Towards a High-Level Implementation of Flexible Parallelism Primitives for Symbolic Languages

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

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

PASCO'07 - July 28^{th}, 2007

Introduction (I) - Motivation

- Parallelism (finally!) becoming mainstream thanks to multicore architectures – even on laptops!
- Declarative languages interesting for parallelization:
  - Notion of control provides more flexibility.
  - Amenability to semantics-preserving automatic parallelization.
- And also well-suited to write symbolic computation algorithms:
  - Program close to problem description.
- Much previous work:
  - Logic programming (LP) languages.
  - Functional languages: Erlang, Sisal, etc.
- Two objectives in this work:
  - New, efficient, more flexible approach for exploiting parallelism in LP.
  - Automatic parallelization of logic programs.

Introduction (II) - 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 & operator: fork-join nested parallelism.

Introduction (II) - 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 & 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.

Introduction (III) - Notion of Independence

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

s_1	Y := W+2;	(+ (+ W 2)	Y = W+2,
s_2	X := Y+Z;	Z)	X = Y+Z,
	(imperative)	**(functional)**	**(CLP)**

`main :-`	`p(X) :- X = [1,2,3].`
s_1 `p(X),`	
s_2 `q(X),`	`q(X) :- X = [],` *large computation*.
`write(X).`	`q(X) :- X = [1,2,3].`

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

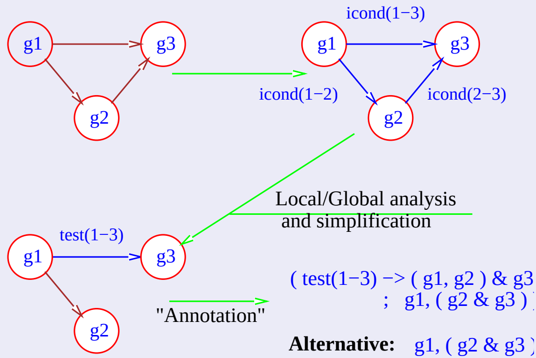
Introduction (IV) - Ciao

- *Ciao*, new generation multi-paradigm language.
  - Supports ISO-Prolog fully (as a library).
- Predicates, functions (including laziness), constraints, higher-order, objects, etc.
- Global analyzer which **infers** many properties such as:
  - Types, pointer aliasing, non-failure, determinacy, termination, data sizes, cost, etc.
- Automatic verification of program assertions (and bug detection if assertions are proved false).
- Parallel, concurrent and distributed execution primitives + automatic parallelization and automatic granularity control.

Automatic Parallelization (I) - CDGs

- Conditional dependency graph:
  - Vertices: possible tasks (statements, calls, etc.).
  - Edges: conditions needed for independence: variable sharing.
- Local or global analysis to remove checks in the edges.
- Annotation process converts graph back to parallel expressions in source.

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



Automatic Parallelization (II) - Flexible Parallelism Primitives (I)

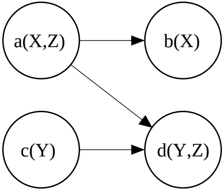
- More flexible constructions to represent parallelism:
  - `G &> H` — schedules goal `G` for parallel execution and continues executing the code after `G &> H`.
    - `H` is a *handler* which contains the state of goal `G`.
  - `H <&` — waits for the goal associated with `H` to finish.
    - Bindings made for the output variables of the parallel goal associated to `H` are available (i.e., goal has produced a complete solution).
- Operator `&` written as:

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

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

Automatic Parallelization (III) - Flexible Parallelism Primitives (II)

- More parallelism can be exploited with these primitives:




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

(sequential)

```
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).
```

(restricted IAP)

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

(unrestricted IAP)

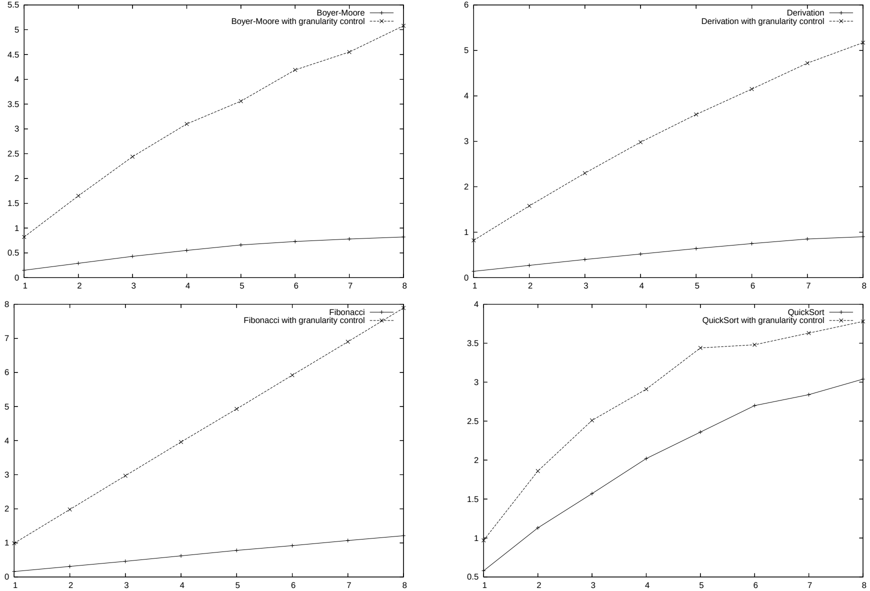
Shared-Memory Implementation

- Versions of and-parallelism previously implementated: &-Prolog, &-ACE, AKL, Andorra-I.
- They rely on complex low-level machinery:
  - Each agent: goal stack, parcall frames, markers, etc.
- Current implementation for shared-memory multiprocessors:
  - Each agent: sequential Prolog machine + goal list + Prolog code.
- Approach: rise components to the source language level:
  - **Prolog-level**: goal publishing, goal searching and goal scheduling.
  - **C-level**: low-level threading, locking, stack management, sharing of memory and untrailing.
  - Simpler machinery and more flexibility.

Performance Results (I) - Restricted And-Parallelism

Benchmark	**Number of processors**								
	Seq.	1	2	3	4	5	6	7	8
AIAKL	1.00	0.94	1.76	1.80	1.80	1.79	1.52	1.77	1.76
Ann	1.00	0.97	1.77	2.61	3.22	3.98	4.52	5.14	**5.61**
Boyer	1.00	0.17	0.33	0.49	0.60	0.70	0.81	0.89	0.94
BoyerGC	1.00	0.86	1.66	2.45	3.13	3.66	4.17	4.63	**5.10**
Deriv	1.00	0.16	0.33	0.45	0.57	0.68	0.80	0.90	0.99
DerivGC	1.00	0.77	1.40	2.05	2.66	3.24	3.66	4.13	**4.57**
FFT	1.00	0.30	0.48	0.59	0.67	0.75	0.77	0.80	0.82
FFTGC	1.00	0.97	1.72	2.16	2.65	2.77	2.94	3.06	3.19
Fibonacci	1.00	0.15	0.29	0.42	0.55	0.67	0.81	0.95	1.09
FibonacciGC	1.00	0.99	1.94	2.88	3.81	4.75	5.69	6.63	**7.52**
Hamming	1.00	0.89	1.19	1.43	1.43	1.43	1.43	1.43	1.43
Hanoi	1.00	0.46	0.83	1.19	1.50	1.75	1.86	2.21	2.44
HanoiDL	1.00	0.24	0.45	0.68	0.85	1.07	1.28	1.47	1.67
HanoiGC	1.00	0.98	1.80	2.33	2.89	3.32	3.70	3.80	**4.07**
MMatrix	1.00	0.76	1.46	2.11	2.82	3.46	4.02	4.59	**5.18**
QuickSort	1.00	0.57	1.08	1.52	1.90	2.25	2.56	2.81	2.98
QuickSortDL	1.00	0.52	0.97	1.32	1.69	2.11	2.35	2.63	2.86
QuickSortGC	1.00	0.98	1.78	2.30	2.85	3.18	3.44	3.62	**3.68**
Takeuchi	1.00	0.11	0.21	0.31	0.40	0.47	0.56	0.61	0.69
TakeuchiGC	1.00	0.87	1.53	2.16	2.59	2.60	2.60	2.60	2.60

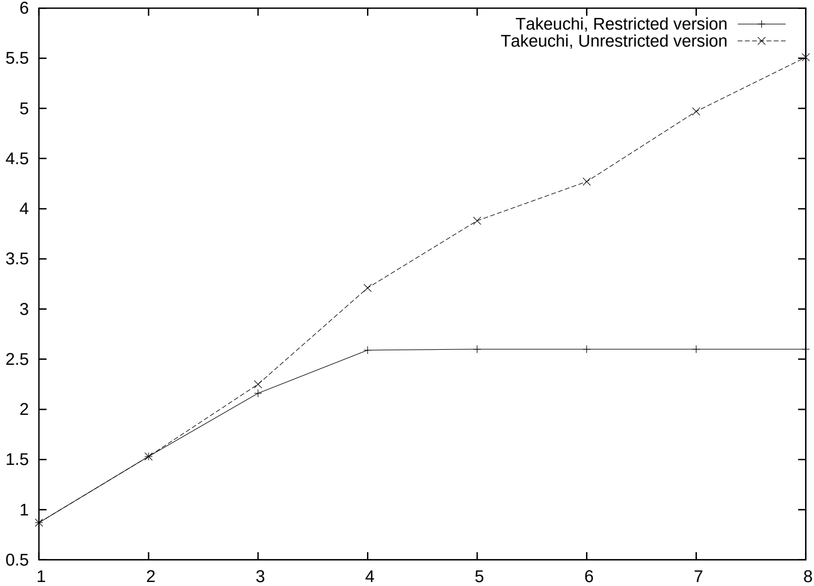
Performance Results (II) - Granularity Control



Performance Results (III) - Unrestricted And-Parallelism

Benchm.	**And-P**	**Number of processors**							
		1	2	3	4	5	6	7	8
FibFun	Restr.	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	Unrestr.	0.99	1.94	2.88	3.81	4.75	5.69	6.63	**7.52**
Takeuchi	Restr.	0.87	1.53	2.16	2.59	2.60	2.60	2.60	2.60
	Unrestr.	0.87	1.53	2.25	3.21	3.88	4.27	4.97	**5.51**
FFT	Restr.	0.97	1.72	2.16	2.65	2.77	2.94	3.06	3.19
	Unrestr.	0.97	1.73	2.19	2.69	2.83	3.01	3.18	3.31
Hamming	Restr.	0.89	1.19	1.43	1.43	1.43	1.43	1.43	1.43
	Unrestr.	0.89	1.21	1.51	1.51	1.51	1.51	1.51	1.51
WMS2	Restr.	0.99	1.01	1.01	1.01	1.01	1.01	1.01	1.01
	Unrestr.	0.99	1.10	1.10	1.10	1.10	1.10	1.10	1.10

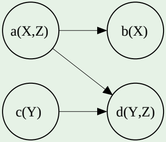
Performance Results (IV) - Comparison Restr./Unrestr. Takeuchi



Conclusions and Future Work

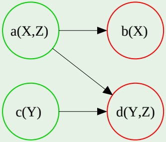
- New implementation approach for exploiting and-parallelism:
  - Simpler machinery.
  - More flexibility.
- Preliminary results:
  - Reasonable speedups are achievable.
  - Additional overhead $\longrightarrow$ granularity control.
  - Also, advances in compilation and improved implementation should (partly) recover efficiency lost due to overhead.
  - Unrestricted and-parallelism provides better speedups.
- Currently working on improving implementation and developing compile-time (automatic) parallelizers for this approach.

Example (Unrestricted Annotation)



Indep	Dep	Joinable	Fork	And	Join	Pub
						$\emptyset$

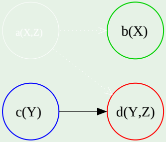
Example (Unrestricted Annotation)



Indep	Dep	Joinable	Fork	And	Join	Pub
						$\emptyset$
$\{a, c\}$	$\{b, d\}$	$\{a\}$	$\{c\}$	$\{a\}$	$\emptyset$	$\{a, c\}$

```
p(X,Y,Z) :-
    c(Y) &> Hc,
    a(X,Z),
```

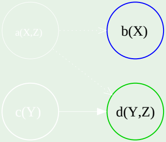
Example (Unrestricted Annotation)



Indep	Dep	Joinable	Fork	And	Join	Pub
						$\emptyset$
$\{a, c\}$	$\{b, d\}$	$\{a\}$	$\{c\}$	$\{a\}$	$\emptyset$	$\{a, c\}$
$\{b, c\}$	$\{d\}$	$\{c\}$	$\{b\}$	$\emptyset$	$\{c\}$	$\{a, c\}$

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

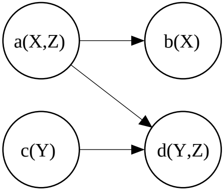
Example (Unrestricted Annotation)



Indep	Dep	Joinable	Fork	And	Join	Pub
						$\emptyset$
$\{a,c\}$	$\{b,d\}$	$\{a\}$	$\{c\}$	$\{a\}$	$\emptyset$	$\{a,c\}$
$\{b,c\}$	$\{d\}$	$\{c\}$	$\{b\}$	$\emptyset$	$\{c\}$	$\{a,c\}$
$\{b,d\}$	$\emptyset$	$\{b,d\}$	$\emptyset$	$\{d\}$	$\{b\}$	$\{a,b,c\}$

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

Minimum Time to Execute a Parallel Expression (I)



fj1

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

$$T_{fj1} = \max(T_a, T_c) + \max(T_b, T_d)$$

fj2

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

$$T_{fj2} = \max(T_a + T_b, T_c) + T_d$$

Minimum Time to Execute a Parallel Expression (II)

dep

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

$$T_1 = 0$$

$$T_2 = T_1$$

$$T_3 = T_2 + T_a$$

$$T_4 = T_3$$

$$T_5 = \max(T_3, T_1 + Tc)$$

$$T_6 = T_5 + T_d$$

$$T_7 = \max(T_6, T_3 + T_b) = T_{dep}$$

Minimum Time to Execute a Parallel Expression (III)

Tfj1 = max(a, c) + max(b, d)

```
tfj1(A,B,C,D,T) :-
   positive([A,B,C,D,T]),
   max(A,C,MAC),
   max(B,D,MBD),
   T .=.  MAC + MBD.
```

```
max(X,Y,X):- X .>=.  Y.
max(X,Y,Y):- X .<.  Y.
```

```
positive([]).
positive([X|Xs]):-
   X .>.  0,
   positive(Xs).
```

Tfj2 = max(a+b, c) + d

```
tfj2(A,B,C,D,T) :-
   positive([A,B,C,D,T]),
   AB .=.  A + B,
   max(AB,C,MaxABC),
   T .=.  D + MaxABC.
```

Tdep = max(a+b, d + max(a,c))

```
tdep(A,B,C,D,T):-
   positive([A,B,C,D,T]),
   AB .=.  A + B,
   max(A, C, MaxAC),
   DAC .=.  D + MaxAC,
   max(AB, DAC, T).
```

Minimum Time to Execute a Parallel Expression (IV)

In any fork-join parallelization always better than the other one?

```
?- tfj1(A,B,C,D,T1),
   tfj2(A,B,C,D,T2),
   T1 .<.  T2.

   yes
```

```
?- tfj1(A,B,C,D,T1),
   tfj2(A,B,C,D,T2),
   T2 .<.  T1.

   yes
```

Can fork-join parallelization be better than unrestricted parallelization?

```
?- tfj1(A,B,C,D,T1),
   tdep(A,B,C,D,T2),
   T1 .<.  T2.

   no
```

```
?- tfj2(A,B,C,D,T1),
   tdep(A,B,C,D,T2),
   T1 .<.  T2.

   no
```

- No combination of execution times can make the unrestricted parallelization be worse than the restricted parallelization!