Automatic Unrestricted Independent And-Parallelism in Declarative Multiparadigm Languages

Amadeo Casas

Electrical and Computer Engineering Department
University of New Mexico

Ph.D. Dissertation Thesis

September 2\textsuperscript{nd}, 2008
1. Introduction and Motivation
2. Background
3. Functions and Lazy Evaluation Support for LP Kernels
4. Annotation Algorithms for Unrestricted IAP
5. High-Level Implementation of Unrestricted IAP
6. Concluding Remarks and Future Work
7. Publications
1 Introduction and Motivation

2 Background

3 Functions and Lazy Evaluation Support for LP Kernels

4 Annotation Algorithms for Unrestricted IAP

5 High-Level Implementation of Unrestricted IAP

6 Concluding Remarks and Future Work

7 Publications
Introduction

- Parallelism (finally!) becoming mainstream thanks to multicore architectures — even on laptops!

- Parallelizing programs is a hard challenge.
  - Necessity to exploit parallel execution capabilities as easily as possible.

- Renewed research interest in development of tools to write parallel programs:
  - Design of languages that better support exploitation of parallelism.
  - Improved libraries for parallel programming.
  - Progress in support tools: parallelizing compilers.
Why Logic Programming?

- Significant progress made in parallelizing compilers for regular computations. But further challenges:
  - Parallelization across procedure calls.
  - Irregular computations.
  - Complex data structures (as in C/C++).
    - Much current work in independence analyses: *pointer aliasing analysis*.
  - Speculation.

- Declarative languages are a very interesting framework for parallelization:
  - All the challenges above appear in the parallelization of LP!
  - But:
    - Program much closer to problem description.
    - Notion of control provides more flexibility.
    - Cleaner semantics (e.g., pointers exist, but are declarative).
Declarative / multiparadigm languages

- Multiparadigm languages — building on the best features of each paradigm:
  - *Logic programming*: expressive power beyond that of functional programming.
    - Nondeterminism.
    - Partially instantiated data structures.
  - *Functional programming*: syntactic convenience.
    - Designated output argument: provides more compact code.
    - Lazy evaluation: ability to deal with infinite data structures.

→ We support both logic and functional programming.

- Industry interest:
  - Intel sponsorship of *DPMC* and *DAMP* (colocated with POPL) workshops.

- *Cross-paradigm synergy*: better parallelizing compilers can be developed by mixing results from different paradigms.
Introduction and Motivation

Background

Functions and Lazy Evaluation Support for LP Kernels

Annotation Algorithms for Unrestricted IAP

High-Level Implementation of Unrestricted IAP

Concluding Remarks and Future Work

Publications
Types of parallelism in LP

- Two main types:
  - Or-Parallelism: explores in parallel alternative computation branches.
  - And-Parallelism: executes procedure calls in parallel.
    - Traditional parallelism: parbegin-parend, loop parallelization, divide-and-conquer, etc.
    - Often marked with &/2 operator: fork-join nested parallelism.
Types of parallelism in LP

- Two main types:
  - *Or-Parallelism*: explores in parallel alternative computation branches.
  - *And-Parallelism*: executes procedure calls in parallel.
    - Traditional parallelism: parbegin-parend, loop parallelization, divide-and-conquer, etc.
    - Often marked with &/2 operator: fork-join nested parallelism.

Example (QuickSort: sequential and parallel versions)

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

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

- We will focus on and-parallelism.
  - Need to detect independent tasks.
Parallel execution and independence

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

<table>
<thead>
<tr>
<th>$s_1$</th>
<th>$s_2$</th>
<th>\begin{align*} Y &amp;:= W+2; \ X &amp;:= Y+Z; \end{align*}</th>
<th>\begin{align*} (+ (+ W 2) \quad &amp; \quad Z) \ Y &amp;:= W+2, \ X &amp;:= Y+Z, \end{align*}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td><strong>Imperative</strong></td>
<td><strong>Functional</strong></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td><strong>CLP</strong></td>
<td></td>
</tr>
</tbody>
</table>
Parallel execution and independence

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

<table>
<thead>
<tr>
<th></th>
<th>Imperative</th>
<th>Functional</th>
<th>CLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>( Y := W + 2; )</td>
<td>((+ (+ W 2) Z))</td>
<td>( Y = W + 2, )</td>
</tr>
<tr>
<td>2</td>
<td>( X := Y + Z; )</td>
<td></td>
<td>( X = Y + Z, )</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
\text{main :-} & \quad \text{p(X) :- X = [1,2,3].} \\
& \quad \text{s_1 p(X),} \\
& \quad \text{s_2 q(X),} \\
& \quad \text{write(X).} \\
& \quad \text{q(X) :- X = [], large computation.} \\
& \quad \text{q(X) :- X = [1,2,3].}
\end{align*}
\]

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

USER

Source Code

Annotation Process
MEL, UDG, ...
UDDG, UOUDG, ...

Dependency Info

Global Analysis (A. I.)

Side−effect Analysis

Granularity Analysis

Parallel Prolog Code
(&/2, &>/2, <&/1)

Execution Model
Architecture of parallelizing compiler

Background

USER

Source Code

FLOPS’06

Dependency Info

Annotation Process
MEL, UDG, ...
UUDG, UOUDG, ...

LOPSTR’07

Global Analysis (A. I.)

Side–effect Analysis

Granularity Analysis

Parallel Prolog Code
(ə/2, ə>/2, ə/1)

ICLP’08, PADL’08

Execution Model
CDG-based automatic parallelization

- **Conditional Dependency Graph:**
  - Vertices: possible sequential tasks (statements, calls, etc.)
  - Edges: conditions needed for independence (e.g., variable sharing).
- Local or global analysis to remove checks in the edges.
- Annotation converts graph back to (now parallel) source code.

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

Alternative:

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

**Alternative:**

```
g1, ( g2 & g3 )
```
An alternative, more flexible source code annotation

- Classical parallelism operator &/2: nested fork-join.
  - Rigid structure of &/2.
- However, more flexible constructions can be used to denote parallelism:
  - G &> H — schedules goal G for parallel execution and continues executing the code after G &> H.
    - H is a handler which contains / points to the state of goal G.
  - H <& — waits for the goal associated with H to finish.
    - The goal associated to H has produced a solution: bindings for the output variables are available.

- Operator &/2 can be written as:
  
  A & B :- A &> H, call(B), H <&.

- Optimized deterministic versions: &!/2, <&!/1.
  - Ciao provides a determinacy analysis.
Expressing more parallelism

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

<table>
<thead>
<tr>
<th>Sequential</th>
<th>Restricted IAP</th>
<th>Unrestricted IAP</th>
</tr>
</thead>
<tbody>
<tr>
<td>p(X,Y,Z) :- a(X,Z), b(X), c(Y), d(Y,Z).</td>
<td>p(X,Y,Z) :- a(X,Z) &amp; c(Y), b(X) &amp; d(Y,Z).</td>
<td>p(X,Y,Z) :- c(Y) &amp;&gt; Hc, a(X,Z), b(X) &amp;&gt; Hb, Hc &lt;&amp;, d(Y,Z), Hb &lt;&amp;.</td>
</tr>
</tbody>
</table>

In this case: unrestricted parallelization at least as good (time-wise) as restricted ones, assuming no overhead.
Introduction and Motivation

Background

Functions and Lazy Evaluation Support for LP Kernels

Annotation Algorithms for Unrestricted IAP

High-Level Implementation of Unrestricted IAP

Concluding Remarks and Future Work

Publications
Functional syntax layer

- Syntactic functional layer, with functions, laziness, and HO.
  - Implemented in Ciao, but useful in general for LP-based systems.

- Adding functional features to LP systems not new:
  - A good number of systems integrate functions into some form of LP: NU-Prolog, Lambda-Prolog, HiLog/XSB, Oz, Mercury, HAL,...
  - Or perform a “native” integration of FP and LP (e.g., Babel, Curry,...).

- Our approach: [Published at FLOPS’06]
  - Library-based implementation:
    - Exploits the extension facilities: packages.
    - Makes it independent from, and composable with other extensions: higher-order, constraints, etc.
    - No compiler or abstract machine modification (all done at source level).
  - Functions can retain the power of predicates (it is just syntax!).
  - Functions inherit all other Ciao features (assertions, properties, constraints,...) + (analysis, optimization, verification,...).
Overview of functional notation

Main features (briefly):

- **Function applications**: any term preceded by \( ~/1 \) operator, or declared as function with `:- fun_eval`.
- **Functional definitions**: via `=/2`.
- **Disjunctive and conditional expressions**:
  - \( (A \mid B \mid C), (\text{Cond1} \ ? \ V1), (\text{Cond1} \ ? \ V1 \mid V2) \).
- **Quoting**: `pair(A,B) := ~(A-B)`
- **Laziness**: via `:- lazy`.

Example (FibFun: parallel transformation)

\[
\begin{align*}
\text{fib}(0) & := 0. \\
\text{fib}(1) & := 1. \\
\text{fib}(N) & := \text{fib}(N-1) + \text{fib}(N-2) \quad \text{:- int}(N), N > 1. \\
\text{?- Y = ~fib(10)}. & \quad Y = 55. \\
\text{?- 55 = ~fib(X)}. & \quad X = 10.
\end{align*}
\]
Overview of functional notation

- **Main features (briefly):**
  - *Function applications:* any term preceded by \(~/1\) operator, or declared as function with \:- fun_eval. 
  - *Functional definitions:* via \:+/=2. 
  - *Disjunctive and conditional expressions:* 
    - \((A \mid B \mid C), (\text{Cond1} \ ? \ V1), (\text{Cond1} \ ? \ V1 \mid V2)\).  
  - *Quoting:* pair(A,B) := \(^{(A-B)}\). 
  - *Laziness:* via \:- lazy.

**Example (FibFun: parallel transformation)**

\[
\begin{align*}
\text{fib}(0) & := 0. \\
\text{fib}(1) & := 1. \\
\text{fib}(N) & := \text{fib}(N-1) + \text{fib}(N-2) \\
& := \text{int}(N), \ N > 1. \\
\end{align*}
\]

?- \ Y = \text{fib}(10).
\ Y = 55.
?- 55 = \text{fib}(X).
\ X = 10.
1 Introduction and Motivation

2 Background

3 Functions and Lazy Evaluation Support for LP Kernels

4 Annotation Algorithms for Unrestricted IAP

5 High-Level Implementation of Unrestricted IAP

6 Concluding Remarks and Future Work

7 Publications
New annotation algorithms: general idea

- Remember: $\&/2$ vs. $\&>/2 + <&/1$.

- Main idea: [Published at LOPSTR’07. Submitted to TPLP]
  - 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 of literals can affect the order of the solutions.
  - Order determined by $\&>/2$.
  - Order not respected $\Rightarrow$ more flexibility in annotation.
Automatic parallelization with alternative primitives
Non order-preserving, unrestricted annotation (I)

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

Example (Unrestricted Annotation UUDG)

<table>
<thead>
<tr>
<th></th>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td>a(X,Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>b(X)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>c(Y)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>d(Y,Z)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

∅
Automatic parallelization with alternative primitives
Non order-preserving, unrestricted annotation (I)

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

**Example (Unrestricted Annotation UUDG)**

\[
\begin{array}{ccc|ccc}
\text{Indep} & \text{Dep} & \text{pvt} & \text{ToPub} & \text{ToWait} & \text{Pub} \\
\{a, c\} & \{b, d\} & b & \{a, c\} & \{a\} & \{a, c\} \\
\end{array}
\]

\[
p(X, Y, Z) :-
\begin{align*}
& c(Y) \&> Hc, \\
& a(X, Z) \&> Ha,
\end{align*}
Ha <&,
\]
Automatic parallelization with alternative primitives
Non order-preserving, unrestricted annotation (I)

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

**Example (Unrestricted Annotation UUDG)**

\[
\begin{align*}
\text{a}(X,Z) &\quad \text{b}(X) \\
\text{c}(Y) &\quad \text{d}(Y,Z)
\end{align*}
\]

<table>
<thead>
<tr>
<th></th>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>{a,c}</td>
<td>{b,d}</td>
<td>b</td>
<td>{a,c}</td>
<td>{a}</td>
<td>{a,c}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>\emptyset</td>
</tr>
</tbody>
</table>

\[
p(X,Y,Z) :- \\
c(Y) \ &> \ Hc,  \\
a(X,Z),
\]

Amadeo Casas (ECE-UNM)  
Ph.D. Dissertation Thesis  
September 2\textsuperscript{nd}, 2008  
19 / 34
Automatic parallelization with alternative primitives
Non order-preserving, unrestricted annotation (I)

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

**Example (Unrestricted Annotation UUDG)**

<table>
<thead>
<tr>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td>∅</td>
<td>{b, d}</td>
<td>b</td>
<td>{a, c}</td>
<td>{a}</td>
<td>∅</td>
</tr>
<tr>
<td>{b, c}</td>
<td>{d}</td>
<td>d</td>
<td>{b}</td>
<td>{c}</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

\[
Amadeo Casas (ECE-UNM)
Ph.D. Dissertation Thesis
September 2^{nd}, 2008 19 / 34
\]
Automatic parallelization with alternative primitives
Non order-preserving, unrestricted annotation (I)

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

Example (Unrestricted Annotation UUDG)

<table>
<thead>
<tr>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a, c}</td>
<td>{b, d}</td>
<td>b</td>
<td>{a, c}</td>
<td>{a}</td>
<td>{a, c}</td>
</tr>
<tr>
<td>{b, c}</td>
<td>{d}</td>
<td>d</td>
<td>{b}</td>
<td>{c}</td>
<td>{a, b, c}</td>
</tr>
<tr>
<td>{b, d}</td>
<td>\emptyset</td>
<td>–</td>
<td>{d}</td>
<td>{b, d}</td>
<td>{a, b, c, d}</td>
</tr>
</tbody>
</table>

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

● Goal order switched w.r.t. sequential version.
Automatic parallelization with alternative primitives
Order-preserving, unrestricted annotation (II)

Example (Unrestricted Annotation UUDG)

<table>
<thead>
<tr>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>{}</td>
</tr>
</tbody>
</table>
Automatic parallelization with alternative primitives
Order-preserving, unrestricted annotation (II)

Example (Unrestricted Annotation UUDG)

\[ p(Y,Z) :\neg a(Y,Z), \]

<table>
<thead>
<tr>
<th></th>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a}</td>
<td>{b, e}</td>
<td>b</td>
<td>{a}</td>
<td>{a}</td>
<td>{a}</td>
<td>{a}</td>
</tr>
</tbody>
</table>

∅
Automatic parallelization with alternative primitives
Order-preserving, unrestricted annotation (II)

Example (Unrestricted Annotation UUDG)

\[
p(Y,Z) :-
  a(Y,Z),
  ( b(X), c(X), d(X) ) \&
  ( e(Y), f(Y), g(Y) ).
\]
Automatic parallelization with alternative primitives
Order-preserving, unrestricted annotation

Example (Unrestricted Annotation UOUDG)

<table>
<thead>
<tr>
<th>a(X,Z)</th>
<th>b(X)</th>
<th>c(Y)</th>
<th>d(Y,Z)</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>⊘</td>
</tr>
</tbody>
</table>

Indep = Independent
Dep = Dependent
pvt = Private
ToPub = To Publish
ToWait = To Wait
Pub = Published
Automatic parallelization with alternative primitives
Order-preserving, unrestricted annotation

Example (Unrestricted Annotation UOUDG)

<table>
<thead>
<tr>
<th></th>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>{a, c}</td>
<td>{b, d}</td>
<td>b</td>
<td>{a}</td>
<td>{a}</td>
<td>{a}</td>
</tr>
</tbody>
</table>

p(X, Y, Z) :-
a(X, Z),
Automatic parallelization with alternative primitives
Order-preserving, unrestricted annotation

Example (Unrestricted Annotation UOUDG)

<table>
<thead>
<tr>
<th>Indep</th>
<th>Dep</th>
<th>pvt</th>
<th>ToPub</th>
<th>ToWait</th>
<th>Pub</th>
</tr>
</thead>
<tbody>
<tr>
<td>{a, c}</td>
<td>{b, d}</td>
<td>b</td>
<td>{a}</td>
<td>{a}</td>
<td>{a}</td>
</tr>
<tr>
<td>{b, g}</td>
<td>\emptyset</td>
<td>--</td>
<td>{b, g}</td>
<td>\emptyset</td>
<td>{a, b, c, d}</td>
</tr>
</tbody>
</table>

\[ p(X, Y, Z) :- \]
\[ a(X, Z), \]
\[ b(X) \& \]
\[ (c(Y), d(Y)) . \]

- Goal order maintained but less parallelism exploited!
1 Introduction and Motivation

2 Background

3 Functions and Lazy Evaluation Support for LP Kernels

4 Annotation Algorithms for Unrestricted IAP

5 High-Level Implementation of Unrestricted IAP

6 Concluding Remarks and Future Work

7 Publications
Objectives of the execution model for unrestricted IAP

- Versions of and-parallelism previously implemented:
  - &-Prolog, &-ACE, AKL, Andorra-I,...

- They rely on complex low-level machinery:
  - Each agent: new WAM instructions, goal stack, parcall frames, markers, etc.

- Approach: rise components to the source language level:
  [Published at ICLP’08 and PADL’08]
  - **Prolog-level**: goal publishing, goal searching, goal scheduling, markers creation (through choice-points),...
  - **C-level**: low-level threading, locking, stack management, sharing of memory, untrailing,...
  - Current implementation for shared-memory multiprocessors:
    - Agent: sequential Prolog machine + goal list + (mostly) Prolog code.
    → Simpler machinery and more flexibility.
Memory management problems in nondeterministic IAP execution

- Lots of issues in memory management.
- In particular, dealing with the trapped goals and garbage slots problems:

```
?- a(X) &> Ha, b(Y) &> Hb, c(Z), Hb <&, Ha <&, fail.
```

![Diagram showing memory management issues in nondeterministic IAP execution.](image-url)
Creation of (high-level) markers

Execution of parallel goal

remote_call(Handler) :-
    save_init_execution(Handler),
    retrieve_goal(Handler, Goal),
    call(Goal),
    save_end_execution(Handler),
    set_goal_finished(Handler),
    release(Handler).

remote_call(Handler) :-
    set_goal_failed(Handler),
    release(Handler),
    metacut_garbage_slots(Handler),
    fail.

- Library of concurrency primitives to implement a high-level approach to IAP.
  - Better programming discipline $\Rightarrow$ easier to maintain!
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

  - goal available
  - speculative execution

- **Locally Executing**
  - call/1

  - execution finished
  - fail

- **Remotely Executing**
  - call_handler/1

  - execution finished
  - execution failed

  - goal found

- **Failed**
  - set_goal_failed/1
  - release/1

- **Finished**
  - set_goal_finished/1
  - release/1

- **Cancelled**
  - cancellation/1

- Amadeo Casas (ECE-UNM) Ph.D. Dissertation Thesis September 2\textsuperscript{nd}, 2008
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

- **Locally Executing**
  - call/1
  - execution finished
  - execution failed

- **Remotely Executing**
  - call_handler/1
  - execution finished

- **Failed**
  - set_goal_failed/1
  - release/1

- **Finished**
  - set_goal_finished/1
  - release/1

- **Cancelled**
  - cancellation/1

- **Speculative execution**
- **Goal found**
- **Execution cancelled**

- **Goal available**
- **Read event**
State diagram of a parallel goal

Published
- push_goal/3
- release_some_suspended_agent/0

goal available

Locally Executing
- call/1

Finished
- set_goal_finished/1
- release/1

Failed
- set_goal_failed/1
- release/1

Remotely Executing
- call_handler/1

Cancelled
- cancellation/1

speculative execution

goal found

execution cancelled

read event

execution failed
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

- **Locally Executing**
  - call/1
    - execution finished
    - fail
    - execution failed

- **Remotely Executing**
  - call_handler/1
    - execution finished
    - execution failed

- **Finished**
  - set_goal_finished/1
  - release/1

- **Failed**
  - set_goal_failed/1
  - release/1

- **Cancelled**
  - cancellation/1

- speculative execution
- goal available
- goal found
- execution cancelled
- read event
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0
  - speculative execution
  - goal found

- **Locally Executing**
  - call/1
  - execution finished
  - execution failed
  - goal available

- **Remotely Executing**
  - call_handler/1
  - execution finished
  - execution failed
  - read event
  - execution cancelled

- **Failed**
  - set_goal_failed/1
  - release/1

- **Finished**
  - set_goal_finished/1
  - release/1

- **Cancelled**
  - cancellation/1
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

- **Locally Executing**
  - call/1
    - execution finished
    - fail

- **Remotely Executing**
  - call_handler/1
    - execution finished
    - execution failed

- **Finished**
  - set_goal_finished/1
  - release/1

- **Failed**
  - set_goal_failed/1
  - release/1

- **Cancelled**
  - cancellation/1

- **Speculative execution**
  - goal found
  - execution cancelled
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

- **Locally Executing**
  - call/1
  - execution finished
  - execution failed

- **Remotely Executing**
  - call_handler/1
  - execution finished
  - execution failed

- **Finished**
  - set_goal_finished/1
  - release/1

- **Cancelled**
  - cancellation/1

- **Failed**
  - set_goal_failed/1
  - release/1

Actions:
- goal found
- goal available
- speculative execution
- execution cancelled
- execution failed
- execution finished
- read event
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0
  - speculative execution

- **Locally Executing**
  - call/1
  - execution finished
  - execution failed

- **Remotely Executing**
  - call_handler/1
  - execution finished
  - execution failed

- **Finished**
  - set_goal_finished/1
  - release/1

- **Failed**
  - set_goal_failed/1
  - release/1

- **Cancelled**
  - cancellation/1

- goal available
- goal found
- execution cancelled
- read event
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0
- **Locally Executing**
  - call/1
  - execution finished
  - execution failed
- **Remotely Executing**
  - call_handler/1
  - execution finished
  - execution failed
- **Failed**
  - set_goal_failed/1
  - release/1
- **Finished**
  - set_goal_finished/1
  - release/1
- **Cancelled**
  - cancellation/1

Transitions:
- Goal available
- Goal found
- Speculative execution
- Execution cancelled
- Read event
State diagram of a parallel goal

1. Published
   - push_goal/3
   - release_some_suspended_agent/0
   - speculative execution
   - goal available

2. Locally Executing
   - call/1
   - execution finished
   - execution failed

3. Remotely Executing
   - call_handler/1
   - read event
   - execution finished
   - execution failed

4. Finished
   - set_goal_finished/1
   - release/1

5. Cancelled
   - cancellation/1

6. Failed
   - set_goal_failed/1
   - release/1

Amadeo Casas (ECE-UNM)
Ph.D. Dissertation Thesis
September 2nd, 2008
**State diagram of a parallel goal**

![State diagram](image)

- **Published**
  - `push_goal/3`
  - `release_some_suspended_agent/0`
  - Speculative execution

- **Locally Executing**
  - `call/1`
  - Execution finished
  - Execution failed

- **Remote Executing**
  - `call_handler/1`
  - Execution cancelled

- **Finished**
  - `set_goal_finished/1`
  - Release/1

- **Failed**
  - `set_goal_failed/1`
  - Release/1

- **Cancelled**
  - `cancellation/1`
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

- **Remotely Executing**
  - call_handler/1
- **Locally Executing**
  - call/1

- **Finished**
  - set_goal_finished/1
  - release/1

- **Failed**
  - set_goal_failed/1
  - release/1

- **Cancelled**
  - cancellation/1

- **Execution Cancelled**
- **Execution Finished**
- **Execution Failed**
- **Goal Available**
- **Goal Found**
- **Speculative Execution**
- **Execution Finished**
- **Execution Failed**
- **Execution Cancelled**
- **Read Event**
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

- **Locally Executing**
  - call/1

- **Remotely Executing**
  - call_handler/1

- **Finished**
  - set_goal_finished/1
  - release/1

- **Failed**
  - set_goal_failed/1
  - release/1

- **Cancelled**
  - cancellation/1

- **Speculative execution**
  - goal found
  - execution cancelled

- **Execution finished**
  - execution failed

- **Goal available**
  - goal found

Amadeo Casas (ECE-UNM) Ph.D. Dissertation Thesis September 2nd, 2008 26 / 34
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

- **Locally Executing**
  - call/1
  - execution finished
  - execution failed

- **Remotely Executing**
  - call_handler/1
  - execution finished
  - execution failed

- **Finished**
  - set_goal_finished/1
  - release/1

- **Failed**
  - set_goal_failed/1
  - release/1

- **Cancelled**
  - cancellation/1
  - execution cancelled

- Speculative execution
  - goal found
  - goal available

- Read event
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0
  - speculative execution

- **Locally Executing**
  - call/1
  - execution finished
  - execution failed

- **Remotely Executing**
  - call_handler/1
  - execution finished
  - execution failed

- **Finished**
  - set_goal_finished/1
  - release/1

- **Failed**
  - set_goal_failed/1
  - release/1

- **Cancelled**
  - cancellation/1

Events:
- goal available
- goal found
- read event
- execution cancelled
State diagram of a parallel goal

Published
- push_goal/3
  - goal available
- release_some_suspended_agent/0

Locally Executing
- call/1
  - execution finished
  - fail
- Finished
  - set_goal_finished/1
    - execution finished
  - release/1

Remotely Executing
- call_handler/1
  - goal found
  - speculative execution
- Failed
  - execution failed
- Cancelled
  - execution cancelled
  - cancellation/1

Finished
- execution failed
State diagram of a parallel goal

- **Published**
  - *push_goal/3*
  - *release_some_suspended_agent/0*
  - goal available

- **Locally Executing**
  - *call/1*
  - *execution finished* → *Finished*
  - *execution failed* → *Failed*

- **Remotely Executing**
  - *call_handler/1*
  - *execution finished* → *Finished*
  - *execution failed* → *Failed*

- **Cancelled**
  - *cancellation/1*

- *speculative execution*

- *goal found*

- *read event*
State diagram of a parallel goal

- Published
  - push_goal/3
  - release_some_suspended_agent/0
- Locally Executing
  - call/1
  - execution finished
  - execution failed
- Remotely Executing
  - call_handler/1
  - read event
- Failed
  - set_goal_failed/1
  - release/1
- Finished
  - set_goal_finished/1
  - release/1
- Cancelled
  - cancellation/1

Speculative execution:
- goal found
- execution cancelled

Events:
- goal available
- execution failed
- execution finished
- set_goal_failed/1
- set_goal_finished/1
- cancellation/1

Amadeo Casas (ECE-UNM) Ph.D. Dissertation Thesis September 2\textsuperscript{nd}, 2008 26 / 34
State diagram of a parallel goal

- **Published**
  - push_goal/3
  - release_some_suspended_agent/0

- **Locally Executing**
  - call/1
    - execution finished
    - execution failed
  - goal available

- **Remotely Executing**
  - call_handler/1
    - fail
    - execution finished
    - execution failed

- **Finished**
  - set_goal_finished/1
  - release/1
  - goal found

- **Failed**
  - set_goal_failed/1
  - release/1
  - execution cancelled

- **Cancelled**
  - cancellation/1
  - speculative execution
State diagram of a parallel goal

**Published**
- `push_goal/3`
- `release_some_suspended_agent/0`

**Remotely Executing**
- `call_handler/1`

**Locally Executing**
- `call/1`

**Finished**
- `set_goal_finished/1`
- `release/1`

**Failed**
- `set_goal_failed/1`
- `release/1`

**Cancelled**
- `cancellation/1`

- **Execution**
  - `goal found`
  - `speculative execution`

- **State Transitions**
  - `goal available`
  - `execution finished`
  - `execution failed`
  - `fail`

- **Other Events**
  - `read event`

---

Amadeo Casas (ECE-UNM)  
Ph.D. Dissertation Thesis  
September 2\textsuperscript{nd}, 2008
High-Level Implementation of Unrestricted IAP

Performance results
Restricted vs. unrestricted parallelization

- **Sun Fire T2000:**
  - 8 cores and 8 Gb of memory, each of them capable of running 4 threads in parallel.
  - Speedups with more than 8 threads stop being linear even for completely independent computations, since threads in the same core compete for shared resources.
  - All performance results obtained by averaging 10 runs.

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>And-Par.</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>FibFun</td>
<td>Restricted</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
<td>1.00</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.99</td>
<td>1.95</td>
<td>2.89</td>
<td>3.84</td>
<td>4.78</td>
<td>5.71</td>
<td>6.63</td>
<td>7.57</td>
</tr>
<tr>
<td>FFT</td>
<td>Restricted</td>
<td>0.98</td>
<td>1.76</td>
<td>2.14</td>
<td>2.71</td>
<td>2.82</td>
<td>2.99</td>
<td>3.08</td>
<td>3.37</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.98</td>
<td>1.82</td>
<td>2.31</td>
<td>3.01</td>
<td>3.12</td>
<td>3.26</td>
<td>3.39</td>
<td>3.63</td>
</tr>
<tr>
<td>Hamming</td>
<td>Restricted</td>
<td>0.93</td>
<td>1.13</td>
<td>1.52</td>
<td>1.52</td>
<td>1.52</td>
<td>1.52</td>
<td>1.52</td>
<td>1.52</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.93</td>
<td>1.15</td>
<td>1.64</td>
<td>1.64</td>
<td>1.64</td>
<td>1.64</td>
<td>1.64</td>
<td>1.64</td>
</tr>
<tr>
<td>Takeuchi</td>
<td>Restricted</td>
<td>0.88</td>
<td>1.61</td>
<td>2.16</td>
<td>2.62</td>
<td>2.63</td>
<td>2.63</td>
<td>2.63</td>
<td>2.63</td>
</tr>
<tr>
<td></td>
<td>Unrestricted</td>
<td>0.88</td>
<td>1.62</td>
<td>2.39</td>
<td>3.33</td>
<td>4.04</td>
<td>4.47</td>
<td>5.19</td>
<td>5.72</td>
</tr>
</tbody>
</table>
Performance results

Restricted vs. unrestricted parallelization

(a) FibFun

(b) FFT

(c) Hamming

(d) Takeuchi
Performance results
Deterministic vs. Non-deterministic annotation

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Op.</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>1</td>
</tr>
<tr>
<td>AIAKL</td>
<td>&amp;!</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>0.96</td>
</tr>
<tr>
<td>Ann</td>
<td>&amp;!</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>0.96</td>
</tr>
<tr>
<td>Deriv</td>
<td>&amp;!</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>0.84</td>
</tr>
<tr>
<td>FFT</td>
<td>&amp;!</td>
<td>0.98</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>0.98</td>
</tr>
<tr>
<td>Hanoi</td>
<td>&amp;!</td>
<td>0.89</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>0.89</td>
</tr>
<tr>
<td>MMMatrix</td>
<td>&amp;!</td>
<td>0.91</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>0.90</td>
</tr>
<tr>
<td>QuickSort</td>
<td>&amp;!</td>
<td>0.97</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>0.97</td>
</tr>
<tr>
<td>Takeuchi</td>
<td>&amp;!</td>
<td>0.88</td>
</tr>
<tr>
<td></td>
<td>&amp;</td>
<td>0.88</td>
</tr>
</tbody>
</table>
Performance results
Non-deterministic benchmarks

Performance results obtained in some representative non-deterministic parallel benchmarks:

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Number of processors</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
</tr>
<tr>
<td>Chat</td>
<td>2.31</td>
</tr>
<tr>
<td>Numbers</td>
<td>1.84</td>
</tr>
<tr>
<td>Progeom</td>
<td>0.99</td>
</tr>
<tr>
<td>Queens</td>
<td>0.99</td>
</tr>
<tr>
<td>QueensT</td>
<td>0.99</td>
</tr>
</tbody>
</table>

Super-linear speedups are achievable, thanks to good failure implementation (e.g., eager goal cancellation).
Concluding Remarks and Future Work

1. Introduction and Motivation
2. Background
3. Functions and Lazy Evaluation Support for LP Kernels
4. Annotation Algorithms for Unrestricted IAP
5. High-Level Implementation of Unrestricted IAP
6. Concluding Remarks and Future Work
7. Publications
Concluding Remarks and Future Work

Conclusions and future work

- New approach for exploiting and-parallelism automatically:
  - Support for unrestricted and-parallelism, annotation, multiparadigm, ...
  - Simpler machinery and more flexibility.

- Performance results:
  - Reasonable speedups are achievable.
  - Super-linear speedups can be achieved, thanks to goal cancellation.
  - Unrestricted and-parallelism provides better observed speedups.

- Expanded results to other paradigms:
  - Functional extension of Prolog + lazy evaluation.

- All this work available in **Ciao**: freely downloadable!

- Future work:
  - Support for HO pattern unification in functional syntax extension.
  - Usage of resource information to control the additional inherent overhead due to the nature of the high-level implementation.
  - Improvements in execution model:
    - Usage of existing tools in execution model (e.g., tabling).
    - Exploitation of other sources of parallelism.
    - Design efficient parallel GC algorithms for this approach.
1. Introduction and Motivation

2. Background

3. Functions and Lazy Evaluation Support for LP Kernels

4. Annotation Algorithms for Unrestricted IAP

5. High-Level Implementation of Unrestricted IAP

6. Concluding Remarks and Future Work

7. Publications
Publications in international conferences


- All publications in *Springer LNCS* series (listed in *JCR*).
  - Three A-level (*ICLP/PADL/FLOPS*), one B-level (*LOPSTR*).
  - *LOPSTR* extended version currently submitted for publication in international journal (*TPLP*).