Concurrent Programming Constructs in Multi-Engine Prolog

Parallelism just for the cores (and not more!)

Paul Tarau

Department of Computer Science and Engineering
University of North Texas
tarau@cs.unt.edu

Abstract

We discuss the impact of the separation of logic engines (independent logic processing units) and multi-threading on the design of parallel programming constructs aware of the fundamental invariant of today's typical computer architectures: the presence of *a few* independent cores. We advocate a combination of coroutining constructs with focus on expressiveness and a simplified, multi-threading API that ensures optimal use of a desired number of cores. In this context, native multi-threading is made available to the application programmer as a set of high-level primitives with a declarative flavor.

Keywords *multi-engine Prolog, coroutining and multi-threading, high-level multi-threading, coordination in multi-engine Prolog, Java-based Prolog implementation*

Categories and Subject Descriptors D.3.3 [*PROGRAMMING LANGUAGES*]: Language Constructs and Features—Concurrent programming structures, Coroutines

General Terms Languages, Performance, Algorithms, Design

1. Introduction

Multi-threading has been adopted in today's Prolog implementations as it became widely available in implementation languages like C or Java.

An advantage of multi-threading over more declarative concurrency models like various AND-parallel and OR-parallel execution schemes, is that it maps to the underlying hardware directly: on typical multi-core machines threads and processes are mapped to distinct "CPUs"[1]. Another advantage is that a procedural multi-threading API can tightly control thread creation and thread reuse.

On the other hand, the explicit use of a procedural multi-threading API breaks the declarative simplicity of the execution model of logic based languages. At the same time it opens a Pandora's box of timing and execution order dependencies, resulting in performance overheads for various runtime structures that need to be synchronized. It also elevates risks of software failure due to programmer errors given the mismatch between assumptions about behavior expected to follow the declarative semantics of the core language and the requirements of a procedural multi-threading API.

In this paper we will describe a design emphasizing *the decoupling of the multi-threading API and the logic engine operations and encapsulation of multi-threading in a set of high-level primitives with a declarative flavor*.

We advocate using threads when there are performance benefits and resorting to determinacy[2] through lightweight, cooperative sequential constructs, when the main reason is programming language expressiveness.

We have implemented the API in the context of an experimental, Java-based Prolog implementation. It is called Lean-Prolog as we have consistently tried to keep implementation complexity under control and follow minimalist choices in the design of built-ins, external language interfaces and a layered, modular extension mechanism.

Lean-Prolog is a new Java-based reimplementation of BinProlog's virtual machine, the BinWAM. It succeeds our *Jinni Prolog* implementation that has been used in various applications [13, 14, 16, 18] as an *intelligent agent infrastructure*, by taking advantage of Prolog's knowledge processing capabilities in combination with a simple and easily extensible runtime kernel supporting a flexible reflexion mechanism [23]. Naturally, this has suggested to investigate whether some basic agent-oriented language design ideas can be used for a refactoring of Prolog's interaction with the external world, including interaction with other instances of the Prolog processor itself.

Agent programming constructs have influenced design patterns at "macro level", ranging from interactive Web services to mixed initiative computer human interaction. *Performatives* in Agent communication languages [4] have made these constructs reflect explicitly the intentionality, as well as the negotiation process involved in agent interactions. At the same time, it has been a long tradition of logic programming languages [6, 9] to use multiple logic engines for supporting concurrent execution.

In this context we have centered our implementation around logic engine constructs providing an API that supports reentrant instances of the language processor. This has naturally led to a view of logic engines as instances of a generalized family of iterators called *Fluents* [17], that have allowed the separation of the first-

[1] In the presence of features like "hyper-threading" CPUs have a somewhat virtual flavor by sharing a core, but typically the OS recognizes them as distinct and allocates threads and processes to them independently. An upper estimate on the maximum amount of parallelism available through multi-threading or multiple processes is then given by the number of CPUs seen by the OS.

[2] Used here to mean "results independent of execution order or timing assumptions". Not to be confused with determinism which in logic programming parlance means returning unique solutions.

order language interpreters from the multi-threading mechanism, while providing a very concise source-level reconstruction of Prolog's built-ins. Later we have extended the original *Fluents* with a few new operations [22] supporting bi-directional, mixed-initiative exchanges between engines.

The resulting language constructs, that we have called *Interactors*, express control, metaprogramming and interoperation with stateful objects and external services. They complement Horn Clause Prolog with a significant boost in expressiveness, to the point where they allow emulating at source level virtually all Prolog built-ins, including dynamic database operations.

On the other hand, our multi-threading layer has been designed to be independent of the interactor API. This allows assumptions of determinacy when working with multiple engines (and other sequential interactors) within a thread. Such assumptions would be complex to emulate in combination with unrestricted multi-threading and would require adding inefficient and error prone synchronization constructs. The multi-threading API integrates thread-construction with interactors called `Hubs` that provide synchronization between multiple consumers and producers. It supports high-level performance-centered concurrency patterns while removing the possibility of programming errors involving explicit synchronization.

The general architectural principle advocated in the paper can be stated concisely as follows: *separate concurrency for performance from concurrency for expressiveness*. Arguably, it is a good fit with the general idea behind declarative programming languages – delegate as much low level detail to underlying implementation as possible rather than burdening the programmer with complex control constructs.

The paper is organized as follows.

Section 2 overviews logic engines and describes their basic operations and the interactor API that extends the same view to various other built-in predicates. Section 3 introduces `Hubs` - flexible synchronization devices that allow interoperation and coordination between threads. Section 4 describes a set of high-level multi-threading operations that ensure concurrent execution seen as a means to accelerate computations while keeping the semantics as close as possible to a declarative interpretation. An evaluation of practical performance gains is given in section 5. Section 6 introduces inner servers - a programming abstraction providing a simple client-server style communication between threads. A similar model described in section 7 provides a "dialog" mechanism between processes for atomic remote predicate calls on a network. Section 8 shows that fairly complex "concurrency for expressiveness" patterns like Linda blackboards can be implemented cooperatively in terms of sequential operations on logic engines.

Finally, section 9 discusses related work and section 10 concludes the paper.

2. Logic engines as answer generators

Our *Interactor API* has evolved progressively into a practical Prolog implementation framework starting with [17] and continued with [19] and [22]. We summarize it here and refer to [22] for the details of a semantic description in terms of Horn Clause Logic of various engine operations.

An *logic engine* is simply a language processor reflected through an API that allows its computations to be controlled interactively from another *Engine* very much the same way a programmer controls Prolog's interactive toplevel loop: launch a new goal, ask for a new answer, interpret it, react to it.

A logic engine can be seen as an instance of the *Prolog runtime system* implementing LD-resolution [20] on a given clause database, together with a set of built-in operations. The command

```
new_engine(AnswerPattern, Goal, Interactor)
```

creates a new logic engine, uniquely identified by `Interactor`, which shares code with the currently running program and is initialized with `Goal` as its starting point. `AnswerPattern` is a term, usually a list of variables occurring in `Goal`, of which answers returned by the engine will be instances. Note however that `new_engine/3` acts like a typical constructor, no computations are performed at this point, except for initial allocation of data areas.

In our Lean-Prolog implementation, with all data areas dynamic, engines are lightweight and engine creation is fast and memory efficient[3] to the point where using them as building blocks for a significant number of built-ins and various language constructs is not prohibitive in terms of performance.

2.1 Iterating over computed answers

Note that our logic engines are seen, in an object oriented-style, as implementing the *interface* `Interactor`. This supports a *uniform* interaction mechanism with a variety of objects ranging from logic engines to file/socket streams and iterators over external data structures.

The `ask_interactor/2` operation is used to retrieve successive answers generated by an `Interactor`, on demand. It is also responsible for actually triggering computations in the engine. The query

```
ask_interactor(Interactor, AnswerInstance)
```

tries to harvest the answer computed from `Goal`, as an instance of `AnswerPattern`. If an answer is found, it is returned as `the(AnswerInstance)`, otherwise the atom `no` is returned. As in the case of the `Maybe` Monad in Haskell, returning distinct functors in the case of success and failure, allows further case analysis in a pure Horn Clause style, without needing Prolog's CUT or if-then-else operation.

Note that bindings are not propagated to the original `Goal` or `AnswerPattern` when `ask_interactor/2` retrieves an answer, i.e. `AnswerInstance` is obtained by first standardizing apart (renaming) the variables in `Goal` and `AnswerPattern`, and then backtracking over its alternative answers in a separate Prolog interpreter. Therefore, backtracking in the caller interpreter does not interfere with `Interactor`'s iteration over answers. Backtracking over `Interactor`'s creation point, as such, makes it unreachable and therefore subject to garbage collection.

An interactor is stopped with the

```
stop_interactor(Interactor)
```

operation, that, in the case of logic engines, allows reclaiming resources held by the engine. In our latest implementation *Lean Prolog*, we are using a fully automated memory management mechanism where unreachable engines are automatically garbage collected. While this API clearly refers to operations going beyond Horn Clause logic, it can be shown that a fairly high-level pure Prolog semantics can be given to them in a style somewhat similar to what one would do when writing a Prolog interpreter in Haskell, as shown in section 4 of [22].

So far, these operations provide a minimal API, powerful enough to switch tasks cooperatively between an engine and its client and emulate key Prolog built-ins like `if-then-else` and `findall` [17], as well as typical higher order operations like *fold* and *best_of* [22].

[3] The additional operation `load_engine(Interactor, AnswerPattern, Goal)` that clears data areas and initializes an engine with `AnswerPattern, Goal` has also been available as a further optimization, by providing a mechanism to reuse an existing engine.

2.2 A yield/return operation

The following operations provide a "mixed-initiative" interaction mechanism, allowing more general data exchanges between an engine and its client.

First, like the `yield return` construct of `C#` and the `yield operation` of Ruby and Python, our `return/1` operation

```
return(Term)
```

will save the state of the engine and transfer *control* and a *result* `Term` to its client. The client will receive a copy of `Term` simply by using its `ask_interactor/2` operation.

Note that an interactor returns control to its client either by calling `return/1` or when a computed answer becomes available. By using a sequence of `return/ask_interactor` operations, an engine can provide a stream of *intermediate/final results* to its client, without having to backtrack. This mechanism is powerful enough to implement a complete exception handling mechanism simply by defining

```
throw(E) :- return(exception(E)).
```

When combined with a `catch(Goal, Exception, OnException)`, on the client side, the client can decide, upon reading the exception with `ask_interactor/2`, if it wants to handle it or to throw it to the next level.

2.3 Coroutining logic engines

Coroutining has been in use in Prolog systems mostly to implement constraint programming extensions. The typical mechanism involves *attributed variables* holding suspended goals that may be triggered by changes in the instantiation state of the variables. We discuss here a different form of coroutining, induced by the ability to switch back and forth between engines.

The operations described so far allow an engine to return answers from any point in its computation sequence. The next step is to enable an engine's *client*[4] to *inject* new goals (executable data) to an arbitrary inner context of an engine. Two new primitives are needed:

```
to_engine(Engine, Data)
```

that is called by the client [5] to send data to an Engine, and

```
from_engine(Data)
```

that is called by the engine to receive a client's Data.

A typical use case for the *Interactor API* looks as follows:

1. the client creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*, parameterized as follows:
   (a) the *client* passes some data and a new goal to the *engine* and issues an `ask_interactor/2` operation that passes control to it
   (b) the *engine* starts a computation from its initial goal or the point where it has been suspended and runs (a copy of) the new goal received from its *client*
   (c) the *engine* returns (a copy of) the answer, then suspends and returns control to its *client*
3. the *client* interprets the answer and proceeds with its next computation step
4. the process is fully reentrant and the *client* may repeat it from an arbitrary point in its computation

[4] Another engine, that uses an engine's services.

[5] Equivalently the `tell_interactor/2` generic interactor predicate can be also used here.

For instance, using `to_engine` and `from_engine`, one can implement a close equivalent of Ruby's `yield` statement as follows:

```
ask_engine(Engine, (Answer:-Goal), Result):-
  to_engine(Engine, (Answer:-Goal)),
  ask_interactor(Engine, Result).

engine_yield(Answer):-
  from_engine((Answer:-Goal)),
  call(Goal),
  return(Answer).
```

The predicate `ask_engine/3` sends a query (possibly built at runtime) to an engine, which in turn, executes it and returns a result with an `engine_yield` operation. The query is typically a goal or a pattern of the form `AnswerPattern:-Goal` in which case the engine interprets it as a request to instantiate `AnswerPattern` by executing `Goal` before returning the answer instance.

As the following example shows, this allows the client to use, from outside, the (infinite) recursive loop of an engine as a form of *updatable persistent state*.

```
sum_loop(S1):-engine_yield(S1 => S2), sum_loop(S2).

inc_test(R1, R2):-
   new_engine(_, sum_loop(0), E),
   ask_engine(E, (S1 => S2 :- S2 is S1+2), R1),
   ask_engine(E, (S1 => S2 :- S2 is S1+5), R2).

?- inc_test(R1, R2).
R1=the(0 => 2),
R2=the(2 => 7)
```

Note also that after parameters (the increments 2 and 5) are passed to the engine, results dependent on its state (the sums so far 2 and 7) are received back. Moreover, note that an arbitrary goal is injected in the local context of the engine where it is executed. The goal can then access the engine's *state variables* `S1` and `S2`. As engines have separate garbage collectors (or in simple cases as a result of tail recursion), their infinite loops run in constant space, provided that no unbounded size objects are created.

3. Hubs and threads

As a key difference with typical multi-threaded Prolog implementations like Ciao-Prolog [1] and SWI-Prolog [25], our Interactor API is designed up front with a clear separation between *engines* and *threads* as we prefer to see them as orthogonal language constructs.

To ensure that communication between logic engines running concurrently is is safe and synchronized, we hide the engine handle and provide a producer/consumer data exchanger object, called a `Hub`, when multi-threading.

A `Hub` can be seen as an interactor used to synchronize threads. On the Prolog side it is introduced with a constructor `hub/1` and works with the standard interactor API:

```
hub(Hub)
ask_interactor(Hub, Term)
tell_interactor(Hub, Term)
stop_interactor(Hub)
```

On the Java side, each instance of the `Hub` class provides a synchronizer between M producers and N consumers. A `Hub` supports data exchanges through a private object `port` and it implements the `Interactor` interface:

```
private Object port;
```

```
synchronized public Object ask_interactor() {
    while(null==port) {
      try {
        wait();
      } catch(InterruptedException e) {
        if(stopped)
          break;
      }
    }
    Object result=port;
    port=null;
    notifyAll();
    return result;
}

synchronized public Object tell_interactor(Object T) {
    while(null!=port) {
      try {
        wait();
      } catch(InterruptedException e) {
        if(stopped)
          break;
      }
    }
    port=T;
    notifyAll();
    return stopped?null:T;
}

synchronized public void stop_interactor() {
    this.stopped=true;
}
```

Consumers issue `ask_interactor/2` operations that correspond to `tell_interactor/2` operations issued by producers.

A group of related threads are created around a `Hub` that provides both basic synchronization and data exchange services. The built-in

```
new_logic_thread(Hub, X, G, Clone, Source)
```

creates a new thread by either "cloning" the current Prolog code and symbol spaces or by loading new Prolog code in a separate name space from a `Source` (typically a precompiled file or a stream). Usually a default constructor

```
new_logic_thread(Hub, X, G)
```

is used. It shares the code but it duplicates the symbol table to allow independent symbol creation and symbol garbage collection to occur safely in multiple threads without the need to synchronize or suspend thread execution.

4. High-level concurrency with higher-order constructs

Encapsulating concurrent execution patterns in high-level abstractions, when performance gains are the main reason for using multiple threads, avoids forcing a programmer to suddenly deal with complex procedural issues when in the middle of working with declarative constructs in a (mostly) declarative language like Prolog. It is also our experience that in an untyped language this reduces software risks significantly.

The predicate `multi_all(XGs, Xs)` runs list of goals `XGs` of the form `Xs:-G` (on a new thread each) and collects all answers to a list `Xs`.

```
multi_all(XGs, Xs):-
  hub(Hub),
  length(XGs, ThreadCount),
  launch_logic_threads(XGs, Hub),
  collect_thread_results(ThreadCount, Hub, Xs),
  stop_interactor(Hub).
```

When launching the threads we ensure that they share the same `Hub` for communication and synchronization.

```
launch_logic_threads([], _Hub).
launch_logic_threads([(X:-G)|Gs], Hub):-
  new_logic_thread(Hub, X, G),
  launch_logic_threads(Gs, Hub).
```

Collecting the bag of results computed by all the threads involves consuming them as soon as they arrive to the `Hub`.

```
collect_thread_results(0, _Hub, []).
collect_thread_results(ThreadCount, Hub, MoreXs):-
  ThreadCount>0,
  ask_interactor(Hub, Answer),
  count_thread_answer(Answer, ThreadCount, ThreadsLeft,
    Xs, MoreXs),
  collect_thread_results(ThreadsLeft, Hub, Xs).
```

Note also that termination is detected by counting the "`no`" answers indicating that a given thread has nothing new to produce.

```
count_thread_answer(no, ThreadCount, ThreadsLeft, Xs, Xs):-
  ThreadsLeft is ThreadCount-1.
count_thread_answer(the(X), ThreadCount, ThreadCount,
  Xs, [X|Xs]).
```

One can try out the code as follows:

```
?-multi_all([(I:-for(I, 1, 10)),
             (J:-member(J, [a, b, c]))], Rs).
Rs = [1, 2, 3, a, 4, b, 5, 6, 7, 8, 9, 10, c].
```

A typical use (reminding of the *map-reduce* design pattern) is to call `multi_all` with a list of goals of length larger or equal to the number of available CPUs that can work independently in producing answers to the same problem. The next stage consists of filtering these answers (a reduce step) followed by another call to `multi_all`.

However, when one wants to be able to tell apart the answers produced by different goals a variation of the same pattern, that we have called `multi_findall` can be used.

The predicate `multi_findall(XGs, Xss)` works as follows: for each `(X:-G)` on the list `XGs` it starts a new thread and then aggregates solutions as if `findall(X, G, Xs)` were called. It collects all the answers `Xs` to a list of lists `Xss` in the order defined by the list of goals `XGs`. It adopts a well-known technique from TCP/IP networking - answer patterns are marked with the number of the goal producing them, such that the results of out-of-order multi-threaded execution can be sorted and grouped using these consecutive integer markers.

```
multi_findall(XGs, Xss):-
  mark_answer_patterns(XGs, MarkedXGs, 0),
  multi_all(MarkedXGs, MXs),
  collect_marked_answers(MXs, Xss).

mark_answer_patterns([], [], _).
mark_answer_patterns([(X:-G)|XGs],
                     [(N-X:-G)|MarkedXGs], N):-
  N1 is N+1,
  mark_answer_patterns(XGs, MarkedXGs, N1).

collect_marked_answers(MXs, Xss):-
  findall(Xs, keygroup(MXs, _, Xs), Xss).


?-multi_findall([(I:-for(I, 1, 10)),
                 (J:-member(J, [a, b, c]))], Rss).
Rss = [[1, 2, 3, 4, 5, 6, 7, 8, 9, 10], [a, b, c]]
```

Note the use of the keygroup/3 LeanProlog built-in that backtracks over key-list of answers pairs. The list of results collected at the end is the same as if a sequence of `findall/3` operations were used – a pattern familiar to the Prolog programmer - except that the results are produced by parallel threads. Note also that the only thing the programmer needs to know is that *grouping together at least as many goals as the number of CPUs available will speed up execution accordingly*.

One of deficiencies of these multi-threaded `findall`-like operations is that they might build large lists of answers unnecessarily. With inspiration from combinators in functional languages one can implement a more flexible multi-threaded `fold` operation.

The predicate `multi_fold(F, XGs, Xs)` runs a list of goals `XGs` of the form `Xs:-G` and combines with `F` their answers to accumulate them into a single final result without building intermediate lists.

```
multi_fold(F, XGs, Final):-
  hub(Hub),
  length(XGs,ThreadCount),
  launch_logic_threads(XGs, Hub),
  ask_interactor(Hub, Answer),
  (Answer = the(Init) ->
    fold_thread_results(ThreadCount, Hub, F, Init, Final)
  ; true
  ),
  stop_interactor(Hub),
  Answer=the(_).
```

Like the predicate `multi_all`, `multi_fold` relies on the predicate `launch_logic_threads` to run threads initiated by the goal list `XGs`. I relies on `fold_thread_results` to collect results computed by various threads from `Hub` and to combine them into a single result, while keeping track of the number of threads that have finished their work.

```
fold_thread_results(0, _Hub, _F, Best, Best).
fold_thread_results(ThreadCount, Hub, F, SoFar, Best):-
  ThreadCount > 0,
  ask_interactor(Hub, Answer),
  count_thread_answer(Answer, ThreadCount, ThreadsLeft,
                      F, SoFar, Better),
  fold_thread_results(ThreadsLeft, Hub, F, Better, Best).

count_thread_answer(no, ThreadCount, ThreadsLeft,
                                   _F, SoFar, SoFar):-
  ThreadsLeft is ThreadCount-1.
count_thread_answer(the(X), ThreadCount,
                    ThreadCount, F, SoFar, Better):-
  call(F, X, SoFar, Better).
```

A typical application is the predicate `multi_best(F, XGs, M)`, which runs a list of goals `XGs` of the form `N:-G` where `N` is instantiated to a numeric value. By using `max/3` to combine the current best answers with a candidate one it extracts at the the maximum `M` of all answers computed (in an arbitrary order) by all threads.

```
multi_best(XGs,R):-multi_fold(max,XGs,R).
```

Note that, as in the case of its `fold` cousins in functional languages, `fold` can be used to emulate various other higher order predicates. For instance the `findall`-like predicate `multi_all` is emulated by the predicate `multi_all(XGs,Xs)` which runs a list of goals `XGs` of the form `Xs:-G` and combines all answers to a list using `list_cons`.

```
multi_all(XGs, Rs):-
   multi_fold(list_cons,[([]:-true)|XGs],Rs).

list_cons(X, Xs, [X|Xs]).
```

A different pattern arises from combinatorial search algorithms where one wants to stop multiple threads as soon as a first solution is found. Things like restarts in SAT solvers and various Monte Carlo algorithms fit in this category.

The predicate `multi_first(K, XGs, Xs)` runs list of goals `XGs` of the form `Xs:-G` until the first `K` answers `Xs` are found (or fewer if less then `K`) answers exist. It uses a very simple mechanism built into Lean Prolog's multi-threading API: when a `Hub` interactor is stopped, all threads associated to it are notified to terminate.

```
multi_first(K, XGs, Xs):-
  hub(Hub),
  length(XGs, ThreadCount),
  launch_logic_threads(XGs, Hub),
  collect_first_results(K, ThreadCount, Hub, Xs),
  stop_interactor(Hub).

collect_first_results(_, 0, _Hub, []).
collect_first_results(0, _, Hub, []):-
  stop_interactor(Hub).
collect_first_results(K, ThreadCount, Hub, MoreXs):-
  K>0, ThreadCount>0,
  ask_interactor(Hub, Answer),
  count_thread_answer(Answer, ThreadCount,
                      ThreadsLeft, Xs, MoreXs),
  ( ThreadCount=:=ThreadsLeft->K1 is K-1
  ; K1 is K
  ),
  collect_first_results(K1, ThreadsLeft, Hub, Xs).
```

In particular, searching for at most one solution is possible:

```
multi_first(XGs,X):-multi_first(1,XGs,[X]).
```

The `multi_first/3 and multi_first/2` predicates provide an alternative to using `CUT` in Prolog as a means to limit search while supporting a scalable mechanism for concurrent execution. Note also that it is more flexible than `CUT` as it can be used to limit the search to a window of `K` solutions. However, in contrast with `CUT`, the order in which these first solutions are found is arbitrary.

5. Measuring the practical performance gains

The main focus of the high level multi-threading predicates described so far is to provide performance gains without major changes in existing Prolog code.

To evaluate the maximum amount of parallelism provided by `multi_all` with a number of independent goals related to the actual number of hardware threads and virtual-threads provided by Intel's hyper-threading, we have run the naive reverse benchmark on a MacPro with two 2.26GHz Quad-Core Intel Xeon processor giving a total of 8 cores with 2 virtual threads each (provided by hyper-threading) and 16GB of memory (Figure 1).

The benchmark indicates that performance increases almost linearly with the number of cores (up to 8) as they provide genuine hardware threads, but it will not benefit from hyper-threading which shares the same arithmetic-logic units among virtual threads.

Rather than working on other artificial examples where performance can use the maximum parallelism available, we have reorganized the the Lean Prolog parser and compiler to speed-up the self-compilation test. To better evaluate the practical performance improvements we have tried it on 3 computers (Figure 2) featuring a blend or real and virtual threads (hyper-threading). Note that hyper-threading provides 2 virtual CPUs sharing one core and typically a less than 1.5 performance increase from multithreading.

The maximum performance increase of only about 2.3 times despite of the large number of available threads on the 8 core MacPro is explained by the fact that there are big variations in the files sizes of LeanProlog's modules and by the fact that the time taken by the largest file dominates the benchmark. To a smaller extent, the final concatenation of the resulting binaries has added a constant sequential cost reducing the overall speed-up ratio as well. The relatively good performance on the 4 GB 2.13 GHz

Threads	Execution time (ms)	Thousands of LIPS
1	1599	9664
2	821	18810
3	548	28181
4	424	36445
5	355	43405
6	297	52001
7	256	60262
8	227	67925
9	235	65633
10	231	66636
11	227	67884
12	232	66559
13	222	69583
14	224	68898
15	218	70690
16	218	70581

Figure 1. Lean Prolog on naive reverse on an 8-core MacPro

System/Program	Sequential	Multithreaded
8 core MacPro slowest	11.46	4.89
8 core MacPro fastest	10.16	4.49
2 core MacAir slowest	14.29	8.53
2 core MacAir fastest	12.56	7.55
1 core NetBook slowest	84.39	62.21
1 core NetBook fastest	79.04	56.27

Figure 2. Lean Prolog bootstrapping time (in seconds)

MacAir with a CoreDuo CPU which does not have hyper-threading and provides a maximum of 2 hardware threads, is partly due to its fast SSD drive and partly to its comparably fast (in terms of GHz) CPU. The results on the Sony Linux NetBook with a single hyper-threading enabled 1.66 GHz Intel N450 CPU match the performance increase one can expect from the 2 virtual threads provided by hyper-threading.

6. Inner servers

A simple "inner server" API, similar to a socket based client/server connection can be used to delegate tasks to a set of threads for concurrent processing.

The predicate `new_inner_server(IServer)` creates an inner server consisting of a thread and 2 hubs.

```
new_inner_server(IServer):-
  IServer = hubs(In, Out),
  hub(In), hub(Out),
  new_logic_thread(In, _, inner_server_loop(In, Out)).
```

The predicate `inner_server_loop(In, Out)` loops consuming data from hub `In` and returning answers to hub `Out`.

```
inner_server_loop(In, Out):-
  ask_interactor(In, (X:-G)),
  ( G==stop, !, tell_interactor(Out, done), fail
  ; G->R=the(X)
  ; R=no
  ),
  tell_interactor(Out, R),
  inner_server_loop(In, Out).
```

The predicate `tell_inner_server(IServer, X, G)` gives a task that the server starts executing. It works in cooperation with the predicate `ask_inner_server(IServer, Answer)` that collects the answer from an inner server (and possibly waits until is done). Typically, after giving a task to the inner server, one can run some other code in parallel and when finished collect the answers using `ask_inner_server`.

```
tell_inner_server(hubs(In, _Out), X, G):-
  tell_interactor(In, (X:-G)).

ask_inner_server(hubs(_In, Out), Answer):-
  ask_interactor(Out, Answer).
```

Finally the predicate `stop_inner_server(IServer)` stops an inner server by notifying its thread to terminate and then shutting down the two hubs.

```
stop_inner_server(IServer):-
  IServer=hubs(In, Out),
  query_inner_server(IServer, _, stop, done),
  stop_interactor(In),
  stop_interactor(Out).
```

7. Sequentializing remote predicate calls

It is typical in various programming languages to associate client-server connections and multi-threading: when a new client connects, the server launches a new thread to service it. On the other hand, one can use client-server connections to ensure sequential – one transaction at a time – cooperation between clients connected to a server. The predicate `seq_server(Port)` uses the built-in `new_seq_server(Port,Server)` that creates a server listening on a port and then handles client requests sequentially.

```
seq_server(Port):-
  new_seq_server(Port, Server),
  repeat,
    new_seq_service(Server, ServiceSocket),
    seq_server_step(ServiceSocket),
  fail.

seq_server_step(Service):-
   recv_canonical(Service, (X:-Goal)),
   (  Goal ==stop->!, R=stopped
    ; call(Goal)->R=the(X)
    ; R=no
   ),
   send_canonical(Service, R),
   R\==stopped,
   seq_server_step(Service).
```

On the client side, an atomic operation called a "dialog" is initiated with the built-in

```
open_socket_dialog(Host, Port, Client)
```

that waits for the server to be available. After exchanging a sequence of remote predicate calls with the server over the open socket using

```
 ask_over_socket(S, Query, Answer):-
  send_canonical(S, Query),
  recv_canonical(S, Answer).
```

the client terminates the "dialog" with

```
close_socket_dialog(Socket).
```

Note the use of `send_canonical` and `recv_canonical` built-in predicates that send and receive Prolog terms (translated to a canonical form as byte sequences) on both the client and server side. While it is possible to have several such servers running in parallel and executing tasks concurrently on a given machine, one should be aware that performance benefits will only be as good as the total number of available hardware threads. Note also that having atomicity of remote predicate call transactions with the "dialog" mechanism ensures simple and safe coordination between multiple clients connecting to the same server on networked computers.

8. Integrating cooperative multi-tasking constructs

The message passing style interaction shown in the previous sections between engines and their clients, can be easily generalized to associative communication through a unification based blackboard interface [3]. Exploring this concept in depth promises more flexible interaction patterns, as out of order operations would become possible, matched by association patterns. An interesting question arises at this point. Can blackboard-based coordination be expressed directly in terms of engines, and as such, can it be seen as independent of a multi-threading API?

We have shown so far that when focusing on performance on multi-core architectures, multi-threading can be encapsulated in high-level constructs that provide its benefits without the need of a complex procedural API.

To support our claim that "concurrency for expressiveness" works quite naturally with coroutining logic engines we will describe here a cooperative implementation of Linda blackboards. In contrast to multi-threaded or multi-process implementations, it ensures atomicity "by design" for various operations. It is an example of *concurrency for expressiveness* that can be used orthogonally from *concurrency for performance* to coordinate cooperatively multiple logic engines within a single thread.

The predicate `new_coordinator(Db)` uses a database parameter `Db` (a synthetic name, if given as a variable, provided by `db_ensure_bound`) to store the state of the Linda blackboard. The state of the blackboard is described by the dynamic predicates `available/1` that keeps track of terms posted by `out` operations, `waiting/2` that collects pending `in` operations waiting for matching terms, and `running/1` that helps passing control from one engine to the next.

```
new_coordinator(Db):-
  db_ensure_bound(Db),
  db_dynamic(Db, available/1),
  db_dynamic(Db, waiting/2),
  db_dynamic(Db, running/1).
```

The predicate `new_task` initializes a new coroutining engine, driven by goal `G`. We shall call such an engine "an agent" in the next paragraphs.

```
new_task(Db, G):-
  new_engine(nothing, (G, fail), E),
  db_assertz(Db, running(E)).
```

Three cooperative Linda operations are available to an agent. They are all expressed by returning a specific pattern to the `Coordinator`.

```
coop_in(T):-return(in(T)), from_engine(X), T=X.

coop_out(T):-return(out(T)).

coop_all(T, Ts):-return(all(T, Ts)), from_engine(Ts).
```

The `Coordinator` implements a handler for the patterns returned by the agents as follows:

```
handle_in(Db, T, E):-
  db_retract1(Db, available(T)),
  !,
  to_engine(E, T),
  db_assertz(Db, running(E)).
handle_in(Db, T, E):-
  db_assertz(Db, waiting(T, E)).

handle_out(Db, T):-
  db_retract(Db, waiting(T, InE)),
  !,
  to_engine(InE, T),
  db_assertz(Db, running(InE)).
handle_out(Db,T):-
  db_assertz(Db, available(T)).

handle_all(Db, T, Ts, E):-
  findall(T, db_clause(Db, available(T), true), Ts),
  to_engine(E, Ts),
  db_assertz(Db, running(E)).
```

The `Coordinator`'s dispatch loop `coordinate/1` (failure driven here to run without requiring garbage collection) works as follows:

```
coordinate(Db):-
  repeat,
    ( db_retract1(Db, running(E))->
        ask_interactor(E, the(A)),
        dispatch(A, Db, E),
        fail
     ; !
    ).
```

Its `dispatch/3` predicate calls the handlers as appropriate.

```
dispatch(in(X), Db, E):-handle_in(Db, X, E).
dispatch(out(X), Db, E):-handle_out(Db, X),
  db_assertz(Db, running(E)).
dispatch(all(T, Ts), Db, E):-handle_all(Db, T, Ts, E).
dispatch(exception(Ex), _, _):-throw(Ex).
```

Note also that the `dispatch/3` predicate propagates exceptions - in accordance with a "fail fast" design principle.

```
stop_coordinator(C):-
  foreach(db_clause(C, running(E), true), stop(E)),
  foreach(db_clause(C, waiting(_, E), true), stop(E)).
```

When the coordinator is stopped using `stop_coordinator` the database is cleaned of possible records of unfinished tasks in either `running` or `waiting` state. The code uses a few Lean Prolog extensions - like multiple dynamic databases (with operations like `db_assert/2` similar to `assert/1` except for the extra first argument that names the database on which they act). Another extension, `foreach` makes failure-driven loops more readable - it is defined as follows:

```
foreach(When,Then):- When,once(Then),fail.
foreach(_,_).
```

The following test predicate shows that out-of-order `in` and `out` operations are exchanged as expected between engines providing a simple transactional implementation of Linda coordination.

```
test_coordinator:-
  new_coordinator(C),
  new_task(C,
    foreach(
      member(I, [0, 2]),
      ( coop_in(a(I, X)),
        println(coop_in=X)
      )
    )
  ),
  new_task(C,
    foreach(
      member(I, [3, 2, 0, 1]),
      (
        println(coop_out=f(I)),
        coop_out(a(I, f(I)))

      )
    )
  ),
  new_task(C,
    foreach(
      member(I, [1, 3]),
      ( coop_in(a(I, X)),
        println(coop_in=X)
      )
    )
  ),
  coordinate(C),
  stop_coordinator(C).
```

When running the code, one can observe that explicit coroutining control exchanges between engines have been replaced by operations of the higher level Linda coordination protocol.

```
?- test_coordinator.
coop_out = f(3)
coop_out = f(2)
coop_out = f(0)
coop_in = f(0)
coop_out = f(1)
coop_in = f(2)
coop_in = f(1)
coop_in = f(3)
true.
```

This shows that "concurrency for expressiveness" in terms of the logic-engines-as-interactors API provides flexible building blocks for the encapsulation of non-trivial high-level concurrency patterns.

A final detail, important for optimizing engine-based cooperative multi-tasking is *engine reuse*. Thread pools have been in use either at kernel level or user level in various operating system and language implementations [10] to avoid costly allocation and deallocation of resources required by threads. Likewise, for first-class logic engine implementations that cannot avoid high creation/initialization costs, it makes sense to build *interactor pools*. For this, an additional operation is provided by our API, allowing the *reuse* of an existing logic engine (`load_engine/3`). An interactor pool can be maintained by a dedicated logic engine that keeps track of the state of various interactors and provides recently freed handles, when available, to `new_engine` requests.

9. Related work

Multiple logic engines have been present in one form or another in various parallel implementation of logic programming languages [5, 11, 12, 24]. Among the earliest examples of parallel execution mechanisms for Prolog, AND-parallel [6] and OR-parallel [9] execution models are worth mentioning.

However, with the exception of the author's papers on this topic [15–17, 19, 21, 22] there are relatively few examples of using first-class logic engines as a mechanism to enhance language expressiveness, independently of their use for parallel programming. A notable exception is [2] where such an API is discussed for parallel symbolic languages in general.

In combination with multithreading, our own engine-based API bears similarities with various other Prolog systems, notably [1, 25]. Coroutining has also been present in logic languages to support constraint programming extensions requiring suspending and resuming execution based on changes of the binding state of variables. In contrast to these mechanisms that focus on transparent, fine-grained coroutining, our engine-based mechanisms are coarse-grained and programmer controlled. Our coroutining constructs can be seen, in this context, as focussed on expressing cooperative design patterns that typically involve the use of a procedural multi-threading API.

Finally, our `multi_findall` and `multi_fold` predicates have similarities with design patterns like *ForkJoin* [8] or *MapReduce* [7] coming from sharing a common inspiration source: higher-order constructs like and `map fold` in functional programming.

10. Conclusion

We have shown that by decoupling logic engines and threads, programming language constructs can be kept simple when their purpose is clear – *multi-threading for performance* is separated from *concurrency for expressiveness*. This is achieved via communication between independent language interpreters independent of the multi-threading API.

Our language constructs are particularly well-suited to take advantage of today's multi-core architectures where keeping busy a relatively small number of parallel execution units is all it takes to get predictable performance gains, while reducing the software risks coming from more complex concurrent execution mechanisms designed with massively parallel execution in mind.

The current version of LeanProlog containing the implementation of the constructs discussed in this paper, as well as a number of draft papers describing other aspects of the system are available at `http://logic.cse.unt.edu/research/LeanProlog`.

Acknowledgment

We thank NSF (research grant 1018172) for support.

References

[1] M. Carro and M. V. Hermenegildo. Concurrency in Prolog Using Threads and a Shared Database. In *ICLP*, pages 320–334, 1999.

[2] A. Casas, M. Carro, and M. Hermenegildo. Towards a high-level implementation of flexible parallelism primitives for symbolic languages. In *PASCO '07: Proceedings of the 2007 international workshop on Parallel symbolic computation*, pages 93–94, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-741-4. doi: http://doi.acm.org/10.1145/1278177.1278193.

[3] K. De Bosschere and P. Tarau. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

[4] FIPA. FIPA 97 specification part 2: Agent communication language, Oct. 1997. URL `http://www.fipa.org/spec/f8a22.zip`. Version 2.0.

[5] G. Gupta, E. Pontelli, K. A. Ali, M. Carlsson, and M. V. Hermenegildo. Parallel execution of prolog programs: a survey. *ACM Trans. Program. Lang. Syst.*, 23(4):472–602, 2001. ISSN 0164-0925. doi: http://doi.acm.org/10.1145/504083.504085.

[6] M. V. Hermenegildo. An abstract machine for restricted and-parallel execution of logic programs. In *Proceedings on Third international conference on logic programming*, pages 25–39, New York, NY, USA, 1986. Springer-Verlag New York, Inc. ISBN 0-387-16492-8.

[7] R. Lämmel. Google's MapReduce programming model revisited. *Sci. Comput. Program.*, 68:208–237, October 2007. ISSN 0167-6423.

[8] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM. ISBN 1-58113-288-3.

[9] E. Lusk, S. Mudambi, E. Gmbh, and R. Overbeek. Applications of the aurora parallel prolog system to computational molecular biology. In *In Proc. of the JICSLP'92 Post-Conference Joint Workshop on Distributed and Parallel Implementations of Logic Programming Systems, Washington DC*. MIT Press, 1993.

[10] I. Pyarali, M. Spivak, R. Cytron, and D. C. Schmidt. Evaluating and optimizing thread pool strategies for real-time CORBA. In *LCTES/OM*, pages 214–222, 2001. URL `citeseer.ist.psu.edu/451897.html`.

[11] E. Shapiro. The family of concurrent logic programming languages. *ACM Comput. Surv.*, 21(3):413–510, 1989. ISSN 0360-0300. doi: http://doi.acm.org/10.1145/72551.72555.

[12] K. Shen and M. V. Hermenegildo. A simulation study of Or- and independent And-parallelism. In V. Saraswat and K. Ueda, editors, *Logic Programming Proceedings of the 1991 International Symposium*, pages 135–151, Cambridge, Massachusetts London, England, 1991. MIT Press.

[13] P. Tarau. Towards Inference and Computation Mobility: The Jinni Experiment. In J. Dix and U. Furbach, editors, *Proceedings of JELIA'98, LNAI 1489*, pages 385–390, Dagstuhl, Germany, Oct. 1998. Springer. invited talk.

[14] P. Tarau. Intelligent Mobile Agent Programming at the Intersection of Java and Prolog. In *Proceedings of The Fourth International Conference on The Practical Application of Intelligent Agents and Multi-Agents*, pages 109–123, London, U.K., 1999.

[15] P. Tarau. Multi-Engine Horn Clause Prolog. In G. Gupta and E. Pontelli, editors, *Proceedings of Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages*, Las Cruces, NM, Nov. 1999. URL `http://www.cs.nmsu.edu/lldap/iclp99/`.

[16] P. Tarau. Inference and Computation Mobility with Jinni. In K. Apt, V. Marek, and M. Truszczynski, editors, *The Logic Programming Paradigm: a 25 Year Perspective*, pages 33–48. Springer, 1999. ISBN 3-540-65463-1.

[17] P. Tarau. Fluents: A Refactoring of Prolog for Uniform Reflection and Interoperation with External Objects. In J. Lloyd, editor, *Computational Logic–CL 2000: First International Conference*, London, UK, July 2000. LNCS 1861, Springer-Verlag.

[18] P. Tarau. Agent Oriented Logic Programming Constructs in Jinni 2004. In B. Demoen and V. Lifschitz, editors, *Logic Programming, 20-th International Conference, ICLP 2004*, pages 477–478, Saint-Malo, France, Sept. 2004. Springer, LNCS 3132.

[19] P. Tarau. Logic Engines as Interactors. In M. Garcia de la Banda and E. Pontelli, editors, *Logic Programming, 24-th International Conference, ICLP*, pages 703–707, Udine, Italy, Dec. 2008. Springer, LNCS.

[20] P. Tarau and M. Boyer. Nonstandard Answers of Elementary Logic Programs. In J. Jacquet, editor, *Constructing Logic Programs*, pages 279–300. J.Wiley, 1993.

[21] P. Tarau and V. Dahl. High-Level Networking with Mobile Code and First Order AND-Continuations. *Theory and Practice of Logic Programming*, 1(3):359–380, May 2001. Cambridge University Press.

[22] P. Tarau and A. Majumdar. Interoperating Logic Engines. In *Practical Aspects of Declarative Languages, 11th International Symposium, PADL 2009*, pages 137–151, Savannah, Georgia, Jan. 2009. Springer, LNCS 5418.

[23] S. Tyagi and P. Tarau. A Most Specific Method Finding Algorithm for Reflection Based Dynamic Prolog-to-Java Interfaces. In I. Ramakrishan and G. Gupta, editors, *Proceedings of PADL'2001*, Las Vegas, Mar. 2001. Springer-Verlag.

[24] K. Ueda. Guarded horn clauses. In E. Wada, editor, *Logic Programming '85, Proceedings of the 4th Conference, Tokyo, Japan, July 1-3, 1985*, volume 221 of *Lecture Notes in Computer Science*, pages 168–179. Springer, 1985.

[25] J. Wielemaker. Native Preemptive Threads in SWI-Prolog. In C. Palamidessi, editor, *ICLP*, volume 2916 of *Lecture Notes in Computer Science*, pages 331–345. Springer, 2003. ISBN 3-540-20642-6.