Concurrency in Prolog Using Threads and a Shared Database*

Manuel Carro mcarro@fi.upm.es

Manuel Hermenegildo herme@fi.upm.es

School of Computer Science, T.U. Madrid (UPM)

Abstract

Concurrency in Logic Programming has received much attention in the past. One problem with many proposals, when applied to Prolog, is that they involve large modifications to the standard implementations, and/or the communication and synchronization facilities provided do not fit as naturally within the language model as we feel is possible. In this paper we propose a new mechanism for implementing synchronization and communication for concurrency, based on atomic accesses to designated facts in the (shared) database. We argue that this model is comparatively easy to implement and harmonizes better than previous proposals within the Prolog control model and standard set of built-ins. We show how in the proposed model it is easy to express classical concurrency algorithms and to subsume other mechanisms such as Linda, variable-based communication, or classical parallelism-oriented primitives. We also report on an implementation of the model and provide performance and resource consumption data.

1 Introduction

Concurrency has been studied in the context of a wide range of programming paradigms, and many different mechanisms have been devised for expressing concurrent computations in procedural programming languages [4]. In this paper we are interested in developing a model of concurrency for Prolog. In fact, concurrency has also received much attention in the context of logic languages. However, most previous proposals have the drawback that they either involve large modifications to standard Prolog implementations, and/or the communication and synchronization facilities provided do not fit as naturally within the Prolog language model as we feel is possible.

One approach to concurrency is represented by the family of *concurrent logic languages*, which includes PARLOG, Concurrent Prolog, Guarded Horn Clauses, Janus, and others (see [25, 30] and their references). These languages share a number of characteristics. First, concurrency is implicit, i.e., every literal in a clause body represents a concurrent process. While this can be attractive in principle, it can cause an unnecessarily high number of processes to be generated, and can also make it difficult to write sequential code. Consequently, we feel that the spawning of a concurrent computation should preferably be done via an explicit language primitive (or, conversely, that the language should have an explicit operator for sequential composition of processes) [17, 8]. While there is certainly still much debate on this issue, it is noteworthy that designs which started out as implicitly concurrent languages, such as Oz [13], have opted for the explicit concurrency approach in more recent versions.

*This work has been partially supported by the CICYT Project ELLA, TIC 96-1012-C02-01, and Fulbright UPM-NMSU Technology Exchange Program project ECCOSIC. We also want to thank the anonymous referees for their very valuable comments.

Another common characteristic of the family of concurrent logic languages is that communication and synchronization between processes is performed by means of shared variables. This initially extremely attractive model has turned out to suffer from a number drawbacks in practice. The most important one is that backtracking has been found very difficult to implement. As a result, most of these languages have simply eliminated backtracking altogether. However, we feel that backtracking is an integral part of the Prolog control model, and that eliminating it is therefore not an option in our context.

A more recent family of languages, including Andorra-I [24] and AKL [18] directly addresses the backtracking problem. In Andorra-I, choices are suspended until a deterministic path can be taken. AKL allows *encapsulated search* while, at the same time, communicating deterministic bindings outside such encapsulated operations. While these very interesting proposals solve to some extent the backtracking problem, they still have what we perceive as drawbacks in our context: apart from supporting the implicit concurrency approach, they require quite specialized implementation technology, particularly with regard to the representation of variables (which sometimes also results in a slowdown of sequential execution). As a result, they require what would be major changes to standard Prolog abstract machines, which are almost invariably based on the WAM [1, 31]. Also, the operational semantics of these models (specially in the case of AKL) are far removed from that of Prolog and it does not appear straightforward to adapt the related ideas to Prolog without affecting the language in a significant way.

Another interesting approach to concurrency is to use the capabilities of parallel Prolog implementations such as &-Prolog [16], Aurora [22], MUSE [2], ACE [12], etc. These systems proved early on that it is possible to construct very efficient multi-worker (i.e., multi-thread) Prolog engines. While these systems were designed with parallelism in mind, they contain many useful basic building blocks for a concurrent system. In fact, some of these engines have been used for some time now to implement concurrent applications (e.g., Aurora in [28]) or even fully concurrent and distributed Prolog systems (e.g., &-Prolog in [8, 15], which uses explicit creation of threads and a combination of a blackboard and marked shared variables for communication). Our approach builds on these experiences, but tries to improve on them in several areas. First, from the engine point of view, although these systems were all designed as extensions of the WAM, we feel that for practical purposes a model that requires smaller modifications to the WAM is useful. Furthermore, we are interested in finding better solutions to the communication and synchronization among threads.

The difficulties associated with shared variable-based communication have led to the development of other communication and synchronization primitives. One of them is *ports* [18], used for example by MT-SICStus [10]. MT-SICStus implements a relatively simple design, some aspects of which are derived from Erlang [3]. Threads can be `spawn`ed (with an initial goal) and `kill`ed. They can also `send` and `receive` fresh copies of terms, using a single port per thread. This allows creating a simple *goal server* which executes whichever goal it receives. This is certainly a quite useful model, but we feel that it can be improved upon: ports (similarly to streams) do not have a clean interaction with backtracking (what is sent down a port is not put back on backtracking), the overall model is somewhat restrictive, and the explicit codification of the above mentioned server seems to be needed for most applications. In Oz 2.0 [13] threads are started using an explicit construction which returns a value as if it were an expression. Message passing and synchronization use (in a similar way to AKL) shared variables and also an abstract data type `Port`, which can be shared among threads and passed to other functions/procedures, perhaps as parts of other data structures. A shared stack allows fast communication among threads. Exceptions can be *injected* into other threads. A very interesting characteristic of Oz is the caching and coherence mechanism used in concurrent, distributed execution.

An alternative to ports is to use Linda blackboards [9]. Linda is a simple but powerful concurrency paradigm which focuses on (and unifies) the mechanisms for communication and synchronization. The Linda model assumes a shared memory area (the blackboard) where tuples are written and read (using pattern matching) by concurrent processes. Writing and reading are performed through a number of primitives and are atomic actions. Reading may suspend if the requested tuple is not present in the blackboard, this being the main synchronization mechanism. This approach has been made available either natively or as libraries in several logic programming systems (e.g., [6, 27, 8, 15, 29, 5]). A practical example of this approach is the BinProlog-based Jinni system [29]. Jinni has a relatively rich set of primitives for creating threads (called *engines*), also providing them with a starting goal and means for recovering the answers returned. Obtaining multiple answers from an engine is possible by `ask`ing the engine, until this call finally fails. Coordination among engines is achieved using a Linda-style set of primitives, which access and modify a shared blackboard. These operations allow writing and reading (with pattern matching) from the blackboard, and gathering in a list and at once all the tuples which match a pattern. A form of constraint-based synchronization is also available, as well as the possibility of migrating computations to remote places. Jinni is quite attractive, but still the main means for communication, the blackboard, is an external object to the Prolog model.

Although the approaches proposed to date are undoubtedly interesting and useful, we feel that they either do not provide all the features we perceive as most interesting in a practical concurrent Prolog system or they require complex implementations. The main novelty of the model we present in this paper is that both communication and synchronization are based on concurrent, atomic accesses to the shared Prolog database, which we argue can be used in the same way as a blackboard. We will show that, apart from the conceptual simplification, this choice creates very useful synergies in the overall language design, while remaining reasonably easy to implement. In our approach, an extension of the `assert/retract` family of Prolog calls allows suspension on calls and redos. We show that these primitives, when combined with the Prolog module system, have the same or richer functionality than blackboard-based systems, while fitting well within the Prolog model: they offer full unification instead of pattern matching on tuples and provide a clean interaction with Prolog control, naturally supporting backtracking. The model, as described in this paper, is available in the current distribution of CIAO (a next-generation, public domain Prolog system – see `http://www.clip.dia.fi.upm.es/Software`).

2 A First Level Interface

As mentioned before, the significant effort realized by the logic programming community in building parallel Prolog systems has proven that it is possible to construct sophisticated and very efficient multi-worker Prolog engines. However, it is also true that the inherent complication of these systems has prevented their availability as part of mainstream Prolog systems (with the possible exception of SICStus/MUSE, and, to a more limited extent, &-Prolog, Aurora, Andorra-I, and ACE). One of our main objectives in the design of the proposed concurrency model has been to simplify the low-level implementation, i.e., the modifications to the Prolog *engine* required, in order to make it as easy as possible to incorporate into an existing WAM-based Prolog system. Such simplicity should also result in added robustness.

With this in mind, we start by defining a set of basic building blocks for concurrency which we argue can be efficiently implemented with small effort. We will then stack higher-level functionality as abstractions over these basic building blocks.

2.1 Primitives for Creating Threads

A thread corresponds conceptually to an independent Prolog evaluator, capable of executing a Prolog goal to completion in a local environment, i.e., unaffected by other threads. It is related to the notion of *agent* [16] or *worker* [22, 2] used in parallel logic programming systems.

Basic Thread Creation and Management: Thread creation is performed by calling `launch_goal/2`, which is similar in spirit to the `&/1` operators of &-Prolog, the `spawn` of MT-SICStus, or the `new_engine` of BinProlog. A call to `launch_goal(Goal, GoalHandle)` first copies `Goal` and its arguments to the address space of a new thread and returns a *handle* in `GoalHandle` which allows the creating thread to have (restricted) control of and access to the state of the created thread. Execution of `Goal` then proceeds in the new thread within an independent environment. An exception may be raised if the spawning itself is not successful, but otherwise no further communication or synchronization with the caller occurs until a call to `join_goal(GoalHandle)` is made, unless explicitly programmed using the synchronization and communication primitives (Section 2.2).

A call to `join_goal(GoalHandle)` waits for the success or failure of the goal corresponding to `GoalHandle`. If a solution is found by the concurrent goal, this goal can at a later time be forced to backtrack and produce another solution (or fail) using `backtrack_goal(GoalHandle)`. When no more solutions are needed from a given goal, the builtin `release_goal(GoalHandle)` must be called to release the corresponding thread. This also frees the memory areas used during the execution of the goal, and makes them available for other goals.

A number of specializations of `launch_goal/2` are useful in practice. A simplified version `launch_goal(Goal)` causes `Goal` to be executed to completion (first solution or failure) in a new thread, which (conceptually) then dies silently. This behavior is useful when the created thread is to run completely detached from its parent, or when all the communication is performed using the communication / synchronization primitives (Section 2.2). There are also other primitives, such as `kill_goal(GoalHandle)`, which kills the thread executing that goal and releases the memory areas taken up by it, which are useful in practice to handle exceptions and recover from errors.

Implementation Issues and Performance: The implementation requires the Prolog engine to be *reentrant*, i.e., several invocations of the engine code must be able to proceed concurrently with separate states. The modifications required are well understood from the parallel Prolog implementation work (we follow [16]).

A concurrent goal is launched by copying it with fresh, new variables, to the storage areas of a separate *engine*, which has its own working storage (*stack set*[1]) and attaching a thread (*agent*) to this stack set. The code area is shared and visible by all engines. Goal copying ensures that execution is completely local to the receiving WAM. This avoids many complications related to the concurrent binding and unbinding (on backtracking) of shared variables, since bindings/trailing, backtracking, and garbage collection are always local to a WAM, and thus need no changes with respect to the original, single-threaded implementation.

An important issue is how to handle goals which are suspended (e.g., are waiting at a *join*, or have executed other primitives which may cause suspension, to be described later) and goals which have returned a solution but are waiting to be joined. This issue was studied in [14, 26]. There are two basic solutions: one is to *freeze* the corresponding stack set, which then cannot be used until the goal is resumed. The same occurs with a stack set containing the execution of a goal which has produced a solution but has pending alternatives. When this stack set

[1] The memory areas used by a WAM, which are usually managed using a stack policy.

Thread creation	Engine coupling	Engine creation
2.03 ms / 702 LI	3.16 ms / 1091 LI	10.3 ms / 3579 LI

Table 1: Profile of engine and thread creation (average for 800 threads).

is asked for another alternative a thread attaches to it and forces backtracking. This approach has the advantage of great simplicity at the cost of some memory consumption: it causes memory areas of the WAM (the upper parts of frozen stack sets) to be unused, since a new WAM is created for every new goal if the other WAMs are frozen and cannot be reused. WAMs are reused when a goal detaches after completion or when they are explicitly freed via a call to `release_goal/1`, in which case they are left empty and ready to execute another goal. The alternative is to reuse frozen stacks using *markers* to separate executions corresponding to different concurrent goals [14, 26]. This can be more efficient in memory consumption, but is also more complicated to implement. We have implemented an intermediate approach which is possible if the stacks in the engine can be resized dynamically. We start with very small stacks which are expanded automatically as needed. This has allowed us to run quite large benchmarks with a considerable number of threads without running into memory exhaustion problems. It is also possible to shrink the stacks upon goal success, so that no memory is wasted in exchange for a small overhead. This is planned for future versions.

For our experiments we implemented the proposed primitives in the CIAO Prolog engine (essentially a simplified version of the &-Prolog engine, itself an independent evolution from SICStus 0.5-0.7, and whose performance is comparable to current SICStus versions running emulated code). We have used a minimal set of the POSIX thread primitives, in the hope of abstracting away the quite different management of threads provided in different operating systems, and to favor porting among UNIX flavors. All the experiments reported in this paper have been run on a SparcCenter 2000, with 10 55MHz processors, Solaris 2.5, CIAO-Prolog 0.9p75. All the measurements have been made using *walltime* clock.

Table 1 provides figures for several operations involving threads. Since the overhead per thread seems to remain fairly constant with the number of threads used, we show the average behavior for 800 (simultaneous) threads. Measurements correspond to the Prolog view of the execution: they reflect the time from a `launch_goal(Goal)` is issued, to the time `Goal` is started. Times are given in ms. and, to abstract away from the processor speed, in "number of naive-reverse Logical Inferences" (at the ratio of 345 logical inferences per millisecond, the result given by `nrev` in the machine used). Although these numbers depend heavily on the implementation of O.S. primitives, we feel that providing them is interesting, since they are real indications of the cost of thread management.

The column labeled "Thread creation" reflects the time needed to start a thread, including the time used in copying the goal. The column labelled "Engine coupling" adds the time needed to locate an already created, free WAM, and to attach to it, and includes the initialization of the WAM registers. The column "Engine creation" takes into account the time used in actually creating a new engine (i.e., memory areas) and attaching to it. The last one is, as expected, larger, and this supports the idea of not disposing of the engines which are not being used. These figures are also useful in order to determine the threshold which should be used to decide whether execution should be sequential or parallel, based on granularity considerations [11]. Regarding memory consumption, the addition of thread support increased only very marginally the memory space needed per WAM.

A Note on Avoiding the Copy of the Calling Goal: Copying goals on launch, despite its advantages, may be very expensive. We support an additional set of

Operation	**Linda**	`concurrent/1` **Facts**	
Put tuple	`out`	`asserta/1, assertz/1`	
Wait, read tuple	`read`	`call/1`, simple call	
Wait, read, delete tuple	`in`	`retract/1`	
Read tuple or fail	`read_noblock`	`call_nb/1`	(+)
Read and delete or fail	`in_noblock`	`retract_nb/1`	(+)
No more tuples	—	`close_predicate/1`	
(More tuples may appear)	—	(`open_predicate/1`)	

Table 2: Comparing Linda primitives and database-related Prolog primitives

primitives which perform sharing of arguments instead of copying.[2] To simplify the implementation and avoid a performance impact on sequential execution, concurrent accesses to the shared variables are not protected. The programmer has to ensure correct, locked accesses to them, including the effects of backtracking in other agents.

More complex management of variables can be built on top of these primitives by using, for example, attributed variables for automatic locking and publication of deterministic bindings, with techniques similar to those in [15], and incremental, on demand copying of goal arguments, as shown in [8, 20].

2.2 Synchronization and Communication Primitives

For the reasons argued previously, in this design we would like to use communication and synchronization primitives simpler to implement than those based on shared-variable instantiation. The use of the dynamic database that we propose as a concurrent shared repository of terms for communication and synchronization requires some (local) modifications to the semantics of the accesses to the dynamic database, but also results in some very interesting synergies.

Making the Database Concurrent: We start by assuming that we can mark certain dynamic predicates as concurrent by using a `concurrent/1` declaration. The implication is that these predicates can be updated concurrently and atomically by different threads. We also assume for simplicity that these predicates will only contain facts, i.e., they are `data/1` predicates in the sense of [7] and Ciao (this makes them faster and helps global analysis). Finally, we assume that if a concurrent predicate is called and no matching fact exists at that time in the database, then the calling thread *suspends* and is resumed only when such a matching fact appears (i.e., is asserted by a different thread, instead of failing). With these assumptions, there is a relationship between the Linda primitives (Table 2, middle column) and the Prolog *assert/retract/call* family of builtins in the context of concurrent predicates (right column). The first three Linda operations, `out/read/in`, have now clear counterparts in terms both of information sharing and synchronization. In the following example:

```
:- concurrent state/1.
p:- launch_goal(q),                    q:- <...produce Result...>, !,
    state(X), !,                           asserta(state(correct(Result))).
    (X = failed -> ... ; ...).         q:- asserta(state(failed)).
```

`p` launches predicate `q` and waits for notification of its final state, which may be a `Result` or a *failed* state (the use of the Prolog cuts will be clarified further later). Making the dynamic predicate `state/1` be `concurrent` ensures atomic updates and the suspension of the call to `state(X)` in `p`.

[2] That is, as long as the goals are being executed in the same machine (Section 3).

One interesting difference with Linda primitives appears at this point: it is clear that we may want to be able to backtrack into a call to a concurrent predicate (such as the one to `state(X)` above). The behavior on backtracking of calls to concurrent predicates is as follows: if an alternative unifying fact exists in the database, then the call matches with it and proceeds forward again. If no such fact exists, then execution suspends until one is asserted. This is the natural extension of the behavior when the predicate is called the first time, and makes sense in our concurrent environment where facts of this predicate can be generated by another thread and may appear at any time. It allows, for example, implementing producer-consumer relations using simple failure-driven loops. In the following temperature example a thread accesses a device for making temperature readings, and asserts these, while a concurrent reader accesses them in a failure driven loop as they become available.

```
:- concurrent temp/1.

temperature:- launch_goal(read_temp), produce_temp.

produce_temp:-
    ( read_temp_device(Temp) ->
         assertz(temp(Temp)),
         produce_temp
      ;  assertz(temp(end)) ).

read_temp:- temp(Temp),
          ( Temp = end ->
            true
          ; <...work with Temp...>,
            fail ).
```

When no more temperature readings are possible, `read_temp_device/1` fails and the `end` token, instead of a a temperature, is stored, which causes the reader to exit. Conceptually, when backtracking is performed, the *next clause pointer* moves downwards in the clauses of the `temp/1` predicate until the last fact is reached. Then, the calling thread waits for more facts to appear. Note that assertion is done using `assertz/1`, which adds new clauses at the end of the predicate, so that they can be seen by the reader waiting for them. If `asserta/1` were used, newly added facts would not be visible, and thus the reader would not wake up and read the new data available. Also, note that the data produced remains, so that other readers could process it as well by backtracking over it. For example, assume the temperature asserted has the time of the reading associated with it. Different readers can then to consult the temperature at a given time concurrently, suspending if the temperature for the desired time has not been posted yet. Alternatively, if the call to `temp(Temp)` above is replaced with a call to `retract(temp(Temp))`, then each consumer will eliminate a piece of data which will then not be seen by the other consumers. This is useful for example for implementing a task scheduler, where consumers "steal" a task which will then not be performed by others.

The concurrent database thus allows representing a changing outside world in a way that is similar to other recent proposals in computational logic, such as condition-action rules [19]. A sequence of external states can be represented by a predicate to which a series of suitably timestamped facts are added monotonically, as in the temperature sensor example above. Processes can sense this state and react to it or suspend waiting for a given outside event to happen.

One nice characteristic of the approach, apart from naturally supporting backtracking, is that many concurrent programs using shared facts are very similar to the non-concurrent ones. There is, however, a subtle difference which must be taken into account: when calling standard, non-concurrent facts with alternatives, the choicepoint disappears when the last fact is accessed. In the reader the "last fact" was assumed to be that with the "`end`" token, but this did not make the choicepoint pushed in by the access to the database go away. A failure at a later point of the reader would cause it to backtrack to this choicepoint, and probably suspend — which may or may not be desired. Getting around this behavior is possible by simply putting an explicit cut at the point in which we decide that no more facts are needed (i.e., the communication channel has been conceptually closed), so that the dynamic concurrent choicepoint is removed. This is the reason for the cut in the first example of synchronization: we just wanted to wait for a fact to be present, and then we did not want to leave the choicepoint lying around.

Closing Concurrent Predicates: There are cases in which we prefer failure instead of suspension, if no matching is possible. This can be achieved in two ways. The first one is using non-blocking (`_nb`) versions of the retract and call primitives (marked + in Table 2), which fail instead of suspending, while still ensuring atomic accesses and updates. The second, and more interesting one, is explicitly *closing* the predicate using `close_predicate/1`. This states that all alternatives for the predicate have been produced, and any reader backtracking over the *last* asserted fact will then fail rather than suspending. The example can now be coded as:

```
:- concurrent temp/1.
temperature:- launch_goal(read_temp), produce_temp.
produce_temp:-                              read_temp:-
    ( read_temp_device(Temp) ->                 temp(Temp),
         assertz(temp(Temp)),                   <...work with Temp...>,
         produce_temp                           fail.
      ;  close_predicate(temp/1) ).         read_temp.
```

where the call to `temp(Temp)` eventually fails after the predicate is closed. This is useful for example for marking that a stream modeled by a concurrent predicate is closed: all the threads reading/consuming facts from this predicate will fail upon the end of the data. For completeness, a symmetrical `open_predicate/1` call is available in order to make a closed predicate behave concurrently again (although it is arguably best practice not to re-open closed predicates).

Local Concurrent Predicates: New, concurrent predicates can be created dynamically by calling the builtin `concurrent/1`. The argument to `concurrent/1` can be a new predicate. Also, if the argument of the call to `concurrent/1` contains a variable in the predicate name, the system will create dynamically a new, local predicate name. This allows *encapsulating* the communication, which is now private to those threads having access to the variable:

```
temp:- concurrent(T/1), launch_goal(read_temp(T)), produce_temp(T).
produce_temp(T):-                           read_temp(T):-
    ( read_temp_device(Temp) ->                 T(Temp),
         assertz(T(Temp)),                      <...work with Temp...>,
         produce_temp                           fail.
      ;  close_predicate(T/1) ).            read_temp(_).
```

where we could replace the higher-order syntax `T(X)` supported by CIAO Prolog with calls to `=..` and `call/n` (e.g., `call(T,Data)`). Note that the functionality at this point is not unlike that of a *port*, but with a richer backtracking behavior.

Another way of encapsulating communication stems from an interesting synergy between the concurrent database and the module system. Concurrent predicates are, as usual, in principle local to the module in which they appear. If they are not exported, they constitute is a channel which is local to the module and can only be used by the predicates in it. The module-local database thus acts as a local blackboard. By exporting and reexporting concurrent predicates between modules, separate, *private blackboards*, can be easily created whose accessibility is restricted to those importing the corresponding module. This is particularly useful when several instantiations (objects) are created from a given module (class) – see [23].

Logical View vs. Immediate Update: A final difference between concurrent predicates and standard dynamic predicates is that the logical view of database updates [21], while convenient for many reasons, is not really appropriate for them. In fact, if this view were implemented then consumers would not see the facts produced by sibling producers. Thus, an immediate update view is implemented for concurrent predicates so that changes are immediately visible to all threads.

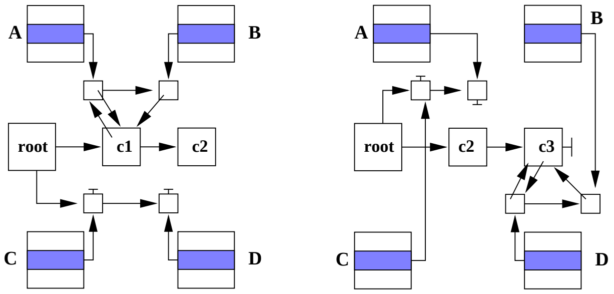

Figure 1: Choicepoints and suspended calls before and after updating clauses

Locks on Atoms/Predicate Names: A method for associating semaphores [4] to atoms / predicate names is available. Mimicking those in procedural languages, a counter is associated with each atom which can be tested or set atomically using `atom_lock_state(+Atom, ?Value)`. It can be atomically tested and decremented if its value is non-zero, or waited on if it is zero, using `lock_atom(+Atom)`, and incremented atomically using `unlock_atom(+Atom)`. The implementation is very cheap, avoiding the overkill of simulating semaphores with concurrent predicates, when only a simple means of synchronization is needed.

Implementation Issues and Performance: Concurrent accesses are made atomic by using internal, user-transparent locks, one per predicate. Every call to a concurrent predicate creates a dynamic choicepoint with special fields. In particular, its *next alternative* field points to the next clause to try on backtracking through an indirect pointer. All the indirect pointers from different choicepoints leading to a given clause are linked together into a chain reachable from that clause, so that any goal updating the predicate can access and relocate all of them atomically if needed (for example, if the clause is removed). Calls which suspend do not have their associated choicepoint removed, and the corresponding indirect pointers are linked in a separate suspension chain (Figure 1, left). When a thread tries to access its next alternative and no alternative matching clause exists, the thread waits on changes to that indirect pointer instead of failing. This behavior ultimately depends on whether the call was blocking or not, and on whether the predicate was closed or not at the time. We discuss the interesting case of blocking calls on open predicates.

When a clause is removed, the chain of indirect pointers leading to it is checked: some of the pointers might be moved forwards in the clause list to the next possibly matching instance (as dictated by indexing), and in some cases it can be determined that no matching clause exists. In the latter case, they are linked to the chain of suspended calls. On the other hand, every time a new clause is appended, the list of suspended calls is checked, and those which may match the new clause (again, according to indexing), are made to point to that new clause. This is performed even if the affected goal is not actively waiting on an update of the clause, but executing some other code.

Figure 1 depicts a possible state of the database before and after some clause updates take place. On the left, choicepoints **A** and **B** point (indirectly) to clause `c1` as next clause to try. Choicepoints **C** and **D** point to *null* clauses, and they are either suspended, or they would suspend should they backtrack now. Let us remove `c1` and add `c3`. The thread which adds / removes clauses is in charge of updating the affected pointers, based on indexing considerations. The call from **A**, which was pointing to `c1`, does not match neither `c2` nor `c3`, so it points to the *null* clause now, and is enqueued in the list of calls to suspend. The call from **B** may match `c3` but does not match `c2`, and its indirect pointer is set accordingly. The call from **C** does not match `c3` and so its state does not change. And, finally, the

Primes	Conc	Data
5000	1511	1915
10000	2475	5204
15000	4775	9100
20000	6386	12560
25000	9061	17804
30000	11900	24298
35000	13252	29450

Table 3: Sieve of Erathostenes

Fact spec	Memory	bytes/fact	bytes/arg
p/0	1264	21.57	—
p/1	1753	29.91	8.34
p/2	1871	31.93	5.18
p/4	2105	35.92	3.58
p/8	2571	43.87	2.78
p/16	3507	59.85	2.39
p(g/1)	1615	27.56	5.99
p(g/2)	1753	29.91	4.17
p(g/4)	1967	33.57	3.00
p(g/8)	2435	41.86	2.53
p(g/16)	3373	57.56	2.24

Table 4: Memory usage, 60000 facts

call from **D** might match `c3`, and it is updated to point to this clause.

The cut needs some additional machinery to retain its semantics. Not only the (dynamic) choicepoints in the scope of a cut should be swept away (which boils down to updating a pointer), but also the possibly suspended goals corresponding to the concurrent choicepoints must be removed. This is currently done by traversing part of the choicepoint stack, following the links to suspended calls, and removing them.

The implementation of concurrent predicates is not trivial, but we argue that it is much simpler than implementing variable-based communication that behaves well on backtracking. Also, it affects only one part of the abstract machine, database access, which is typically well isolated from the rest. In our experience, the changes to be performed are fairly local. The resulting communication among threads based on access to the database may be slower than communication using shared variables, although, depending on the implementation, reading can be faster. However, note that in this design we are not primarily interested in speed, but rather in flexibility and robustness, for which we believe the proposed solution is quite appealing. Also, in the proposed implementation the execution speed of sequential code which makes no use of concurrency is not affected in any way, which is not as easy with a shared-variable approach. Furthermore, the fact that concurrent predicates should not meet the logical view of database updates [21], eliminates the need to check whether a fact is alive or not within the time window of a call, which makes, in some cases, the access and modification of concurrent predicates up to more than twice as fast as that of standard dynamic predicates.

As an example of the impact on speed of the immediate database updates, Table 3 shows timings (in milliseconds) for a database implementation of the well-known Sieve of Erathostenes, using a failure loop to both traverse the table of live elements and to remove multiples. The "Data" column corresponds to the version which uses the CIAO `data/1` declaration (which is faster than `dynamic`, and specialized for facts). The "Conc" row uses the `concurrent/1` declaration. Clause liveness (i.e., whether a given clause should or not be seen by a given call) must be tested quite often in this case, which accounts for the performance jump. On the other hand, other patterns of accesses to database perform this liveness test quite sparingly (if at all), and benefit less from the immediate update, suffering instead from the mandatory lock of the predicates being accessed. However, the factors seem to compensate even in the worst cases since we have not been able to find noticeable slowdowns.

With respect to memory consumption, Table 4 lists average memory usage per fact and per argument for the CIAO Prolog implementation in a benchmark which asserts 60000 facts to the database. A fact `p` with different arguments (integers)

1	2	4	6	12
4327	2823	1687	1400	1625

Table 5: Adding and removing facts from a database, 10 processors available

```
:- concurrent fork/1.
                                        philosopher(ForkLeft):-
philosophers:-                              ForkRight is (ForkLeft mod 5) + 1,
    atom_lock_state(room, 0),               think,
    launch_goal(philosopher(1)),            lock_atom(room),
    launch_goal(philosopher(2)),            retract(fork(ForkLeft)),
    launch_goal(philosopher(3)),            retract(fork(ForkRight)), !,
    launch_goal(philosopher(4)),            eat,
    launch_goal(philosopher(5)),            assertz(fork(ForkLeft)),
    atom_lock_state(room, 4).               assertz(fork(ForkRight)),
                                            unlock_atom(room),
eat:- ...         fork(1). fork(2).         philosopher(ForkLeft).
think:- ...       fork(3). fork(4). fork(5).
```

Figure 2: Code for the Five Dining Philosophers

was asserted, as well as a fact with a single argument, containing a functor with different numbers of arguments (integers again). It is encouraging that these figures are well behaved, as we may expect large numbers of facts asserted in the database.

Another interesting issue is the impact of contention in concurrent predicate accesses. Our implementation ensures that concurrent accesses to different predicates will not interfere with each other: Table 5 shows speeds for the access and removal of a total of 10000 facts using different numbers of threads. Each thread accesses a different predicate name, which results in speedups until the number of threads is greater than the number of available processors, when other contention factors appear. However, there is obviously some interference in the concurrent accesses to the same predicate.

3 Some Applications and Examples

We now illustrate the use of the proposed concurrency scheme with some examples.

The Five Dining Philosophers: Figure 2 presents the code for the problem of the Five Dining Philosophers, with the aim of showing how a standard solution can be adapted to the concurrent database approach. The code mimics the solution presented in [4]. Each philosopher is modeled as a concurrent goal which receives its number as an argument. Fork-related actions are modeled by accesses to a concurrent predicate `fork/1`. A global semaphore, associated with the atom `room`, controls the maximum number of philosophers in the dining room, and also makes sure that all philosophers start at once.[3] No attempt is made to record when a philosopher is thinking or eating, but this can be done by asserting a concurrent predicate recording what every philosopher is doing at each time.

A Skeleton for a Server: A *server* is a perpetual process which receives requests from other programs (clients) and attends them. Typically, the server should accept more queries while previous ones are being serviced, since otherwise the service would stop temporarily. Therefore, servers usually are multithreaded, and children

[3] Actually, this is not strictly needed: letting philosophers think and eat as they become alive does not change the behavior of the algorithm, but this decision illustrates the use of atom-based locks for global synchronization.

	Start thread	Gather bindings
No handle, local	`..., G &, ...`	—
Handle, local	`..., G &> H, ...`	`H <&`
Remotely concurrent	`(G &) @ S`	—
Locally concurrent, remote execution	`(G @ S) &`	—
Remote handle, remotely concurrent	`(G &> H) @ S`	`(H <&) @ S`
Local handle, remote execution	`(G @ S) &> H`	`H <&`

Table 6: Starting concurrent / distributed goals and waiting for bindings.

fork from the parent in order to handle individual requests. A simple skeleton for a server is shown in Figure 3. The main thread waits for a request and, when one arrives, launches a child thread to process it. The server itself is started within the context of a catch/throw construction which will exit the execution should the server receive any external signal.[4]

Possible internal errors of the server can be dealt with by the `service/1` predicate itself, since each one of its invocations is detached from the main thread. The shared database provides a communication means in case the children have to report any data to the dispatcher.

```
main:- catch(server, _AnyError, halt).

server:-
        wait_for_request(Query),
        launch_goal(service(Query)),
        server.
```

Figure 3: A skeleton for a server

Implementing Higher-Level Concurrency Primitives: The interface offered by the primitives related to threads, locks, and database is sufficient for building many different concurrent programs, but it is somewhat low-level. For example, the number of simultaneous threads has to be controlled explicitly as part of the application code. Similarly, waiting for completion of the computation of a thread and accessing the bindings created by it need the execution of a (fixed) sequence of steps. Also, implementing backtracking over concurrent goals requires some often repeated coding sequences. Such sequences are clear candidates to be abstracted as higher-level constructs.

Using the basic primitives, we have implemented the set of concurrency and distributed execution constructs proposed in [15, 8], some examples of which are shown in table 6. Remote goals are executed in a server `S`, specified with the placement operator `@/2` (so that, for example, `G @ S` means "execute `G` at `S`, wait for its completion, and import the bindings performed"). Handles (`H`) allow waiting for the (remote) completion of the goal, and gathering the bindings. Lack of space prevents us from including the actual implementation code, but it is easy to port the implementations given in [15, 8]. Using concurrent predicates instead of the external blackboard used there results in a simplification of the code. Significant simplifications also stem from the fact that with the proposed primitives goals which have produced a solution can be left frozen and then asked for additional solutions. Thus, concurrent and distributed goals now need not be called in the context of `findall`.

As a simple example, we discuss the implementation of a version of the traditional &-Prolog `&/2` operator, which, placed instead of a comma, specifies that the two adjacent goals are to be executed in parallel and independently: `GoalA & GoalB`. This operator was implemented at a very low-level (i.e., modifying the underlying abstract machine) in the &-Prolog system [16] and in other systems [12], which resulted in very good performance, but at the cost of a non-trivial amount of implementation work. Figure 4 shows the code for our source-level implementation

[4]Exceptions in CIAO Prolog are installed on a per-thread basis, so every concurrent goal can have its own exception handlers without altering the behavior of the other threads.

```
:- concurrent goal_to_execute/2.
:- concurrent solution/3.

GoalA & GoalB :- new_id(IdA), assertz(goal_to_execute(IdA, GoalA)),
                 call_with_result(GoalB, ResultB),
                 (   retract_nb(goal_to_execute(IdA, GoalA)) ->
                     call_with_result(GoalA, ResultA)
                 ;   repeat, perform_some_other_work(IdA, GoalA, ResultA), !),
                 ResultA = success, ResultB = success.

perform_some_other_work(Id, Sol, Res):- retract_nb(solution(Id, Sol, Res)).
perform_some_other_work(_Id, _Sols, _Result):-
        retract_nb(goal_to_execute(Id, Goal)), !,
        call_with_result(Goal, Result),
        assertz(solution(Id, Goal, Result)),
        fail.

scheduler:- retract(goal_to_execute(Id, Goal)),
            call_with_result(Goal, Result),
            assertz(solution(Id, Goal, Result)),
            fail.

call_with_result( Goal, success) :- call(Goal), !.
call_with_result(_Goal, failure).
```

Figure 4: Code for an and-parallel scheduler for deterministic goals

which assumes that the goals to be executed are deterministic. Extending it for non-deterministic goals is easy, but makes the code too long for our space limitations. However, we will compare performance results for both the simple implementation and the one which fully supports backtracking.

In this implementation, the parallel operator `&/2` assigns a unique identifier to every parallel conjunction. One of the parallel goals is executed locally, while the other is stored in the database, together with its identifier, waiting for a scheduler to pick it up. Such a scheduler is implemented by the predicate `scheduler/0`. To use N processors of a parallel machine, $N-1$ threads should be created, all running initially `scheduler/0`. As soon as one goal is posted to the database, one of the threads running the scheduler grabs and executes it, leaves the solution in the database, and fails in order to wait for another goal. If no free schedulers are available, the main thread may find, upon completion of the local goal, that the goal stored in the database is still there. Then, this local thread picks it up and executes it locally. On the other hand, if the solution waited for is not in the database, and the goal left there from the conjunction has been taken, the main thread switches personality and tries to execute any other goals present in the database while also checking whether the solution it requires for the original goal has been posted or not.

This very naive implementation cannot, of course, achieve the same performance as &-Prolog (and this is obviously not the objective of the exercise). However, it is interesting that a correct selection of the granularity level [11] does produce speedups due to parallel execution on at least some benchmarks. Table 7 shows times (in milliseconds) for the parallel execution of the doubly recursive Fibonacci benchmark (computing the 24^{th} Fibonacci number) using the scheduler for deterministic goals. Each column is labelled with a different granularity level, i.e., the column labeled "17" corresponds to a call which stops spawning goals from the call to compute the 17^{th} Fibonacci number downwards. Table 8 shows results for the same benchmark using a scheduler which supports non-deterministic goals. The

	17	18	19	20	21	22	23	24	25
1	1289	1253	1253	1287	1266	1264	1285	1305	1309
3	463	455	480	491	524	623	533	820	1309
5	287	290	312	312	376	319	510	818	1309
7	219	220	210	244	309	318	504	823	1309
9	178	178	197	220	213	330	519	836	1309

Table 7: Deterministic and-parallel scheduler: granularity against no. of agents

	17	18	19	20	21	22	23	24	25
1	1621	1555	1530	1549	1541	1582	1584	1613	1325
3	571	570	570	588	611	746	635	1062	1332
5	367	363	380	392	453	411	617	1029	1332
7	286	290	266	299	378	393	628	1081	1332
9	246	229	247	256	262	403	633	1040	1332

Table 8: Non deterministic and-parallel scheduler: granularity against no. of agents

lower the granularity level, the more goals are executed in parallel, and the smaller they are. The speedups shown approach linearity when execution is performed at a large enough granularity level. As expected, execution also speeds up as more parallel goals are available, until a turning point is reached (at the level of granularity of 17). At this level of granularity the cost of accessing the database for copying goals and recovering the solutions exceeds the speedup obtained from parallel execution. The nondeterministic scheduler, additionally, adds an overhead to the execution, which for this benchmark case ranges from 16% to 30%, with an isolated peak of 39%—and therefore, has a higher granularity, with the "turning point" in 18.

References

[1] Hassan Ait-Kaci. *Warren's Abstract Machine, A Tutorial Reconstruction*. MIT Press, 1991.

[2] K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[3] J. Armstrong, R. Virding, C. Wistrom, and M. Williams. *Concurrent Programming in Erlang*. Prentice Hall, 1996.

[4] M. Ben-Ari. *Principles of Concurrent Programming*. Prentice Hall International, 1982.

[5] K. De Bosschere. Multi–Prolog, Another Approach for Parallelizing Prolog. In *Proceedings of Parallel Computing*, pages 443–448. Elsevier, North Holland, 1989.

[6] A. Brogi and P. Ciancarini. The Concurrent Language, Shared Prolog. *ACM Transactions on Programming Languages and Systems*, 13(1):99–123, 1991.

[7] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[8] D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint Conference on Declarative Programming*, pages 67–78, July 1996.

[9] N. Carreiro and D. Gelernter. Linda in Context. *Comm. of the ACM*, 32(4), 1989.

[10] J. Eskilson and M. Carlsson. SICStus MT—A Multithreaded Execution Environment for SICStus Prolog. In *PLILP'98*, volume 1490 of *LNCS*. Springer, September 1998.

[11] P. López García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.

[12] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos-Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.

[13] S. Haridi. A Tutorial of Oz 2.0. Technical report, SICS, 1996.

[14] M. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, May 1987.

[15] M. Hermenegildo, D. Cabeza, and M. Carro. Using Attributed Variables in the Implementation of Concurrent and Parallel Logic Programming Systems. In *ICLP'95*. MIT Press, June 1995.

[16] M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.

[17] M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In Evan Tick, editor, *Proc. of the 1994 ICOT/NSF Workshop on Parallel and Concurrent Programming*. U. of Oregon, March 1994.

[18] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *ILPS'91*, pages 167–183. MIT Press, 1991.

[19] R. A. Kowalski. Logic Programming with Integrity Constraints. In *Proceedings of JELIA*, pages 301–302, 1996.

[20] E. Lamma, P. Mello, C. Stefanelli, and P. Van Hentenryck. Improving Distributed Unification through Type Analysis. In *Proceedings of Euro-Par 1997*, volume 1300 of *LNCS*, pages 1181–1190. Springer-Verlag, 1997.

[21] T. G. Lindholm and R. A. O'Keefe. Efficient Implementation of a Defensible Semantics for Dynamic Prolog Code. In Jean-Louis Lassez, editor, *JICSLP'87*. The MIT Press, 1987.

[22] E. Lusk et al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.

[23] A. Pineda and M. Hermenegildo. O'ciao: An Object Oriented Programming Model for (Ciao) Prolog. Technical Report CLIP 5/99.0, Facultad de Informática, UPM, July 1999.

[24] V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proc. 3rd. ACM SIGPLAN PPoPP Symposium*. ACM, April 1990.

[25] E.Y. Shapiro. The Family of Concurrent Logic Programming Languages. *ACM Computing Surveys*, 21(3):412–510, September 1989.

[26] K. Shen and M. Hermenegildo. Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 635–640. Springer-Verlag, August 1996.

[27] Swedish Institute of Computer Science, P.O. Box 1263, S-16313 Spanga, Sweden. *Sicstus Prolog V3.0 User's Manual*, 1995.

[28] P. Szeredi, K. Molnár, and R. Scott. Serving Multiple HTML Clients from a Prolog Application. In *Proc- of the 1st Workshop on Logic Programming Tools for INTERNET Applications*, JICSLP"96, Bonn, September 1996.

[29] Paul Tarau. Jinni: Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *PAAM'9*. The Practical Applications Company, 1999.

[30] E. Tick. The Deevolution of Concurrent Logic Programming Languages. *The Journal of Logic Programming*, 23(1–3):89–125, 1995.

[31] D.H.D. Warren. An Abstract Prolog Instruction Set. Technical Report 309, SRI International, 1983.