Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs

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

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

LOPSTR'07 - August 24th

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.
- Objective in this work:
  - An efficient and more flexible approach for (automatically) exploiting and-parallelism in logic programs.

Background: 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.

Background: 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.

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

```
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*.

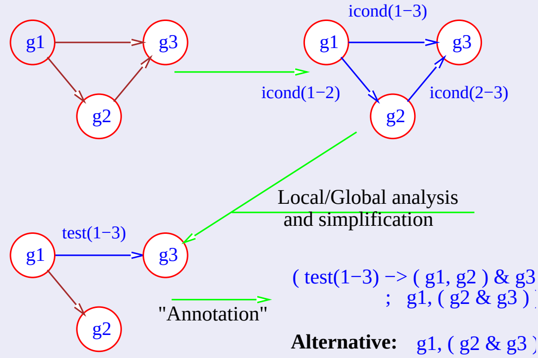
Ciao

- *Ciao*, new generation multi-paradigm language.
  - Supports ISO-Prolog (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.

CDG-based automatic parallelization

- **C**onditional **D**ependency **G**raph:
  - 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(...).
```



(test(1−3) −> (g1, g2) & g3
; g1, (g2 & g3)

Alternative: g1, (g2 & g3)

An alternative, more flexible source code annotation

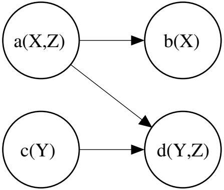
- Classical parallelism operator `&/2`: nested fork-join.
- However, more flexible constructions can be used to denote parallelism:
  - `G &>` $\mathtt{H}_\mathtt{G}$ — schedules goal `G` for parallel execution and continues executing the code after `G &>` $\mathtt{H}_\mathtt{G}$.
    - ★ $\mathtt{H}_\mathtt{G}$ is a *handler* which contains / points to the state of goal `G`.
  - $\mathtt{H}_\mathtt{G}$ `<&` — waits for the goal associated with $\mathtt{H}_\mathtt{G}$ to finish.
    - ★ The goal $\mathtt{H}_\mathtt{G}$ 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.

New annotation algorithms: general idea

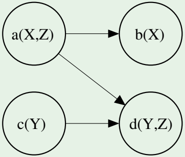
- Main idea:
    - *Publish goals* (e.g., `G &> H`) as soon as possible.
    - *Wait for results* (e.g., `H <&`) as late as possible.
    - One clause at a time.
- Limits to how soon a goal is published + how late results are gathered are given by the dependencies with the rest of the goals in the clause.
- As with `&/2`, annotation may respect or not relative order of goals in clause body.
    - Order determined by `&>/2`.
    - Order not respected ⇒ more flexibility in annotation.

Automatic parallelization with alternative primitives

Non order-preserving, unrestricted annotation

pvt: nearest goal to be scheduled among those dependent on already scheduled but not finished goals.

Example (Unrestricted Annotation UUDG)



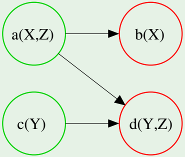
Indep	Dep	pvt	ToPub	ToWait	Pub
					$\emptyset$

Automatic parallelization with alternative primitives

Non order-preserving, unrestricted annotation

pvt: nearest goal to be scheduled among those dependent on already scheduled but not finished goals.

Example (Unrestricted Annotation UUDG)



Indep	Dep	pvt	ToPub	ToWait	Pub
					∅
{*a*, *c*}	{*b*, *d*}	*b*	{*a*, *c*}	{*a*}	{*a*, *c*}

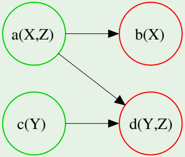
```
p(X,Y,Z) :-
        c(Y) &> Hc,
        a(X,Z) &> Ha,
        Ha <&,
```

Automatic parallelization with alternative primitives

Non order-preserving, unrestricted annotation

pvt: nearest goal to be scheduled among those dependent on already scheduled but not finished goals.

Example (Unrestricted Annotation UUDG)



Indep	Dep	pvt	ToPub	ToWait	Pub
					$\emptyset$
$\{a, c\}$	$\{b, d\}$	b	$\{a, c\}$	$\{a\}$	$\{a, c\}$

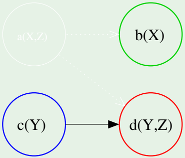
```
p(X,Y,Z) :-
    c(Y) &> Hc,
    a(X,Z),
```

Automatic parallelization with alternative primitives

Non order-preserving, unrestricted annotation

pvt: nearest goal to be scheduled among those dependent on already scheduled but not finished goals.

Example (Unrestricted Annotation UUDG)



Indep	Dep	pvt	ToPub	ToWait	Pub
					$\emptyset$
$\{a, c\}$	$\{b, d\}$	b	$\{a, c\}$	$\{a\}$	$\{a, c\}$
$\{b, c\}$	$\{d\}$	d	$\{b\}$	$\{c\}$	$\{a, b, c\}$

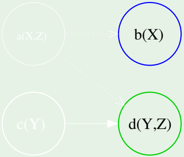
```
p(X,Y,Z) :-
        c(Y) &> Hc,
        a(X,Z),
        b(X) &> Hb,
        Hc <&,
```

Automatic parallelization with alternative primitives

Non order-preserving, unrestricted annotation

pvt: nearest goal to be scheduled among those dependent on already scheduled but not finished goals.

Example (Unrestricted Annotation UUDG)



Indep	Dep	pvt	ToPub	ToWait	Pub
					$\emptyset$
$\{a, c\}$	$\{b, d\}$	b	$\{a, c\}$	$\{a\}$	$\{a, c\}$
$\{b, c\}$	$\{d\}$	d	$\{b\}$	$\{c\}$	$\{a, b, c\}$
$\{b, d\}$	$\emptyset$	$-$	$\{d\}$	$\{b, d\}$	$\{a, b, c, d\}$

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

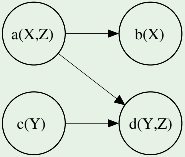
- Note goal order switched w.r.t. sequential version of clause.

Automatic parallelization with alternative primitives

Order-preserving, unrestricted annotation

Seq also constrained by left-to-right order.

Example (Unrestricted Annotation UOUDG)



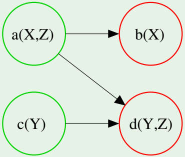
Indep	Dep	pvt	ToPub	ToWait	Pub
					$\emptyset$

Automatic parallelization with alternative primitives

Order-preserving, unrestricted annotation

Seq also constrained by left-to-right order.

Example (Unrestricted Annotation UOUDG)



Indep	Dep	pvt	ToPub	ToWait	Pub
					$\emptyset$
$\{a, c\}$	$\{b, d\}$	b	$\{a\}$	$\{a\}$	$\{a\}$

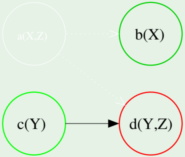
```
p(X,Y,Z) :-
        a(X,Z),
```

Automatic parallelization with alternative primitives

Order-preserving, unrestricted annotation

Seq also constrained by left-to-right order.

Example (Unrestricted Annotation UOUDG)



Indep	Dep	pvt	ToPub	ToWait	Pub
					$\emptyset$
$\{a, c\}$	$\{b, d\}$	b	$\{a\}$	$\{a\}$	$\{a\}$
$\{b, c\}$	$\{d\}$	d	$\{b, c\}$	$\{c\}$	$\{a, b, c\}$

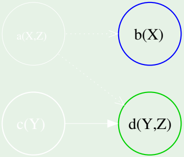
```
p(X,Y,Z) :-
    a(X,Z),
    b(X) &> Hb,
    c(Y),
```

Automatic parallelization with alternative primitives

Order-preserving, unrestricted annotation

Seq also constrained by left-to-right order.

Example (Unrestricted Annotation UOUDG)



Indep	Dep	pvt	ToPub	ToWait	Pub
					$\emptyset$
$\{a, c\}$	$\{b, d\}$	b	$\{a\}$	$\{a\}$	$\{a\}$
$\{b, c\}$	$\{d\}$	d	$\{b, c\}$	$\{c\}$	$\{a, b, c\}$
$\{b, d\}$	$\emptyset$	$-$	$\{d\}$	$\{b, d\}$	$\{a, b, c, d\}$

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

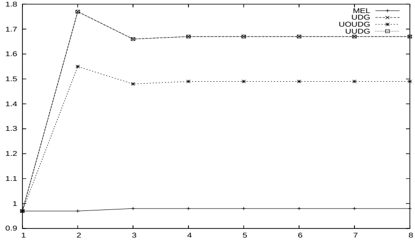
Performance results

Speedups

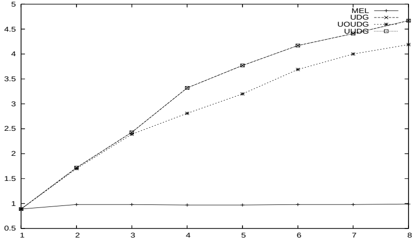
Benchm.	**Ann.**	**Number of processors**							
		1	2	3	4	5	6	7	8
AIAKL	UMEL	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98
	UOUDG	0.97	1.55	1.48	1.49	1.49	1.49	1.49	1.49
	UDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
	UUDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
Hanoi	UMEL	0.89	0.98	0.98	0.97	0.97	0.98	0.98	0.99
	UOUDG	0.89	1.70	2.39	2.81	3.20	3.69	4.00	4.19
	UDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
	UUDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
FibFun	UMEL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UOUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
	UDG	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Takeuchi	UMEL	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UOUDG	0.88	1.62	2.17	2.64	2.67	2.67	2.67	2.67
	UDG	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UUDG	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72

Performance results

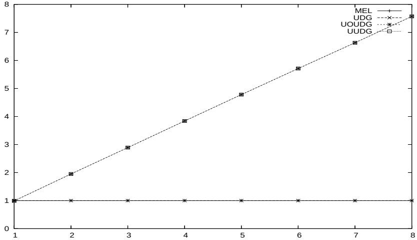
Restricted vs. Unrestricted And-Parallelism



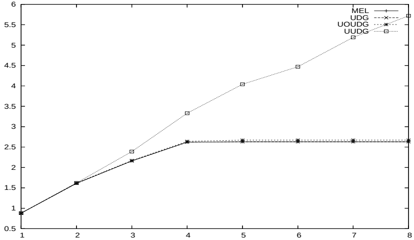
(a) AIAKL



(b) Hanoi



(c) FibFun



(d) Takeuchi

Conclusions and future work

- We have presented two algorithms to perform source-to-source transformation of a logic program into an unrestricted independent and-parallel version of itself.
  - Both respecting or not the order of the solutions.
- Unrestricted and-parallelism:
  - Provides better observed speedups.
  - Benefits are potentially larger for programs with high numbers of goals in their clauses.
- Currently improving parallelizers to take into account additional compile-time information.

FibFun

```
fib(0) := 0.
fib(1) := 1.
fib(N) := N>1 ?
          fib(N+1) +
          fib(N+2).
```

Functional

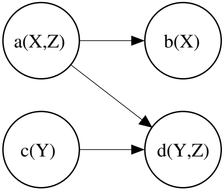
```
fib(0,0).
fib(1,1).
fib(N,M) :-
   N>1,
   N1 is N-1,
   fib(N1,M1),
   N2 is N-2,
   fib(N2,M2),
   M is M1 + M2.
```

Logic

```
fib(0,0).
fib(1,1).
fib(N,M) :-
   N>1,
   N1 is N-1,
   fib(N1,M1) &> H,
   N2 is N-2,
   fib(N2,M2),
   H <&,
   M is M1 + M2.
```

Unrestricted IAP

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.   positive([]).
max(X,Y,Y):- X .<.  Y.    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!