A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism

Amadeo Casas[1] Manuel Carro[2] Manuel Hermenegildo[1,2]

[1]University of New Mexico (USA)

[2]Technical University of Madrid (Spain) and IMDEA-Software (Spain)

December 12th, 2008

Introduction

- Parallelism (finally!) becoming mainstream thanks to multicore architectures — even on laptops!
- Parallelizing programs is a hard challenge.
  - Necessity to exploit parallel execution capabilities as easily as possible.
- Renewed research interest in development of tools to write parallel programs:
  - Design of languages that better support exploitation of parallelism.
  - Improved libraries for parallel programming.
  - Progress in support tools: **parallelizing compilers**.

  (Different objectives from "multi-threading" –already supported.)

Why Logic Programming?

- Declarative languages (and logic programming languages among them) are a very interesting framework for parallelization:
  - Program much closer to problem description.
  - Notion of control provides more flexibility.
  - Cleaner semantics (e.g., pointers exist, but are declarative).
  - Amenability to semantics-preserving automatic parallelization.
- Industry interest:
  - E.g., Intel sponsorship of *DAMP* workshops (colocated with POPL).
- Previous work by same authors:
  - **LOPSTR'07**: annotation algorithms for *unrestricted* IAP.
  - **PADL'08**: execution model for parallel execution of *deterministic* goals.

Types of parallelism in LP

- Two main types:
  - *Or-Parallelism*: explores in parallel **alternative computation branches**.
  - *And-Parallelism*: executes **procedure calls** in parallel.
    - ⋆ Traditional parallelism: parbegin-parend, loop parallelization, divide-and-conquer, etc.
    - ⋆ Often marked with `&/2` operator: fork-join nested parallelism.

Types of parallelism in LP

- Two main types:
  - *Or-Parallelism*: explores in parallel **alternative computation branches**.
  - *And-Parallelism*: executes **procedure calls** in parallel.
    - ★ Traditional parallelism: parbegin-parend, loop parallelization, divide-and-conquer, etc.
    - ★ Often marked with &/2 operator: fork-join nested parallelism.

Example (QuickSort: sequential and parallel versions)

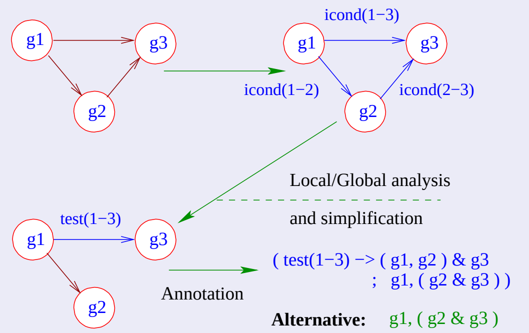
```
qsort([], []).                       qsort([], []).
qsort([X|L], R) :-                   qsort([X|L], R) :-
   partition(L, X, SM, GT),             partition(L, X, SM, GT),
   qsort(GT, SrtGT),                    qsort(GT, SrtGT) &
   qsort(SM, SrtSM),                    qsort(SM, SrtSM),
   append(SrtSM, [X|SrtGT], R).         append(SrtSM, [X|SrtGT], R).
```

- We will focus herein on and-parallelism.

CDG-based automatic parallelization

- Conditional Dependency Graph:
  - Vertices: possible sequential tasks (statements, calls, etc.)
  - Edges: conditions needed for independence (e.g., variable sharing).
- Local or global analysis to remove checks in the edges.
- Annotation converts graph back to (now parallel) source code.

```
foo(...)  :-
        g1(...),
        g2(...),
        g3(...).
```



An alternative, more flexible source code annotation

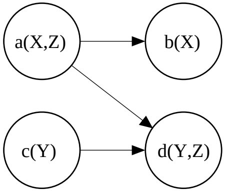
- Classical parallelism operator `&/2`: nested fork-join.
  - Rigid structure of `&/2`.
- However, more flexible constructions can be used to denote parallelism:
  - `G &>` H_G — schedules goal `G` for parallel execution and continues executing the code after `G &>` H_G.
    - H_G is a *handler* which contains / points to the state of goal `G`.
  - H_G `<&` — waits for the goal associated with H_G to finish.
    - The goal associated to H_G has produced a solution: bindings for the output variables are available.
- Operator `&/2` can be written as:

```
A & B :- A &> H, call(B), H <&.
```

- Optimized deterministic versions: `&!>/2`, `<&!/1`.

Expressing more parallelism

- More parallelism can be exploited with these primitives.
- Consider sequential code below (dep. graph at the right) and two possible parallelizations:



Sequential	Restricted IAP	Unrestricted IAP
`p(X,Y,Z) :- a(X,Z), b(X), c(Y), d(Y,Z).`	`p(X,Y,Z) :- a(X,Z) & c(Y), b(X) & d(Y,Z).` `p(X,Y,Z) :- c(Y) & (a(X,Z),b(X)), d(Y,Z).`	`p(X,Y,Z) :- c(Y) &> Hc, a(X,Z), b(X) &> Hb, Hc <&, d(Y,Z), Hb <&.`

- In this case: unrestricted parallelization guaranteed equal to or better (time-wise) than restricted ones, assuming no overhead.

Objectives of the execution model for unrestricted IAP

- Several previous implementations supporting and-parallelism:
  - &-Prolog, &-ACE, DASWAM, AKL, Andorra-I,...
- Most based on multi-sequential, marker-based ("&-Prolog") model.
  - A set of *WAM-like* agents.
- Implementation has relied on low-level machinery –complex.
  - New WAM instructions.
  - Goal stacks, parcall frames, markers, etc.
- Objective of current work:
  - Rise a good portion to the source language (Prolog/ImProlog) level.
  - Try to keep sufficient performance.

(... in the Ciao spirit of keeping the kernel small.)

High-level implementation of unrestricted IAP

- What to do at what level:
  - **Prolog-level**: goal publishing / searching etc. (goal stealing-based scheduling), marker creation, backtracking management, ...
  - **C-level**: low-level threading, locking, stack management, sharing of memory, untrailing, ...
  - Current implementation for shared-memory multiprocessors:
    - ⋆ Agent: sequential Prolog machine + goal list + (mostly) Prolog code.

$\longrightarrow$ Simpler machinery and more flexibility.

- Some issues:
  - A goal *list* for each agent (instead of a goal stack)
    - ⋆ Unrestricted parallelism.
    - ⋆ Makes goal cancellation easier.
  - Implement *parcall frames* as *heap structures*. Accessible at source level as *goal handlers*.
  - Markers implemented through normal choice points at source level (+ some fields in handlers).

Creation of (high-level) markers / canceling

Non-deterministic goal publishing

```
Goal &> Handler :-
   add_goal(Goal,nondet,Handler),
   undo(cancellation(Handler)),
   release_some_suspended_thread.
```

Goal startup

```
Handler <& :-
   enter_mutex_self,
   (
      goal_available(Handler) ->
      exit_mutex_self,
      retrieve_goal(Handler,Goal),
      call(Goal)
   ;
      check_if_finished_or_failed(Handler)
   ).
Handler <& :-
   add_goal(Handler),
   release_some_suspended_thread,
   fail.
```

Creation of (high-level) markers / canceling

Goal startup

```
work :-
   (  read_event(Handler) ->
      ...
   ;  (
         find_goal(H) ->
         exit_mutex_self,
         call_handler(H)
      ;  ...
```

Execution of parallel goal

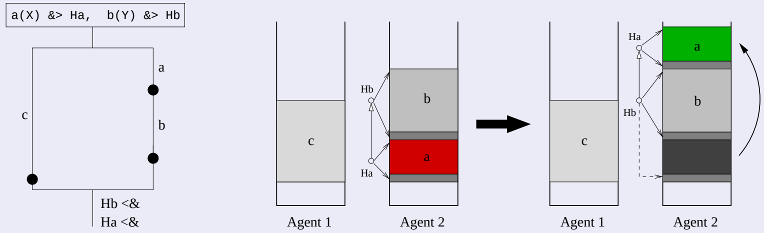
```
call_handler(Handler) :-
   retrieve_goal(Handler,Goal),
   save_init_execution(Handler),
   call(Goal),
   save_end_execution(Handler),
   enter_mutex(Handler),
   set_goal_finished(Handler),
   release(Handler),
   exit_mutex(Handler).
```

```
call_handler(Handler) :-
   enter_mutex(Handler),
   set_goal_failed(Handler),
   release(Handler),
   metacut_garbage_slots(Handler),
   exit_mutex(Handler),
   fail.
```

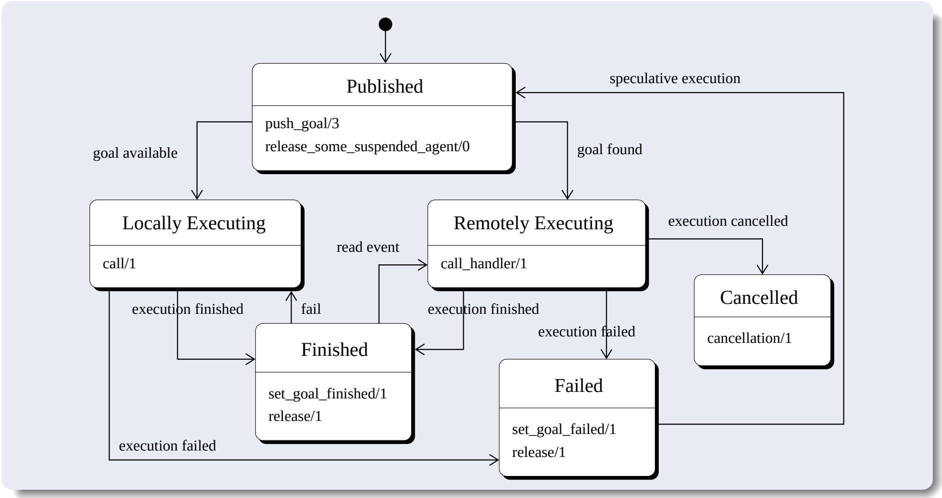
Memory management problems in nondeterministic IAP execution

- Lots of issues in memory management.
- In particular, dealing with the *trapped goals* and *garbage slots* problems:
- Agents created with small stacks which grow on demand.

```
?- a(X) &> Ha,  b(Y) &> Hb,  c(Z),  Hb <&,  Ha <&, fail.
```



State diagram of a parallel goal



Performance results

- *Sun Fire T2000:*
  - 8 cores and 8 Gb of memory, each of them capable of running 4 threads in parallel.
    - ⋆ Speedups with more than 8 threads stop being linear even for completely independent computations, since threads in the same core compete for shared resources.
  - Implemented in *Ciao*.
  - All performance results obtained by averaging 10 runs.

Performance results

Deterministic vs. Non-deterministic annotation

Benchmark	Op.	Number of processors							
		1	2	3	4	5	6	7	8
AIAKL	&!	0.97	1.82	1.82	1.82	1.83	1.83	1.83	1.82
	&	0.96	1.70	1.71	1.72	1.74	1.75	1.72	1.72
Ann	&!	0.98	1.86	2.72	3.56	4.38	5.16	5.88	6.64
	&	0.96	1.85	2.72	3.57	4.35	5.14	5.87	6.61
Deriv	&!	0.91	1.63	2.37	3.05	3.78	4.49	4.98	5.49
	&	0.84	1.60	2.34	2.99	3.73	4.43	4.56	4.85
FFT	&!	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
	&	0.98	1.72	1.97	2.65	2.67	2.75	2.93	2.97
Hanoi	&!	0.89	1.76	2.47	3.32	3.77	4.17	4.61	5.25
	&	0.89	1.77	1.91	2.84	3.13	3.54	3.96	4.47
MMatrix	&!	0.91	1.74	2.55	3.32	4.18	4.83	5.55	6.28
	&	0.90	1.48	2.16	2.88	3.51	4.13	4.71	5.25
QuickSort	&!	0.97	1.78	2.31	2.87	3.19	3.46	3.67	3.75
	&	0.97	1.71	2.17	2.43	2.60	2.93	3.06	3.19
Takeuchi	&!	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72
	&	0.88	1.45	2.02	2.85	3.41	3.80	4.23	4.66

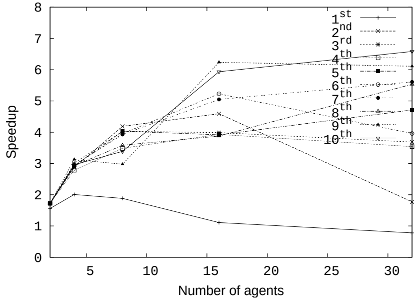
Performance results

Non-deterministic benchmarks

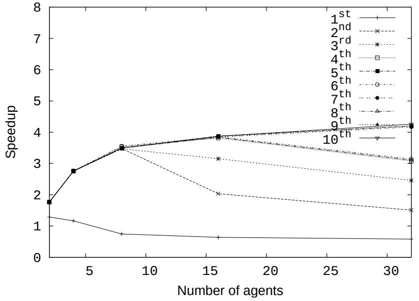
- Performance results obtained in some representative non-deterministic parallel benchmarks:

Benchmark	**Number of processors**							
	1	2	3	4	5	6	7	8
Chat	2.31	4.49	5.42	6.91	9.79	9.95	11.10	17.29
Numbers	1.84	1.79	1.79	1.79	1.79	1.79	1.78	1.78
Progeom	0.99	0.96	0.97	0.98	0.98	0.98	0.98	0.98
Queens	0.99	0.94	0.94	0.94	0.94	0.94	0.94	0.94
QueensT	0.99	1.90	2.41	3.18	4.71	4.61	4.58	4.57

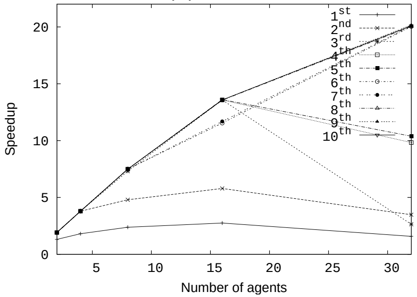
- Super-linear speedups are achievable, thanks to good failure implementation (e.g., eager goal cancellation).



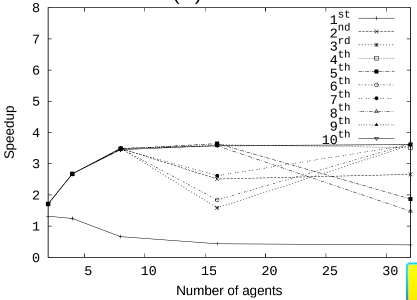
(a) Boyer



(b) FFT



(c) Fibonacci



(d) QuickSort

Conclusions and future work

- Improved high-level implementation of and-parallelism:
  - Main implementation components raised to the source level.
  - Simpler machinery and more flexibility.
  - *Full support for non-determinism / backtracking.*
- Performance results:
  - Reasonable speedups are achievable.
  - Super-linear speedups can be achieved, thanks to goal cancellation.
  - Unrestricted and-parallelism provides better observed speedups.
  - Parallel backtracking support has limited impact on deterministic execution efficiency.
- Future work involves improvements in execution model:
  - Design efficient parallel garbage collection algorithms for this implementation.
  - Exploitation of other sources of parallelism.
  - Combination with concurrency models.