Towards High-Level Execution Primitives for And-parallelism: Preliminary Results

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

[1]University of New Mexico (USA)
[2]Technical University of Madrid (Spain)

CICLOPS'07 - September 8^{th}

Introduction and motivation

- Parallelism (finally!) becoming mainstream thanks to multicore architectures – even on laptops!
- Declarative languages interesting for parallelization:
  - Program close to problem description.
  - Notion of control provides more flexibility.
  - Amenability to semantics-preserving automatic parallelization.
- Significant previous work in logic and functional programming.
- Two objectives in this work:
  - New, efficient, and more flexible approach for exploiting (unrestricted) (and-)parallelism in LP.
  - Take advantage of new automatic parallelization for LP.

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

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

- We will focus on and-parallelism.
  - Need to detect *independent* tasks.

Background: parallel execution and independence

- **Correctness:** same results as sequential execution.
- **Efficiency:** execution time $\leq$ than seq. program (no slowdown), assuming parallel execution has no overhead.

	Imperative	Functional	CLP
s_1	Y := W+2;	(+ (+ W 2)	Y = W+2,
s_2	X := Y+Z;	Z)	X = Y+Z,

```
main :-
    s1   p(X),
    s2   q(X),
         write(X).
```

```
p(X) :- X = [1,2,3].

q(X) :- X = [], large computation.
q(X) :- X = [1,2,3].
```

- Fundamental issue: p *affects* q (prunes its choices).
  - q ahead of p is *speculative*.
- **Independence:** *correctness* + *efficiency*.

Related work and proposed solution

- Versions of and-parallelism previously implementated: &-Prolog, &-ACE, AKL, Andorra-I,...
- They rely on complex low-level machinery:
  - Each agent: new WAM instructions, goal stack, parcall frames, markers, etc.
- Current implementation for shared-memory multiprocessors:
  - Each agent: sequential Prolog machine + goal list + (mostly) Prolog code.
- Approach: rise components to the source language level:
  - **Prolog-level**: goal publishing, goal searching, goal scheduling, "marker" creation (through choice-points),...
  - **C-level**: low-level threading, locking, stack management, sharing of memory, untrailing,...

$\longrightarrow$ Simpler machinery and more flexibility.

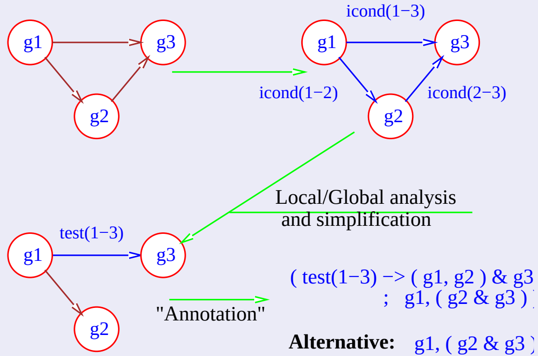
Ciao and CiaoPP

- *Ciao*: new generation multi-paradigm language.
  - Supports ISO-Prolog (as a library).
  - Predicates, functions (including laziness), constraints, higher-order, objects, tabling, etc.
  - Parallel, concurrent and distributed execution primitives.
- Preprocessor / environment (CiaoPP):
  - Infers many properties such as types, pointer aliasing, non-failure, determinacy, termination, data sizes, cost, etc.
  - Performs automatic verification of program assertions (and bug detection if assertions are proved false).
  - Performs *automatic parallelization and automatic granularity control*.

CDG-based automatic parallelization

- Conditional Dependency Graph: **[TOPLAS'99, JLP'99]**
  - 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

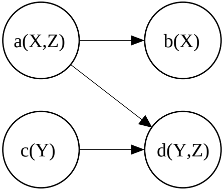
- Classical parallelism operator &/2: nested fork-join.
- However, more flexible constructions can be used to denote parallelism:
  - G &> H_G — schedules goal G for parallel execution and continues executing the code after G &> H_G.
    - H_G is a *handler* which contains / points to the state of goal G.
  - H_G <& — waits for the goal associated with H_G to finish.
    - The goal H_G was associated to 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.
- Take the sequential code below (dep. graph at the right) and three possible parallelizations:



Sequential

```
p(X,Y,Z) :-
    a(X,Z),
    b(X),
    c(Y),
    d(Y,Z).
```

Restricted IAP

```
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).
```

Unrestricted IAP

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

- In this case: unrestricted parallelization at least as good (time-wise) as any restricted one, assuming no overhead.

Low-level support

- Low-level parallelism primitives:

```
apll:push_goal(+Goal,+Det,-Handler).
apll:find_goal(-Handler).
apll:goal_available(+Handler).
apll:retrieve_goal(+Handler,-Goal).
apll:goal_finished(+Handler).
apll:set_goal_finished(+Handler).
apll:waiting(+Handler).
```

- Synchronization primitives:

```
apll:suspend.
apll:release(+Handler).
apll:release_some_suspended_thread.
apll:enter_mutex(+Handler).
apll:enter_mutex_self.
apll:release_mutex(+Handler).
apll:release_mutex_self.
```

Prolog-level algorithms (I)

- Thread creation:

```
create_agents(0) :- !.
create_agents(N) :-
  N > 0,
  conc:start_thread(agent),
  N1 is N - 1,
  create_agents(N1).
```

```
agent :-
  apll:enter_mutex_self,
  (
    find_goal_and_execute -> true
  ;
    apll:exit_mutex_self,
    apll:suspend
  ),
  agent.
```

- High-level goal publishing:

```
Goal &!> Handler :-
  apll:push_goal(Goal,det,Handler),
  apll:release_some_suspended_thread.
```

Prolog-level algorithms (II)

- Performing goal joins:

```
Handler <&!  :-
  apll:enter_mutex_self,
  (
    apll:goal_available(Handler) ->
    apll:retrieve_goal(Handler,Goal),
    apll:exit_mutex_self,
    call(Goal)
  ;
    apll:exit_mutex_self,
    perform_other_work(Handler)
  ).
```

```
perform_other_work(Handler) :-
  apll:enter_mutex_self,
  (
    apll:goal_finished(Handler),
    apll:exit_mutex_self,
  ;
    (
      find_goal_and_execute -> true
    ;
      apll:exit_mutex_self,
      apll:suspend
    ),
    perform_other_work(Handler)
  ).
```

Prolog-level algorithms (III)

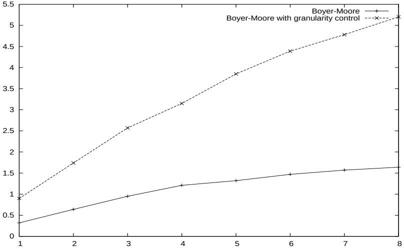
- Search for parallel goals:

```
find_goal_and_execute :-
   apll:find_goal(Handler),
   apll:exit_mutex_self,
   apll:retrieve_goal(Handler,Goal),
   call(Goal),
   apll:enter_mutex(Handler),
   apll:set_goal_finished(Handler),
   (
     apll:waiting(Handler) ->
     apll:release(Handler)
   ;
     true
   ),
   apll:exit_mutex(Handler).
```

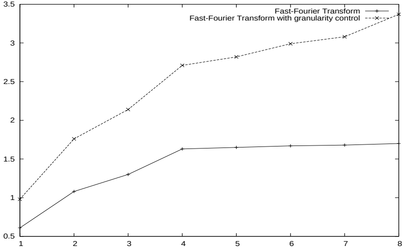
(Preliminary) performance results for restricted and-parallelism (I)

Benchmark	Number of processors								
	Seq.	1	2	3	4	5	6	7	8
AIAKL	1.00	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
Ann	1.00	0.98	1.86	2.65	3.37	4.07	4.65	5.22	**5.90**
Boyer	1.00	0.32	0.64	0.95	1.21	1.32	1.47	1.57	1.64
BoyerGC	1.00	0.90	1.74	2.57	3.15	3.85	4.39	4.78	**5.20**
Deriv	1.00	0.32	0.61	0.86	1.09	1.15	1.30	1.55	1.75
DerivGC	1.00	0.91	1.63	2.37	3.05	3.69	4.21	4.79	**5.39**
FFT	1.00	0.61	1.08	1.30	1.63	1.65	1.67	1.68	1.70
FFTGC	1.00	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
Fibonacci	1.00	0.30	0.60	0.94	1.25	1.58	1.86	2.22	2.50
FibonacciGC	1.00	0.99	1.95	2.89	3.84	4.78	5.71	6.63	**7.57**
Hanoi	1.00	0.67	1.31	1.82	2.32	2.75	3.20	3.70	4.07
HanoiDL	1.00	0.47	0.98	1.51	2.19	2.62	3.06	3.54	3.95
HanoiGC	1.00	0.89	1.72	2.43	3.32	3.77	4.17	4.41	**4.67**
MMatrix	1.00	0.91	1.74	2.55	3.32	4.18	4.83	5.55	**6.28**
Palindrome	1.00	0.44	0.77	1.09	1.40	1.61	1.82	2.10	2.23
PalindromeGC	1.00	0.94	1.75	2.37	2.97	3.30	3.62	4.13	**4.46**
QuickSort	1.00	0.75	1.42	1.98	2.44	2.84	3.07	3.37	3.55
QuickSortDL	1.00	0.71	1.36	1.95	2.26	2.76	2.96	3.18	3.32
QuickSortGC	1.00	0.94	1.78	2.31	2.87	3.19	3.46	3.67	**3.75**
Takeuchi	1.00	0.23	0.46	0.68	0.91	1.12	1.32	1.49	1.72
TakeuchiGC	1.00	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63

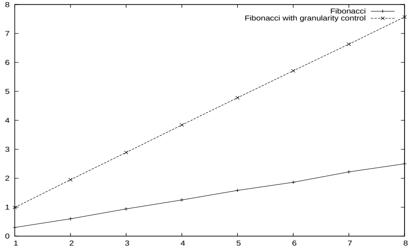
(Preliminary) performance results for restricted and-parallelism (II)



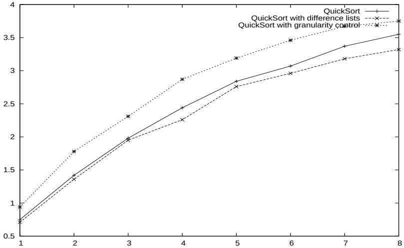
(a) Boyer-Moore



(b) Fast-Fourier Transform



(c) Fibonacci

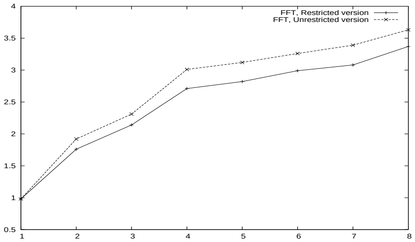


(d) Quicksort

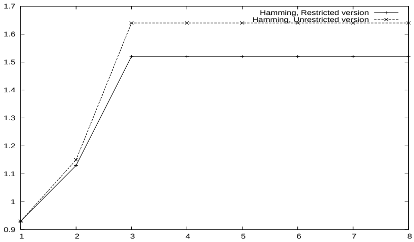
Restricted vs. unrestricted and-parallelism (I)

Benchm.	**And-P**	**Number of processors**							
		1	2	3	4	5	6	7	8
FibFunGC	Restricted	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Unrestricted	0.99	1.95	2.89	3.84	4.78	5.71	6.63	**7.57**
TakeuchiGC	Restricted	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	Unrestricted	0.88	1.62	2.39	3.33	4.04	4.47	5.19	**5.72**
FFTGC	Restricted	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	Unrestricted	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
Hamming	Restricted	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	Unrestricted	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
WMS2	Restricted	0.99	1.01	1.01	1.01	1.01	1.01	1.01	1.01
	Unrestricted	0.99	1.10	1.10	1.10	1.10	1.10	1.10	1.10

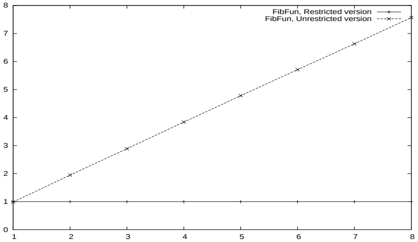
Restricted vs. unrestricted and-parallelism (II)



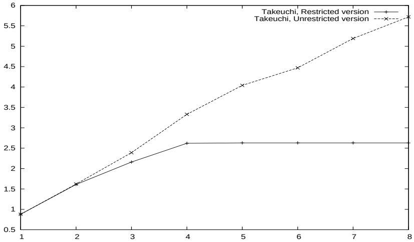
(e) FFT



(f) Hamming



(g) FibFun



(h) Takeuchi

Conclusions and future work

- New implementation approach for exploiting and-parallelism:
  - Simpler machinery.
  - More flexibility.
- Preliminary results:
  - Reasonable speedups are achievable.
  - Additional overhead makes it necessary to perform granularity control.
- Unrestricted and-parallelism:
  - Provides better observed speedups!
- Currently working on:
  - Improving implementation.
  - Developing compile-time (automatic) parallelizers for this approach **[LOPSTR'07]**.