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, Jesús Correas,
María José García de la Banda, Manuel Hermenegildo, Pedro López, Mario Méndez, Edison Mera, José 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** \(^{\text{Con83}}\): execute simultaneously different search space branches.
  - Present in general search problems, enumeration part of constr. problems, etc.
    
    ```prolog
    money(S,E,N,D,M,O,R,Y) :-
    digit(0),
    digit(S),
    digit(E),
    ...,
    carry(I),
    ...,
    N is E+0-10*I,
    digit(9),
    carry(0),
    carry(1).
    ```

- **And-parallelism** \(^{\text{Con83}}\): execute simultaneously different clause body goals.
  - Comprises traditional parallelism (parallel loops, divide and conquer, etc.).
  - *Concurrent languages* also generally based on and-parallelism.
    
    ```prolog
    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:
  \[
  g'_1 \leftarrow g'_{11}, \ldots, g'_{1m}, \\
  g''_1 \leftarrow g''_{11}, \ldots, g''_{1k}.
  \]

- 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), ...

\[ p_1(X) :- \ldots, X=a, \ldots, !, \ldots \]\n\[ p_2(X) :- \ldots, X=b, \ldots \]\n
main :- l, s.
\[ :- \text{parallel } l/0. \]\n\[ l :- \text{large\_work\_a}. \]
\[ l :- \text{large\_work\_b}. \]
\[ :- \text{parallel } s/0. \]\n\[ s :- \text{small\_work\_a}. \]
\[ s :- \text{small\_work\_b}. \]
**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 \cite{CGH93,FCH96}, ViMust, Parsee \cite{PK96}, VisAll \cite{FIVC98}, ...
  - Program transformation to increase granularity \cite{Pre93}.
  - Compile-time/run-time granularity control; automatically introduce parallel annotations \cite{LGHD96}.
Some Results in OR Parallelism

- Quite successful systems built (ECLIPSE, SICSTUS/MUSE, Aurora, OrpYap\textsuperscript{RSS99}, 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.

<table>
<thead>
<tr>
<th>Program</th>
<th>1</th>
<th>2</th>
<th>4</th>
<th>8</th>
<th>10</th>
<th>Sicstus 0.6</th>
</tr>
</thead>
<tbody>
<tr>
<td>parse1</td>
<td>1</td>
<td>1.8</td>
<td>2.8</td>
<td>2.93</td>
<td>2.76</td>
<td>1.25</td>
</tr>
<tr>
<td>parse5</td>
<td>1</td>
<td>1.97</td>
<td>3.74</td>
<td>6.92</td>
<td>7.72</td>
<td>1.27</td>
</tr>
<tr>
<td>db5</td>
<td>1</td>
<td>1.93</td>
<td>3.74</td>
<td>6.92</td>
<td>7.34</td>
<td>1.37</td>
</tr>
<tr>
<td>8queens</td>
<td>1</td>
<td>1.99</td>
<td>3.95</td>
<td>7.88</td>
<td>9.6</td>
<td>1.25</td>
</tr>
<tr>
<td>tina</td>
<td>1</td>
<td>2.07</td>
<td>4.06</td>
<td>7.81</td>
<td>9.59</td>
<td>1.43</td>
</tr>
</tbody>
</table>

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

- Easy to extend to CLP (e.g., VanHentenryck\textsuperscript{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 $\theta_1$ and $\theta_2$,
  - 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: $\theta \circ \eta(t) = \text{mgu}(E(\theta) \cup E(\eta))(t)$
  - Change parallel model.
  - Not an issue in CLP: conjunction instead of composition [GHM93] [GHM00].
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.)

<table>
<thead>
<tr>
<th></th>
<th>Imperative</th>
<th>Functions</th>
<th>Constraints</th>
</tr>
</thead>
<tbody>
<tr>
<td>( s_1 )</td>
<td>( Y := W+2; )</td>
<td>(+ W 2)</td>
<td>( Y = W+2, )</td>
</tr>
<tr>
<td>( s_2 )</td>
<td>( X := Y+Z; )</td>
<td>(+ Z)</td>
<td>( X = Y+Z, )</td>
</tr>
</tbody>
</table>

  *Running at \( s_2 \) “seeing \( s_1 \):”*

  For *Predicates* (multiple procedure definitions):

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td><code>main :-</code></td>
<td><code>p(X) :- X=a.</code></td>
<td></td>
</tr>
<tr>
<td>( s_1 )</td>
<td><code>q(X),</code></td>
<td><code>q(X) :- X=b, large computation.</code></td>
</tr>
<tr>
<td>( s_2 )</td>
<td><code>write(X).</code></td>
<td><code>q(X) :- X=a.</code></td>
</tr>
</tbody>
</table>

  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 [GHM93, Gar94].

- Detection of independence:
  - Run-time \textit{(a-priori conditions)} [Con83, LK88, JH91].
  - Compile-time [CDD85].
  - Mixed: conditional execution graph expressions [DeG84, Her86b]. (1)
  - User control: explicit parallelism (concurrent languages). (2)

- \( (1) + (2) = \&\text{-Prolog} \) [DeG84, Her86b]: view parallelization as a source to source transformation of original program into a parallelized (“annotated”) one in a \textit{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+09] 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

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

  ```prolog
  main :-
      t(X,Y,Z),
      $s_1$ p(X,Y),
      $s_2$ 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 $t(X,Y,Z) :- X=a$. 

• The “pointers” view:

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

\[
\text{main :- } X=f(K,g(K)), \ Y=a, \\
\text{ } \ Z=g(L), \ W=h(b,L), \\
\text{ } \ \longrightarrow \\
\text{ } p(X,Y), \\
\text{ } q(Y,Z), \\
\text{ } r(W).
\]

\( p \) and \( q \) are strictly independent, but \( q \) and \( r \) are not.
A Priori Independence: Strict Independence-III

• Example 2:

\[
\text{qs}([X|L], R) :- \text{part}(L,X,L1,L2), \\
\quad \text{qs}(L2,R2), \text{qs}(L1,R1), \\
\quad \text{app}(R1,[X|R2],R).
\]

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

\[
\text{qs}([X|L], R) :- \\
\quad \text{part}(L,X,L1,L2), \\
\quad ( \text{indep}(L1,L2) \rightarrow \text{qs}(L2,R2) \& \text{qs}(L1,R1) \\
\quad \quad \quad ; \ \text{qs}(L2,R2), \text{qs}(L1,R1) ), \\
\quad \text{app}(R1,[X|R2],R).
\]

• Not always possible to determine locally/statically:

\[
\text{main} :- \text{t}(X,Y), \quad \text{p}(X), \text{q}(Y).
\]

\[
\text{main} :- \text{read}([X,Y]), \text{p}(X), \text{q}(Y).
\]

• 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 \[ \text{ECR93, CDO88} \]:
  - Simplicity
  - Relatively high overhead → higher need for granularity control
  - Used, e.g., in ECLIPSE system.

- Direct implementation \[ \text{Her86b} \]:
  - Abstract machine needs to be modified: e.g., PWAM / Marker model \[ \text{Her87, Her86a, SH96, PG98} \], EAM/AKL box machine \[ \text{War90, JH90} \].
    - 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.
- 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, GHPSC94a, PGPF97]
Sequent Symmetry, **hand parallelized** programs.
(Speedup over state of the art sequential systems.)
Visualization of And-parallelism – (small) qsort, 1 processor

(VisAndOr \textsc{CGH}93 output.)
Visualization of And-parallelism – (small) qsort, 4 processors

(VisAndOr CGH93 output.)
Independence – Strict Independence (Contd.)

- Not always possible to determine locally/statically:
  
  \[
  \text{main} :- \ t(X,Y), \ p(X), \ q(Y).
  \]
  \[
  \text{main} :- \ \text{read}([X,Y]), \ p(X), \ q(Y).
  \]

- Alternatives: run-time independence tests, global analysis, ...
  
  \[
  \text{main} :- \ \text{read}([X,Y]), \ ( \ \text{indep}(X,Y) \ \rightarrow \ p(X) \ \& \ q(Y) \ ; \ p(X), q(Y) \ ).
  \]
  \[
  \text{main} :- \ t(X,Y), \ p(X) \ \& \ q(Y). \quad \%\% \ (\text{After analysis})
  \]
Parallelization Process: CDG-based Automatic Parallelization

- **Conditional Dependency Graph** (of some code segment) \[HW87, BGH99, GPA+01\]:
  - Vertices: possible tasks (statements, calls, bindings, 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.

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

Alternative:

"Annotation"

Local/Global analysis and simplification

\[
\text{Local/Global analysis and simplification}
\]

Alternative: \( g1, (g2 \& g3) \)
Simplifying Independence Conditions (Strict Ind.)

Recall that \( b_1 \) and \( b_2 \) are strictly independent for \( \theta \) iff

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

\( \text{indep}(b_1, b_2) \) 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 \( \text{indep}\{x, y\}, \{y, z\} \).

Equivalent to \( \{\text{indep}(x, y), \text{indep}(x, z), \text{indep}(y, y), \text{indep}(y, z)\} \).

Domain of interpretation \( DI \): subset of propositional logic.

For clause \( C \), it contains predicates of the form \( \text{ground}(x) \) and \( \text{indep}(y, z) \), \( \{x, y, z\} \subseteq \text{vars}(C) \), with axioms:

\[
\begin{align*}
\{\text{ground}(x) \rightarrow \text{indep}(x, y) | \{x, y\} \subseteq \text{vars}(C)\} \\
\{\text{indep}(x, x) \rightarrow \text{ground}(x) | x \in \text{vars}(C)\}
\end{align*}
\]

The set \( \{\text{indep}(x, y), \text{indep}(x, z), \text{indep}(y, y), \text{indep}(y, z)\} \) can be simplified to \( \{\text{ground}(y), \text{indep}(x, z)\} \).
Simplifying Independence Conditions (Strict Ind.)

BGH99

Identify Dependencies

\[ p(x,y) \quad q(x,z) \quad s(z,w) \]

\[ \text{ind}(y,w) \]

false

true

Simplify Dependencies

\[ h(x,y,z) : - (p(x,y) \& q(x,z)), s(z,w). \]

\[ h(x,y,z) : - \text{ind}(y,w) \rightarrow p(x,y) \& (q(x,z), s(z,w)) ; p(x,y), q(x,z), s(z,w). \]
&-Prolog/Ciao Parallelizer Overview

USER

Ciao:
(C)LP, FP, (Java) ...

Annotators (local dependency analysis)
MEL/CDG/UDG/URLP/...

Parallelized Code (&)

Ciao/&–Prolog
Parallel RT system

PARALLELIZING COMPILER (CiaoPP)

Abstract Interpretation
(Sharing, Sharing+Freeness, Aeqs, Def, Lsign, ...)

Dependency Info

side–effect analysis

granularity analysis
Parallelizing compiler \cite{HW87} (now integrated in CiaoPP \cite{HBPLG99, HPBLG03}):

- **Global Analysis**: infers independence information.
- **Annotator(s)**: Prolog $\rightarrow$ &-Prolog parallelization \cite{DeG87, MH90, BGH94a, CH94, PGPF97, MBdlBH99}.
  - MEL: Maximum Expression Length — simple heuristic.
  - CDG: Conditional Graph Expressions — graph partitioning of clauses.
  - UDG: Unconditional Graph Expressions.
  - 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 \cite{DLH90, DL91, DL93, DLGHL94, DLGHL97, DLGH97, SCK98, MLGCH08}.
- **Granularity Control**: restricts parallelism based on task sizes \cite{DLH90, LGHD96, SCK98}.
- **Other modules**: *side effect analyzer* (sequencing of side-effects, coded in &-Prolog), *multiple specializer / partial evaluator, invariant eliminator*, etc.
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)
\textbf{\textit{&-Prolog compilation: examples - II}}

\begin{verbatim}
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.
\end{verbatim}

Parallelized program (\textit{&-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 $A_{Sub}$ (linearity) domain (“P”) \[Søn86\] \[MS93\].
  - Type domains, depth-K, etc.
  - (Constraints:) Definiteness \[dIBH93\] \[AMSS94\], Freeness \[dIBHB+96\], LSign \[KMM+96\] domains.

- Domains combined using \[CMB+95\] framework: e.g. $A_{Sub}$+$SH$, $A_{Sub}$+$ShF$

- 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) \[dIBH93\] \[dIBHB+96\], 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 $Occ(\theta, U) = \{X | X \in \text{dom}(\theta) \land U \in \text{var}(X\theta)\}$,

- **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**
  - $\text{independent}(x\theta, y\theta) \iff \exists v \in Uvar, x \in Occ(\theta, v) \land y \in Occ(\theta, v)$

Other additional axioms are encoded in the sharing patterns.
:- entry multiply(g,g,f).

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.

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:**
\[ D_\alpha = D_{\alpha-sharing} \times D_{\alpha-freeness} \]

- \( D_{\alpha-sharing} = \varnothing(\varnothing(P\text{var})) \)
- \( D_{\alpha-freeness} = \varnothing(P\text{var}) \)

- Abstraction (freeness) of a substitution:
  \[ \alpha_{freeness}(\theta) = \{ X \mid X \in \text{dom}(\theta), \exists Y \in U\text{var} \ (X\theta = Y) \} \]

- Example:
  \[ \theta = \{ W/P, X/f(P, Q), Y/g(Q, R), Z/f(a) \} \]
  \[ \alpha(\{\theta\}) = (\lambda_{sharing}, \lambda_{freeness}), \text{ where} \]
  - \( \lambda_{sharing} = \{\emptyset, \{Y\}, \{W, X\}, \{X, Y\}\} \)
  - \( \lambda_{freeness} = \{W\} \)
Two components: sharing & freeness ($\hat{\theta}_{SH}, \hat{\theta}_{FR}$)

The freeness information restricts the possible combinations of sharing patterns.

Pictorial representation:

- $p(X,Y,Z)$
  - $\hat{\theta}_{SH} = [[XY]]$
  - $\hat{\theta}_{FR} = [Y]$
  
- $p(X,L)$
  - $\hat{\theta}_{SH} = [[X][XL]]$
  - $\hat{\theta}_{FR} = [L]$

- $p(X,Y,Z)$
  - $\hat{\theta}_{SH} = [[XY][Z]]$
  - $\hat{\theta}_{FR} = [Z]$

- $X = f(Y)$
- $Z = b$
- $X = [Y|L]$
- $X = f(A)$
- $Y = f(A)$
&-Prolog compilation: examples - IV

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

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.

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!)

<table>
<thead>
<tr>
<th>Program</th>
<th>Average time in seconds</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Prol.</td>
</tr>
<tr>
<td>aiakl</td>
<td>0.17</td>
</tr>
<tr>
<td>ann</td>
<td>1.76</td>
</tr>
<tr>
<td>bid</td>
<td>0.46</td>
</tr>
<tr>
<td>boyer</td>
<td>1.12</td>
</tr>
<tr>
<td>browse</td>
<td>0.38</td>
</tr>
<tr>
<td>deriv</td>
<td>0.21</td>
</tr>
<tr>
<td>fib</td>
<td>0.03</td>
</tr>
<tr>
<td>hanoiapp</td>
<td>0.11</td>
</tr>
<tr>
<td>mmatrix</td>
<td>0.07</td>
</tr>
<tr>
<td>occur</td>
<td>0.34</td>
</tr>
<tr>
<td>peephole</td>
<td>1.36</td>
</tr>
<tr>
<td>qplan</td>
<td>1.68</td>
</tr>
<tr>
<td>qsortapp</td>
<td>0.08</td>
</tr>
<tr>
<td>read</td>
<td>1.07</td>
</tr>
<tr>
<td>serialize</td>
<td>0.20</td>
</tr>
<tr>
<td>tak</td>
<td>0.04</td>
</tr>
<tr>
<td>warplan</td>
<td>0.80</td>
</tr>
<tr>
<td>witt</td>
<td>1.86</td>
</tr>
</tbody>
</table>

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

[BGH94b, MBdI BH99, BGH99]
Dynamic tests (’96 numbers!)

(1-10 processors actual speedups on Sequent Symmetry; 10+ projections using IDRA simulator on execution traces) [BGH94b, MBdBH99, BGH99]

M. Hermenegildo – Parallel Execution of Logic Programs
Computlog/ALP Summer School – Las Cruces, NM, July 24-27 2008
A Closer Look at Some Speedups

Benchmark: mmatrix

Simple matrix mul. (> 12 simulated)

The parallelizer, self-parallelized
Independence – Non-Strict Independence

- Pure goals: only one thread “touches” each shared variable. Example:
  
  ```prolog
  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:
  
  ```prolog
  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.”

\[
\text{flatten}(Xs, Ys) :- \text{flatten}(Xs, Ys, []). \\
\text{flatten}([], Xs, Xs). \\
\text{flatten}([X|Xs], Ys, Zs) :- \text{flatten}(X, Ys, Ys1), \text{flatten}(Xs, Ys1, Zs). \\
\text{flatten}(X, [X|Xs], Xs) :- \text{atomic}(X), X \neq [].
\]

- Another example:

\[
\text{qsort}([], S, S). \\
\text{qsort}([X|Xs], S, S2) :- \\
\quad \text{partition}(Xs, X, L, R), \\
\quad \text{qsort}(L, S, [X|S1]), \\
\quad \text{qsort}(R, S1, S2).
\]
We consider the parallelization of pairs of goals.

Let the situation be: \( \{\tilde{\beta}\} \, p \, \{\tilde{\psi}\} \ldots q \).

We define:

\[
S(p) = \{L \in \tilde{\beta}_{\text{SH}} \mid L \cap \text{var}(p) \neq \emptyset\}
\]

\[
\text{SH} = S(p) \cap S(q) = \{L \in \tilde{\beta}_{\text{SH}} \mid L \cap \text{var}(p) \neq \emptyset \wedge L \cap \text{var}(q) \neq \emptyset\}
\]

Conditions for non-strict independence for \( p \) and \( q \):

\begin{align*}
\text{C1} & \quad \forall L \in \text{SH} \, L \cap \tilde{\psi}_{\text{FR}} \neq \emptyset \\
\text{C2} & \quad \neg (\exists N_1 \ldots N_k \in S(p) \, \exists L \in \tilde{\psi}_{\text{SH}} \, \exists i, j \, 1 \leq i < j \leq k \, \exists L \in \tilde{\psi}_{\text{SH}} \, \forall i, j \, 1 \leq i < j \leq k \, N_i \cap N_j \cap \tilde{\beta}_{\text{FR}} = \emptyset)
\end{align*}

- **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:
  - \( \text{ground}(X) \): \( X \) is ground.
  - \( \text{allvars}(X,F) \): every free variable in \( X \) is in the list \( F \).
  - \( \text{indep}(X,Y) \): \( X \) and \( Y \) do not share variables.
  - \( \text{sharedvars}(X,Y,F) \): every free variable shared by \( X \) and \( Y \) is in the list \( F \).
- 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.

<table>
<thead>
<tr>
<th>P</th>
<th># of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.78</td>
</tr>
<tr>
<td>2</td>
<td>0.54</td>
</tr>
<tr>
<td>3</td>
<td>0.56</td>
</tr>
<tr>
<td>4</td>
<td>0.91</td>
</tr>
<tr>
<td>5</td>
<td>0.99</td>
</tr>
</tbody>
</table>
Independence – Constraint Independence

- Standard Herbrand notions do not carry over to general constraint systems.
  
  \[\text{main} : - Y > X, Z > X, p(Y) & q(Z), \ldots\]
  \[\text{main} : - Y > X, X > Z, p(Y) & q(Z), \ldots\]

- 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 \textit{a-priori} condition: given \(g_1(\bar{x})\) and \(g_2(\bar{y})\):
  \[
  (\bar{x} \cap \bar{y} \subseteq \text{def}(c)) \text{ and } (\exists_{-\bar{x}} c \land \exists_{-\bar{y}} c \rightarrow \exists_{-\bar{y} \cup \bar{x}} c)
  \]
  \(\text{def}(c)\) is the set of variables constrained to a unique value in \(c\)

- For \(c = \{y > x, z > x\}\)
  \[\exists_{-\{y\}} c = \exists_{-\{z\}} c = \exists_{-\{y,z\}} c = \text{true}\]

- For \(c = \{y > x, x > z\}\)
  \[\exists_{-\{y\}} c = \exists_{-\{z\}} c = \text{true}, \quad \exists_{\{y,z\}} c = y > z\]

- Approximation: presence of “links” through the store.

- Run-time checks: \(\text{def}(X), \text{indep}(X,Y), \text{unlinked}(X,Y)\)
Some Preliminary CLP &-Parallelization Results (Compiler)

- Parallel expressions:

<table>
<thead>
<tr>
<th>Bench. Program</th>
<th>Total CGEs</th>
<th>Uncond. CGEs</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Def</td>
<td>Free</td>
</tr>
<tr>
<td>amp</td>
<td>5</td>
<td>–</td>
</tr>
<tr>
<td>bridge</td>
<td>0</td>
<td>–</td>
</tr>
<tr>
<td>circuit</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>dnf</td>
<td>14</td>
<td>14</td>
</tr>
<tr>
<td>laplace</td>
<td>1</td>
<td>–</td>
</tr>
<tr>
<td>mining</td>
<td>5</td>
<td>4</td>
</tr>
<tr>
<td>mmatrix</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>mg_extend</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>num</td>
<td>16</td>
<td>16</td>
</tr>
<tr>
<td>pic</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>power</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>runge_kutta</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>trapezoid</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Some Preliminary CLP &-Parallelization Results (Compiler)

- Conditional checks:

<table>
<thead>
<tr>
<th>Bench. Program</th>
<th>Conditions: def/unlinked</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Def</td>
</tr>
<tr>
<td>amp</td>
<td>1/10</td>
</tr>
<tr>
<td>bridge</td>
<td>0/0</td>
</tr>
<tr>
<td>circuit</td>
<td>1/5</td>
</tr>
<tr>
<td>dnf</td>
<td>0/2</td>
</tr>
<tr>
<td>laplace</td>
<td>0/0</td>
</tr>
<tr>
<td>mining</td>
<td>3/5</td>
</tr>
<tr>
<td>mmatrix</td>
<td>0/2</td>
</tr>
<tr>
<td>mg_extend</td>
<td>0/0</td>
</tr>
<tr>
<td>num</td>
<td>0/24</td>
</tr>
<tr>
<td>pic</td>
<td>2/9</td>
</tr>
<tr>
<td>power</td>
<td>3/40</td>
</tr>
<tr>
<td>runge_kutta</td>
<td>5/0</td>
</tr>
<tr>
<td>trapezoid</td>
<td>0/9</td>
</tr>
</tbody>
</table>
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 than 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 \[GHM96\], 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.

→ “No such thing as dependent and-parallelism.”
Dealing with Speculation

- Computations can be speculative (or even non-terminating!):
  
  ```prolog
  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, Gar94].

- Static detection of non-failure [BCHM94, DLGH97]: avoids speculativeness / guarantees theoretical speedup.

→ *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

- 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.

![Diagram showing granulation control]
Granularity Control Example

- For the previous example:

  \[
  \text{main} :- \text{read}(X), \text{read}(Z), \text{inc\_all}(X,Y) \& \text{r}(Z,M), \ldots \\
  \text{inc\_all}([]) := []. \\
  \text{inc\_all}([I|Is]) := [ I+1 \mid \neg \text{inc\_all}(Is) ].
  \]

- Assume \( X \) determined to be input, \( Y \) output, cost function inferred \( 2 \times \text{length}(X) + 1 \), threshold 100 units:

  \[
  \text{main} :- \text{read}(X), \text{read}(Z), (2\times\text{length}(X)+1 > 100 \rightarrow \text{inc\_all}(X,Y) \& \text{r}(Z,M) ) \\
  \hspace{2cm} \text{; inc\_all}(X,Y), \text{r}(Z,M))
  \]

- 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:
   \[\text{:- true inc_all}(X,Y) : \text{list}(X,\text{int}), \text{var}(Y) \Rightarrow \text{list}(Y,\text{int}).\]

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:

   \[
   \text{Size}^2_{\text{inc_all}}(0) = 0 \text{ (boundary condition from base case)}, \\
   \text{Size}^2_{\text{inc_all}}(n) = 1 + \text{Size}^2_{\text{inc_all}}(n - 1).
   \]

   \[\text{Sol} = \text{Size}^2_{\text{inc_all}}(n) = n.\]

4. Use this, set up recurrence equations for the computational cost of procedures:

   \[
   \text{Cost}^L_{\text{inc_all}}(0) = 1 \text{ (boundary condition from base case)}, \\
   \text{Cost}^L_{\text{inc_all}}(n) = 2 + \text{Cost}^L_{\text{inc_all}}(n - 1).
   \]

   \[\text{Sol} = \text{Cost}^L_{\text{inc_all}}(n) = 2n + 1.\]

- 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:
  ...,
  \( \text{length}(X) > 50 \rightarrow \text{inc}_\text{all}(X,Y) \& r(Z,M) \)
  \( ; \text{inc}_\text{all}(X,Y) \& r(Z,M) ; \)
  ...,

  ...,
  \( \text{length}_\text{gt}(LX,50) \rightarrow \text{inc}_\text{all}(X,Y) \& r(Z,M) \)
  \( ; \text{inc}_\text{all}(X,Y) \& r(Z,M) ; \)
  ...

- Complex thresholds: use also communication cost functions, load, ...

**Example:** Assume \( \text{CommCost}(\text{inc}_\text{all}(X)) = 0.1 \) \( (\text{length}(X) + \text{length}(Y)) \).

We know \( \text{ub}_\text{length}(Y) \) (actually, exact size) = \( \text{length}(X) \); thus:

\[
2 \text{length}(X) + 1 > 0.1 (\text{length}(X) + \text{length}(X)) \equiv 2 \text{length}(X) > 0.2 \text{length}(X) \equiv
\]

Guaranteed speedup for any data size!

\[\Rightarrow\] 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):
  
  \[ ... \text{, } (\text{has\_more\_elements\_than}(X, 5) \rightarrow \text{inc\_all\_2}(X, Y) \& r(X)) \; ; \; \text{inc\_all\_2}(X, Y), \; r(X)) \text{, } ... \]

  \[
  \text{inc\_all}([X_1, X_2, X_3, X_4, X_5 | R) := [X_1 + 1, X_2 + 1, X_3 + 1, X_4 + 1, X_5 + 1 \mid \sim \text{inc\_all}(R)].
  \]

  \[
  \text{inc\_all}([]) := [].
  \]

  (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, GHPSC94b, 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:

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

- Cluster:

<table>
<thead>
<tr>
<th>programs</th>
<th>seq. prog.</th>
<th>no gran.ctl</th>
<th>gran.ctl</th>
<th>gc.stopping</th>
<th>gc.argsize</th>
</tr>
</thead>
<tbody>
<tr>
<td>fib(19)</td>
<td>1.839</td>
<td>0.970</td>
<td>1.389</td>
<td>1.009</td>
<td>0.639</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-43%</td>
<td>-4.0%</td>
<td>+34%</td>
</tr>
<tr>
<td>hanoi(13)</td>
<td>6.309</td>
<td>2.690</td>
<td>2.839</td>
<td>2.419</td>
<td>2.419</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-5.5%</td>
<td>+10.1%</td>
<td>+10.1%</td>
</tr>
<tr>
<td>unbmatrix</td>
<td>2.099</td>
<td>1.039</td>
<td>1.349</td>
<td>0.870</td>
<td>0.870</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-29.84%</td>
<td>+16.27%</td>
<td>+16.27%</td>
</tr>
<tr>
<td>qsort(1000)</td>
<td>3.670</td>
<td>1.819</td>
<td>2.009</td>
<td>1.649</td>
<td>1.429</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>-11%</td>
<td>+9.3%</td>
<td>+21%</td>
</tr>
</tbody>
</table>
Refinements (2): Granularity-Aware Annotation

- 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:

  \[ \begin{align*}
  &a \rightarrow b \\
  &c \rightarrow d \\
  &d \rightarrow e
  \end{align*} \]

- 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 \( j \).

<table>
<thead>
<tr>
<th>MEL annotator:</th>
<th>( a, b &amp; c, d &amp; e )</th>
</tr>
</thead>
<tbody>
<tr>
<td>UDG annotator:</td>
<td>( c &amp; ( a, b, e ), d )</td>
</tr>
<tr>
<td>Granularity-aware:</td>
<td>( a, c, ( b &amp; d ), e )</td>
</tr>
</tbody>
</table>
Refinements (3): Using Execution Time Bounds/Estimates

- 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.
  - 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 \texttt{nrev} with mode:
  \[
  \text{:- pred nrev/2 : list(int) * var.}
  \]

- Estimation of execution time for a concrete input — consider:
  \[
  A = [1,2,3,4,5], \quad n = \text{length}(A) = 5
  \]

<table>
<thead>
<tr>
<th>Component</th>
<th>Once $K_{\omega_i}$</th>
<th>Static Analysis $\text{Cost}_p(I(\omega_i), n) = C_i(n)$</th>
<th>Application $C_i(5)$</th>
<th>$K_{\omega_i} \times C_i(5)$</th>
</tr>
</thead>
<tbody>
<tr>
<td>step</td>
<td>21.27</td>
<td>$0.5 \times n^2 + 1.5 \times n + 1$</td>
<td>21</td>
<td>446.7</td>
</tr>
<tr>
<td>nargs</td>
<td>9.96</td>
<td>$1.5 \times n^2 + 3.5 \times n + 2$</td>
<td>57</td>
<td>567.7</td>
</tr>
<tr>
<td>giunif</td>
<td>10.30</td>
<td>$0.5 \times n^2 + 3.5 \times n + 1$</td>
<td>31</td>
<td>319.3</td>
</tr>
<tr>
<td>gounif</td>
<td>8.23</td>
<td>$0.5 \times n^2 + 0.5 \times n + 1$</td>
<td>16</td>
<td>131.7</td>
</tr>
<tr>
<td>viunif</td>
<td>6.46</td>
<td>$1.5 \times n^2 + 1.5 \times n + 1$</td>
<td>45</td>
<td>290.7</td>
</tr>
<tr>
<td>vounif</td>
<td>5.69</td>
<td>$n^2 + n$</td>
<td>30</td>
<td>170.7</td>
</tr>
</tbody>
</table>

Execution time $\overline{K_{\Omega}} \cdot \overline{\text{Cost}_p(I(\Omega), n)}$: 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)
Dependent And-parallelism: DDAS (I)

- Exploits Independent + “Dependent” And-parallelism.
- Goals communicate through shared variables.
- Shared variables are marked (dep/1 annotation).
- Example: \( \text{example}(X) : - (\text{dep}(X) \Rightarrow a(X) \& b(X)) \).
  \[
a(X). \quad b(1).
\]
- 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 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 → no dependent backtracking needed.
  ◦ Prolog support: preprocessor + engine (interpreter).
  ◦ Exploits both and- and or-parallelism. (Good speedups in practice)
  ◦ Problem: no nondeterministic steps can proceed in parallel.
  ◦ 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

- 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:

  \[
  \begin{align*}
  p(X,Y,Z) & : - \\
  & a(X,Z), \\
  & b(X), \\
  & c(Y), \\
  & d(Y,Z).
  \end{align*}
  \]

  \[
  \begin{align*}
  p(X,Y,Z) & : - \\
  & a(X,Z), \\
  & b(X) & & d(Y,Z).
  \end{align*}
  \]

  \[
  \begin{align*}
  p(X,Y,Z) & : - \\
  & c(Y) & & (a(X,Z),b(X)), \\
  & d(Y,Z).
  \end{align*}
  \]

Sequential | Restricted IAP | 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

<table>
<thead>
<tr>
<th>Benchm.</th>
<th>Ann.</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>AIAKL</td>
<td>UMEL</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>UOUDG</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>UDG</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>UUDG</td>
<td>0.97</td>
</tr>
<tr>
<td>Hanoi</td>
<td>UMEL</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>UOUDG</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>UDG</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>UUDG</td>
<td>0.89</td>
</tr>
<tr>
<td>FibFun</td>
<td>UMEL</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>UOUDG</td>
<td>0.99</td>
</tr>
<tr>
<td></td>
<td>UDG</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>UUDG</td>
<td>0.99</td>
</tr>
<tr>
<td>Takeuchi</td>
<td>UMEL</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>UOUDG</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>UDG</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>UUDG</td>
<td>0.88</td>
</tr>
</tbody>
</table>
Performance results - Restricted vs. Unrestricted And-Parallelism

AIAKL

FibFun

Sun Fire T2000 - 8 cores

Hanoi

Takeuchi

M. Hermenegildo – Parallel Execution of Logic Programs
Compulog/ALP Summer School – Las Cruces, NM, July 24-27 2008
Towards a higher-level implementation

- 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**

---

M. Hermenegildo – Parallel Execution of Logic Programs

Compulog/ALP Summer School – Las Cruces, NM, July 24-27 2008
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.)
  [HN90, CGH93, 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 parallelism 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])


[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.


<table>
<thead>
<tr>
<th>Reference</th>
<th>Title</th>
<th>Details</th>
</tr>
</thead>
</table>


