

Computational Logic
Distributed/Internet Programming

LP/CLP, the Internet, and the WWW

- Can Logic and Constraint Logic Programming be an attractive alternative for Internet/WWW programming?
- Shared with other net programming tools:
 - ◇ dynamic memory management,
 - ◇ well-behaved structure (and pointer!) manipulation,
 - ◇ robustness, compilation to architecture-independent bytecode, ...
- In addition:
 - ◇ powerful symbolic processing capabilities,
 - ◇ dynamic databases,
 - ◇ search facilities,
 - ◇ grammars,
 - ◇ sophisticated meta-programming / higher order,
 - ◇ easy code (agent) motion,
 - ◇ well understood semantics, ...

LP/CLP, the Internet, and the WWW

- Most public-domain and commercial LP/CLP systems:
 - ◇ either already have Internet connection capabilities (e.g., socket interfaces),
 - ◇ or it is relatively easy to add it to them (e.g., through the C interface)(e.g., Quintus, LPA, PDC, Amzi!, IF-Prolog, Eclipse, SICStus, BinProlog, SWI, PrologIV, CHIP, Ciao, etc.)
- Some additional “glue” needed to make things really convenient:
 - ◇ We present several techniques for “filling in these gaps” (many implemented as public domain libraries).
- In doing this we also work towards answering the question:
 - ◇ Is there anything fundamental missing in current LP/CLP systems?
- Commercial systems add packages that provide higher-level functionality.

- Additional motivation: the WWW can be an excellent showcase for LP/CLP applications!

Outline

- (PART I: *WWW programming*)
 - PART II: *Distributed/agent programming*
 - ◇ (Modeling and accessing information servers –active modules).
 - ◇ A simple distributed LP/CLP language using “worker teams”.
 - ◇ Communicating via Blackboards.
 - ◇ Implementing distributed variable-based communication using attributed variables.
-
- Different concurrent/distributed execution scenarios:
 - ◇ Request/provide remote services in a distributed network (including database servers, WWW servers, etc.)
 - ◇ (Distributed) networks of concurrent, communicating agents
 - ◇ Coarse-grained Parallelism (granularity control required)
 - Most functionality can be obtained using current LP/CLP systems! (again, concurrency in the underlying engine is very useful)

Distributed Teams of Workers (Ciao)

- Team: set of workers (threads) that share the same code and cooperate to run it.
- Concurrency and/or parallelism occurs between workers.
- Worker management:
 - ◇ add_worker Add (possibly remote) worker to the team. Intuition:
 - * The system starts with one worker.
 - * If a worker is added at a remote site, it makes it possible to run goals at that site (similar to opening a file).
 - * If more than one worker is added (locally or at a given remote site) it is often either to achieve parallelism (in multiprocessor machines) or fairness (giving “gas” to different goals).
 - ◇ delete_worker Delete (possibly remote) worker from the team
- The workers are kept coherent from the point of view of code management, global state, etc.

Some Concurrency & Parallelism Operators (Ciao)

- Objective: express concurrency, independent and-parallelism, dependent and-parallelism, etc. (and support a notion of fairness), within a team of workers.
- Basic operators (in addition to sequential conjunction, etc.):
 - ◇ $A \&$ – Schedules goal A for execution (when a worker is free).
 - ◇ $A \&\&$ – “Fair” version of the $\&/1$ operator: if there is no idle worker, it creates one to execute A (new thread).
 - ◇ $A @ Id$ – Placement operator: goal A is to be executed on worker Id (which may be remote). Can be combined with the other operators.
 - ◇ $A \&> H$ – Schedules goal A, returns in H a *handler*.
 - ◇ $H <\&$ – waits for end of execution of goal pointed to by H, back-unifies bindings.
 - ◇ $A \& B$ – Schedules A and B, waits for the execution of both to finish.
 - ◇ Last one can be implemented using previous two:
$$A \& B :- B \&> H, A, H <\& .$$
 - ◇ Bindings in shared variables not guaranteed to be seen until threads join.
 - ◇ Full support for backtracking.

Using Basic Concurrency & Parallelism Operators

- `move(red), move(green)`.
- `move(red) &, move(green)`.
- `add_worker(I), move(red) &, move(green)`.
- `delete_workers, move(red) &&, move(green)`.
- `delete_workers, add_worker(alba,I), move(green) @ I`.

Using Parallelism: Examples

```
main :-
    read_input_list(L),
    collect_unloaded_hosts(Hosts),
    add_workers(Hosts, _Ids),
    process_list(L),
    halt.

process_list([]).
process_list([H|T]) :-
    process(H) &
    process_list(T).

add_workers([Host|Hosts], [Id|Ids]) :-
    add_worker(Host, Id),
    add_workers(Hosts, Ids).
add_workers([], []).
```

Using Parallelism: Examples

- One of the Ciao libraries is a parallelizing preprocessor
- Uses source-to-source transformation
- Includes some automatic granularity control
- Possible alternative using granularity control:

```
process_list([]).  
process_list([H|T]) :-  
    ( H < 5 ->  
        process_list(T), process(H)  
    ;   process(H) & process_list(T) ).
```

Implementation Issues

- Creating workers / threads:

- ◇ In standard systems: standard process creation primitives (e.g., “fork”, “rsh”, etc.) can be used.
- ◇ Better approach (for local threads): use engine capable of supporting multiple workers natively in an efficient way.
- ◇ The machines developed for parallel systems provide exactly the required functionality (e.g., RAP-WAM, ACE-WAM, DASWAM, etc., and even Aurora, Muse, ...).
Also starting to appear in other Prolog systems (e.g., BinProlog, SICStus).
- ◇ Interesting issue: how to support several independent executions without creating too many “stack sets”.
The “marker” models used in parallel systems address this issue.
- ◇ Scheduling: classical goal stacks and goal stealing strategies still appear most suitable.
- ◇ Distributed scheduling: through sockets (or blackboards)

Communication: Using Blackboards

- Blackboards (linda stile): basic but very useful means of communication and synchronization (higher level than using sockets directly)
- Present in many systems: SICStus, BinProlog/ μ^2 -Prolog, &-Prolog/Ciao, ...
- Basic features:
 - ◇ out/1: write tuple
 - ◇ rd/1: read tuple
 - ◇ in/1: remove tuple
 - ◇ rd_noblock/1 and in_noblock/1
 - ◇ in/2 and rd/2 (on disjunctions)
- Sometimes, several (possibly hierarchical) blackboards allowed – then, extra argument to primitives specifies which blackboard.

Producer-Consumer: Linda Version

(using Ciao / SICStus BB primitives)

```
?- create_bb(B,local), N=10,  
    lproducer(N,B) @ alba &, lconsumer(B).
```

```
lproducer(N,B) :-  
    lproducer(N,1,B).
```

```
% second argument is message order
```

```
lproducer(0,C,B) :- !,  
    linda:out(message(end(C)),B).
```

```
lproducer(N,C,B) :-  
    N>0,  
    linda:out(message(C,N),B),  
    N1 is N-1,  
    C1 is C+1,  
    lproducer(N1,C1,B).
```

Producer-Consumer: Linda Version

```
lconsumer(B) :-  
    lconsumer(1,B).
```

```
lconsumer(C,B) :-  
    linda:rd([message(end(C)),  
            message(_,C)], T, B),  
    lconsumer_data(T,B).
```

```
lconsumer_data(message(end(_)),B).  
lconsumer_data(message(N,C),B) :-  
    C1 is C+1,  
    lconsumer(C1,B).
```

Implementation Issues

- Implementation approaches and techniques:
 - ◇ Blackboard can be a Prolog process. Tuples maintained via assert/retract. Communication, e.g., via sockets (allows Internet-wide use of the blackboard).
 - ◇ Support blackboard internally in system (possibly, in conjunction with asserted database).
 - ◇ Mixed approach: local vs. remote blackboards.
 - ◇ The blackboard can also be a completely special purpose program (e.g., BinProlog's "Java blackboard").

Other Forms of Communication: Shared Variables

- Variable sharing/communication:
 - ◇ `share(X)` – bindings on the variables of `X` (tells) will be exported to other workers in the team
 - ◇ `unshare(X)` – bindings on the variables of `X` (tells) will be local
 - ◇ `wait(X)` – Suspends the execution until `X` is bound (also, `d_wait(X)`)
 - ◇ `ask(C)` – Suspends the execution until the constraint `C` is satisfied

- Example:

`share(X), (move(red), X=done) &, move(green), wait(X).`

A Simple Producer/Consumer Program (using Shared Vars)

```
go(L) :-
    share(L),
    consumer(L) &,
    producer(3,L).

producer(0,T) :- !, T = [].
producer(N,T) :- N > 0,
    T = [N|Ns],
    N1 is N-1,
    report(N,produced),
    producer(N1,Ns).

consumer(L) :-
    ask(L=[]), !.
consumer(L) :-
    ask(L=[H|T]),
    report(H,consumed),
    consumer(T).
```

Implementation Issues

- Shared variables can be implemented using attributed variables [Huitouze '90, Neumerkel '90] + blackboard:
 - ◇ variables involved in a parallel call are marked as a “communication” variable (i.e., shared)
 - ◇ done by attaching an attribute
 - ◇ communication variables are given unique identifiers
 - ◇ “shared” character is inherited during unification
 - ◇ standard tells done in place, tells to comm. variables posted on blackboard
 - ◇ asks do a blocking `rd` (read) on the blackboard
- All implementation done at source (Prolog) level (see our ICLP'95 paper)
- Blackboard-based systems and shared variable communication-based systems – “different camps:” they can be easily unified using this technique!

Other Issues

- Code and heap structure caching and coherence maintenance in distributed environments:
 - ◇ Very interesting work being done in the context of the OZ language, using techniques related to those used in multiprocessor cache coherence.
 - ◇ BinProlog and LogicWeb also support a form of code caching.
- Security: only a few proposals (e.g., BinProlog's)
- Alternative means of communication: Ports ([AKL], related to sockets), direct use of sockets, ...
- Logical views of reactivity? Use of linear logic, or condition-action rules as proposed by Kowalski?

Other Conclusions/Issues

- Some concurrency and parallelism operators proposed.
- Several forms of communication: blackboards, active objects, shared variables, sockets, ports, ...
- Attributed variables can be used for implementing distributed shared variable communication.
- All implementation can be done at source (Prolog) level.
- Native support for concurrency in underlying system very useful (e.g., in the Ciao run-time system, the &-Prolog abstract machine is used; similarly in BinProlog).
- Security, caching...

- Ciao code provided as public domain Prolog libraries (<http://www.clip.dia.fi.upm.es>)
- Put your LP/CLP-agent applications on the WWW!

Appendix: The Ciao System and its Libraries

- Ciao is an LP/CLP system developed at UPM, in collaboration with several other industrial and academic centers.
- In the Ciao project:
 - ◇ We try to design useful extensions of LP and CLP for distributed execution, WWW programming, concurrency, higher-order, powerful debugging, ...
 - ◇ We try to keep as much as possible compatibility with ISO-Prolog.
 - ◇ We develop the extensions as much as possible in the form of libraries.
 - ◇ We build public domain versions of these libraries for standard LP/CLP systems.
 - ◇ We identify aspects that are difficult or inefficient and for which native engine support is needed .
 - ◇ We develop abstract machine modifications and advanced compilation and support technology.
- I.e., we try to answer the question of what really needs to be added to/changed in current systems.

PiLLoW and Other Ciao Libraries

- For concreteness we will often refer to *PiLLoW* and other Ciao system libraries.
- Ciao Libraries (freely available, and in different stages of development) include:
 - ◇ *PiLLoW*: WWW/HTML interface
 - ◇ prolog shell: Prolog shell scripts
 - ◇ Distribution: blackboards, concurrency, agents, ...
 - ◇ PLAI: Global analysis (including type checking/inferencing)
 - ◇ APC: Global optimization (source to source, including specialization and parallelization)

Ciao Compiler Transformations/Optimizations (Source to Source)

- Examples of transformations/techniques used:
 - ◇ Supporting CLP via attributed variables.
 - ◇ Distributed execution on standard CLP/LP.
 - ◇ Supporting CC on standard CLP/LP systems (with delay).
 - ◇ Supporting the Andorra model in CLP/LP systems.
 - ◇ Functions/higher order.
- Analyses used / characteristics:
 - ◇ Top-down framework with efficient dynamic fixpoint (PLAI).
 - ◇ Modes, types, sharing (aliasing), independence, etc.
 - ◇ Several domains over Herbrand: *SH*, *SH+FR*, *ASub*, *SH+ASub*, *SH+FR+ASub*, *Path*, *Types*, ...
 - ◇ Over constraints: *Def*, *Fr*, *FD*, *LSign*, *DiffLSign*, ...
 - ◇ Support for dynamic scheduling (concurrency).
 - ◇ Support for incremental analysis.

- ◇ Support for full languages (e.g., ISO-prolog).
- ◇ Cost analysis (upper and lower bounds).
- Examples of optimizations performed:
 - ◇ Compile-time elim. of run-time tests via (abstract) PE.
 - ◇ Multiple (abstract) specialization (e.g., loop invariants).
 - ◇ LP/CLP/CC parallelization.
 - ◇ Optim. of synchronization / sched. anal. (for delays and CC).
 - ◇ Goal and constraint reordering (optimization of search).
 - ◇ Granularity control.

Ciao and Other CC Systems

- Input from other LP/CC systems:
 - ◇ CC: entailment-based synchronization.
 - ◇ NU-Prolog/Par NU-Prolog: transformation to delay declaration for support of Ciao on conventional systems.
 - ◇ AKL: encapsulation.
 - ◇ OZ: modules, applications of records.
 - ◇ Shared with QE-Janus: “quiche eating” implementation approach.
- Main differences:
 - ◇ “Sequential by default” vs. “concurrent by default.”
 - ◇ Explicit concurrency (and parallelism) operators (“threads”).
 - ◇ Distributed implementation.
 - ◇ Extensive global analysis and optimization (e.g., automatic static parallelization, suspension reduction).
 - ◇ Designed to be portable to conventional LP/CLP systems.

- Other issues:
 - ◇ Active modules.
 - ◇ WWW interface.
 - ◇ Functions, HO, scripts, ...

Language Visions?

- On the future LP Language: can Ciao offer some interesting ideas?
 - ◇ Backwards compatible with LP/CLP (ISO standard).
 - ◇ Can use existing implementation technology.
 - ◇ Incorporates some language solutions:
 - * Sequential operator.
 - * Separation of parallelism and concurrency.
 - * Explicit request for fairness.
 - * Distribution primitives.
 - * Active modules/objects.
 - * Separation of control rules (e.g., Andorra) from parallelism and optimizations.
 - * Integration of several in the same framework.
 - * ...
 - ◇ Final thoughts – minor things matter, e.g., in Ciao:
 - * tcl/tk interface.

- * Stand-alone executables, linkables, *and scripts*.
- * Small executables.
- * *html* interface.
- * ...