Parallel Execution of Logic Programs
A Tutorial

(Or: Multicores are here! Now, what do we do with them?)

Manuel Hermenegildo

IMDEA Software
Tech. University of Madrid
U. of New Mexico

Compulog/ALP Summer School – Las Cruces, NM, July 24-27 2008

The UPM work presented is a joint effort with members of the CLIP group at the UPM School of Computer Science and IMDEA Software including: Francisco Bueno, Daniel Cabeza, Manuel Carro, Amadeo Casas, Pablo Chico, Jess Correas, María José García de la Banda, Manuel Hermenegildo, Pedro López, Mario Mndez, Edison Mera, Jos Morales Jorge Navas, and Germán Puebla.

Introduction / Motivation

- *Multicore chips* have moved parallelism from niche (HPC) to mainstream –even on laptops!
- According to vendors (and Intel in particular [e.g., DAMP workshops]):
  - Feature size reductions will continue for foreseeable future (12 generations!).
  - But power consumption does not allow increasing clock speeds much.
  - Multicore is the way to use this space without raising power consumption.
  - Number of cores expected to *double* with each generation!
- But writing parallel programs hard/error-prone –how to exploit all those cores?
  - Ideal situation: *Conventional Program* + *Multiprocessor* = *Higher Perf.* → automatic parallelization.
  - More realistically: compiler-*aided* parallelization.
  - Languages (dialects, constructs) for parallelization+parallel programming.
  - Scheduling techniques [BW93, Cie92], memory management, abstract machines, etc.

LP and CLP – quite interesting from the parallelism point of view

- Many parallelism-friendly aspects:
  - ◇ program close to problem description → less hiding of intrinsic parallelism
  - ◇ well understood mathematical foundation → simplifies formal treatment
  - ◇ relative purity (well behaved variable scoping, fewer side-effects, generally single assignment) → more amenable to automatic parallelization.

- At the same time, requires *dealing with the most complex problems* [Her97, Her00]:
  - ◇ irregular computations; complex data structures; (well behaved) pointers; dynamic memory management; recursion; ...

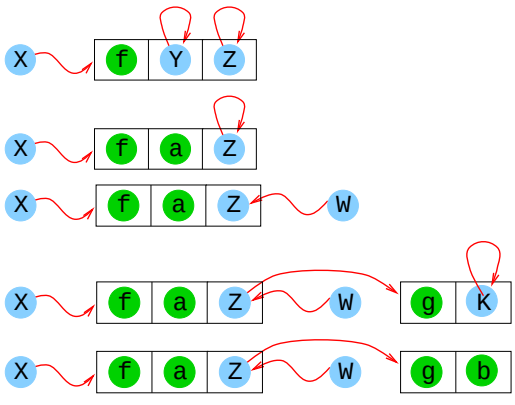
  but in a much more elegant context;
  and brings up some upcoming issues (e.g., speculation, search, constraints).

→ Very good platform for developing *universally useful techniques*:
Examples to date: conditional dep. graphs, abstract interpretation w/interesting domains, cost analysis / gran. control, dynamic sched. and load balancing, ...

Complex Data Structures / Pointers

- Example:

```
main :- X = f(Y,Z),
        Y = a,
        W = Z,
        W = g(K),
        X = f(a,g(b)).
```



Parallelism in Logic Programs and CLP

- Or-parallelism [Con83]: execute simultaneously different search space branches.
  - Present in general search problems, enumeration part of constr. problems, etc.

```
money(S,E,N,D,M,O,R,Y) :-          digit(0).
        digit(S),                  digit(1).
        digit(E),                  ...
        ...,                       digit(9).
        carry(I),
        ...,                       carry(0).
        N is E+O-10*I,             carry(1).
```

- And-parallelism [Con83]: execute simultaneously different clause body goals.
  - Comprises traditional parallelism (parallel loops, divide and conquer, etc.).
  - *Concurrent languages* also generally based on and-parallelism.

```
qsort([X|L],R) :-
        partition(L,X,L1,L2),
        qsort(L2,R2),
        qsort(L1,R1),
        append(R1,[X|R2],R).
```


Objective and Issues

- Temptation: make use of all this potential.
- Problem: this can yield a slowdown or even erroneous results.
- Objective [HR89]: and/or-parallel execution of (some of the goals in) logic programs (and full Prolog, CLP, CC, ...), while:
  - obtaining the same solutions as the sequential execution (i.e., **correctness**)
  - taking a shorter or equal execution time (*speedup* or, at least, *no-slowdown* over state-of-the-art sequential systems) (i.e., **efficiency**).
- Above conditions may not always be met:
  - Independence: conditions that the run-time behavior of the goals must satisfy to guarantee correctness and efficiency (under ideal conditions – no overhead).
- The presence of overheads complicates things further:
  - Granularity Control: techniques for ensuring efficiency in the presence of overheads.

Sequential and Parallel Execution Framework: OR

- Model [HR95]: consider a state $G = \langle g_1 : g_2 : \ldots : g_n, \theta \rangle$ where we select g_1.
- If there are two clauses:
  $g_1^{'} \leftarrow g_{11}^{'}, \ldots, g_{1m}^{'}.$ s.t. $mgu(g_1\theta, g_1^{'}) = \theta'$
  $g_1^{''} \leftarrow g_{11}^{''}, \ldots, g_{1k}^{''}.$ s.t. $mgu(g_1\theta, g_1^{''}) = \theta''$
- We construct two states:
  $G' = \langle g_{11}^{'} : \ldots : g_{1m}^{'} : g_2 : \ldots : g_n, \theta\theta' \rangle$
  $G'' = \langle g_{11}^{''} : \ldots : g_{1k}^{''} : g_2 : \ldots : g_n, \theta\theta'' \rangle$
- Sequential execution: execute G' first and then G''.
- Parallel execution: execute G' and G'' in parallel.
- Since G' and G'' are completely independent [HR95]:
  - Same results are obtained in parallel or sequentially.
  - All branches can be explored in parallel.
  - Same number of branches explored (only if “all sols”!).
- Thus, or-parallelism: mostly implementation issues.
  (but side-effects, cuts, and aggregation predicates complicate things)

Issues in OR Parallelism

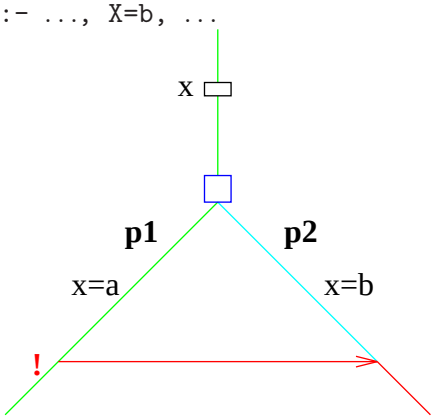
- System organization:
  - System comprises a collection of workers (processes/processors).
  - Each worker is an LP/CLP engine with a full set of stacks.
  - A scheduler assigns unexplored branches to idle workers.
- Main implementation problem: alternative bindings – efficiently maintaining different environments per branch (e.g., p_1 and p_2 in example):
  - **Sharing** (e.g. *Aurora* [LBD+88], PEPSys/ECLIPSE [CSW88, ECR93], etc.)
  - **Recomputation** (e.g. *Delphi* model) [Clo87].
  - **Copying** (e.g. *Muse* system) [AK90] ECLIPSE [ECR93], SICStus, OZ).
  - Theoretical limitations [GJ93]. Desirable:
    * Constant–time access to variables
    * Constant–time task creation
    * Constant–time task switching

    Impossible to meet all three with a finite number of processors.
    (Hence, they are not met in sequential execution!)

Issues in Or-parallelism: Illustration

```
..., p(X), ...

p1(X) :- ..., X=a, ..., !, ...
p2(X) :- ..., X=b, ...
```




```
main :- l, s.

:- parallel l/0.
l :- large_work_a.
l :- large_work_b.

:- parallel s/0.
s :- small_work_a.
s :- small_work_b.
```


Issues in OR Parallelism

- Speculation (e.g., p_2 in example).
  - To guarantee speedup: avoid speculative work – too strong/difficult?
  - To guarantee no-slowdown:
    - Left-biased scheduling.
    - Instantaneous killing on cut.
- Granularity: avoid parallelizing work that is too small.
- Parallelization can be done:
  - Adding `parallel/1` annotations to selected predicates (ANL,ECLIPSE)
  - Others (Aurora, MUSE) automatically via the scheduler.
- Useful supporting techniques identified:
  - Visualization/trace analysis: ANL, VisAndOr/IDRA [CGH93, FCH96], ViMust, Parsee [PK96], VisAll [FIVC98], ...
  - Program transformation to increase granularity [Pre93].
  - Compile-time/run-time granularity control; automatically introduce `parallel` annotations [LGHD96].

Some Results in OR Parallelism

- Quite successful systems built (ECLIPSE, SICSTUS/MUSE, Aurora, OrpYap[RSC99], etc.)
- MUSE is quite easy to add to an existing Prolog system (done with Prolog by BIM, also added to SICStus Prolog V3.0)
- Significant speedups w.r.t. state-of-the-art Prolog systems can be obtained with Aurora and Muse *for search-based applications*.

Program	1	2	4	8	10	Sicstus 0.6
parse1	1	1.8	2.8	2.93	2.76	1.25
parse5	1	1.97	3.74	6.92	7.72	1.27
db5	1	1.93	3.74	6.92	7.34	1.37
8queens	1	1.99	3.95	7.88	9.6	1.25
tina	1	2.07	4.06	7.81	9.59	1.43

- Much work done on schedulers (left bias, cut, side effects, ...)

- Easy to extend to CLP (e.g., VanHentenryck [Van89], ECLIPSE system).

Simple Goal-level And-Parallel Exec. Framework

- Model [HR90, HR95]:
  consider a state $G = \langle g_1 : g_2 : \ldots : g_n, \theta \rangle$, to execute g_1 and g_2 in parallel:
  - ◇ execute $\langle g_1, \theta \rangle$ and $\langle g_2, \theta \rangle$ in parallel (fork) obtaining θ_1 and θ_2,
  - ◇ continue with $\langle g_3 : \ldots : g_n, \theta_1\theta_2 \rangle$ (join).
- Regarding multiple solutions – two possibilities:
  - ◇ Gather all solutions for both goals separately.
  - ◇ Perform "parallel backtracking".
- *Multiple problems*, related to *variable binding conflicts*: during parallel execution of $\langle g_1, \theta \rangle$ and $\langle g_2, \theta \rangle$ the same variable may be bound to inconsistent values.
- Correctness problems (due to the definition of composition of substitutions – e.g. x/a composed with x/b succeeds!) [HR89]
  Solutions (proved correct in case of "pure" goals):
  - ◇ Modify definition of composition: $\theta \circ \eta(t) = mgu(E(\theta) \cup E(\eta))(t)$
  - ◇ Change parallel model.
  - ◇ Not an issue in CLP: conjunction instead of composition [dlBHM93, dlBHM00].

Issues in And-Parallelism – Independence

- *Correctness:* "same" solutions as sequential execution.
- *Efficiency:* execution time $<$ than seq. program (or, at least, *no-slowdown*: $\leq$).
  (We assume parallel execution has no overhead in this first stage.)
- Running at s_2 "seeing s_1":

	Imperative	*Functions*	*Constraints*
s_1	`Y := W+2;`	`(+ W 2)`	`Y = W+2,`
s_2	`X := Y+Z;`	`(+ Z)`	`X = Y+Z,`
	read-write deps	*strictness*	*cost!*

For *Predicates* (multiple procedure definitions):

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

```
p(X) :- X=a.
---------------------------
q(X) :- X=b, large computation.
q(X) :- X=a.
```

Again, cost *issue: if* `p` affects `q` *(prunes its choices) then* `q` *ahead of* `p` *is* speculative.

- *Independence:* condition that guarantees correctness *and efficiency*.

Independence and its Detection

- Informal notion: a computation "does not affect" another (also referred to as "stability" in, e.g., EAM/AKL).
- Greatly clarified when put in terms of Search Space Preservation (SSP) – shown SSP sufficient and necessary condition for efficiency [dlBHM93, dlB94].
- Detection of independence:
  - Run-time (a-priori conditions) [Con83, LK88, JH91].
  - Compile-time [CDD85].
  - Mixed: conditional execution graph expressions [DeG84, Her86b]. (1)
  - User control: explicit parallelism (concurrent languages). (2)
- (1)+(2) = &-Prolog [DeG84, Her86b]: view parallelization as a source to source transformation of original program into a parallelized ("annotated") one in a *concurrent/parallel* language. Allows:
  - Automatic parallelization — and understanding the result).
  - User parallelization — and the compiler checking it).

Concrete System Used in Examples: `Ciao`

- For concreteness, hereafter we use &-Prolog (now Ciao) as a target. The relevant minimal subset of &-Prolog/Ciao:
  - Prolog (with if-then-else, etc.).
  - Parallel conjunction "`&/2`" (with correct and complete forwards and backwards semantics).
  - A number of primitives for run-time testing of instantiation state.
- Ciao [HC94, HBC+99, HBC+08, BCC+06] is one of the popular Prolog/CLP systems (supports ISO-Prolog fully). Many other features: new-generation *multi-paradigm* language/prog.env. with:
  - Predicates, constraints, functions (including lazyness), higher-order, ...
  - Assertion language for expressing rich program properties (types, shapes, pointer aliasing, non-failure, determinacy, data sizes, cost, ...). Static debugging, verification, program certification, PCC, ...
  - Parallel, concurrent, and distributed execution primitives.
    - Automatic parallelization.
    - Automatic granularity and resource control.

A Priori Independence: Strict Independence-I

- Approach (goal level). Consider parallelizing `p(X,Y)` and `q(X,Z)`:

```
main :-
     t(X,Y,Z),
 s1  p(X,Y),
 s2  q(X,Z).
```

  We compare the behaviour of s_2`q(X,Z)` and s_1`q(X,Z)`.

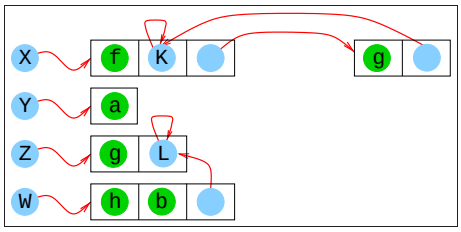
- A-priori Independence: when reasoning only about s_1.
  Can be checked at run-time before execution of the goals.
- A priori independence in the Herbrand domain:
  Strict Independence [DeG84, HR89]: goals do not share variables at run-time.
- *Example 1:* Above, if `t(X,Y,Z) :- X=a.`

A Priori Independence: Strict Independence-II

- The “pointers” view:

  correctness and efficiency (search space preservation) guaranteed for `p & q` if there are no “pointers” between `p` and `q`.

```
main :- X=f(K,g(K)), Y=a,
        Z=g(L), W=h(b,L),
        ---------------------->
        p(X,Y),
        q(Y,Z),
        r(W).
```



  `p` and `q` are strictly independent, but `q` and `r` are not.

A Priori Independence: Strict Independence-III

- *Example 2:*

```
qs([X|L],R) :- part(L,X,L1,L2),
               qs(L2,R2), qs(L1,R1),
               app(R1,[X|R2],R).
```

Might be annotated in &-Prolog (or Ciao) as:

```
qs([X|L],R) :-
        part(L,X,L1,L2),
        ( indep(L1,L2) -> qs(L2,R2) & qs(L1,R1)
                       ;  qs(L2,R2) , qs(L1,R1) ),
        app(R1,[X|R2],R).
```

- Not always possible to determine locally/statically:

```
main :- t(X,Y),       p(X), q(Y).

main :- read([X,Y]), p(X), q(Y).
```

- Alternatives: run-time independence tests, global analysis, ...

Fundamental issues:

- Can we build a system which obtains speedups w.r.t. a state of the art sequential LP system using such annotations?

- Can those annotations be generated automatically?

And-Parallelism Implementation

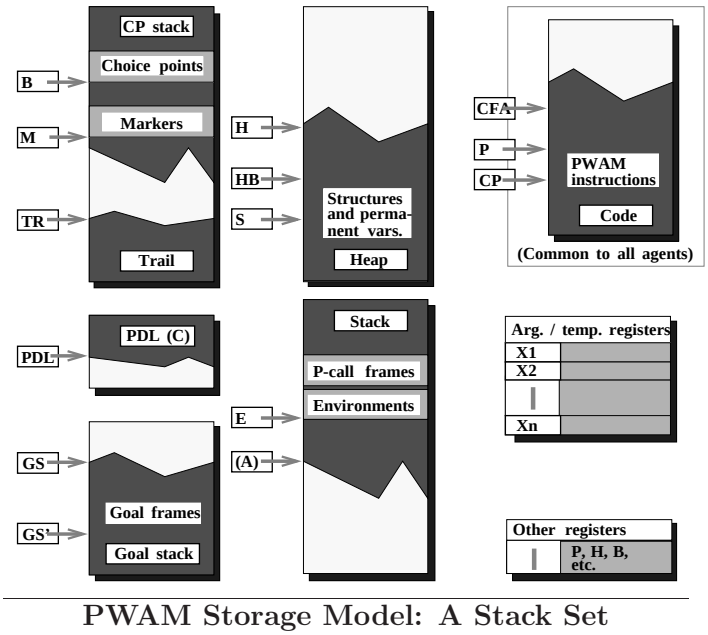
- By translation to or-parallelism [ECR93, CDO88]:
  - Simplicity
  - Relatively high overhead → higher need for granularity control
  - Used, e.g., in ECLIPSE system.
- Direct implementation [Her86b]:
  - Abstract machine needs to be modified: e.g., PWAM / Marker model [Her87, Her86a, SH96, PG98], EAM/AKL box machine [War90, JH90].
    - System comprises a collection of agents (processes/processors).
    - Each agent is an LP/CLP engine with a full set of stacks.
    - Scheduling is normally done lazily through goal stacks.
  - Low overhead (but granularity control still useful)
  - Direct support for concurrent/parallel language
  - Used in &-Prolog/Ciao and most other systems: ACE, IDIOM, DDAS, ...
- Also, higher-level implementations (see later).

And-Parallelism Implementation

- Issues in direct implementation:
  - Scheduling / fast task startup.
  - Memory management.
  - Use of analysis information to improve indexing.
  - Local environment support.
  - Recomputation vs. copying.
  - Efficient implementation of parallel backtracking (and opportunities for intelligent backtracking).
  - Efficient implementation of “ask” (for communication among threads).
  - etc.

&-Prolog Run-time System: PWAM architecture

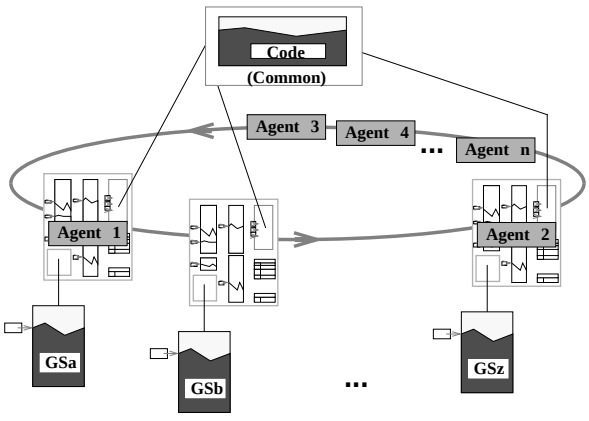
- Evolution of the RAP-WAM (the first Multisequential Model?) and Sicstus WAM.
- Defined as a storage model + an instruction set



PWAM Storage Model: A Stack Set

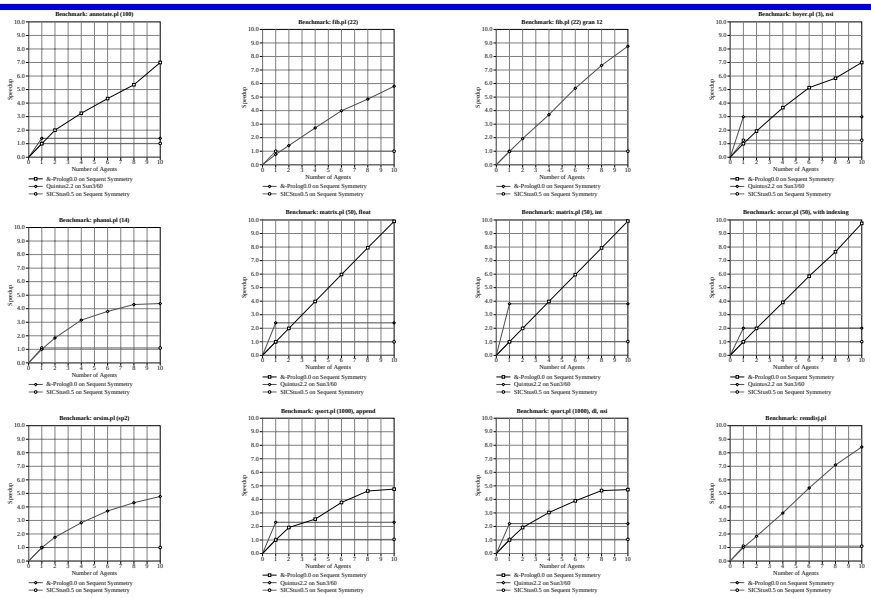
&-Prolog Run-time System: Agents and Stack Sets

- Agents separate from Stack Sets; Dynamic creation/deletion of S.Sets/Agents
- Lazy, on demand scheduling



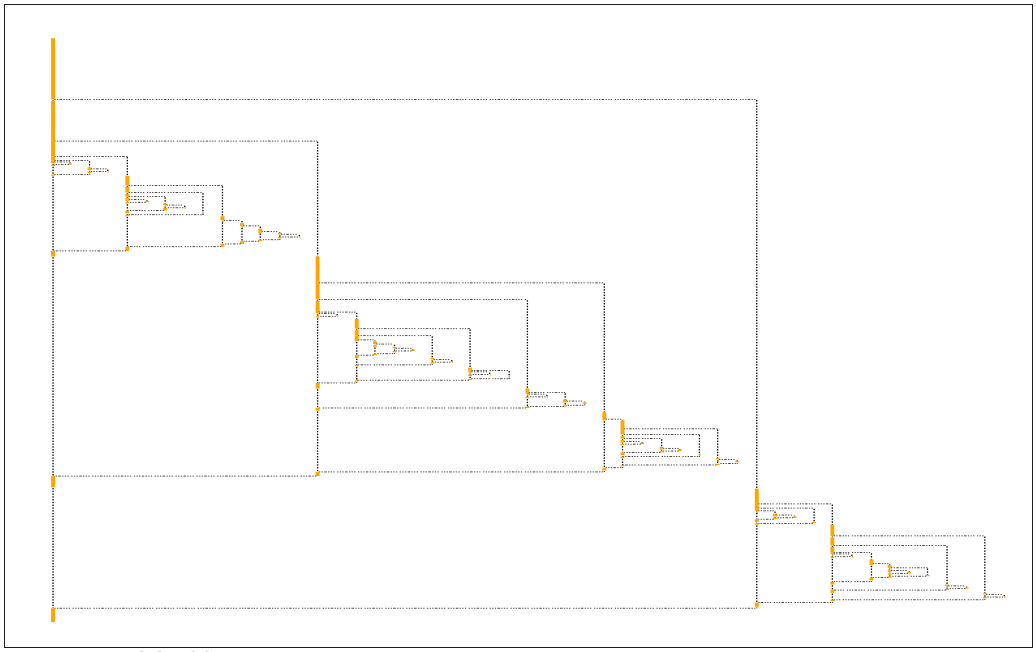
- Extensions / optimizations:
  - ◇ DASWAM / DDAS System (dependent and-//) [She92, She96]
  - ◇ &ACE, ACE Systems (or-, and-, dep-//) [PG95a, GHPSC94, PGPF97]

&-Prolog Run-time System: Performance



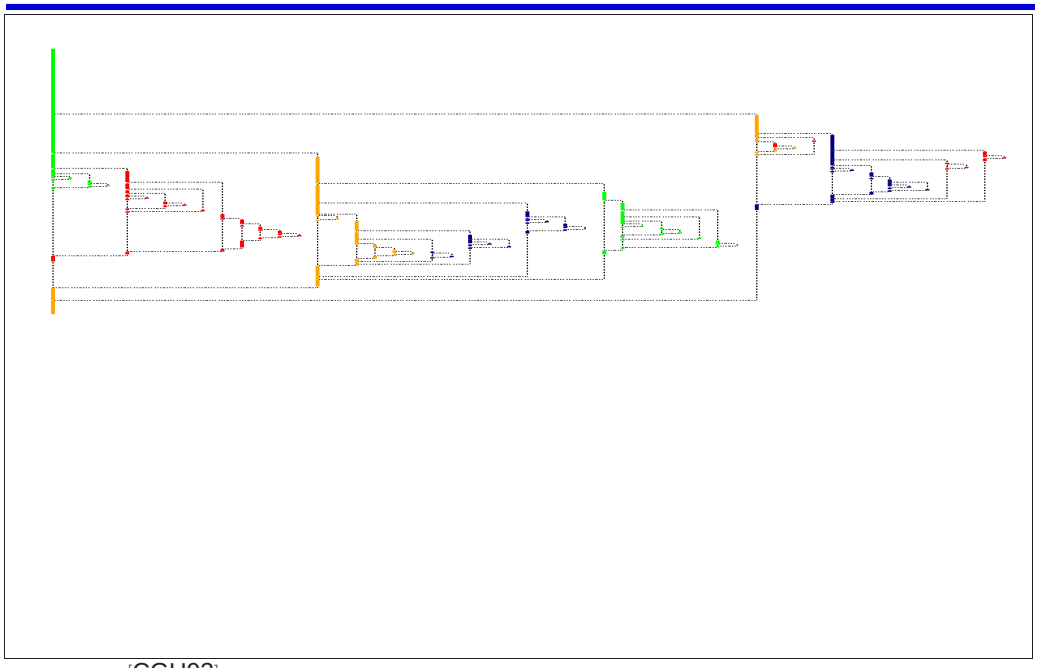
Sequent Symmetry, **hand parallelized** programs.
(Speedup over state of the art sequential systems.)

Visualization of And-parallelism – (small) qsort, 1 processor



(VisAndOr [CGH93] output.)

Visualization of And-parallelism – (small) qsort, 4 processors



(VisAndOr [CGH93] output.)

Independence – Strict Independence (Contd.)

- Not always possible to determine locally/statically:

```
main :- t(X,Y),       p(X), q(Y).

main :- read([X,Y]), p(X), q(Y).
```

- Alternatives: run-time independence tests, global analysis, ...

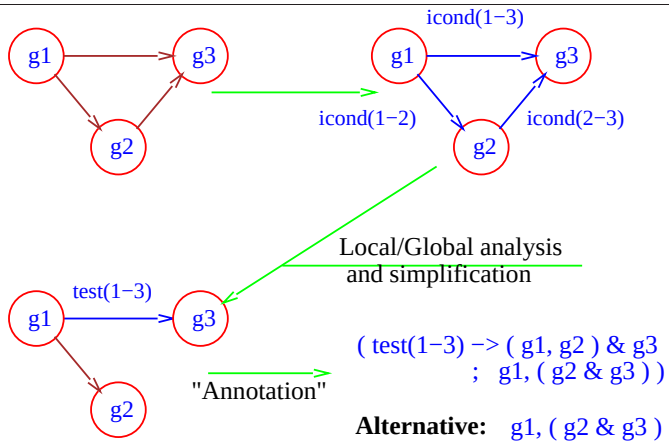
```
main :- read([X,Y]), ( indep(X,Y) -> p(X) & q(Y)
                                  ;  p(X) , q(Y) ).

main :- t(X,Y), p(X) & q(Y).       %% (After analysis)
```

Parallelization Process: CDG-based Automatic Parallelization

- Conditional Dependency Graph (of some code segment) [HW87, BdlBH99, GPA+01]:
  - ◇ Vertices: possible tasks (statements, calls, bidings, etc.).
  - ◇ Edges: possible dependencies (labels: conditions needed for independence).
- Local or global analysis used to reduce/remove checks in the edges.
- Annotation process converts graph back to parallel expressions in source.

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



Simplifying Independence Conditions (Strict Ind.)

[BdlBH99]

- Recall that b_1 and b_2 are strictly independent for θ iff

$$\text{vars}(b_1\theta) \cap \text{vars}(b_2\theta) = \emptyset$$

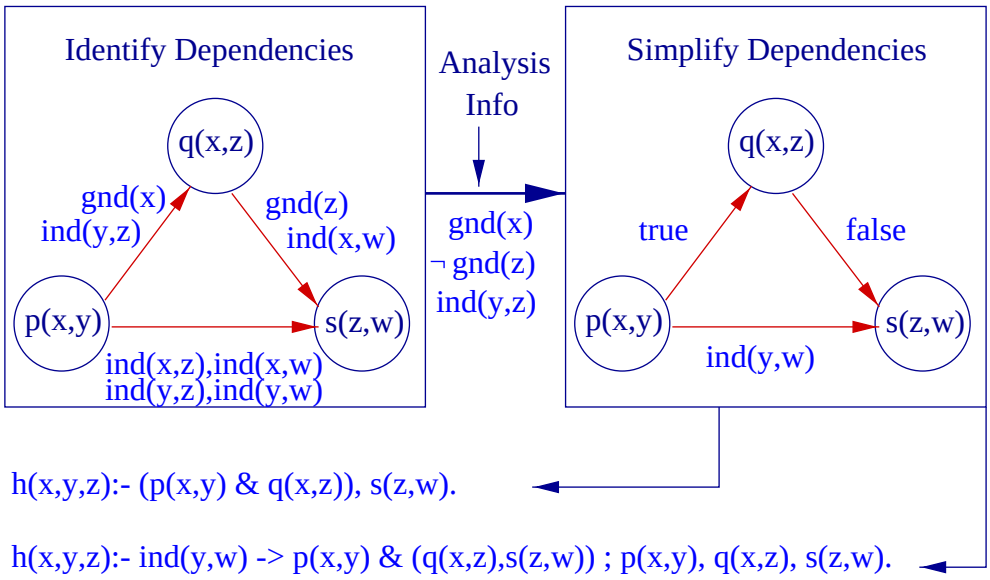
- $indep(b_1, b_2)$ iff b_1 and b_2 do not share variables at run–time.
- $p(x, y)$ and $q(y, z)$ are strictly independent at run–time iff $indep(\{x, y\}, \{y, z\})$.
- Equivalent to $\{indep(x, y), indep(x, z), indep(y, y), indep(y, z)\}$.
- Domain of interpretation DI: subset of propositional logic.
- For clause C, it contains predicates of the form $ground(x)$ and $indep(y, z)$, $\{x, y, z\} \subseteq vars(C)$, with axioms:

$$\{ground(x) \rightarrow indep(x, y) | \{x, y\} \subseteq vars(C)\}$$
$$\{indep(x, x) \rightarrow ground(x) | x \in vars(C)\}$$

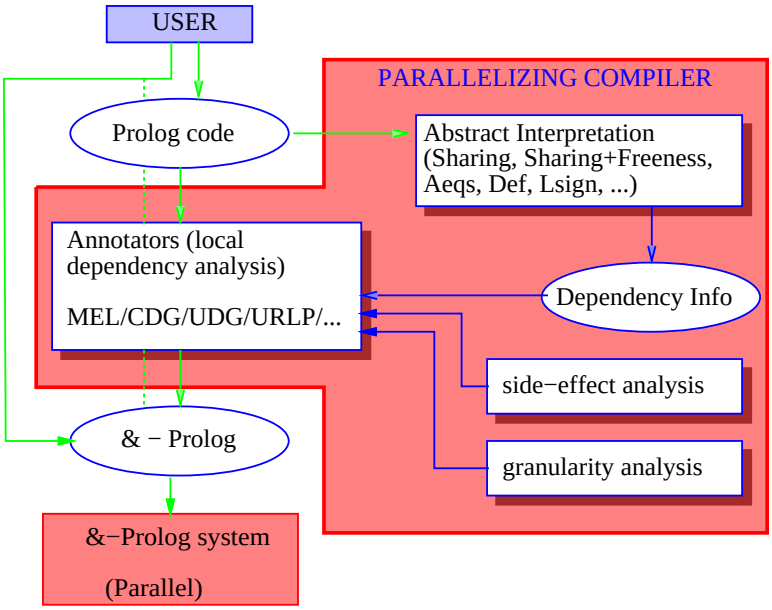
- The set $\{indep(x, y), indep(x, z), indep(y, y), indep(y, z)\}$ can be simplified to $\{ground(y), indep(x, z)\}$.

Simplifying Independence Conditions (Strict Ind.)

[BdlBH99]



&-Prolog/Ciao Parallelizer Overview



&-Prolog/CIAO compiler overview (Contd.)

Parallelizing compiler [HW87] (now integrated in CiaoPP [HBPLG99, HPBLG03]):

- **Global Analysis**: infers independence information.
- **Annotator(s)**: Prolog → &-Prolog parallelization [DeG87, MH90, BdlBH94a, CH94, PGPF97, MBdlBH99].
  - MEL: Maximum Expression Length —simple heuristic.
  - CDG: Conditional Graph Expressions —graph partitioning of clauses.
  - UDG: Unconditional Graph Expressions.
  - URLP: Uncond. Recursive Linear Parallelizer —recursive application of simple rules.
  - Variants of CDG/UDG.
  - Enhanced to better use global analysis info and granularity information (still on–going).
- **Low-level PWAM compiler**: extension of Sicstus V0.5
- **Granularity Analysis**: determines task size or size functions [DLH90, DL91, DL93, DLGHL94, DLGHL97, DLGH97, SCK98, MLGCH08].
- **Granularity Control**: restricts parallelism based on task sizes [DLH90, LGHD96, SCK98].
- **Other modules**: *side effect analyzer* (sequencing of side-effects, coded in &-Prolog), *multiple specializer / partial evaluator*, *invariant eliminator*, etc.

&-Prolog compilation: examples - I

```
multiply([],_,[]).
multiply([V0|V0s],V1,[Vr|Vrs]) :-
    vmul(V0,V1,Vr),
    multiply(V0s,V1,Vrs).

vmul([],[],0).
vmul([H1|T1],[H2|T2],Vr) :-
    scalar_mult(H1,H2,H1xH2),
    vmul(T1,T2,T1xT2),
    Vr is H1xH2+T1xT2.

scalar_mult(H1,H2,H1xH2) :- H1xH2 is H1*H2.
```

Source (Prolog)

&-Prolog compilation: examples - II

```
multiply([],_,[]).
multiply([V0|V0s],V1,[Vr|Vrs]) :-
    (  ground([V1]), indep([[V0,V0s],[V0,Vrs],[V0s,Vr],[Vr,Vrs]])
    -> vmul(V0,V1,Vr) & multiply(V0s,V1,Vrs)
    ;  vmul(V0,V1,Vr),  multiply(V0s,V1,Vrs) ).

vmul([],[],0).
vmul([H1|T1],[H2|T2],Vr) :-
    (  indep([[H1,T1],[H1,T2],[T1,H2],[H2,T2]])
    -> scalar_mult(H1,H2,H1xH2) & vmul(T1,T2,T1xT2)
    ;  scalar_mult(H1,H2,H1xH2),  vmul(T1,T2,T1xT2) ),
    Vr is H1xH2+T1xT2.

scalar_mult(H1,H2,H1xH2) :- H1xH2 is H1*H2.
```

Parallelized program (&-Prolog/Ciao)—no global analysis

Dependency Analysis: Global Analysis Subsystem

- "PLAI" analyzer – top-down driven bottom up analysis [MH89, MH92] (enhanced version of Bruynooghe's scheme [Bru91]).
- Optimized fixpoint algorithm (keeps track of dependencies and approximation state of information, avoids recomputation) [MH89, HPMS00, PH96].
- Some useful abstract domains:
  - Sharing Domain Abstraction ("S") [JL89, MH89, JL92, MH92].
  - Sharing+Freeness Domain Abstraction ("SF") [MH91].
  - Sondergaard's $ASub$ (linearity) domain ("P") [Søn86, MS93]
  - Type domains, depth-K, etc.
  - (Constraints:) Definiteness [dlBH93, AMSS94], Freeness [dlBHB+96], LSign [KMM+96] domains.
- Domains combined using [CMB+95] framework: e.g. $ASub$+SH, $ASub$+ShF
- Automatic elimination of repetitive checks [GH91, PH99].
- Current analyzer quite robust, with support for a relatively complete set of builtins.
- Support for full Prolog [BCHP96], CLP(R) [dlBH93, dlBHB+96], etc.

"Sharing" Abstraction (Groundness + Set Sharing)

- Definitions:
  - $Uvar$: universe of all variables,
  - $Pvar$: set of program variables in a clause,
  - $Subst$: set of all possible mappings from variables in $Pvar$ to terms.
- **Abstract Domain:** $D_\alpha = \wp(\wp(Pvar))$
- **Abstraction of a substitution**:
  $\alpha(A) : Subst \rightarrow D_\alpha$
  $\alpha(\theta) = \{Occ(\theta, U) | U \in Uvar\}$ where $Occ(\theta, U) = \{X | X \in dom(\theta) \wedge U \in var(X\theta)\}$,
- **Example**: Let $\theta = \{W = a, X = f(A_1, A_2), Y = g(A_2), Z = A_3\}$.
  $\alpha(\theta) = \{\emptyset, \{X\}, \{X, Y\}, \{Z\}\}$.
- Note that
  - $independent(x\theta, y\theta) \iff \nexists v \in Uvar, x \in Occ(\theta, v) \wedge y \in Occ(\theta, v)$

  Other additional axioms are encoded in the sharing patterns.

&-Prolog compilation: examples - III

```
:- entry multiply(g,g,f).

multiply([],_,[]).
multiply([V0|V0s],V1,[Vr|Vrs]) :-   % [[Vr],[Vr,Vrs],[Vrs]]
    multiply(V0s,V1,Vrs),           % [[Vr]]
    vmul(V0,V1,Vr).                 % []

vmul([],[],0).
vmul([H1|T1],[H2|T2],Vr) :-         % [[Vr],[H1xH2],[T1xT2]]
    scalar_mult(H1,H2,H1xH2),       % [[Vr],[T1xT2]]
    vmul(T1,T2,T1xT2),              % [[Vr]]
    Vr is H1xH2+T1xT2.              % []

scalar_mult(H1,H2,H1xH2) :-         % [[H1xH2]]
    H1xH2 is H1*H2.                 % []
```

Sharing information inferred by the analyzer

&-Prolog compilation: examples - III

```
multiply([],_,[]).
multiply([V0|V0s],V1,[Vr|Vrs]) :-
    ( indep([[Vr,Vrs]]) ->
        multiply(V0s,V1,Vrs) &
        vmul(V0,V1,Vr)
    ;
        multiply(V0s,V1,Vrs),
        vmul(V0,V1,Vr) ).

vmul([],[],0).
vmul([H1|T1],[H2|T2],Vr) :-
    scalar_mult(H1,H2,H1xH2) &
    vmul(T1,T2,T1xT2),
    Vr is H1xH2+T1xT2.

scalar_mult(H1,H2,H1xH2) :- H1xH2 is H1*H2.
```

. . . and the parallelized program with this information.

Sharing + Freeness Domain

- Allows detecting failure of groundness checks.
- Increases accuracy of sharing information.
- **Abstract Domain:** $D_\alpha = D_{\alpha-sharing} \times D_{\alpha-freeness}$
  - $D_{\alpha-sharing} = \wp(\wp(Pvar))$
  - $D_{\alpha-freeness} = \wp(Pvar)$
- **Abstraction (freeness) of a substitution**:
  $\alpha_{freeness}(\theta) = \{\ X \mid X \in dom(\theta), \exists Y \in Uvar\ (X\theta = Y)\}$
- Example:
  $\theta = \{W/P, X/f(P,Q), Y/g(Q,R), Z/f(a)\}$.
  $\alpha(\{\theta\}) = (\lambda_{sharing}, \lambda_{freeness})$, where
  - $\lambda_{sharing} = \{\emptyset, \{Y\}, \{W,X\}, \{X,Y\}\}$
  - $\lambda_{freeness} = \{W\}$

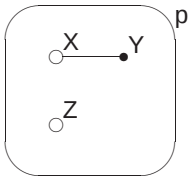
The Sharing+Freeness Abstract Domain – A Pictorial Representation

[CH94]

- Two components: sharing & freeness $(\hat{\theta}_{\mathrm{SH}}, \hat{\theta}_{\mathrm{FR}})$
- The freeness information restricts the possible combinations of sharing patterns.
- Pictorial representation:

p(X,Y,Z)

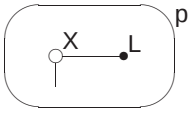
$\hat{\theta}_{\mathrm{SH}} = [[\mathrm{XY}]]$
$\hat{\theta}_{\mathrm{FR}} = [\mathrm{Y}]$



X = f(Y)
Z = b

p(X,L)

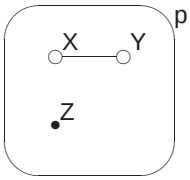
$\hat{\theta}_{\mathrm{SH}} = [[\mathrm{X}][\mathrm{XL}]]$
$\hat{\theta}_{\mathrm{FR}} = [\mathrm{L}]$



X = [Y|L]

p(X,Y,Z)

$\hat{\theta}_{\mathrm{SH}} = [[\mathrm{XY}][\mathrm{Z}]]$
$\hat{\theta}_{\mathrm{FR}} = [\mathrm{Z}]$



X = f(A)
Y = f(A)

&-Prolog compilation: examples - IV

```
:- entry multiply(g,g,f).

multiply([],_,[]).
multiply([V0|V0s],V1,[Vr|Vrs]) :-   % [[Vr],[Vrs]],[Vr,Vrs]
    multiply(V0s,V1,Vrs),           % [[Vr]],[Vr]
    vmul(V0,V1,Vr).                 % [],[]

vmul([],[],0).
vmul([H1|T1],[H2|T2],Vr) :-         % [[Vr],[H1xH2],[T1xT2]],
                                    %          [Vr,H1xH2,T1xT2]
    scalar_mult(H1,H2,H1xH2),       % [[Vr],[T1xT2]],[Vr,T1xT2]
    vmul(T1,T2,T1xT2),              % [[Vr]],[Vr]
    Vr is H1xH2+T1xT2.              % [],[]

scalar_mult(H1,H2,H1xH2) :-         % [[H1xH2]],[H1xH2]
    H1xH2 is H1*H2.                 % [],[]
```

Sharing+Freeness information inferred by the analyzer

&-Prolog compilation: examples - IV

```
multiply([],_,[]).
multiply([V0|V0s],V1,[Vr|Vrs]) :-
    multiply(V0s,V1,Vrs) &
    vmul(V0,V1,Vr).

vmul([],[],0).
vmul([H1|T1],[H2|T2],Vr) :-
    scalar_mult(H1,H2,H1xH2) &
    vmul(T1,T2,T1xT2),
    Vr is H1xH2+T1xT2.

scalar_mult(H1,H2,H1xH2) :- H1xH2 is H1*H2.
```

. . . and the parallelized program with this information.

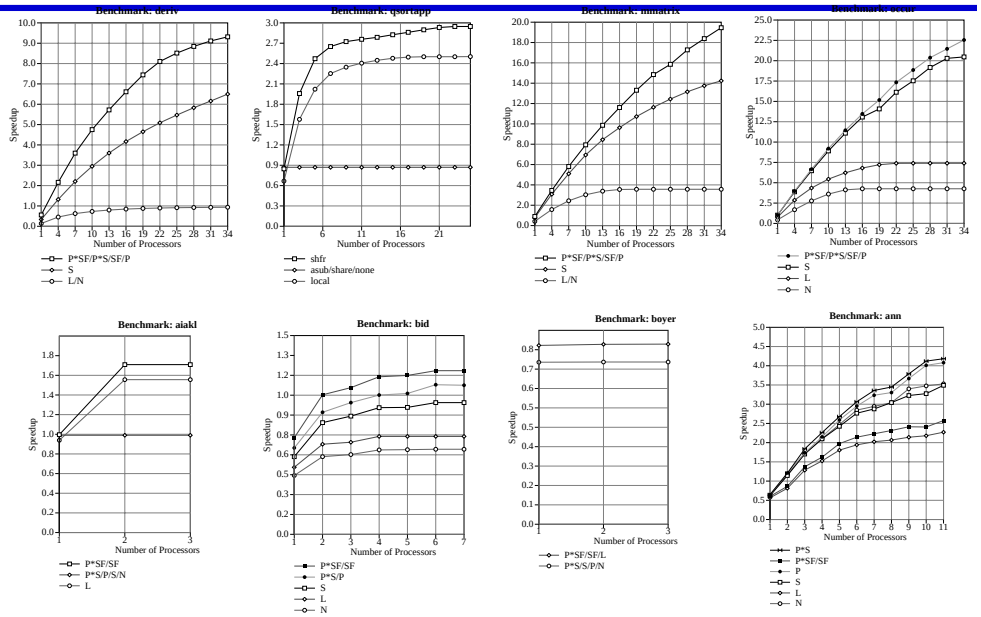
Efficiency of the analyzers — Seconds ('94 numbers!)

Program	Average time in seconds					
	Prol.	S	P	SF	P*S	P*SF
aiakl	0.17	0.20	0.43	0.22	0.32	0.37
ann	1.76	19.40	5.54	10.50	16.37	17.68
bid	0.46	0.32	0.27	0.36	0.46	0.56
boyer	1.12	3.56	1.38	4.17	2.91	3.65
browse	0.38	0.13	0.17	0.15	0.21	0.24
deriv	0.21	0.06	0.05	0.07	0.09	0.11
fib	0.03	0.01	0.01	0.02	0.02	0.02
hanoiapp	0.11	0.03	0.03	0.04	0.06	0.07
mmatrix	0.07	0.03	0.03	0.03	0.04	0.05
occur	0.34	0.04	0.03	0.05	0.06	0.07
peephole	1.36	5.45	2.54	3.94	7.00	7.45
qplan	1.68	1.54	11.52	1.84	2.60	3.36
qsortapp	0.08	0.04	0.05	0.05	0.08	0.09
read	1.07	2.09	1.89	2.35	2.99	3.51
serialize	0.20	2.26	0.23	0.62	0.52	0.67
tak	0.04	0.02	0.02	0.02	0.02	0.04
warplan	0.80	15.71	5.02	8.71	15.74	17.68
witt	1.86	1.98	16.24	2.26	2.87	3.42

Prol.	Standard Prolog compiler time
S	(Set) Sharing
P	Pair sharing (Sondergaard)
SF	Sharing + Freeness
X*Y	*Combinations*

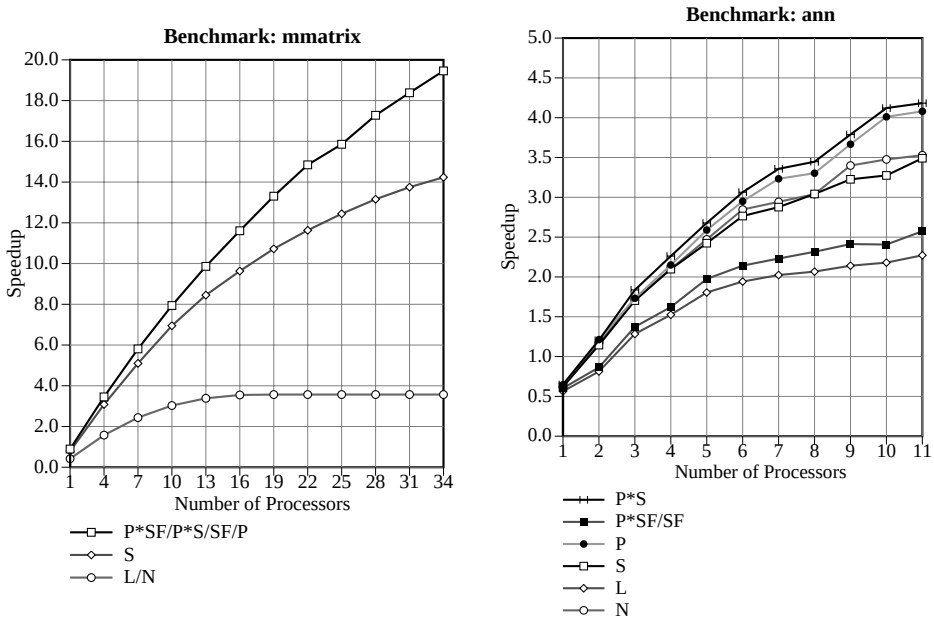
[BdlBH94b, MBdlBH99, BdlBH99]

Dynamic tests ('96 numbers!)



(1-10 processors actual speedups on Sequent Symmetry; 10+ projections using IDRA simulator on execution traces) [BdlBH94b, MBdlBH99, BdlBH99]

A Closer Look at Some Speedups



Simple matrix mul. (> 12 simulated)

The parallelizer, self-parallelized

Independence – Non-Strict Independence

[HR90, HR95, dlB94]

- Pure goals: only one thread "touches" each shared variable. Example:

```
main :- t(X,Y), p(X), q(Y).

t(X,Y) :- Y = f(X).
```

  `p` is independent of `t` (but `p` and `q` are dependent).

- Impure goals: only rightmost "touches" each shared variable. Example:

```
main :- t(X,Y), p(X), q(Y).

t(X,Y) :- Y = a.    p(X) :- var(X), ..., X=b, ...
```

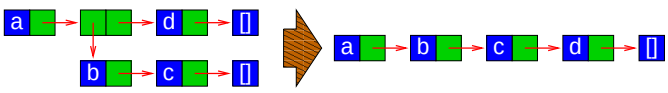
- More parallelism.
- But cannot be detected "a-priori:" requires global analysis.

Independence – Non-Strict Independence

- Very important in programs using "incomplete structures."

```
flatten(Xs,Ys) :- flatten(Xs,Ys,[]).

flatten([], Xs, Xs).
flatten([X|Xs],Ys,Zs)  :- flatten(X,Ys,Ys1), flatten(Xs,Ys1,Zs).
flatten(X, [X|Xs], Xs) :- atomic(X), X \== [].
```



- Another example:

```
qsort([],S,S).
qsort([X|Xs],S,S2) :-
        partition(Xs,X,L,R),
        qsort(L,S,[X|S1]),
        qsort(R,S1,S2).
```

Conditions for Non-Strict Independence Based on ShFr Info

[CH94]

- We consider the parallelization of pairs of goals.
- Let the situation be: $\{\widehat{\beta}\}\ p\ \{\widehat{\psi}\}\ \ldots\ q$.
  We define:

$$\mathrm{S}(p) = \{L \in \widehat{\beta}_{\mathrm{SH}} \mid L \cap \mathrm{var}(p) \neq \emptyset\}$$
$$\mathrm{SH} = \mathrm{S}(p) \cap \mathrm{S}(q) = \{L \in \widehat{\beta}_{\mathrm{SH}} \mid L \cap \mathrm{var}(p) \neq \emptyset \wedge L \cap \mathrm{var}(q) \neq \emptyset\}$$

- Conditions for non-strict independence for p and q:

$$\mathrm{C1}\ \ \forall L \in \mathrm{SH}\ L \cap \widehat{\psi}_{\mathrm{FR}} \neq \emptyset$$
$$\mathrm{C2}\ \ \neg\,(\exists N_1 \ldots N_k \in \mathrm{S}(p)\ \ \exists L \in \widehat{\psi}_{\mathrm{SH}}\ \ L = \cup_{i=1}^{k} N_i\ \wedge\ N_1, N_2 \in \mathrm{SH}\ \wedge \forall i, j\ \ 1 \leq i < j \leq k\ \ N_i \cap N_j \cap \widehat{\beta}_{\mathrm{FR}} = \emptyset)$$

- C1: preserves freeness of shared variables.
- C2: preserves independence of shared variables.
- More relaxed conditions if information re. partial answers and purity of goals.

Run-Time Checks for NSI Based on ShFr Info

- Run-time checks can be automatically included to ensure NSI when the previous conditions do not hold.
- The method uses analysis information.
- Possible checks are:
  - `ground(X)`: `X` is ground.
  - `allvars(X,`$\mathcal{F}$`)`: every free variable in `X` is in the list $\mathcal{F}$.
  - `indep(X,Y)`: `X` and `Y` do not share variables.
  - `sharedvars(X,Y,`$\mathcal{F}$`)`: every free variable shared by X and Y is in the list $\mathcal{F}$.
- The method generalizes the techniques previously proposed for detection of SI.
- Even when only SI is present, the tests generated may be better than the traditional tests.

Experimental Results

Speedups of five programs that have NSI but no SI:

1. **array2list** translates an extendible array into a list of index–element pairs.
2. **flatten** flattens a list of lists of any complexity into a plain list.
3. **hanoi_dl** solves the towers of Hanoi problem using difference lists.
4. **qsort** is the sorting algorithm quicksort using difference lists.
5. **sparse** transforms a binary matrix into an optimized notation for sparse matrices.

P	# of processors									
	1	2	3	4	5	6	7	8	9	10
1	0.78	1.54	2.34	3.09	3.82	4.64	5.41	5.90	6.50	7.22
2	0.54	1.07	1.61	2.07	2.52	3.05	3.62	4.14	4.46	4.83
3	0.56	1.13	1.68	2.25	2.73	3.23	3.70	4.34	4.84	5.25
4	0.91	1.65	2.20	2.53	2.75	2.86	3.00	3.14	3.30	3.33
5	0.99	1.92	2.79	3.68	4.50	5.06	5.78	6.75	8.10	8.26

Independence – Constraint Independence

[dlBHM93, dlBHM00]

- Standard Herbrand notions do not carry over to general constraint systems.

```
main :- Y > X, Z > X, p(Y) & q(Z), ...
main :- Y > X, X > Z, p(Y) & q(Z), ...
```

- General notion [91-94]: “all constraints posed by second thread are consistent with the output constraints of the first thread.” (Better also for Herbrand!)
- Sufficient a-priori condition: given $g_1(\bar{x})$ and $g_2(\bar{y})$:
  $$(\bar{x} \cap \bar{y} \subseteq def(c)) \; and \; (\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow \exists_{-\bar{y} \cup \bar{x}} c)$$
  ($def(c)$ is the set of variables constrained to a unique value in c)
- For $c = \{y > x, z > x\}$ $\quad \bar{\exists}_{-\{y\}} c = \bar{\exists}_{-\{z\}} c = \bar{\exists}_{-\{y,z\}} c = true$
  For $c = \{y > x, x > z\}$ $\quad \bar{\exists}_{-\{y\}} c = \bar{\exists}_{-\{z\}} c = true, \quad \bar{\exists}_{\{y,z\}} c = y > z$
- Approximation: presence of “links” through the store.
- Run-time checks: $def(X)$, $indep(X, Y)$, $unlinked(X, Y)$

Some Preliminary CLP &-Parallelization Results (Compiler)

[dlBBH96]

- Parallel expressions:

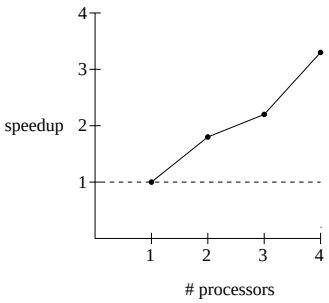
Bench. Program	Total CGEs			Uncond. CGEs		
	Def	Free	FD	Def	Free	FD
amp	5	–	5	0	–	0
bridge	0	–	0	0	–	0
circuit	3	2	2	0	0	0
dnf	14	14	14	12	0	12
laplace	1	–	1	1	–	1
mining	5	4	4	1	0	2
mmatrix	2	2	2	0	0	0
mg_extend	0	0	0	0	0	0
num	16	16	16	5	10	10
pic	4	3	3	0	0	0
power	5	5	5	1	1	1
runge_kutta	2	1	1	0	0	0
trapezoid	1	1	1	0	0	0

Some Preliminary CLP &-Parallelization Results (Compiler)

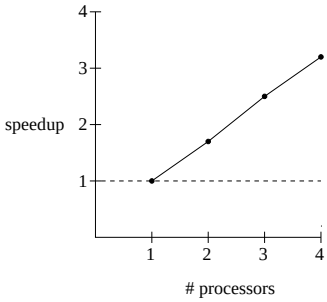
- Conditional checks:

Bench. Program	Conditions: def/unlinked		
	Def	Free	FD
amp	1/10	–	1/10
bridge	0/0	–	0/0
circuit	1/5	0/10	0/3
dnf	0/2	0/30	0/2
laplace	0/0	–	0/0
mining	3/5	5/5	2/4
mmatrix	0/2	2/8	0/2
mg_extend	0/0	0/0	0/0
num	0/24	0/20	0/19
pic	2/9	6/8	1/3
power	3/40	3/29	3/29
runge_kutta	5/0	6/0	3/0
trapezoid	0/9	0/9	0/9

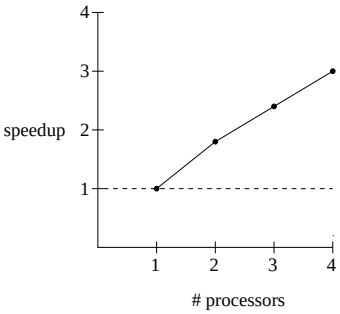
Some Preliminary CLP &-Speedup Results (Run-time System)



Speedups for `mmatrix`



Speedups for `critical` with `go2` input



Speedups for `critical` with `go3` input

Some Preliminary CLP &-Parallelization Results (Summary)

1. Tests on LP programs:
   - Analysis: compares well to LP-specific domains, but worse relative precision (except $Def\ x\ Free$).
   - Annotation:
     - Efficiency shows the relative precision of the information.
     - Effectiveness comparable for $Def\ x\ Free$. Def and $Free$ alone less precise.
2. Tests on CLP programs:
   - Analysis: acceptable, but comparatively more expensive than for LP.
   - Annotation:
     - Efficiency in the same ratio to analysis as for LP.
     - Effectiveness: $Def\ x\ Free$ comparably more effective that Def and $Free$ alone. But still less satisfactory than for LP.
     - Key: none are specific purpose domains.
   - Still, useful speedups.
3. Generalization for LP/CLP with dynamic scheduling and CC [G.Banda Ph.D.].

Other Forms of Independence

- Seen so far:
  - ◇ Strict independence / Non-strict independence / Constraint independence
- Independence in CLP + delay [dlBHM96], and non-deter. CC [BHMR94, BHMR98].
- Determinacy also a form of independence (e.g., Andorra, AKL, EAM –see later).
  - ◇ If/when goals are deterministic they are independent (no-slowdown).
  - ◇ If also non-failing then also no speculation (extra work).

  Determinacy actually subsumed by non-strict/search space preserv. definitions!
- Inconsistency-based independence ("local independence"): finest granularity level, subsumes previous ones [BHMR94, BHMR98].
- Independence can be applied dynamically and at finer grain levels (e.g., "Local Independence", DDAS model, AKL stability, etc.) [HC94]
  Some levels of granularity at which independence is applied:
  - ◇ Goal level / Binding level / Unification level / Across procedures / Etc.

  $\longrightarrow$ "No such thing as *dependent* and-parallelism."

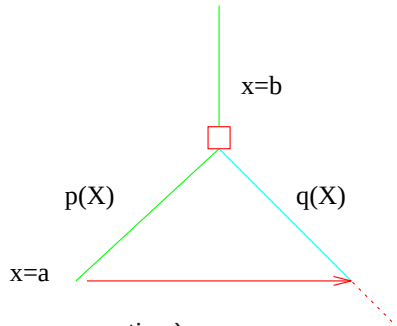
Dealing with Speculation

- Computations can be speculative (or even non-terminating!):

```
foo(X) :- X=b, ..., p(X) & q(X), ...
foo(X) :- X=a, ...

p(X) :- ..., X=a, ...

q(X) :- large computation.
```



  but "no slow-down" guaranteed if
  - ◇ left-biased scheduling,
  - ◇ instantaneous killing of siblings (failure propagation).
- Left biased schedulers, dynamic throttling of speculative tasks,non-failure, etc. [HR89, HR95, dlB94].
- Static detection of non-failure [BCMH94, DLGH97]:
  avoids speculativeness / guarantees theoretical speedup.
  $\rightarrow$ *importance of non-failure analysis*.

Dealing with Overheads, Irregularity

- Independence not enough:
  overheads (task creation and scheduling, communication, etc.)
- In CLP compounded by the fact that the number and size of tasks is highly irregular and dependent on run-time parameters.
- Dynamic solutions:
  - Minimize task management and data communication overheads (micro tasks, shared heaps, compile-time elimination of locks, ...)
  - Efficient dynamic task allocation (e.g., non-centralized task stealing)
- Quite good results for shared-memory multiprocessors early on (e.g., Sequent Balance 1986-89).
- Not sufficient for clusters or over a network.

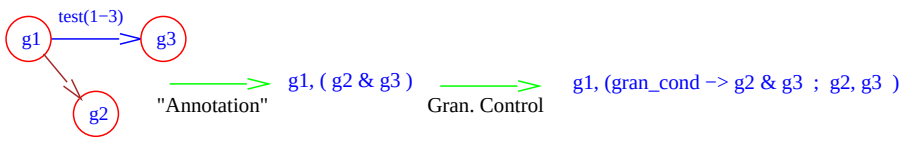
Dealing with Overheads, Irregularity: Granularity Control

[DLH90, DL91, DL93, LGHD94, DLGHL94, LGHD96, DLGHL97, DLGH97, SCK98, MLGCH08]

- Replace parallel execution with sequential execution (or vice-versa) based on bounds (or estimations) on task size and overheads.
- Cannot be done completely at compile-time: cost often depends on input (hard to approximate at compile time, even w/abstract interpretation).

```
main :- read(X), read(Z), inc_all(X,Y) & r(Z,M), ...
inc_all([])     := [].
inc_all([I|Is]) := [ I+1 | ~inc_all(Is) ].
```

- Our approach:
  - Derive at compile-time cost *functions* (to be evaluated at run-time) that efficiently bound task size (lower, upper *bounds*).
  - Transform programs to carry out run-time granularity control.



Granularity Control Example

- For the previous example:

```
main :- read(X), read(Z), inc_all(X,Y) & r(Z,M), ...
inc_all([])     := [].
inc_all([I|Is]) := [ I+1 | ~inc_all(Is) ].
```

- Assume `X` determined to be input, `Y` output, cost function inferred $2 * length(X) + 1$, threshold 100 units:

```
main :- read(X), read(Z), (2*length(X)+1 > 100 -> inc_all(X,Y) & r(Z,M)
                                                ;  inc_all(X,Y) , r(Z,M) ), .
```

- Provably correct techniques (thanks to abstract interpretation): can *ensure speedup* if assumptions hold.
- Issues: derivation of data measures, data size functions, task cost functions, program transformations, optimizations...

Inference of Bounds on Argument Sizes and Procedure Cost in `CiaoPP`

1. Perform type/mode inference:
   `:- true inc_all(X,Y) : list(X,int), var(Y) => list(Y,int).`
2. Infer size measures: list length.
3. Use data dependency graphs to determine the relative sizes of structures that variables point to at different program points – infer argument size relations:

   $\mathtt{Size}^2_{\mathtt{inc\_all}}(0) = 0$ (boundary condition from base case),
   $\mathtt{Size}^2_{\mathtt{inc\_all}}(n) = 1 + \mathtt{Size}^2_{\mathtt{inc\_all}}(n-1)$.

   Sol = $\mathtt{Size}^2_{\mathtt{inc\_all}}(n) = n$.
4. Use this, set up recurrence equations for the computational cost of procedures:

   $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(0) = 1$ (boundary condition from base case),
   $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n) = 2 + \mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n-1)$.

   Sol = $\mathtt{Cost}^{\mathtt{L}}_{\mathtt{inc\_all}}(n) = 2\,n + 1$.

- We obtain lower/upper bounds on task granularities.
- Non-failure (absence of exceptions) analysis needed for lower bounds.

Granularity Control: Some Refinements/Optimizations (1)

- Simplification of cost functions:

```
..., ( length(X) > 50 -> inc_all(X,Y) & r(Z,M)
                       ;  inc_all(X,Y) , r(Z,M) ), ...

..., ( length_gt(LX,50) -> inc_all(X,Y) & r(Z,M)
                         ;  inc_all(X,Y) , r(Z,M) ), ...
```

- Complex thresholds: use also communication cost functions, load, ...
  Example: Assume $CommCost(inc\_all(X)) = 0.1\ (length(X) + length(Y))$.
  We know $ub\_length(Y)$ (actually, exact size) $= length(X)$; thus:

$$2\ length(X) + 1 > 0.1\ (length(X) + length(X)) \cong$$
$$2\ length(X) > 0.2\ length(X) \equiv$$
$$2 > 0.2$$

  Guaranteed speedup for any data size! $\Leftarrow$

  $\Rightarrow$ Sometimes static decisions can be made despite dynamic sizes and costs (e.g., when ratios are independent of input).

Granularity Control: Some Refinements/Optimizations (1)

- Static task clustering (loop unrolling / data parallelism):

```
..., ( has_more_elements_than(X,5) -> inc_all_2(X,Y) & r(X)
                                    ;  inc_all_2(X,Y),  r(X) ), ...

inc_all([X1,X2,X3,X4,X5|R) := [X1+1,X2+1,X3+1,X4+1,X5+1 | ~inc_all(R)].
inc_all([]) := [].
```

  (actually, cases for 4, 3, 2, and 1 elements also have to be included); this is also useful to achieve *fast task startup* [BB93, DJ94, HC95, HC96, ?, PG95b].

- Sometimes static decisions can be made despite dynamic sizes and costs (e.g., when the ratios are independent of input).
- Data size computations can often be done on-the-fly.
- Static placement.

Granularity Control System Output Example

```
g_qsort([], []).
g_qsort([First|L1], L2) :-
  partition3o4o(First, L1, Ls, Lg, Size_Ls, Size_Lg),
  Size_Ls > 20 -> (Size_Lg > 20 -> g_qsort(Ls, Ls2) & g_qsort(Lg, Lg2)
                                ;  g_qsort(Ls, Ls2),  s_qsort(Lg, Lg2))
              ;  (Size_Lg > 20 -> s_qsort(Ls, Ls2),  g_qsort(Lg, Lg2)
                                ;  s_qsort(Ls, Ls2),  s_qsort(Lg, Lg2))),
  append(Ls2, [First|Lg2], L2).

partition3o4o(F, [], [], [], 0, 0).
partition3o4o(F, [X|Y], [X|Y1], Y2, SL, SG) :-
  X =< F, partition3o4o(F, Y, Y1, Y2, SL1, SG), SL is SL1 + 1.
partition3o4o(F, [X|Y], Y1, [X|Y2], SL, SG) :-
   X > F, partition3o4o(F, Y, Y1, Y2, SL, SG1), SG is SG1 + 1.
```


Granularity Control: Experimental Results

- Shared memory:

programs	seq. prog.	no gran.ctl	gran.ctl	gc.stopping	gc.argsize
fib(19)	1.839	0.729 1	1.169 -60%	0.819 -12%	0.549 +24%
hanoi(13)	6.309	2.509 1	2.829 -12.8%	2.399 +4.4%	2.399 +4.4%
unbmatrix	2.099	1.009 1	1.339 -32.71%	0.870 +13.78%	0.870 +13.78%
qsort(1000)	3.670	1.399 1	1.790 -28%	1.659 -19%	1.409 -0.0%

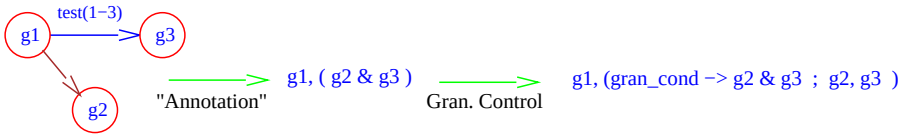
- Cluster:

programs	seq. prog.	no gran.ctl	gran.ctl	gc.stopping	gc.argsize
fib(19)	1.839	0.970 1	1.389 -43%	1.009 -4.0%	0.639 +34%
hanoi(13)	6.309	2.690 1	2.839 -5.5%	2.419 +10.1%	2.419 +10.1%
unbmatrix	2.099	1.039 1	1.349 -29.84%	0.870 +16.27%	0.870 +16.27%
qsort(1000)	3.670	1.819 1	2.009 -11%	1.649 +9.3%	1.429 +21%

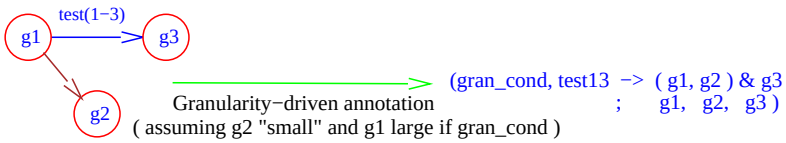
Refinements (2): Granularity-Aware Annotation

[Cas08]

- With classic annotators (MEL, UDG, CDG, . . .) we applied granularity control after parallelization:

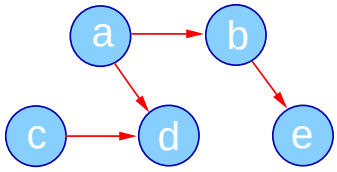


- Developed new annotation algorithm that takes task granularity into account:
  - Annotation is a heuristic process (several alternatives possible).
  - Taking task granularity into account during annotation can help make better choices and speed up annotation process.
  - Tasks with larger cost bounds given priority, small ones not parallelized.



Granularity-Aware Annotation: Concrete Example

- Consider the clause: p `:-` a, b, c, d, e.
- Assume that the dependencies detected between the subgoals of `p` are given by:



- Assume also that:

$$T(a) < T(c) < T(e) < T(b) < T(d),$$

where $T(i) < T(j)$ means: cost of subgoal `i` is smaller than the cost of `j`.

MEL annotator:	`( a, b & c, d & e)`
UDG annotator:	`( c & ( a, b, e ), d )`
Granularity-aware:	`( a, c, ( b & d ), e )`

Refinements (3): Using Execution Time Bounds/Estimates

[MLGCH08]

- Use estimations/bounds on *execution time* for controlling granularity (instead of steps/reductions).
- Execution time generally dependent on platform characteristics ($\approx$ constants) and input data sizes (unknowns).
- Platform-dependent, one-time calibration using fixed set of programs:
  - ◇ Obtains value of the platform-dependent constants (costs of basic operations).
- Platform-independent, compile-time analysis:
  - ◇ Infers cost functions (using modification of previous method), which return count of *basic operations* given input data sizes.
  - ◇ Incorporate the constants from the calibration.

  $\longrightarrow$ we obtain functions yielding *execution times* depending on size of input.
- Predicts execution times with *reasonable* accuracy (challenging!).
- Improving by taking into account lower level factors (current work).

Execution Time Estimation: Concrete Example

- Consider `nrev` with mode:

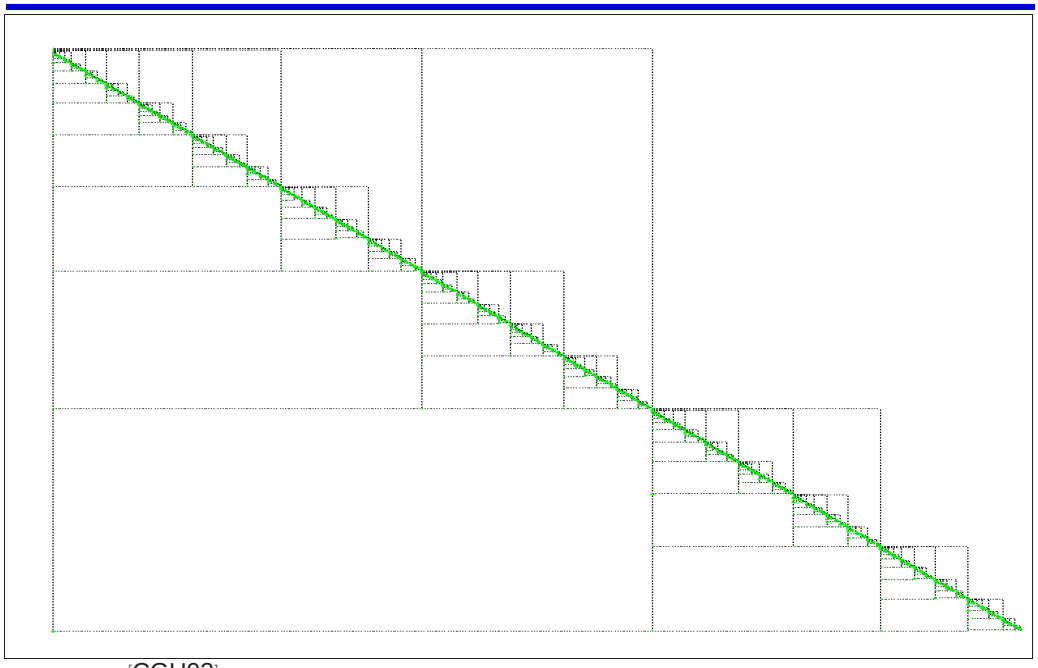
```
:- pred nrev/2 :  list(int) * var.
```

- Estimation of execution time for a concrete input —consider:

`A = [1,2,3,4,5]`, $\overline{n}$ `= length(A) = 5`

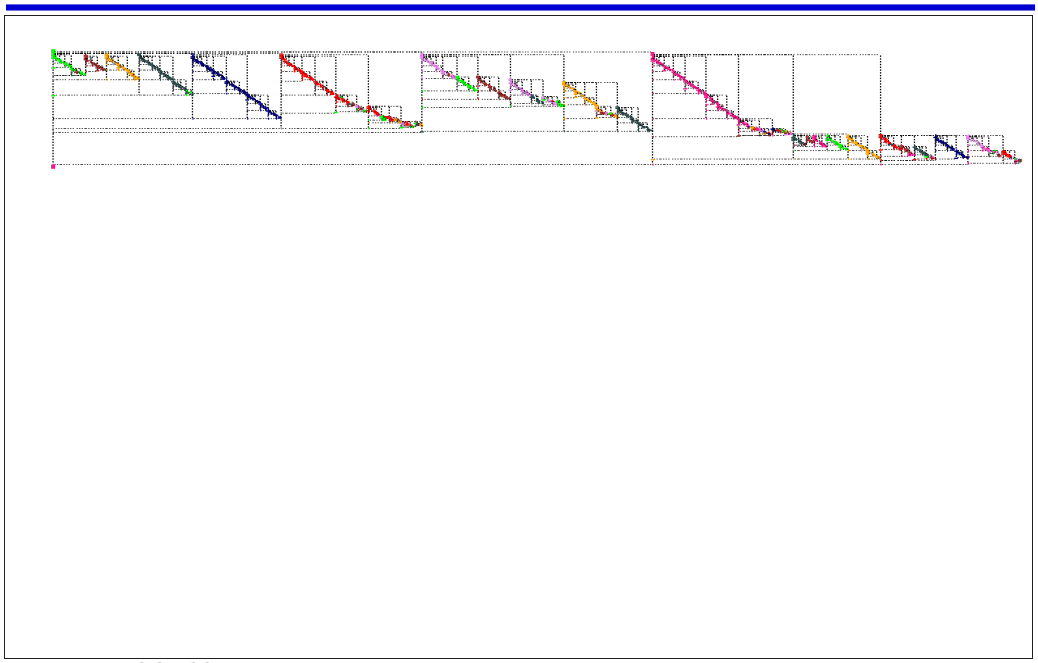
component	Once K_{ω_i}	Static Analysis $\texttt{Cost}_\texttt{p}(I(\omega_i),\overline{n}) = C_i(\overline{n})$	Application $C_i(5)$	$K_{\omega_i} \times C_i(5)$
step	21.27	$0.5 \times n^2 + 1.5 \times n + 1$	21	446.7
nargs	9.96	$1.5 \times n^2 + 3.5 \times n + 2$	57	567.7
giunif	10.30	$0.5 \times n^2 + 3.5 \times n + 1$	31	319.3
gounif	8.23	$0.5 \times n^2 + 0.5 \times n + 1$	16	131.7
viunif	6.46	$1.5 \times n^2 + 1.5 \times n + 1$	45	290.7
vounif	5.69	$n^2 + n$	30	170.7
Execution time $\overline{K_\Omega} \bullet \overline{\texttt{Cost}_\texttt{p}}(\overline{I(\Omega)},\overline{n})$:				1926.8

Fib 15, 1 processor



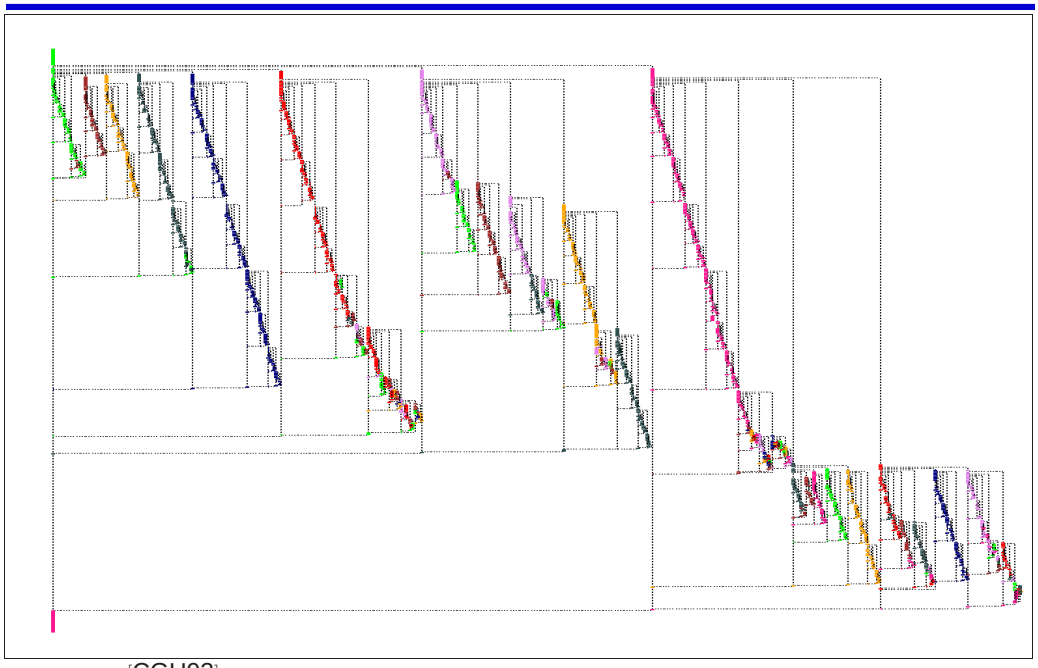
(VisAndOr [CGH93] output.)

Fib 15, 8 processors (same scale)



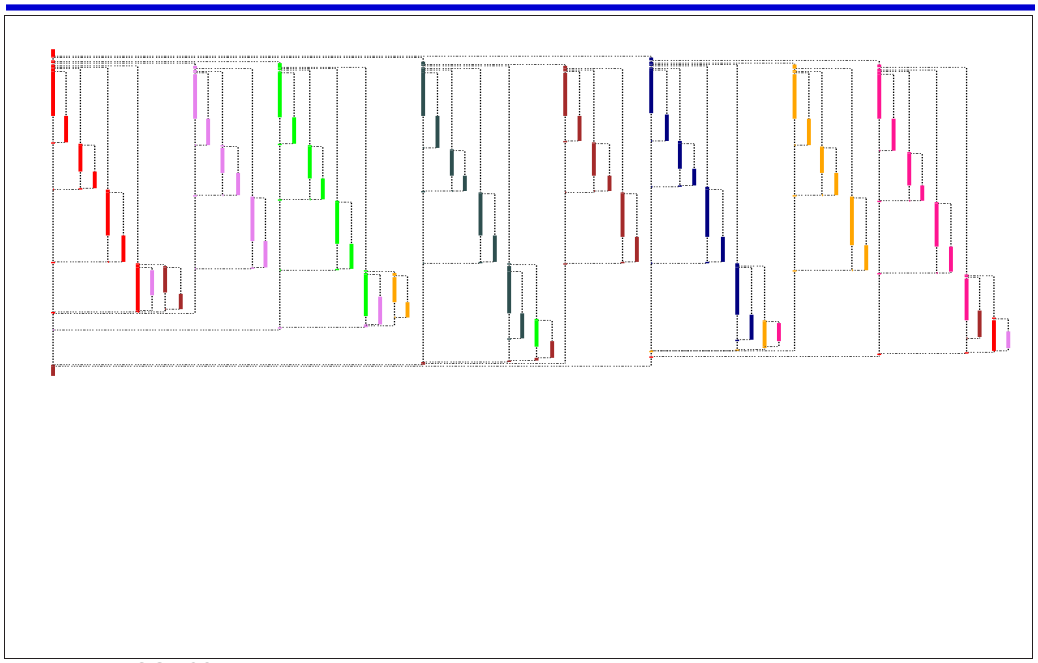
(VisAndOr [CGH93] output.)

Fib 15, 8 processors (full scale)



(VisAndOr [CGH93] output.)

Fib 15, 8 processors, with granularity control (same scale)



(VisAndOr [CGH93] output.)

Dependent And–parallelism: DDAS (I)

[She92, She96]

- Exploits Independent + "Dependent" And–parallelism.
- Goals communicate through shared variables.
- Shared variables are marked (`dep/1` annotation).
- Example:
```
example(X):- (dep(X) => a(X) & b(X)).
a(X).      b(1).
```
- To retain sequential search space: dependent variables are bound by only one producer and received by some consumers.
  - The producer can bind the variable.
  - A consumer suspends if it tries to bind the variable.
  - A suspended consumer is resumed if the variable on which it is suspended is bound *or if it becomes leftmost*.
  - Producer for a given variable changes dynamically as goals finish execution:
    "The producer for a dependent variable is the (lexicographically) leftmost **active** task which has access to that variable."

Dependent And–parallelism: DDAS (II)

- Performance:
  - IAP speedups + new dependent-and speedups
  - IAP programs with one agent run at about 50% speed w.r.t. sequential execution (due to locking and other overheads).
  - DAP programs run at 30%–40% lower speed.

Andorra

- Basic Andorra model [D.H.D.Warren]: goals for which at most one clause matches should be executed first (inspired by Naish's PNU-Prolog).
- If a solution exists, computation rule is complete and correct for pure programs (switching lemma). (But otherwise finite failures can become infinite failures.)
- Determinate reductions can proceed in parallel without the need of choice points $\longrightarrow$ no dependent backtracking needed.
- An implementation: Andorra–I [D.H.D. Warren, V.S. Costa, R. Yang, I. Dutra. . .]
  - Prolog support: preprocessor + engine (interpreter).
  - Exploits both and- and or-parallelism. (Good speedups in practice)
  - Problem: no nondeterministic steps can proceed in parallel.
- "Extended" Andorra Model [Warren] – add independent and-parallelism.
  - With implicit control (unspecified) [Warren, Gupta]
  - With explicit/implicit control: AKL [Janson, Haridi ILPS91]
    (implicit rule – "stability": non-deterministic steps can proceed if "they cannot affected" by other steps)

Non-restricted And-Parallelism

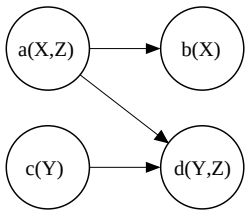
[CH96, Cab04]

- Classical parallelism operator `&/2`: nested fork-join.
- However, more flexible constructions can be used to denote (non-restricted) and-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.
- Optimized deterministic versions: `&!>/2`, `<&!/1`.
- Operator `&/2` can be written as:

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

Non-restricted And-Parallelism

- More parallelism can be exploited with these primitives.
- Take the sequential code below (dep. graph to the right) and three possible parallelizations:




```
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

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

Annotation algorithms for non-restricted &-par.: general idea

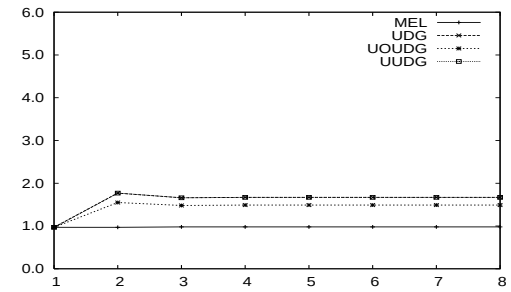
[CCH07]

- 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.

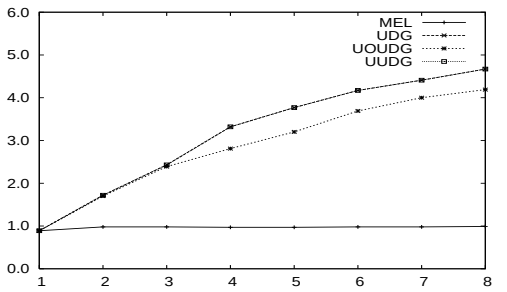
Performance Results – Speedups

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

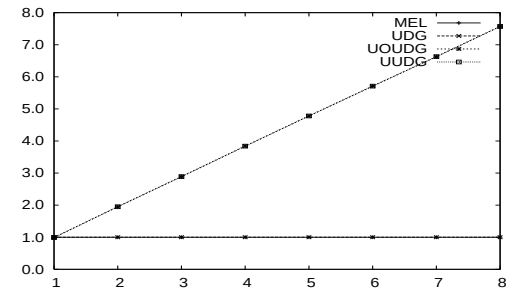
Performance results - Restricted vs. Unrestricted And-Parallelism



AIAKL

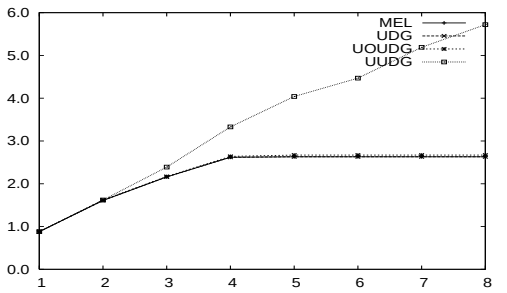


Hanoi



FibFun

Sun Fire T2000 - 8 cores



Takeuchi

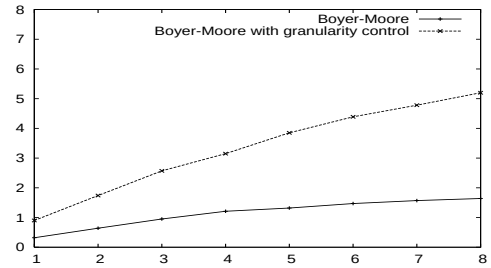
Towards a higher-level implementation

[CCH08b, CCH08a]

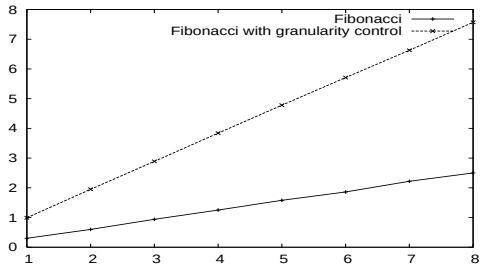
- Versions of and-parallelism previously implemented:
  &-Prolog, &-ACE, AKL, Andorra-I,...
  rely on complex low-level machinery. Each agent:
- Our objective: alternative, easier to maintain implementation approach.
- Fundamental idea: raise non-critical components to the source language level:
  - **Prolog-level**: goal publishing, goal searching, goal scheduling, “marker” creation (through choice-points),...
  - **C-level**: low-level threading, locking, untrailing,...

  → Simpler machinery and more flexibility.
  → Easily exploits unrestricted IAP.
- Current implementation (for shared-memory multiprocessors):
  - Each agent: sequential Prolog machine + goal list + (mostly) Prolog code.
- Recently added full parallel backtracking!

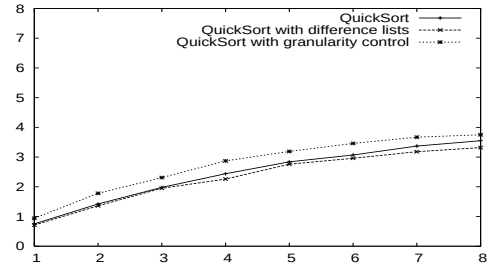
(Preliminary) performance results Sun Fire T2000 - 8 cores



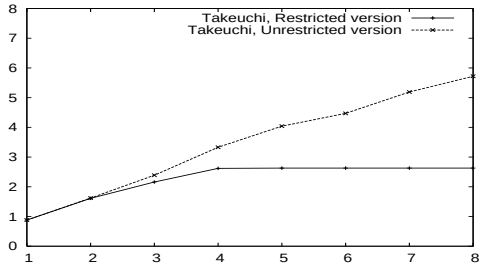
Boyer-Moore



Fibonacci



Quicksort



Takeuchi

And–parallel Execution Models: Summary (I)

- Different types of parallelism, with different costs associated:
  - Complexity considerations (search space, speculation).
  - Coordination cost for agreeing on unifiable bindings.
- Overheads / granularity control.
- Approaches:
  - IAP: goals do not restrict each other's search space.
    - Ensures no slow-down w.r.t. sequential execution.
    - Retains as much as possible WAM optimizations.
    - Some parallelism lost.
- NSIAP: IAP +. . .
  - At most one goal can bind to non-variable a shared variable (or they make *compatible bindings*) and no goal aliases shared variables.
  - Generalization: search space preservation.
  - Reduced to IAP via program analysis and transformation.

And–parallel Execution Models: Summary (II)

- DDAS: goals communicate bindings.
  - Incorporate a suspension mechanism to ensure no more work than in a sequential system – "fine grained independence".
  - Handle dependent backtracking.
  - Some locking and variable-management overhead.
- Andorra I: determinate depend. and– + or–parallelism
  - Dependent determinate goals run in parallel.
  - Allows incorporating also or–parallelism easily.
  - Some locking and goal-management overhead.
- Extended Andorra Model – adding independent and parallelism to Andorra-I.
  - With implicit control.
  - With explicit control: AKL.

Other developments

- ACE: combining MUSE and &-Prolog (And/or Copy-based Execution model) [Being developed by New Mexico S.U. and UPM]

  ngc-recomputation dep-compiler
- Interesting work on memory management [Pontelli ICLP'95].
- Visualization Tools (VisiPAL, ViMust, VisAndOr, Vista, etc.)
  [?, ?, ?, VPG97, FIVC98, Tic92]
- Fine-grained compile-time parallelization ("local indep" [Bueno et al 1994])
- Distributed systems:
  - Significant progress made (e.g. UCM work [Araujo et. al] and Ciao).
  - Vital component: granularity control.
- Ciao: Concurrent Constraint Independent And/Or-Parallel System ['92-present]
  - Non-deterministic concurrent constraint language.
  - Subsumes Prolog, CLP, CC (+Andorra via transformation), ...

  - Distributed / net execution.
- Most Prolog systems have a notion of threads nowadays (SICStus, Ciao, SWI, Yap, XSB, B-Prolog,), adequate for hand-coding *coarse-grain parallelism*.

Some comparison with work in other paradigms

- Much progress (e.g., in FORTRAN) for regular computations. But comparatively less on:
  - parallelization across procedure calls,
  - irregular computations,
  - complex data structures / pointers,
  - speculation, etc.

Wrap-up: (C)LP strong points

- Several generations of parallelizing compilers for LP and CLP [85-...]:
  - Good compilation speed, proved correct and efficient.
  - Speedups over state-of-the-art sequential systems on applications.
  - Good demonstrators of abstract interpretation as data-flow analysis technique.
  - Now including granularity control.

  Improved on hand parallelizations on several large applications.
- Areas of particularly good progress:
  - Concepts of independence (pointers, search/speculation, constraints...).
  - Inter-procedural analysis (dynamic data, recursion, pointers/aliasing, etc.).
  - Parallelization algorithms for conditional dependency graphs.
  - Dealing with irregularity:
    - efficient task representation and fast dynamic scheduling,
    - static inference of task cost functions – granularity control.
  - Mixed static/dynamic parallelization techniques.

Wrap-up: areas for improvement

- Weaker areas / shortcomings:
  - In general, weak in detecting independence in structure traversals based on integer arithmetic (modeled as recursions over recursive data structures to fit parallelizer).
  - Weaker partitioning / placement for regular computations and static data structures.
  - Little work on mutating data structures (e.g., single assignment transformations).
- The objective is to perform *all* these tasks well also!
- Opportunities for synergy.
- A final plug for constraint programming:
  - Merges elegantly the symbolic and the numerical worlds.
  - We believe many of the features of CLP will make it slowly into mainstream languages (e.g., ILOG, ALMA, and other recent proposals).

Some general-purpose contributions from (C)LP

- Some examples so far:
  - Stealing-based scheduling strategies and microthreading.
  - Cactus-like stack memory management techniques.
  - Abstract interpretation-based static dependency analysis.
  - Sharing (aliasing) analyses, Shape analyses, ...
  - Parallelization (“annotation”) algorithms.
  - Cost analysis-based granularity control.
  - Logic variable-based synchronization.
  - Determinacy-based parallelization.
  - ...

Some challenges?

- Parallelism not yet exploited on an everyday basis (real system, real applications).
- Some challenges:
  - Scalability of techniques (from analysis to scheduling).
  - Maintainability of the systems: simplification?
    - Move as much as possible to source level?
      (And explore this same route with many other things –e.g., tabling)
  - Better automatic parallelization:
    - Better granularity control (e.g., time-based).
    - Better granularity-aware annotators.
    - Full scalability of analysis (modular analysis, etc.).
    - Automate program transformations (e.g., loop unrollings).
  - Supporting multiple types of parallism easily is still a challenge.
  - A really elegant (and implementable) concurrent language which includes non-determinism.
  - Combination w/low-level optimization and other features (r.g., or-// YapTab).

Some Bibliography (for a general tutorial see [GPA+01])

[AK90] K. A. M. Ali and R. Karlsson. Full Prolog and Scheduling Or-parallelism in Muse. *International Journal of Parallel Programming*, 19(6):445–475, 1990.

[AMSS94] T. Armstrong, K. Marriott, P. Schachte, and H. Søndergaard. Boolean functions for dependency analysis: Algebraic properties and efficient representation. In Springer-Verlag, editor, *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 266–280, Namur, Belgium, September 1994.

[BB93] Jonas Barklund and Johan Bevemyr. Executing bounded quantifications on shared memory multiprocessors. In Jaan Penjam, editor, *Proc. Intl. Conf. on Programming Language Implementation and Logic Programming 1993*, LNCS 714, pages 302–317, Berlin, 1993. Springer-Verlag.

[BCC+06] F. Bueno, D. Cabeza, M. Carro, M. Hermenegildo, P. López-García, and G. Puebla (Eds.). The Ciao System. Ref. Manual (v1.13). Technical report, C. S. School (UPM), 2006. Available at `http://www.ciaohome.org`.

[BCHP96] F. Bueno, D. Cabeza, M. Hermenegildo, and G. Puebla. Global Analysis of Standard Prolog Programs. In *European Symposium on Programming*, number 1058 in LNCS, pages 108–124, Sweden, April 1996. Springer-Verlag.

[BCMH94] C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of prolog. In *Proc. International Symposium on Logic Programming*, pages 457–471, Ithaca, NY, November 1994. MIT Press.

[BdlBH94a] F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 63–73. World Scientific Publishing Company, September 1994.

[BdlBH94b] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.

[BdlBH99] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Abstract Interpretation in Automatic Parallelization: A Case Study in Logic Programming. *ACM Transactions on Programming Languages and Systems*, 21(2):189–238, March 1999.

[BHMR94] F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. From Eventual to Atomic and Locally Atomic CC Programs: A Concurrent Semantics. In *Fourth International Conference on Algebraic and Logic Programming*, number 850 in LNCS, pages 114–132. Springer-Verlag, September 1994.

[BHMR98] F. Bueno, M. Hermenegildo, U. Montanari, and F. Rossi. Partial Order and Contextual Net Semantics for Atomic and Locally Atomic CC Programs. *Science of Computer Programming*, 30:51–82, January 1998. Special CCP95 Workshop issue.

[Bru91] M. Bruynooghe. A Practical Framework for the Abstract Interpretation of Logic Programs. *Journal of Logic Programming*, 10:91–124, 1991.

[BW93] T. Beaumont and D.H.D. Warren. Scheduling Speculative Work in Or-Parallel Prolog Systems. In *Proceedings of the 10th International Conference on Logic Programming*, pages 135–149. MIT Press, June 1993.

[Cab04] D. Cabeza. *An Extensible, Global Analysis Friendly Logic Programming System*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, August 2004.

[Cas08] A. Casas. *Automatic Unrestricted Independent And-Parallelism in Declarative Multiparadigm Languages*. PhD thesis, University of New Mexico (UNM), Electrical and Computer Engineering Department, University of New Mexico, Albuquerque, NM 87131-0001 (USA), September 2008.

[CCH07] A. Casas, M. Carro, and M. Hermenegildo. Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs. In *17th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'07)*, number 4915 in LNCS, pages 138–153, The Technical University of Denmark, August 2007. Springer-Verlag.

[CCH08a] A. Casas, M. Carro, and M. Hermenegildo. A High-Level Implementation of Non-Deterministic, Unrestricted, Independent And-Parallelism. In M. García de la Banda and E. Pontelli, editors, *24th International Conference on Logic Programming (ICLP'08)*, LNCS. Springer-Verlag, December 2008.

[CCH08b] A. Casas, M. Carro, and M. Hermenegildo. Towards a High-Level Implementation of Execution Primitives for Non-restricted, Independent And-parallelism. In D.S. Warren and P. Hudak, editors, *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, volume 4902 of *LNCS*, pages 230–247. Springer-Verlag, January 2008.

[CDD85] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225. IEEE Computer Society, February 1985.

[CDO88] M. Carlsson, K. Danhof, and R. Overbeek. A Simplified Approach to the Implementation of And-Parallelism in an Or-Parallel Environment. In *Fifth International Conference and Symposium on Logic Programming*, pages 1565–1577. MIT Press, August 1988.

[CGH93] M. Carro, L. Gómez, and M. Hermenegildo. Some Paradigms for Visualizing Parallel Execution of Logic Programs. In *1993 International Conference on Logic Programming*, pages 184–201. MIT Press, June 1993.

[CH94] D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.

[CH96] D. Cabeza and M. Hermenegildo. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint conference on Declarative Programming*, pages 67–78, San Sebastian, Spain, July 1996. U. of the Basque Country. Available from `http://www.cliplab.org/`.

[Cie92] A. Ciepielewski. Scheduling in or-parallel prolog systems: Survey and open problems. *International Journal of Parallel Programming*, 20(6):421–451, 1992.

[Clo87] William Clocksin. Principles of the delphi parallel inference machine. *Computer Journal*, 30(5), 1987.

[CMB+95] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. *ACM Transactions on Programming Languages and Systems*, 17(1):28–44, January 1995.

[Con83] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[CSW88] J. Chassin, J. Syre, and H. Westphal. Implementation of a Parallel Prolog System on a Commercial Multiprocessor. In *Proceedings of Ecai*, pages 278–283, August 1988.

[DeG84] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

[DeG87] D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Int'l Supercomputing Conference*, pages 80–89, Athens, 1987. Springer Verlag.

[DJ94] S. Debray and M. Jain. A Simple Program Transformation for Parallelism. In *1994 International Symposium on Logic Programming*, pages 305–319. MIT Press, November 1994.

[DL91] S. K. Debray and N.-W. Lin. Automatic complexity analysis for logic programs. In *Eighth International Conference on Logic Programming*, pages 599–613, Paris, France, June (1991). MIT Press.

[DL93] S. K. Debray and N. W. Lin. Cost Analysis of Logic Programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.

[dlB94] M. García de la Banda. *Independence, Global Analysis, and Parallelism in Dynamically Scheduled Constraint Logic Programming*. PhD thesis, Universidad Politécnica de Madrid (UPM), Facultad Informatica UPM, 28660-Boadilla del Monte, Madrid-Spain, September 1994.

[dlBBH96] M. García de la Banda, F. Bueno, and M. Hermenegildo. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs*, number 1140 in LNCS, pages 77–91, Aachen, Germany, September 1996. Springer-Verlag.

[dlBH93] M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 437–455. MIT Press, October 1993.

[dlBHB+96] M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, September 1996.

[dlBHM93] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.

[dlBHM96] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in dynamically scheduled logic languages. In *1996 International Conference on Algebraic and Logic Programming*, number 1139 in LNCS, pages 47–61. Springer-Verlag, September 1996.

[dlBHM00] M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in CLP Languages. *ACM Transactions on Programming Languages and Systems*, 22(2):269–339, March 2000.

[DLGH97] S.K. Debray, P. López-García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, pages 48–62, Cambridge, MA, June 1997. MIT Press, Cambridge, MA.

[DLGHL94] S.K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Estimating the Computational Cost of Logic Programs. In *Static Analysis Symposium, SAS'94*, number 864 in LNCS, pages 255–265, Namur, Belgium, September 1994. Springer-Verlag.

[DLGHL97] S. K. Debray, P. López-García, M. Hermenegildo, and N.-W. Lin. Lower Bound Cost Estimation for Logic Programs. In *1997 International Logic Programming Symposium*, pages 291–305. MIT Press, Cambridge, MA, October 1997.

[DLH90] S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.

[ECR93] ECRC. *Eclipse User's Guide*. European Computer Research Center, 1993.

[FCH96] M. Fernández, M. Carro, and M. Hermenegildo. IDRA (IDeal Resource Allocation): Computing Ideal Speedups in Parallel Logic Programming. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 724–734. Springer-Verlag, August 1996.

[FIVC98] N. Fonseca, I.C.Dutra, and V.Santos Costa. VisAll: A Universal Tool to Visualise Parallel Execution of Logic Programs. In J. Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 100–114. MIT Press, 1998.

[GH91] F. Giannotti and M. Hermenegildo. A Technique for Recursive Invariance Detection and Selective Program Specialization. In *Proc. 3rd. Int'l Symposium on Programming Language Implementation and Logic Programming*, number 528 in LNCS, pages 323–335. Springer-Verlag, August 1991.

[GHPSC94] G. Gupta, M. Hermenegildo, E. Pontelli, and V. Santos-Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *International Conference on Logic Programming*, pages 93–110. MIT Press, June 1994.

[GJ93] G. Gupta and B. Jayaraman. Analysis of or-parallel execution models. *ACM Transactions on Programming Languages and Systems*, 15(4):659–680, 1993.

[GPA+01] G. Gupta, E. Pontelli, K. Ali, M. Carlsson, and M. Hermenegildo. Parallel Execution of Prolog Programs: a Survey. *ACM Transactions on Programming Languages and Systems*, 23(4):472–602, July 2001.

[HBC+99] M. Hermenegildo, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López-García, and G. Puebla. The CIAO Multi-Dialect Compiler and System: An Experimentation Workbench for Future (C)LP Systems. In *Parallelism and Implementation of Logic and Constraint Logic Programming*, pages 65–85. Nova Science, Commack, NY, USA, April 1999.

[HBC+08] M. V. Hermenegildo, F. Bueno, M. Carro, P. López, J.F. Morales, and G. Puebla. An Overview of The Ciao Multiparadigm Language and Program Development Environment and its Design Philosophy. In Jose Meseguer Pierpaolo Degano, Rocco De Nicola, editor, *Festschrift for Ugo Montanari*, number 5065 in LNCS, pages 209–237. Springer-Verlag, June 2008.

[HBPLG99] M. Hermenegildo, F. Bueno, G. Puebla, and P. López-García. Program Analysis, Debugging and Optimization Using the Ciao System Preprocessor. In *1999 Int'l. Conference on Logic Programming*, pages 52–66, Cambridge, MA, November 1999. MIT Press.

[HC94] M. Hermenegildo and The CLIP Group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.

[HC95] M. Hermenegildo and M. Carro. Relating Data–Parallelism and And–Parallelism in Logic Programs. In *Proceedings of EURO–PAR'95*, number 966 in LNCS, pages 27–42. Springer-Verlag, August 1995.

[HC96] M. Hermenegildo and M. Carro. Relating Data–Parallelism and (And–) Parallelism in Logic Programs. *The Computer Languages Journal*, 22(2/3):143–163, July 1996.

[Her86a] M. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel*. PhD thesis, Dept. of Electrical and Computer Engineering (Dept. of Computer Science TR-86-20), University of Texas at Austin, Austin, Texas 78712, August 1986.

[Her86b] M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.

[Her87] M. Hermenegildo. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Fourth International Conference on Logic Programming*, pages 556–575. University of Melbourne, MIT Press, May 1987.

[Her97] M. Hermenegildo. Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming. In *Proceedings of EUROPAR'97*, volume 1300 of *LNCS*, pages 31–46. Springer-Verlag, August 1997.

[Her00] M. Hermenegildo. Parallelizing Irregular and Pointer-Based Computations Automatically: Perspectives from Logic and Constraint Programming. *Parallel Computing*, 26(13–14):1685–1708, December 2000.

[HPBLG03] M. Hermenegildo, G. Puebla, F. Bueno, and P. López-García. Program Development Using Abstract Interpretation (and The Ciao System Preprocessor). In *10th International Static Analysis Symposium (SAS'03)*, number 2694 in LNCS, pages 127–152. Springer-Verlag, June 2003.

[HPMS00] M. Hermenegildo, G. Puebla, K. Marriott, and P. Stuckey. Incremental Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 22(2):187–223, March 2000.

[HR89] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[HR90] M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.

[HR95] M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.

[HW87] M. Hermenegildo and R. Warren. Designing a High-Performance Parallel Logic Programming System. *Computer Architecture News, Special Issue on Parallel Symbolic Programming*, 15(1):43–53, March 1987.

[JH90] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. Technical Report PEPMA Project, SICS, Box 1263, S-164 28 KISTA, Sweden, November 1990. Forthcoming.

[JH91] S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

[JL89] D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[JL92] D. Jacobs and A. Langen. Static Analysis of Logic Programs for Independent And-Parallelism. *Journal of Logic Programming*, 13(2 and 3):291–314, July 1992.

[KMM+96] A. Kelly, A. Macdonald, K. Marriott, P.J. Stuckey, and R.H.C. Yap. Effectiveness of optimizing compilation of CLP($\mathcal{R}$). In M.J. Maher, editor, *Logic Programming: Proceedings of the 1992 Joint International Conference and Symposium*, pages 37–51, Bonn, Germany, September 1996. MIT Press.

[LBD+88] E. Lusk, R. Butler, T. Disz, R. Olson, R. Stevens, D. H. D. Warren, A. Calderwood, P. Szeredi, P. Brand, M. Carlsson, A. Ciepielewski, B. Hausman, and S. Haridi. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2/3):243–271, 1988.

[LGHD94] P. López-García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In Hoon Hong, editor, *Proc. of First International Symposium on Parallel Symbolic Computation, PASCO'94*, pages 133–144. World Scientific, September 1994.

[LGHD96] P. López-García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 21(4–6):715–734, 1996.

[LK88] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.

[MBdlBH99] K. Muthukumar, F. Bueno, M. García de la Banda, and M. Hermenegildo. Automatic Compile-time Parallelization of Logic Programs for Restricted, Goal-level, Independent And-parallelism. *Journal of Logic Programming*, 38(2):165–218, February 1999.

[MH89] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.

[MH90] K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[MH91] K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.

[MH92] K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2/3):315–347, July 1992.

[MLGCH08] E. Mera, P. López-García, M. Carro, and M. Hermenegildo. Towards Execution Time Estimation in Abstract Machine-Based Languages. In *10th Int'l. ACM SIGPLAN Symposium on Principles and Practice of Declarative Programming (PPDP'08)*, pages 174–184. ACM Press, July 2008.

[MS93] K. Marriott and H. Søndergaard. Precise and efficient groundness analysis for logic programs. Technical report 93/7, Univ. of Melbourne, 1993.

[PG95a] E. Pontelli and G. Gupta. An Overview of the ACE Project. In *Proc. of Compulog ParImp Workshop*, 1995.

[PG95b] E. Pontelli and G. Gupta. Data And–Parallel Execution of Prolog Programs in ACE. In *IEEE Symposium on Parallel and Distributed Processing*, pages 424–431. IEEE Computer Society, 1995.

[PG98] E. Pontelli and G. Gupta. Efficient Backtracking in And-Parallel Implementations of Non-Deterministic Languages. In T. Lai, editor, *Proc. of the International Conference on Parallel Processing*, pages 338–345. IEEE Computer Society, Los Alamitos, CA, 1998.

[PGPF97] E. Pontelli, G. Gupta, F. Pulvirenti, and A. Ferro. Automatic Compile-time Parallelization of Prolog Programs for Dependent And-Parallelism. In L. Naish, editor, *Proc. of the Fourteenth International Conference on Logic Programming*, pages 108–122. MIT Press, July 1997.

[PH96] G. Puebla and M. Hermenegildo. Optimized Algorithms for the Incremental Analysis of Logic Programs. In *International Static Analysis Symposium*, number 1145 in LNCS, pages 270–284. Springer-Verlag, September 1996.

[PH99] G. Puebla and M. Hermenegildo. Abstract Multiple Specialization and its Application to Program Parallelization. *J. of Logic Programming. Special Issue on Synthesis, Transformation and Analysis of Logic Programs*, 41(2&3):279–316, November 1999.

[PK96] S. Prestwich and A. Kusalik. Programmer–Oriented Parallel Performance Visualizatoin. Technical Report TR–96–01, CS Dept., University of Saskatchewan, 1996.

[Pre93] Steven Prestwich. Improving granularity by program transformation. *ParForce* esprit project report d.wp.1.4.1.m1.2, CEC, July 1993.

[RSC99] Silva F. Rocha, R. and V. Santos Costa. Yapor: an or-parallel prolog system based on environment copying. In *Proceedings of EPPIA'99: The 9th Portuguese Conference on Artificial Intelligence*, 1999.

[SCK98] K. Shen, V. S. Costa, and A. King. Distance: a New Metric for Controlling Granularity for Parallel Execution. In Joxan Jaffar, editor, *Joint International Conference and Symposium on Logic Programming*, pages 85–99, Cambridge, MA, June 1998. MIT Press, Cambridge, MA.

[SH96] K. Shen and M. Hermenegildo. Flexible Scheduling for Non-Deterministic, And-parallel Execution of Logic Programs. In *Proceedings of EuroPar'96*, number 1124 in LNCS, pages 635–640. Springer-Verlag, August 1996.

[She92] K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Proc. Joint Int'l. Conf. and Symp. on Logic Prog.*, pages 717–731. MIT Press, 1992.

[She96] K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.

[Søn86] H. Søndergaard. An application of abstract interpretation of logic programs: occur check reduction. In *European Symposium on Programming, LNCS 123*, pages 327–338. Springer-Verlag, 1986.

[Tic92] Evan Tick. Visualizing Parallel Logic Programming with VISTA. In *International Conference on Fifth Generation Computer Systems*, pages 934–942. Tokio, ICOT, June 1992.

[Van89] P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. In G. Levi and M. Martelli, editors, *Sixth International Conference on Logic Programming*, pages 165–180, Lisbon, Portugal, June 1989. MIT Press.

[VPG97] R. Vaupel, E. Pontelli, and G. Gupta. Visualization of And/Or-Parallel Execution of Logic Programs. In *International Conference on Logic Programming*, Logic Programming, pages 271–285. MIT Press, July 1997.

[War90] D.H.D. Warren. The Extended Andorra Model with Implicit Control. In Sverker Jansson, editor, *Parallel Logic Programming Workshop*, Box 1263, S-163 13 Spanga, SWEDEN, June 1990. SICS.