED PROLOG

The definition of the SP abstract machine presents some problems, that can be summarized under two categories:

– how to organize and manage the blackboard;
– how to control the evaluation of a theory.

Following the operational semantics given in [2], we can design an abstract machine based on two computational entities: a blackboard and a number of agents.

The state of a SP system is composed of:

– the blackboard state $S_B$;
– the states $S_{T_i}$ of every agent.

Def 5.1 Blackboard state

Given a blackboard $B$ its state $S_B$ is defined as:

$$S_B = \approx \sum_{i} A_i \quad (A_i \text{ tuples in the blackboard})$$

The blackboard state is defined by the multiset of all the tuples that are on the blackboard.
**Def 5.2 Agent state**

An agent state corresponds to the elaboration phase in which the agent itself is found. Given an agent named $T$, its state $S_T$ is defined by:

$$S_T = <T(\lambda)>$$

where $\lambda$ is the substitution inherited from the head of the theory. The agent can be in the evaluation phase of its guards, in which case the state is represented simply by the theory name $T(\lambda)$. Otherwise an agent can be active in a local Body computation, and in this case the state is represented by a triple

$$S_T = <T(\lambda), \text{Body}(\lambda), \text{Out}(\lambda)>$$

containing the theory name $T(\lambda)$, the Body, and the Out set of the activated pattern.

**Def 5.3 System state**

Given a running program, let $S_B$ the blackboard state and $S_{T1}, \ldots, S_{Tn}$ the states of the agents, the global system state $S_S$ is defined by the pair:

$$S_S = \{ S_B, < S_{T1}, \ldots, S_{Tn} > \}$$

**5.1 Designing control for SP**

Control of blackboard, agent, and system states can be either centralized or distributed.

Controlling the *blackboard* means handling the shared data structure and the operations that can be executed on the blackboard: reading, writing and deleting atoms. Control of reading and writing operations can be distributed or centralized. Simultaneous readings or writings do not lead the blackboard into a incorrect state. Only the deletion operations have to be really serialized to guarantee that a tuple is consumed by exactly one agent.

Controlling an *agent* is a task that can be divided into the phase of guard evaluation and the phase of internal derivation. During its computation an agent can evolve from a state to another only by commitment or generating the Out set. Since the agent during those transitions must interact with the blackboard, controlling these operations corresponds
to controlling insertions and deletions in the blackboard. Guards evaluation and Body computation can be either centralized or distributed.

Control of the system state depends upon the choices concerning the implementation of operations on the blackboard and agent states. We have a number of possible ways to implement control for SP, each one corresponding to a specific architecture. Table I lists a number of possible implementations, classified with respect to their features for controlling the four critic operations; blackboard read, blackboard modify, guard evaluation, and body evaluation.

<table>
<thead>
<tr>
<th>Implementation</th>
<th>blackboard read</th>
<th>blackboard modify</th>
<th>guard evaluation</th>
<th>body evaluation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Centralized interpreter</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>c</td>
</tr>
<tr>
<td>Agent clients and blackboard server</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>Incremental scheduling</td>
<td>c</td>
<td>c</td>
<td>c</td>
<td>d</td>
</tr>
<tr>
<td>Distributed bb with pattern grained parall.</td>
<td>d</td>
<td>c</td>
<td>d</td>
<td>d</td>
</tr>
<tr>
<td>Distrib. bb with theory grained parallelism</td>
<td>d</td>
<td>d</td>
<td>d</td>
<td>d</td>
</tr>
</tbody>
</table>

We present a short description of the implementation models named in the table.

5.1.1 Centralized Interpreter

The first SP interpreter was totally implemented in Prolog. This prototype consisted of a sequential program running on a uniprocessor. Obviously, all operations were centralized. This amounts to an interpreter that uses forward chaining to choose a pattern to evaluate, as in a production system.

5.1.2 Agents as clients of a blackboard server

In the distributed implementation that was described in [1] there is a scheduler that controls the blackboard state and manages reading, writing, and deletion operations. Control of the guard evaluation for all the patterns is centralized in the scheduler as well. However, control of the internal derivation of each agent is distributed over a set
of (extended) Prolog processes. Each agent corresponds to a separate Prolog process; each process is a client of the shared blackboard manager process.

The scheduler owns all data about the system state, and sequentially evaluates the patterns choosing a subset to be activated; it sends bodies to agents and wait for answers to update the blackboard. When an agent receives a Body it starts a Prolog computation; the result of such a computation is either the Body itself modified with the new substitutions for variables or the failure message.

5.1.3 Incremental evaluation of scheduling

The performance of the preceding architecture can be improved if the scheduler is able to incrementally maintain the blackboard updates with respect to the patterns. The most critical task performed by the scheduler is the guard evaluation searching for activable patterns. We studied an incremental scheduling algorithm based on the “Rete Mach” interpretation technique introduced for production systems [10]. This algorithm is based on the fact that generally the blackboard contents is large, whereas changes between two scheduling cycles are small, thus most of the blackboard contents used during the guard evaluation on a cycle can be reused on the next cycle. A scheduler uses the algorithm to define a set of activable patterns. Given the set of activable patterns, the scheduler arbitrates conflicts and chooses patterns that fire.

5.1.4 Distributed Blackboard with fine-grained parallelism

In this model the blackboard state is distributed among a number of computing units, one for each pattern. Coordination of evaluation is handled by two computing units called respectively dispatcher and clasher. The dispatcher controls the writing operations, whereas the clasher controls the deletion operations. The resulting architecture is depicted in fig. 4.
The dispatcher handles creations and deletions of messages on the blackboard. The blackboard state is distributed: each computing unit owns all the blackboard subset necessary to evaluate the guard of its pattern. If the guard is satisfied the unit sends to the clasher a request of activation. The clasher collects activation requests and arbitrates conflicts, choosing the set of activable patterns, and inhibiting or granting activation to the pattern units. If a computing unit is granted activation it executes the Body, and then communicates to the dispatcher deletions and creations of atoms. The dispatcher propagates this modifications. Thus control of deletions and creations of atoms. The dispatcher

The architecture results in a fine-grained parallelism that could be exploited on a massively parallel system. However, its implementation presents some problems deriving from the necessity of synchronizing all agents to guarantee the correct serialization of the changes in the blackboard; moreover, a large number of communication channels are necessary.

5.1.5 Conclusions

We have studied a number of possible solutions for implementing control of execution of SP programs. We were naturally interested in designing a software architecture that maximizes performance given a very common hardware system: an heterogeneous
network of workstations and mini-mainframes (in our case, SUNs and VAXes). We decided to maintain the granularity of parallelism as in the architecture studied in [1], i.e., large grain logic processes, but aiming at distributing both the blackboard contents and the control of operations on it. Our resulting design is described in the following section.

5.2 Distributed Blackboard with Theory-grained parallelism

In this section we describe an architecture in which most of control is distributed over a number of computing units, and only control of the deletion operations has to be centralized and assigned to a manager called Scheduler. In this solution granularity of parallelism is as fine as agents executing theories (as opposed agents executing patterns, as seen in section 5.1.4). Each agent can evaluate independently all patterns of a theory, accessing the blackboard for reading data. Moreover, each agent controls its own Body evaluation and the related Out set generation on the blackboard.

Since most of the blackboard control is distributed among the agents, we can physically distribute the blackboard itself. Each agent owns only the data it needs to satisfy its guards. At any time during the computation the blackboard is distributed in as many (not disjoint) subset as agents. Since the same atoms can be present in many guards of different theories, it is necessary to guarantee that insertions and deletions in the blackboard are carried over consistently. The architecture that we implemented is based on the theory grained abstract machine described in [1], and it is composed of a distributed set of (extended) Prolog processes.

An agent accesses its own share of the blackboard in order to independently evaluate its guards. When at least one of its guards is successful, the agent sends to a Scheduler a request of activation. The Scheduler receives such a requests, solves the possible conflicts, and then sends back an answer to the agent. Now the agent evaluates the Body related to the chosen pattern, and at the end of the Body evaluation it directly sends each message contained in the Out set to each agent that could use that message. Both the agents and the Scheduler use the graphs generated at compilation time. The
agents use the Out Flow Graph to know the receiving agents, whereas the Scheduler uses the Conflict Flow and In Flow Graphs to arbitrate conflicts and to update the distributed blackboard state.

During the evaluation several agents may be simultaneously active in their internal derivation phase (AND parallelism), while other agents may simultaneously try to satisfy their guards (OR parallelism). It is possible to increase parallelism if the Conflict Flow Graph is not connected. If any, the unconnected subgraphs determine a partition of the patterns. Patterns belonging to different subsets will never be in conflict. Thus each subgraph can be considered as an island of parallelism. In this case the compiler creates a Scheduler for each island, and each Scheduler has to control the activations of a lesser number of agents.

5.2.1 Agents

Each agent has a local knowledge base composed of a static part and a dynamic part. The dynamic part is the subblackboard with data that will be used by the agent to satisfy its guards. In the static part there is a subgraph of the Out Flow Graph, i.e., the part composed of all nodes directly connected to the patterns of the theory. Each agent can independently evaluate its guards using its share of the blackboard. The agent uses its Out Flow subgraph to determine the receivers of its Out set. The receiving agents update their shares of the blackboard with the incoming messages. In fig. 5 the flowchart executed by an agent is shown.
An agent can communicate with other agents or with the Scheduler. Communications between agents take place when generating the Out set. Communications between an agent and the Scheduler take place when an agent sends an activation request containing the list of the satisfied patterns. The Scheduler sends an answer granting an activation or containing an atom list that the agent has to delete from its share of the blackboard state.

### 5.2.2 The Scheduler

The Scheduler controls the evolution of a set of agents. Since the blackboard is distributed among the agents and partially replicated, the Scheduler has to guarantee that each share of the blackboard is updated with respect to the global evolution (especially with respect to the deletion operations).
The Scheduler uses the Conflict Flow and In Flow graphs and loops on its working cycle shown in figure 6. When the Scheduler receives several simultaneous activation requests, using the Conflict Flow graph it computes a set of agents to be granted activation. Then using the In Flow Graph it picks up the agents that must receive deletions lists. It sends the lists to the agents that have to update their shares of the blackboard, and finally it loops to receive other activation requests. The Scheduler stops when it receives an activation request from a pattern that represents the final goal. Such a pattern has the highest priority over every agent, and it terminates the whole system.

Let us consider a communication between an agent T and the Scheduler. T is committing to one or more of its patterns and has sent to the Scheduler an activation request. Suppose that the Scheduler has just sent activation grants to other agents; now it sends to T the list of atoms to be deleted from its share of blackboard because they have been previously deleted by other agents. The activation request from T is no more valid because it was issued with respect to an obsolete blackboard.

The Scheduler rejects all invalid activation requests. It uses a unique mark for each deletion operation. Every time it has to send a deletion list to one or more agents, the Scheduler increases such a global mark, then it associates the mark to messages and updates its data about the agents with whom it has communicated (i.e., it records the last mark sent to each agent). When an agent sends the Scheduler an activation request,
it associates to its request the value of the last mark received from the Scheduler itself. When the Scheduler receives a request from an agent, it compares the mark of the request to the last value sent to that agent. If the pair matches then the blackboard used by the agent to satisfy its patterns was valid. Such a request is considered valid and accepted by the Scheduler. But if the two marks are different, this means that the agent has not deleted yet from its blackboard the atoms consumed by another agent, therefore its request of activation is relative to an invalid state. In this case the Scheduler rejects the request.

5.3 Implementation

The abstract machine described above has been implemented on a physically distributed architecture: a local network including both VAX and SUN machines, using the Internet protocol under Unix.

Figure 7 SP run time architecture

The structure of the actual architecture of the SP runtime system is shown in the figure 7. Communications among agents (i.e., Prolog processes) are based on Internet sockets. Each agent is a Prolog process with a local knowledge base. Each agent executes a control loop. All the messages that flow over the network are full Prolog terms (possibly non ground).

5.4 Compilation of communication flows

All the following discussion is based on the assumption that the most expensive
operations in the chosen implementation are communications between processes. So we will try to reduce their number and their length.

We can simplify a graph removing edges and, where possible, nodes as well. We have defined some optimizations that interest mainly the Out Flow Graph. If we remove an edge A connecting two nodes $P_i$ and $P_j$ we remove either an impossible communication from $P_i$ to $P_j$, or a useless synchronization constraint between the patterns $P_i$ and $P_j$.

### 5.4.1 Eliminating edges

Consider the edges in the Out Flow Graph: we want to see if the atom labeling the edge connecting $P_1$ to $P_2$ is really necessary to $P_2$ to satisfy its guards.

**Example.**

Let $a(2) \in \text{Out}(P_1)$, and let $a(X) \in \text{Read}(P_2)$ under the constrain $X>3$. If an agent executing $P_2$ tries to satisfy its Read guard reading the message $a(2)$ such a guard evaluation fails. The compiler eliminates all the edges labeled with messages causing the failure.

### 5.4.2 Patterns never satisfied

The compiler looks for patterns with guards never satisfied. Given a pattern $P$ and an atom $A$ in its guard, if in the Out Flow graph no edge labeled with $A$ enters in the node $P$, then no pattern can produce $A$ and the atom will never be found on the blackboard. Pattern $P$ and all edges entering in it can be removed by the graph. Moreover, $P$ itself could be removed from the source program. The compiler writes a warning message.

### 5.4.3 Patterns trivially satisfied

The compiler looks for patterns with trivially satisfied atoms in Read. Let $\neg p(X)$ be a negative atom in the Read guard of $P$. If $P$ has not any incoming edge labeled with $p(X)$, then $p(X)$ will never be contained in the blackboard and the goal $\neg p(X)$ will succeed. Thus it can be safely deleted.

### 5.4.4 Overall Structure of the Compiler

In this section we shortly describe the logical organization of the SP source-to-source
compiler. The phase of *source to source transformation* transforms by partial evaluation the SP source program into an optimized SP program. The phase of *graph generation* performs a global analysis of the program and generates the graphs. The phase of *optimization* analysis reduces the graphs, aiming at eliminating useless communications, reducing the time spent by the runtime support to deal with expensive network communications. The phase of *object code generation* produces the object (extended Prolog) code consisting of a set of processes. It generates the runtime environment for each agent, that consists of Prolog code augmented with the parts of the graph necessary to the agent.

### 5.5 Performance Comparison

In order to compare the two current implementations of SP, i.e., the interpreter with centralized blackboard [1] vs. the compiler with distributed blackboard described in this section, we made some simple performance evaluation experiments.

![Figure 8 Comparison between interpreter and compiler](image)

*Figure 8* Comparison between interpreter and compiler

We used the Prolog version found in [16]. The comparisons among the Prolog version, the SP interpreted version, and the SP compiler version are shown in figure 8 and were made using the Mastermind problem varying the size of the problem space. The parallel versions run on a network composed of only two workstations, so this experiment is
not conclusive. However, it is clear that with smaller search spaces the interpreter gives better results with respect to the compiler because fewer communications are necessary to find the solution. In the compiled version, in fact, more communications are necessary and therefore if the problems can be solved in short time, the overhead deriving from the communications overcomes the advantages offered by the compiled version. When the complexity of the problem increases, the compilation time optimizations and the higher parallel granularity have a greater impact. They succeed to overcome the costs of the network communications overhead, showing therefore that the phase of compilation and the physical distribution of the blackboard allow to improve the efficiency of the execution of Shared Prolog programs.

6. Conclusions

In this paper we have shown some compilation techniques that improve the efficiency of the language SP with respect to a distributed implementation based on a network of workstations. The first prototype implementation for SP had been realized by a distributed interpreter with a centralized blackboard. In this paper we have described a novel implementation in which we have distributed the blackboard, statically improving the run time efficiency analyzing the structure of the programs and using techniques based on partial evaluation. We are still improving the communication kernel (we plan to use Linda instead of Internet sockets), so fully fledged performance evaluations are still premature.

From a linguistic point of view the main problems in the compiler based implementation is that it is not possible to dynamically activate new agents executing new theories; this is an important feature of the interpretive version. The difficulties are intrinsic to the transformations and the optimizations executed at compilation time. The dynamic connection in the interpreted version represents an interesting and useful language feature.

Another approach that we are investigating consists of developing an incremental algorithm for speeding up the evaluation of the guards. We are currently studying
parallel versions of scheduling algorithms used in parallel production system languages [9,13].

Acknowledgements

We like to thank M.Danelutto for many useful discussions concerning the SP communication kernel. M.Montanari wrote part of the source-to-source compiler. This paper has been partially supported by C.N.R.- Progetto Finalizzato Sistemi Informatici - Calcolo Parallelo and by the Italian M.U.R.S.T. P.Ciancarini was partially supported in Yale by a N.A.T.O.-C.N.R. grant.

References


