Concurrent Programming Constructs in Multi-Engine Prolog
Parallelism just for the cores (and not more!)

Paul Tarau

Department of Computer Science and Engineering, Univ of North Texas

DAMP’11, Sunday, January 23, 2011, 3:00pm
Motivation

- Logic Programming languages make essential use of unification and backtracking $\Rightarrow$ concurrent programming models are by far more complex than in Functional Programming
- $\Rightarrow$ encapsulate backtracking and unification in independent computational units - *Logic Engines*
- *Interactors*: an abstraction of answer generation and refinement in *Logic Engines*, supporting the agent-oriented view that programming is a dialog between simple, self-contained, autonomous building blocks
- resist temptation to map Logic Engines and Threads directly $\Rightarrow$ encapsulate concurrency in higher order primitives, similar to existing sequential constructs
- $\Rightarrow$ ability to separate concurrency for performance and concurrency for expressiveness
First Class Logic Engines

- an *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 thing as a programmer controlling Prolog’s interactive toplevel loop:
  - launch a new goal
  - ask for a new answer
  - interpret it
  - react to it

- a *Logic Engine* is an *Engine* running a Horn Clause Interpreter with LD-resolution on a given clause database, together with a set of built-in operations
new_engine(AnswerPattern, Goal, Interactor):

- creates a new Horn Clause solver, uniquely identified by Interactor
- shares code with the currently running program
- initialized with Goal as a starting point
- AnswerPattern is a term returned by the engine will be instances
The Engine API: get/2, stop/1

get(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 returns the atom no
- is used to retrieve successive answers generated by an Interactor, on demand
- it is responsible for actually triggering computations in the engine

stop(Interactor):

- stops the Interactor
- no is returned for new queries
A yield/return 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 get/2 operation
- an Interactor returns control to its client either by calling return/1 or when a computed answer becomes available

Application: exceptions

throw(E) :- return(exception(E)).
Exchanging Data with an Interactor

\[\text{to\_engine(Engine,Term)}:\]
\[
\bullet \text{ used to send a client’s data to an Engine}\]

\[\text{from\_engine(Term)}:\]
\[
\bullet \text{ used by the engine to receive a client’s Data}\]
Typical use of the Interactor API

1. the *client* creates and initializes a new *engine*
2. the client triggers a new computation in the *engine*:
   - the *client* passes some data and a new goal to the *engine* and issues a `get` operation that passes control to it
   - 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*
   - 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
Emulating Yield

ask_engine(Engine, Query, Result) :-
    to_engine(Engine, Query),
    get(Engine, Result).

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

- ask_engine/3 sends a query
- the engine executes it and returns a result with an engine_yield operation
- the query is typically AnswerPattern:-Goal, the engine interprets it as a request to instantiate AnswerPattern by executing Goal and returning the answer instance
Encapsulating state in an engine and exchanging data

\[
\text{sum\_loop}(S1) :- \text{engine\_yield}(S1 \Rightarrow S2), \text{sum\_loop}(S2) .
\]

\[
\text{inc\_test}(R1, R2) :-
\quad \text{new\_engine}(_, \text{sum\_loop}(0), E),
\quad \text{ask\_engine}(E, (S1 \Rightarrow S2 : -S2 \text{ is } S1 + 2), R1),
\quad \text{ask\_engine}(E, (S1 \Rightarrow S2 : -S2 \text{ is } S1 + 5), R2).
\]

?- inc\_test(R1, R2).
R1 = the(0 \Rightarrow 2),
R2 = the(2 \Rightarrow 7)
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)
```
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;
}
Interleaving execution with multi_all/2

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).
```
The pattern: “launch and collect”

When launching the threads we ensure that they share the same Hub.

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

\[
\text{count_thread_answer}(\text{no}, \text{ThreadCount}, \text{ThreadsLeft}, \text{Xs}, \text{Xs}) :\neg \\
\text{ThreadsLeft} \text{ is } \text{ThreadCount}-1.
\]

\[
\text{count_thread_answer}(\text{the}(\text{X}), \text{ThreadCount}, \text{ThreadCount}, \text{Xs}, [\text{X}|\text{Xs}]).
\]

How it works:

\[
?-\text{multi_all}([(I:\text{between}(1, 10, I)), \\
\text{(J:\text{member}(J, [a, b, c]))}], \text{Rs}).
\]

\[
\text{Rs} = [1, 2, 3, a, 4, b, 5, 6, 7, 8, 9, 10, c].
\]
A multi-purpose higher order predicate: multi_fold

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.

```prolog
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(_).
```

Paul Tarau
Univ. of North Texas
Concurrent Programming Constructs in Multi-Engine Prolog
A familiar variation: multi\_findall

\[
\text{multi\_findall}(\text{XGs}, \text{Xss}) \text{ marks answers and sorts by goal}
\]
- for each \((X:-G)\) on the list \(\text{XGs}\) it starts a new thread
- then aggregates solutions as if \(\text{findall}(X, G, Xs)\) were called
- It collects all the answers \(Xs\) to a list of lists \(\text{Xss}\) in the order defined by the list of goals \(\text{XGs}\).

\[
\text{multi\_findall}(\text{XGs}, \text{Xss}) :-
\text{mark\_answer\_patterns}(\text{XGs}, \text{MarkedXGs}, 0),
\text{multi\_all}(\text{MarkedXGs}, \text{MXs}),
\text{collect\_marked\_answers}(\text{MXs}, \text{Xss}).
\]

?-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]]
Stopping after the first $K$ answers: multi_first

- **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

- this happens when we detect that the first $K$ answers have been computed or that there are no more answers
## Concurrent Runs of Naive Reverse

<table>
<thead>
<tr>
<th>Threads</th>
<th>Execution time (ms)</th>
<th>Thousands of LIPS</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1599</td>
<td>9664</td>
</tr>
<tr>
<td>2</td>
<td>821</td>
<td>18810</td>
</tr>
<tr>
<td>3</td>
<td>548</td>
<td>28181</td>
</tr>
<tr>
<td>4</td>
<td>424</td>
<td>36445</td>
</tr>
<tr>
<td>5</td>
<td>355</td>
<td>43405</td>
</tr>
<tr>
<td>6</td>
<td>297</td>
<td>52001</td>
</tr>
<tr>
<td>7</td>
<td>256</td>
<td>60262</td>
</tr>
<tr>
<td>8</td>
<td>227</td>
<td>67925</td>
</tr>
<tr>
<td>9</td>
<td>235</td>
<td>65633</td>
</tr>
<tr>
<td>10</td>
<td>231</td>
<td>66636</td>
</tr>
<tr>
<td>11</td>
<td>227</td>
<td>67884</td>
</tr>
<tr>
<td>12</td>
<td>232</td>
<td>66559</td>
</tr>
<tr>
<td>13</td>
<td>222</td>
<td>69583</td>
</tr>
<tr>
<td>14</td>
<td>224</td>
<td>68898</td>
</tr>
</tbody>
</table>
Lean Prolog Compiling Itself

<table>
<thead>
<tr>
<th>System/Program</th>
<th>Sequential</th>
<th>Multithreaded</th>
</tr>
</thead>
<tbody>
<tr>
<td>8 core MacPro slowest</td>
<td>11.46</td>
<td>4.89</td>
</tr>
<tr>
<td>8 core MacPro fastest</td>
<td>10.16</td>
<td>4.49</td>
</tr>
<tr>
<td>2 core MacAir slowest</td>
<td>14.29</td>
<td>8.53</td>
</tr>
<tr>
<td>2 core MacAir fastest</td>
<td>12.56</td>
<td>7.55</td>
</tr>
<tr>
<td>1 core NetBook slowest</td>
<td>84.39</td>
<td>62.21</td>
</tr>
<tr>
<td>1 core NetBook fastest</td>
<td>79.04</td>
<td>56.27</td>
</tr>
</tbody>
</table>

**Figure**: Lean Prolog bootstrapping time (in seconds)
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.

```prolog
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`.
Sequentializing remote predicate calls

seq_server(Port)

- uses the built-in new_seq_server(Port, Server) that creates a server listening on a port
- then it 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)),
    % do some work, check if stopped
    ...
    R\=stopped, seq_server_step(Service).
Integrating cooperative multi-tasking constructs

- we have seen that we can run tasks in parallel with minimal programmer controlled coordination
- use of multithreading (under the hood)
- however, we need sophisticated coordination for more complex tasks
- a fairly powerful model: associative, blackboard-based information exchanges (Linda + unification + indexing)
- 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?
Cooperative Coordination

- `new_coordinator(Db)` uses a database parameter `Db` to store the state of the Linda blackboard.
- The state of the blackboard is described by the dynamic predicates:
  - `available/1` keeps track of terms posted by `out` operations.
  - `waiting/2` collects pending `in` operations waiting for matching terms.
  - `running/1` helps passing control from one engine to the next.

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

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)).
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).
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))) )
        ),
    ),
    ... coordinate(C),
    stop_coordinator(C).
Running the Coordination Example

?- 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)

“concurrency for expressiveness” in terms of the logic-engines-as-interactors API provides flexible building blocks for the encapsulation of high-level concurrency patterns.
Conclusion

- **the context:** Logic Engines encapsulated as Interactors are be used to build on top of pure Prolog a practical Prolog system
- **the contribution:** 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*
- **the benefits:**
  - our language constructs are well-suited for today’s multi-core architectures – performance comes from keeping busy the actual parallel execution units
  - reducing the software risks coming from more complex concurrent execution mechanisms designed with massively parallel execution in mind
  - design patterns reusable in the design and implementation of new logic and functional programming constructs
Questions?

Lean Prolog and a few related papers are at:

- http://logic.cse.unt.edu/research/LeanProlog