# 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 $2^{nd}$, 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.

THE UNIVERSITY of NEW MEXICO

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

## 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.

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

## Parallel execution and independence

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

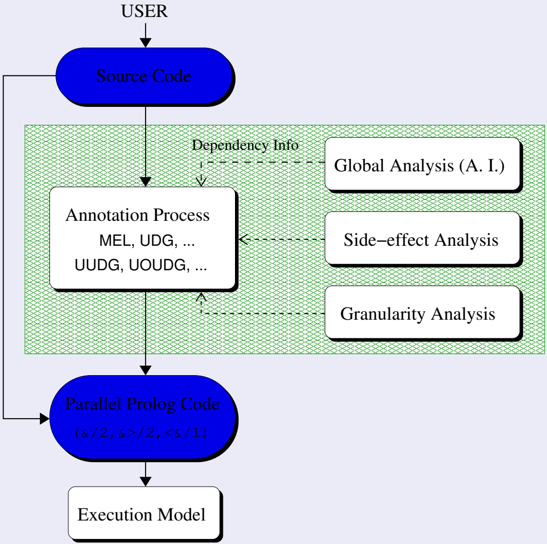
| | | | |
|---|---|---|---|
| $s_1$<br>$s_2$ | Y := W+2;<br>X := Y+Z; | (+ (+ W 2)<br>Z) | Y = W+2,<br>X = Y+Z, |
| | **Imperative** | **Functional** | **CLP** |

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

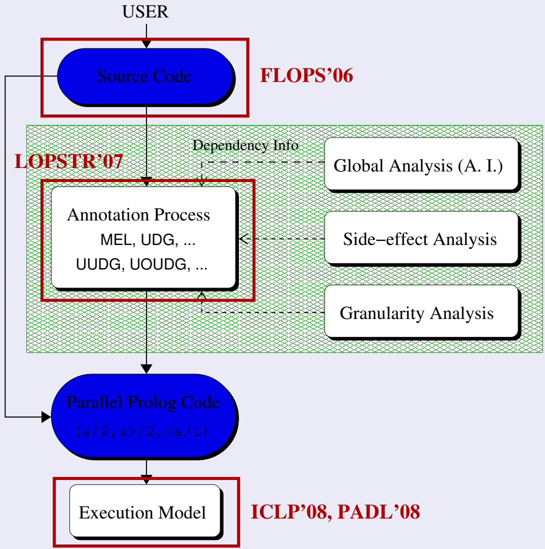
- Fundamental issue: `p` *affects* `q` (prunes its choices).
  - `q` ahead of `p` is *speculative*.
- **Independence:** *correctness* + *efficiency*.

THE UNIVERSITY of NEW MEXICO

# Architecture of parallelizing compiler



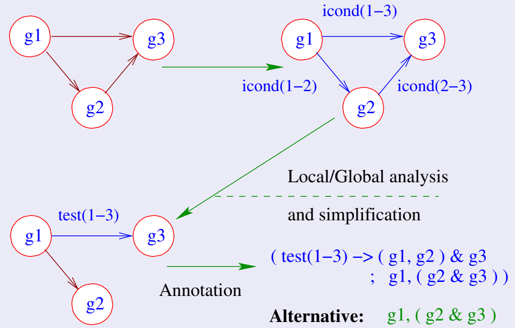
# Architecture of parallelizing compiler



## CDG-based automatic parallelization

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

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



## An alternative, more flexible source code annotation

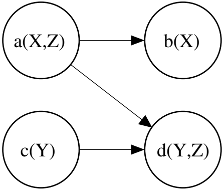
- Classical parallelism operator &/2: nested fork-join.
  - Rigid structure of &/2.
- However, more flexible constructions can be used to denote parallelism:
  - G &> $H_G$ — schedules goal G for parallel execution and continues executing the code after G &> $H_G$.
    - $H_G$ is a *handler* which contains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




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

**Sequential**

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

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

**Restricted IAP**

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

**Unrestricted IAP**

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

THE UNIVERSITY of NEW MEXICO

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

## 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 | B | C)`, `(Cond1 ? V1)`, `(Cond1 ? V1 | V2)`.
  - *Quoting*: `pair(A,B) := ^(A-B).`
  - *Laziness*: via `:- lazy`.

THE UNIVERSITY of NEW MEXICO

## 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 | B | C)`, `(Cond1 ? V1)`, `(Cond1 ? V1 | V2)`.
  - *Quoting*: `pair(A,B) := ^(A-B).`
  - *Laziness*: via `:- lazy`.

### Example (FibFun: parallel transformation)

```
fib(0) := 0.
fib(1) := 1.
fib(N) := fib(N-1) + fib(N-2)
       :- int(N), N > 1.
```

```
?- Y = ~fib(10).
   Y = 55.
?- 55 = ~fib(X).
   X = 10.
```

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

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

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

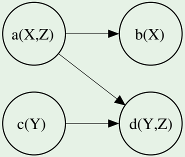
- Remember: `&/2` vs. `&>/2` + `<&/1`.
- Main idea: [Published at **LOPSTR'07**. Submitted to **TPLP**]
  - *Publish goals* (e.g., `G &> H`) as soon as possible.
  - *Wait for results* (e.g., `H <&`) as late as possible.
  - One clause at a time.
- Limits to how soon a goal is published + how late results 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)



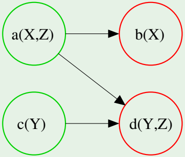

| Indep | Dep | pvt | ToPub | ToWait | Pub |
|---|---|---|---|---|---|
| | | | | | ∅ |
| | | | | | |
| | | | | | |
| | | | | | |

## 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)




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

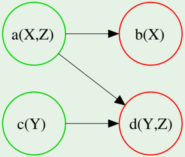
```
p(X,Y,Z) :-
        c(Y) &> Hc,
        a(X,Z) &> Ha,
        Ha <&,
```

# 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)




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

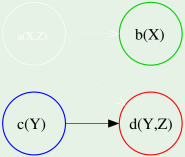
```
p(X,Y,Z) :-
        c(Y) &> Hc,
        a(X,Z),
```

# Automatic parallelization with alternative primitives

## Non order-preserving, unrestricted annotation (I)

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

### Example (Unrestricted Annotation UUDG)




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

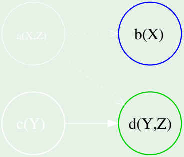
```
p(X,Y,Z) :-
        c(Y) &> Hc,
        a(X,Z),
        b(X) &> Hb,
        Hc <&,
```

# Automatic parallelization with alternative primitives

## Non order-preserving, unrestricted annotation (I)

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

### Example (Unrestricted Annotation UUDG)




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

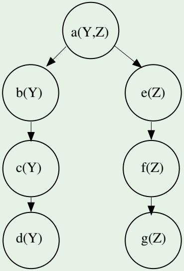
```
p(X,Y,Z) :-
        c(Y) &> Hc,
        a(X,Z),
        b(X) &> Hb,
        Hc <&,
        d(Y,Z),
        Hb <&.
```

- Goal order switched w.r.t. sequential version.

# Automatic parallelization with alternative primitives

## Order-preserving, unrestricted annotation (II)

### Example (Unrestricted Annotation UUDG)

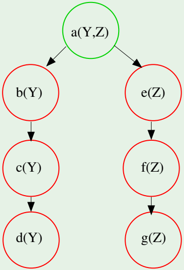



| Indep | Dep | pvt | ToPub | ToWait | Pub |
|---|---|---|---|---|---|
| | | | | | ∅ |
| | | | | | |
| | | | | | |

# Automatic parallelization with alternative primitives

## Order-preserving, unrestricted annotation (II)

### Example (Unrestricted Annotation UUDG)



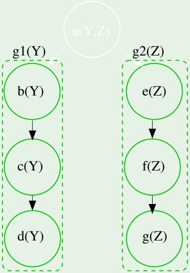

| Indep | Dep | pvt | ToPub | ToWait | Pub |
|---|---|---|---|---|---|
| | | | | | $\emptyset$ |
| $\{a\}$ | $\{b, e\}$ | $b$ | $\{a\}$ | $\{a\}$ | $\{a\}$ |
| | | | | | |

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

# Automatic parallelization with alternative primitives

## Order-preserving, unrestricted annotation (II)

### Example (Unrestricted Annotation UUDG)



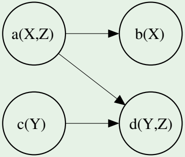

| Indep | Dep | pvt | ToPub | ToWait | Pub |
|---|---|---|---|---|---|
| | | | | | $\emptyset$ |
| $\{a\}$ | $\{b, e\}$ | $b$ | $\{a\}$ | $\{a\}$ | $\{a\}$ |
| $\{\mathbf{g1}, \mathbf{g2}\}$ | $\emptyset$ | – | $\{\mathbf{g1}, \mathbf{g2}\}$ | $\emptyset$ | $\{a, \ldots, g\}$ |

```
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 UODG)

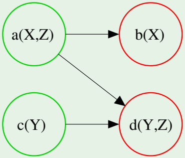



| Indep | Dep | pvt | ToPub | ToWait | Pub |
|---|---|---|---|---|---|
| | | | | | $\emptyset$ |
| | | | | | |
| | | | | | |

# Automatic parallelization with alternative primitives

## Order-preserving, unrestricted annotation

### Example (Unrestricted Annotation UOUDG)



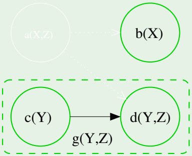

| Indep | Dep | pvt | ToPub | ToWait | Pub |
|---|---|---|---|---|---|
| | | | | | $\emptyset$ |
| $\{a, c\}$ | $\{b, d\}$ | $b$ | $\{a\}$ | $\{a\}$ | $\{a\}$ |
| | | | | | |

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

# Automatic parallelization with alternative primitives

## Order-preserving, unrestricted annotation

### Example (Unrestricted Annotation UOUDG)




| Indep | Dep | pvt | ToPub | ToWait | Pub |
|---|---|---|---|---|---|
| | | | | | ∅ |
| {a, c} | {b, d} | b | {a} | {a} | {a} |
| {b, **g**} | ∅ | – | {b, **g**} | ∅ | {a, b, c, d} |

```
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

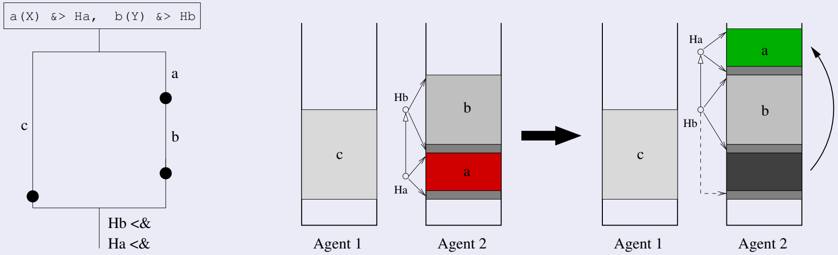
- Versions of and-parallelism previously implemented:
  - &-Prolog, &-ACE, AKL, Andorra-I,...
- They rely on complex low-level machinery:
  - Each agent: new WAM instructions, goal stack, parcall frames, markers, etc.
- Approach: rise components to the source language level: [Published at **ICLP'08** 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.
```



## Creation of (high-level) markers

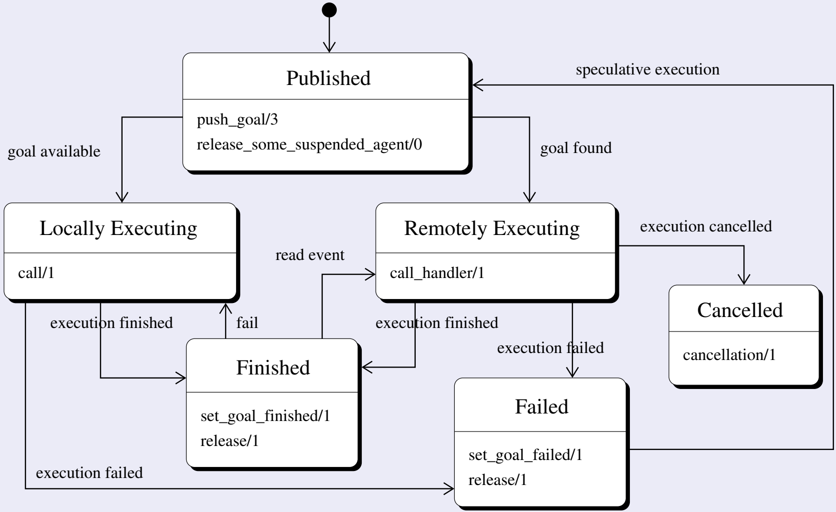
### Execution of parallel goal

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

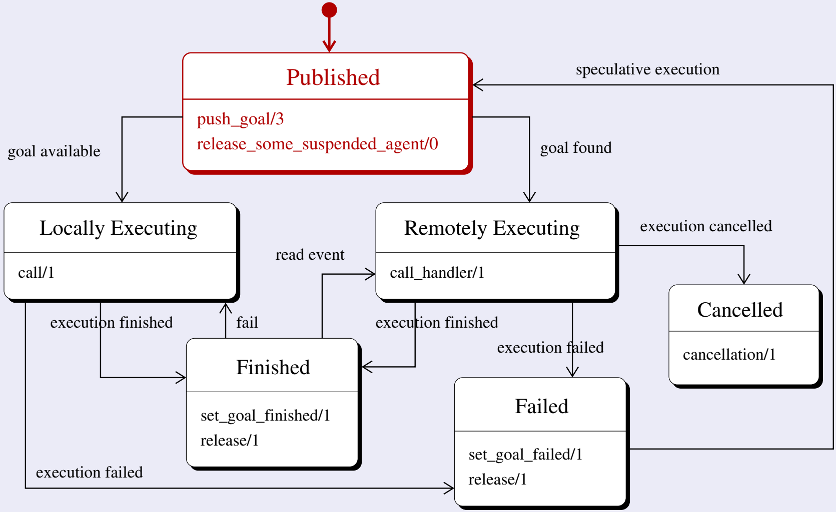
```
remote_call(Handler) :-
    set_goal_failed(Handler),
    release(Handler),
    metacut_garbage_slots(Handler),
    fail.
```

- Library of concurrency primitives to implement a high-level approach to IAP.
    - Better programming discipline ⇒ easier to maintain!

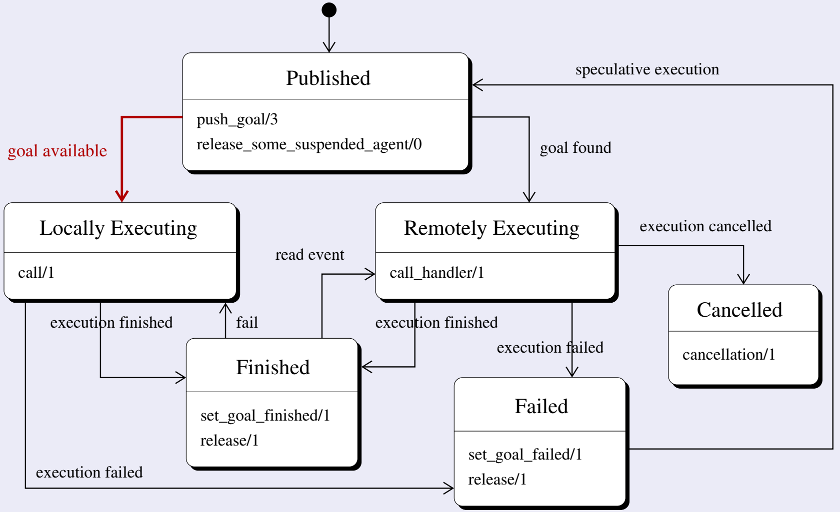
## State diagram of a parallel goal



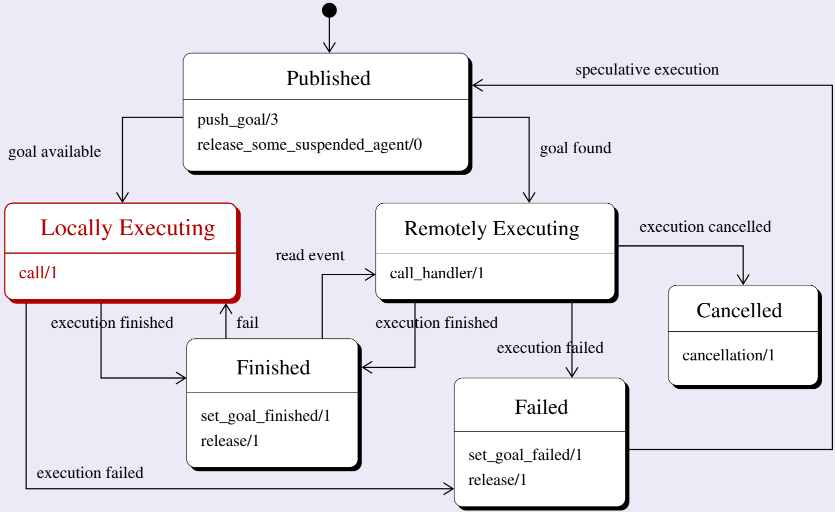
## State diagram of a parallel goal



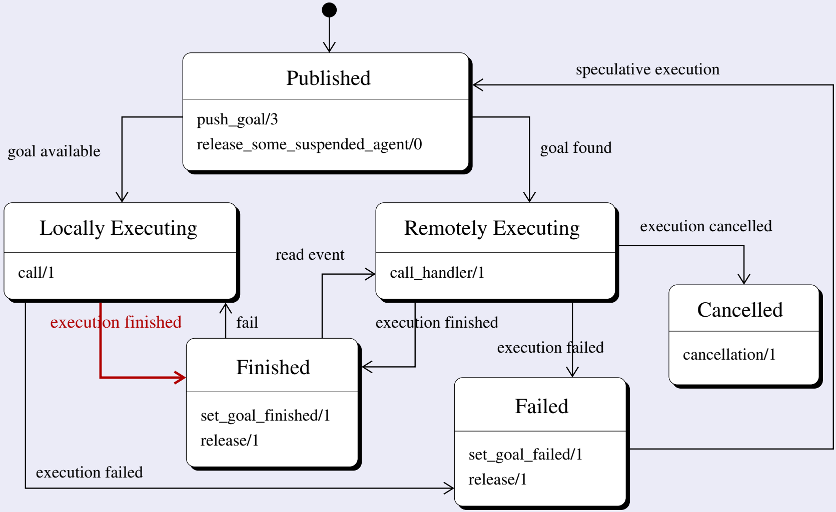
# State diagram of a parallel goal



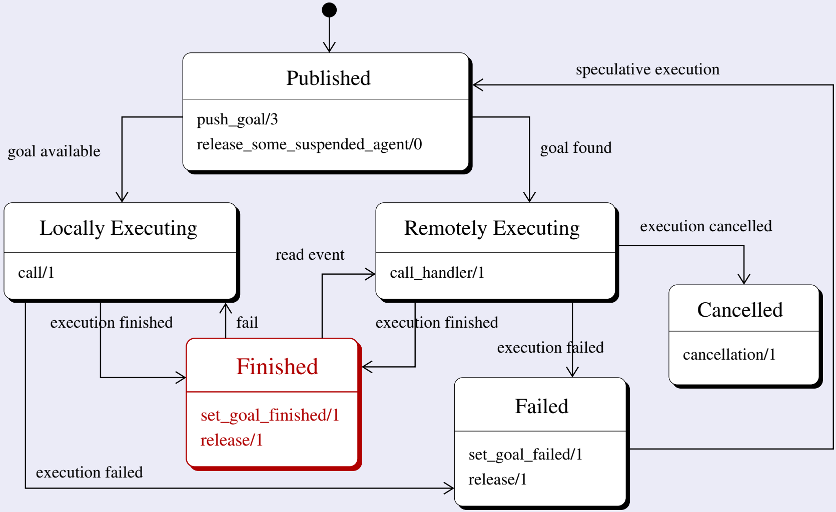
# State diagram of a parallel goal



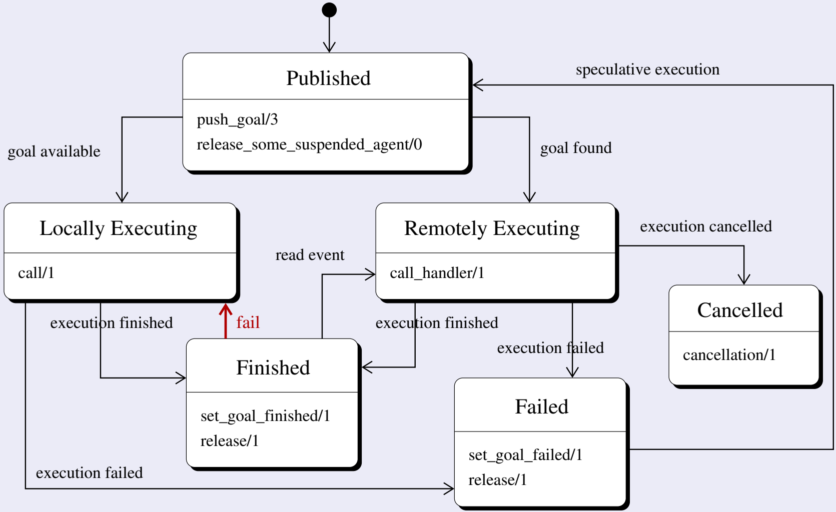
## State diagram of a parallel goal



# State diagram of a parallel goal

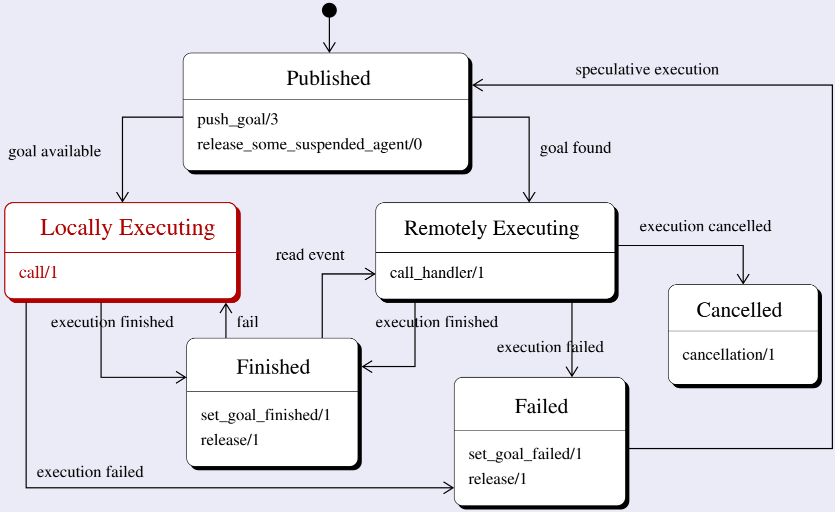


# State diagram of a parallel goal

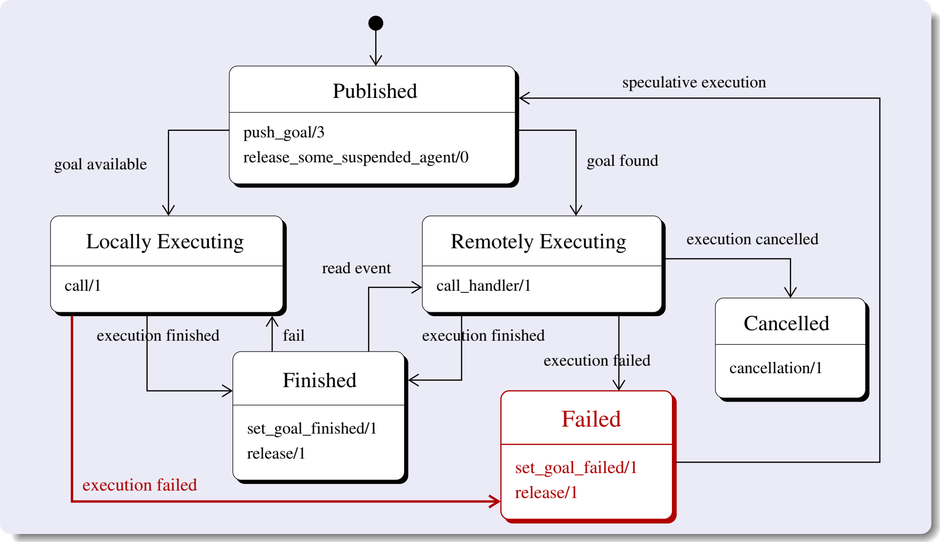



THE UNIVERSITY of NEW MEXICO

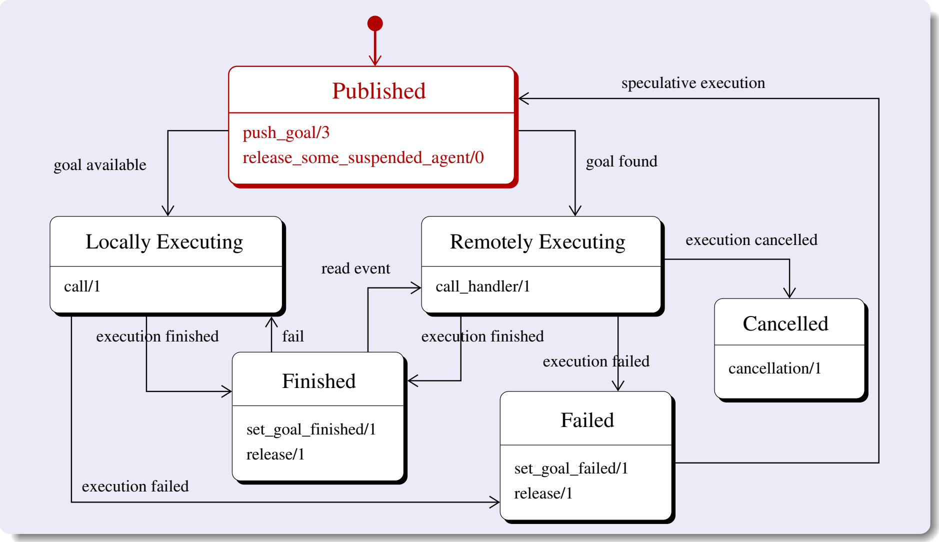
# State diagram of a parallel goal



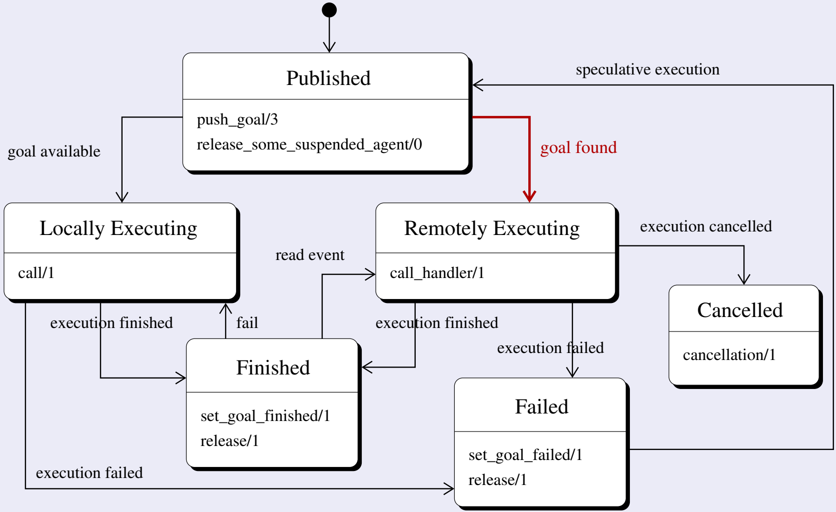
## State diagram of a parallel goal



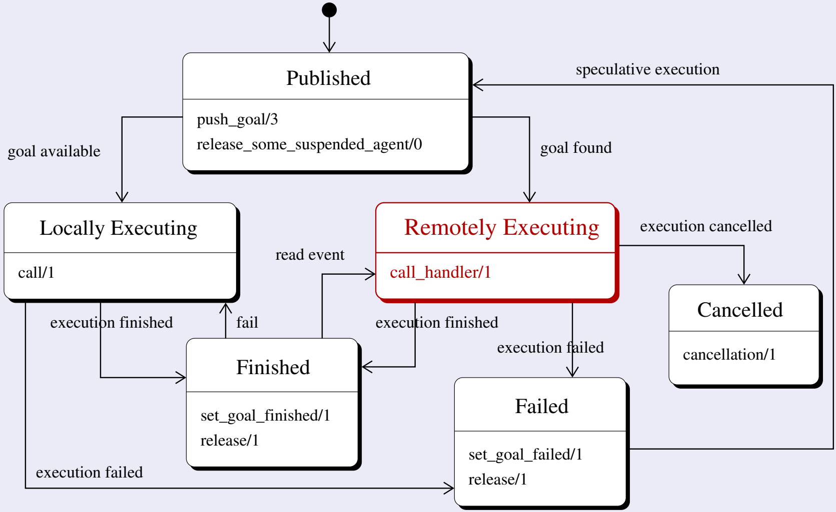
## State diagram of a parallel goal



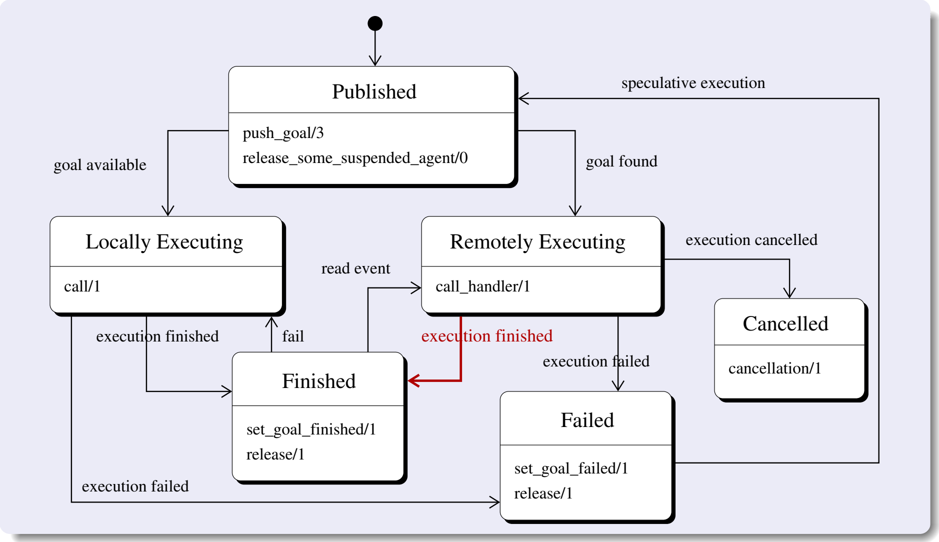
## State diagram of a parallel goal



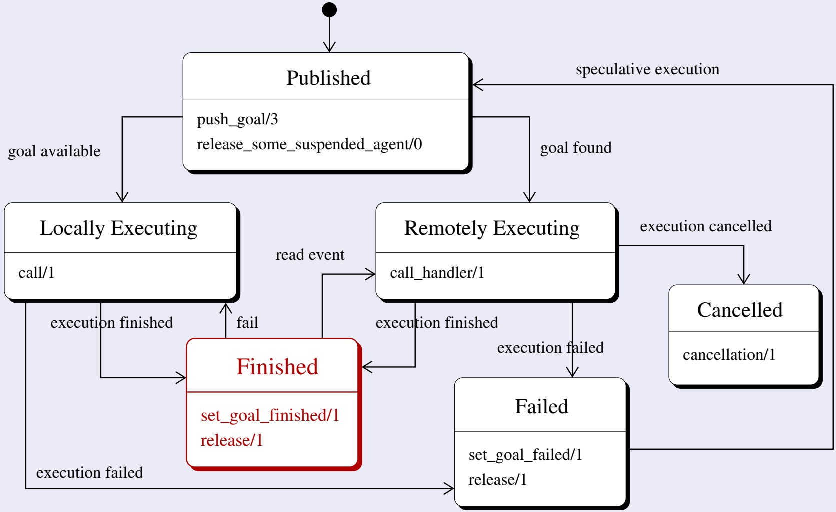
# State diagram of a parallel goal



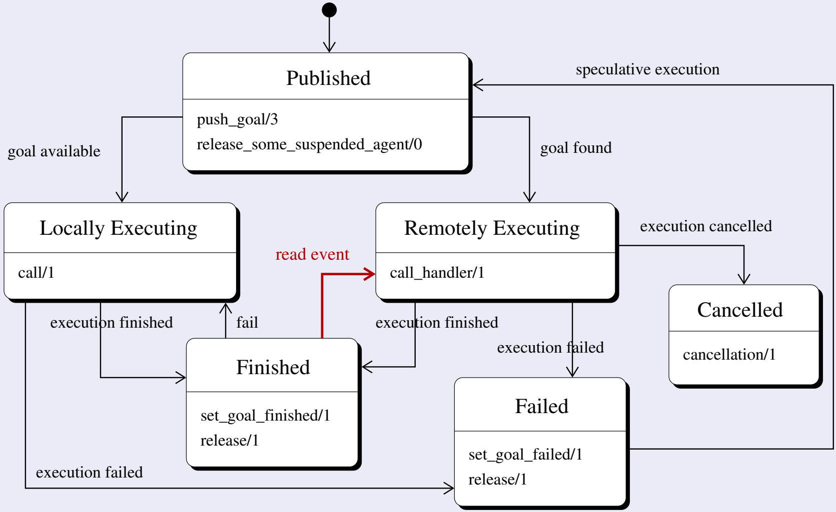
# State diagram of a parallel goal



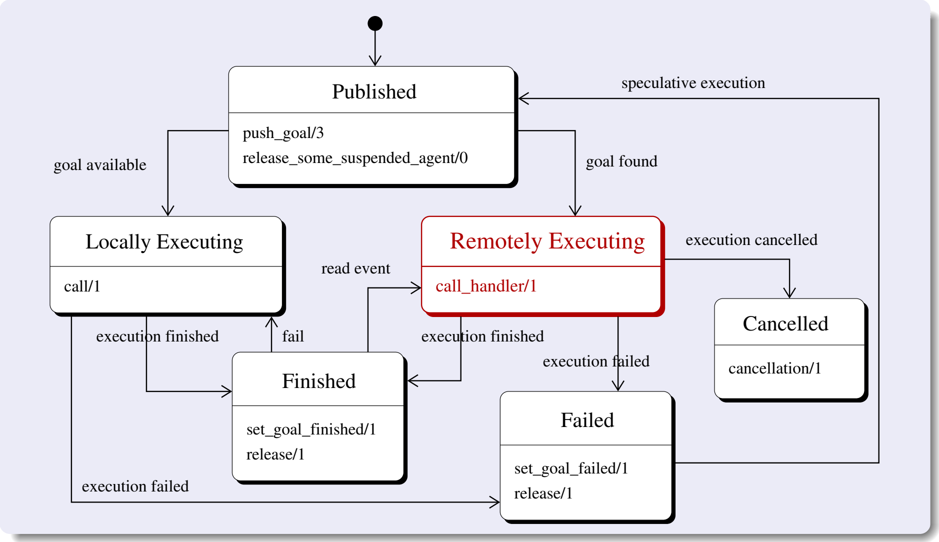
## State diagram of a parallel goal



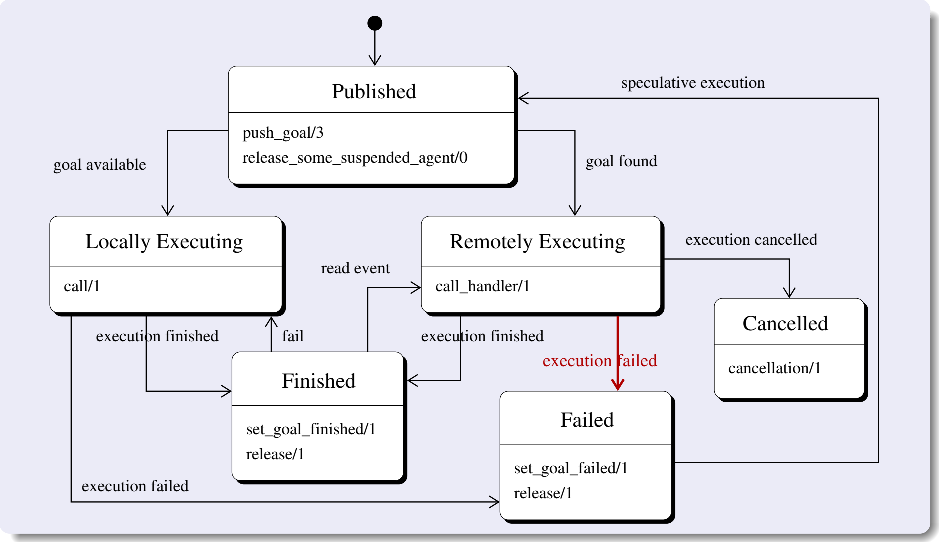
# State diagram of a parallel goal



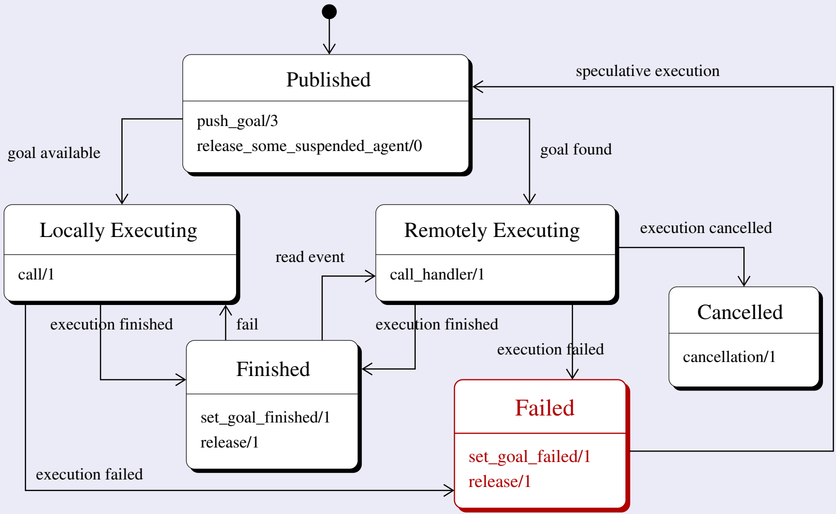
## State diagram of a parallel goal



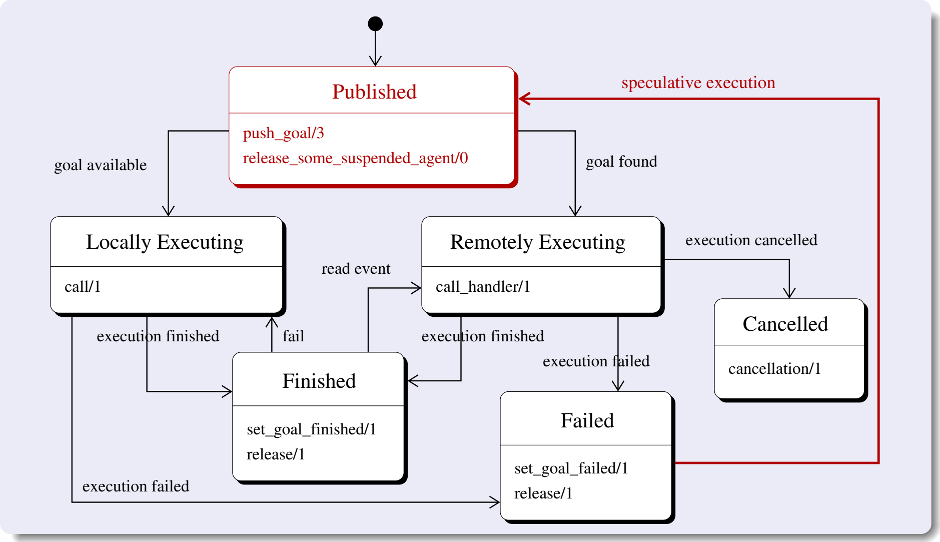
## State diagram of a parallel goal



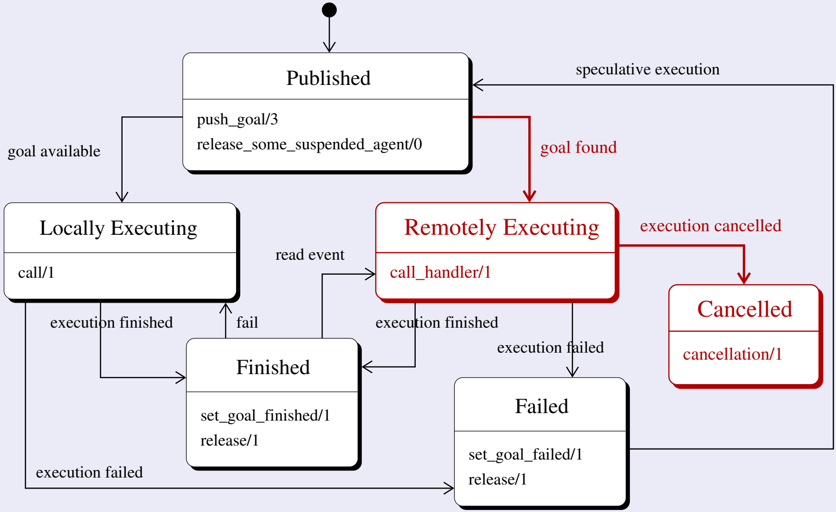
# State diagram of a parallel goal



# State diagram of a parallel goal



# State diagram of a parallel goal



# Performance results

## Restricted vs. unrestricted parallelization

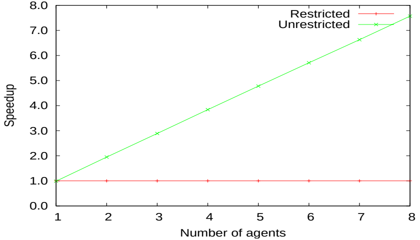
- *Sun Fire T2000:*
  - 8 cores and 8 Gb of memory, each of them capable of running 4 threads in parallel.
    - ★ Speedups with more than 8 threads stop being linear even for completely independent computations, since threads in the same core compete for shared resources.
  - All performance results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

| Benchmark | And-Par. | Number of processors | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| FibFun | *Restricted* | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 | 1.00 |
| | *Unrestricted* | 0.99 | 1.95 | 2.89 | 3.84 | 4.78 | 5.71 | 6.63 | **7.57** |
| FFT | *Restricted* | 0.98 | 1.76 | 2.14 | 2.71 | 2.82 | 2.99 | 3.08 | 3.37 |
| | *Unrestricted* | 0.98 | 1.82 | 2.31 | 3.01 | 3.12 | 3.26 | 3.39 | 3.63 |
| Hamming | *Restricted* | 0.93 | 1.13 | 1.52 | 1.52 | 1.52 | 1.52 | 1.52 | 1.52 |
| | *Unrestricted* | 0.93 | 1.15 | 1.64 | 1.64 | 1.64 | 1.64 | 1.64 | 1.64 |
| Takeuchi | *Restricted* | 0.88 | 1.61 | 2.16 | 2.62 | 2.63 | 2.63 | 2.63 | 2.63 |
| | *Unrestricted* | 0.88 | 1.62 | 2.39 | 3.33 | 4.04 | 4.47 | 5.19 | **5.72** |

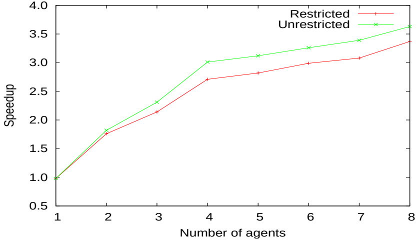
THE UNIVERSITY *of* NEW MEXICO

# Performance results

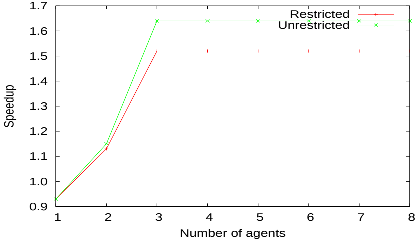
## Restricted vs. unrestricted parallelization



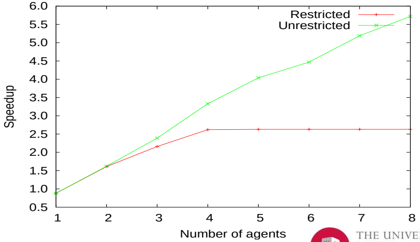

(a) FibFun




(b) FFT




(c) Hamming




(d) Takeuchi

THE UNIVERSITY of NEW MEXICO

# Performance results

## Deterministic vs. Non-deterministic annotation

| Benchmark | Op. | Number of processors | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| AIAKL | &! | 0.97 | 1.82 | 1.82 | 1.82 | 1.83 | 1.83 | 1.83 | 1.82 |
| | & | 0.96 | 1.70 | 1.71 | 1.72 | 1.74 | 1.75 | 1.72 | 1.72 |
| Ann | &! | 0.98 | 1.86 | 2.72 | 3.56 | 4.38 | 5.16 | 5.88 | 6.64 |
| | & | 0.96 | 1.85 | 2.72 | 3.57 | 4.35 | 5.14 | 5.87 | 6.61 |
| Deriv | &! | 0.91 | 1.63 | 2.37 | 3.05 | 3.78 | 4.49 | 4.98 | 5.49 |
| | & | 0.84 | 1.60 | 2.34 | 2.99 | 3.73 | 4.43 | 4.56 | 4.85 |
| FFT | &! | 0.98 | 1.82 | 2.31 | 3.01 | 3.12 | 3.26 | 3.39 | 3.63 |
| | & | 0.98 | 1.72 | 1.97 | 2.65 | 2.67 | 2.75 | 2.93 | 2.97 |
| Hanoi | &! | 0.89 | 1.76 | 2.47 | 3.32 | 3.77 | 4.17 | 4.61 | 5.25 |
| | & | 0.89 | 1.77 | 1.91 | 2.84 | 3.13 | 3.54 | 3.96 | 4.47 |
| MMatrix | &! | 0.91 | 1.74 | 2.55 | 3.32 | 4.18 | 4.83 | 5.55 | 6.28 |
| | & | 0.90 | 1.48 | 2.16 | 2.88 | 3.51 | 4.13 | 4.71 | 5.25 |
| QuickSort | &! | 0.97 | 1.78 | 2.31 | 2.87 | 3.19 | 3.46 | 3.67 | 3.75 |
| | & | 0.97 | 1.71 | 2.17 | 2.43 | 2.60 | 2.93 | 3.06 | 3.19 |
| Takeuchi | &! | 0.88 | 1.62 | 2.39 | 3.33 | 4.04 | 4.47 | 5.19 | 5.72 |
| | & | 0.88 | 1.45 | 2.02 | 2.85 | 3.41 | 3.80 | 4.23 | 4.66 |

THE UNIVERSITY of NEW MEXICO

# Performance results

## Non-deterministic benchmarks

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

| Benchmark | Number of processors | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |
| Chat | 2.31 | 4.49 | 5.42 | 6.91 | 9.79 | 9.95 | 11.10 | 17.29 |
| Numbers | 1.84 | 1.79 | 1.79 | 1.79 | 1.79 | 1.79 | 1.78 | 1.78 |
| Progeom | 0.99 | 0.96 | 0.97 | 0.98 | 0.98 | 0.98 | 0.98 | 0.98 |
| Queens | 0.99 | 0.94 | 0.94 | 0.94 | 0.94 | 0.94 | 0.94 | 0.94 |
| QueensT | 0.99 | 1.90 | 2.41 | 3.18 | 4.71 | 4.61 | 4.58 | 4.57 |

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

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

## 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

- A. Casas, M. Carro, M. Hermenegildo. **A High-Level Implementation of Non-Deterministic Unrestricted Independent And-Parallelism**. The $24^{th}$ Int'l. Conf. on Logic Programming (*ICLP'08*). Dec. 2008.
- A. Casas, M. Carro, M. Hermenegildo. **Towards a High-Level Implementation of Execution Primitives for Unrestricted, Independent And-Parallelism**. The $10^{th}$ Int'l. Symp. on Practical Aspects of Declarative Languages (*PADL'08*). Jan. 2008.
- A. Casas, M. Carro, M. Hermenegildo. **Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs**. The $17^{th}$ Int'l. Symp. on Logic-Based Program Synthesis and Transformation (*LOPSTR'07*). Aug. 2007.
- A. Casas, D. Cabeza, M. Hermenegildo. **A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems**. $8^{th}$ Int'l. Symp. on Functional and Logic Programming (*FLOPS'06*). Apr. 2006.

  - 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*).

THE UNIVERSITY of NEW MEXICO