Automatic Unrestricted Independent And-Parallelism in Logic Programs

Amadeo Casas, Manuel V. Hermenegildo
Depts. of Comp. Science and Electr. and Comp. Eng., Univ. of New Mexico, USA.
(*e-mail:* {amadeo,herme}@{cs,ece}.unm.edu)

Manuel Carro, Manuel V. Hermenegildo
School of Computer Science, T. U. Madrid (UPM), Spain.
(*e-mail:* {mcarro,herme}@fi.upm.es)

submitted 1 January 2003; revised 1 January 2003; accepted 1 January 2003

Abstract

We present new algorithms which perform automatic parallelization via source-to-source transformations. The objective is to exploit goal-level, *unrestricted* independent and-parallelism. The proposed algorithms use as targets new parallel execution primitives which are simpler and more flexible than the well-known `&/2` parallel operator, which makes it possible to generate better parallel expressions by exposing more potential parallelism among the literals of a clause than is possible with `&/2`. The main differences between the algorithms stem from whether the order of the solutions obtained is preserved or not, and on the use of determinacy information. We briefly describe the environment where the algorithms have been implemented and the runtime platform in which the parallelized programs are executed. We also report on an evaluation of an implementation of our approach. We compare the performance obtained to that of previous annotation algorithms and show that relevant improvements can be obtained.

KEYWORDS: Logic Programming, Automatic Parallelization, And-Parallelism, Program Transformation.

1 Introduction

Parallelism capabilities are becoming ubiquitous thanks to the widespread use of multi-core processors. Indeed, most laptops on the market contain two cores (capable of running up to four threads simultaneously) and single-chip, 8-core servers are now in widespread use. Furthermore, the trend is that the number of on-chip cores will double with each processor generation. In this context, being able to exploit such parallel execution capabilities in programs as easily as possible becomes more and more a necessity. However, it is well-known (Karp and Babb 1988) that parallelizing programs is a hard challenge. This has renewed interest in language-related designs and tools which can simplify the task of producing parallel programs.

The comparatively higher level of abstraction of declarative languages and, among them, logic programming languages and also the new multiparadigm languages

based on logic programming kernel languages, allows writing programs which are closer to the specification of the solution. Besides, there is often more freedom in the implementation of different operational semantics which respect the declarative semantics. In particular, the notion of control in declarative languages frequently is separated from the actual specification, which allows for more flexibility to arrange the evaluation order of some operations, including executing them in parallel if deemed convenient, without affecting the semantics of the original program. Additionally, the cleaner semantics that declarative programs pose and the use of logic variables, which can be assigned only one value and thus there is no need to look after flow dependencies, makes it possible to automatically detect more accurately any lack of dependencies among operations and hence to exploit opportunities for parallelism more easily than in imperative languages, increasing the performance through parallel execution on multicore architectures (including multicore embedded systems). At the same time, in most other respects in the case of logic programs the presence of dynamic data structures with "declarative pointers" (logical variables), irregular computations, or complex control makes the parallelization of logic programs a particularly interesting case that allows tackling the more complex parallelization-related challenges in a formally simple and well-understood context (Hermenegildo 2000).

Because of this potential, quite significant progress has been made in automatic parallelization for logic programming (Gupta et al. 2001), where two main forms of parallelism have been studied. *Or-parallelism* is exploited when the alternatives created by non-deterministic goals are explored simultaneously by different processors, in order to reduce the time taken to traverse their (possibly large) search space. The exploitation of this type of parallelism is interesting in applications that involve extensive seach, since the choices that are represented by alternative clauses usually involve a large number of steps before a failure or a success in the search occurs. Some relevant or-parallelism systems are Aurora (Lusk et al. 1990) and MUSE (Ali and Karlsson 1990).

An alternative strategy that is used to parallelize logic programs is referred to as *and-parallelism*, which aims at executing simultaneously conjunctive goals in clauses or in the resolvent, in a similar fashion as in traditional parallelism exploited. While or-parallelism can only obtain speedups when there is search involved, and-parallelism can be used in more algorithmic schemes, with loop parallelization and divide-and-conquer and map-style algorithms being classic representatives. Examples of systems that have exploited and-parallelism are ROPM (Kalé 1987), AO-WAM (Butler et al. 1986), DDAS (Shen 1996) and &-Prolog (Hermenegildo and Greene 1991). Additionally, some systems such as ACE (Gupta et al. 1994), AKL (Janson 1994), and Andorra (Santos-Costa et al. 1991) (Santos-Costa 1993) exploit certain combinations of both and- and or-parallelism. In this paper, we concentrate on the issue of automatic and-parallelization.

The main of objective of a parallelizing compiler is to uncover as much parallelism as possible, but always preserving some conditions to guarantee that the set of solutions obtained is the same one as in the sequential execution and that there is not a decrease in the performance of the execution, i.e., the parallel ex-

ecution is never slower than the sequential execution. Thus, a correct parallelization has been traditionally defined as one that preserves during and-parallel execution some key properties, which are typically correctness and efficiency (i.e., no-slowdown) (Hermenegildo and Rossi 1995). The preservation of these properties is ensured by executing in parallel goals which meet some (non-unique) notion of *independence*, meaning that the goals to be executed in parallel do not interfere with each other in some particular sense. This can include, for instance, absence of competition for binding variables among goals to be run in parallel plus other considerations such as, e.g., absence of side effects. For simplicity, in the rest of the paper we will assume that we are only dealing with side-effect free program sections. Note however that this does not affect the generality of our presentation, as we deal with dependencies in a generic way.

One of the best understood sufficient conditions for ensuring that goals meet the efficiency and correctness criteria for parallelization is called *strict independence* (Hermenegildo and Rossi 1995), which entails the absence of shared variables at runtime between any two literals being parallelized. It should be noted that some proposals exploit and-parallelism between goals which do not meet this condition, but on which other restrictions are imposed which also ensure the no-slowdown property and absence of conflicts due to the binding of shared variables. An example of such restrictions is the *non-strict independence* (Hermenegildo and Rossi 1995), in which two goals share some variables, although there is no competition in their bindings. Although non-strict independence between two literals cannot be determined by inspecting the previous state of execution, and thus global analysis of the original program is required, it is quite interesting because it uncovers some of the parallelism that is present in applications that manipulate open data structures, as for instance difference lists. Another example is the Basic Andorra Model (Santos-Costa et al. 1991), which makes use of determinism information in order to decide whether two goals are to be executed in parallel or not, since two computations that have no alternatives to execute, and its execution will never fail, are independent and can thus be executed in parallel. In addition, an interesting issue is at what level of granularity the notion of independence is applied: at the goal level, at the binding level, etc. Our work in this paper will focus on *goal-level* (strict and non-strict) independent and-parallelism.

One particularly successful approach to automatically parallelize a logic program uses three different stages (Hermenegildo and Warren 1987; Bueno et al. 1999; Gupta et al. 2001). The first one explores the literals in the original clause looking for candidates for parallel execution by detecting data and control dependencies between pairs of those literals. A directed dependency graph (see Figure 1(b) as an example) is then built to capture this extracted information. The nodes in the graph correspond to literals in the body of the clause and the edges represent dependence conditions between them. Edges are labeled with the associated dependence conditions (which may be trivially *true* or *false*). As a second stage, a global analysis (Bueno et al. 1999) needs to be run in order to gather information regarding, e.g., variable aliasing, groundness and side effects, in order to prove statically whether those dependence conditions are statically true or false. Those edges which

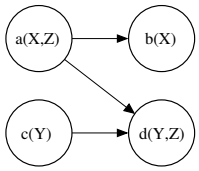
represent a dependence condition which is *true* are eliminated from the graph, since the two literals are independent. If an edge represents a condition which is *false* then it will be left in the graph as an unconditional edge, since the two literals are dependent. For the rest of the edges in the graph, when a condition cannot be completely evaluated at compile-time, it may remain associated to the edge, but possibly in a simplified form.

Finally, the third stage corresponds to the annotation process, which encodes the resulting dependency graph in the target parallel language. This annotation should respect the dependencies found in the original program while, at the same time, exploiting as much parallelism as possible. Several algorithms based on different heuristics have been proposed to compile the dependency graph into parallel code (Muthukumar et al. 1999; Muthukumar and Hermenegildo 1990; Bueno et al. 1994; DeGroot 1987) using fork-join structures. In this process, labeled edges result in run-time checks when conditional parallel expressions are allowed. Since the tasks to be parallelized may not represent an enough amount of computation with respect to the overhead that is incurred when the run-time check is evaluated, unresolved dependencies are sometimes assumed to always hold, and parallel execution will be allowed only between literals which have been statically determined to be independent. This approach saves run-time checks at the expense of losing some parallelism.

This annotation process is the focus of this paper. We will present and evaluate new annotation algorithms which target and-parallelism primitives which can express richer dependency graphs than those which can be encoded with the *nested fork-join* approaches. Our hope is that since the transformed programs will contain in some cases more parallelism, we will be able to obtain better speedups than the *fork-join* variants for such cases, using the proposed approach. Limitations of the fork-join annotations have been previously studied for imperative languages (Sarkar 1990), in which the reordering of the instructions was studied to expose the maximum amount of parallelism, but the non-deterministic nature of logic programs was not covered. Two different algorithms will be provided: the first one preserves the order of the solutions with respect to the sequential program, and the second may change this order if more parallelism can be exploited. This second algorithm has clearly fewer restrictions to satisfy, and thus we expect to obtain a better performance with it. Additionally, these algorithms will benefit from the use of determinism information, both by using specialized versions of the parallelism primitives when the literals are detected to be deterministic and by reording literals, when that is allowed.

The rest of this paper is organized as follows. Section 2 motivates the use of the more flexible operators by presenting the limitations of the traditional fork-join operator. The annotation algorithms are presented in Section 3, in addition to a discussion of the improvements that can be performed when determinism information is available. Section 4 then briefly describes our experimental framework and shows the performance results that we obtained, comparing them with the result of some of the previous annotation algorithms. Finally, Section 5 presents our conclusions and discusses future work.

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



(a) Predicate *p/3*. (b) Dependency graph.

Fig. 1. Predicate `p/3` and its associated dependency graph.

2 Motivation for Unrestricted Independent And-Parallelism

We will first introduce the well-known `&/2` operator for parallelism and its limitations. Regardless of the annotation algorithm used, annotations using `&/2` have to give up parallelizing some goals due to the somewhat rigid structure that this operator imposes on the final program. We will show how better annotations for parallelism are possible when other, simpler primitives, are used.

2.1 Fork-Join-*Style Parallelization*

The &-Prolog language has been the vehicle for expressing goal-level restricted independent and-parallelism in logic programs. A simplified grammar (i.e., without cuts, built-ins and side-effects) that defines the syntax of *restricted* &-Prolog programs follows.

Definition 1 (*Restricted &-Prolog grammar*)
Let $\overrightarrow{t}$ be a tuple of terms and p a predicte symbol. Then the following grammar defines the set of valid sentences in the restricted &-Prolog language:

Program ::= Clause `.` Program | ε
Clause ::= Literal | Literal `:-` Body
Body ::= Literal | Literal `,` Body | Body `->` Body `;` Body | ParExp
ParExp ::= Body `&` Body
Literal ::= $p(\overrightarrow{t})$

The restricted &-Prolog language (Hermenegildo and CLIP Group 1994) is basically an extension of Prolog, in which parallel expressions (i.e., *ParExp* in the grammar above) that make use of a nested parallel fork-join operator `&/2` are added. Essentially, the sequential comma is replaced by the `&/2` operator, in order to mark which goals can be executed in parallel.

The other addition to the restricted &-Prolog language corresponds to the syntactic sugar expression for if-then-else constructions (conds `->` body1 `;` body2), useful to perform different computations depending on the result of evaluating some runtime checks.

In order to show the limitations of the fork-join operator `&/2`, we will use as running example the predicate `p/3` in Figure 1(a). We will assume that the dependencies detected between the literals in the predicate `p/3` are `dep(a,b)`, `dep(a,d)`,

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

(a) *fj1*: Order-preserving

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

(b) *fj2*: Non-order-preserving

Fig. 2. *Fork-Join* annotations for `p/3`.

`dep(c,d)`, where we denote by `dep(X,Y)` that `Y` depends on (and therefore must be executed before) `X`. The conditional dependency graph for the predicate `p/3` is shown in Figure 1(b). The vertices V correspond to the literals of the clause and there exists an edge between two literals L_i and L_j in E if $ind(L_i, L_j) \neq true$ (i.e., the literals L_i and L_j are dependent and thus the literal L_i has to be completed before the literal L_j), where ind is the notion of independence. As mentioned before, this information is obtained in our case from global data-flow analysis (Bueno et al. 1999).

For the sake of simplicity, we will assume in the rest of the paper that all the dependencies are unconditional —i.e., conditional dependencies are assumed to be always false, reducing the use of if-then-else constructions. This brings simplicity and avoids potentially costly run-time checks in the parallelized code at the expense of having fewer opportunities for parallelism. However, it has been experimentally found to be a good compromise (Muthukumar et al. 1999; Bueno et al. 1999) between the degree of parallelism uncovered and the execution time of independence tests.

Conjunctive parallel execution has traditionally been denoted using the `&/2` operator instead of the sequential comma (','). The former binds more tightly than the latter. Thus, the expression "`a(X,Z), b(X) & c(Y), d(Y,Z)`" means that literals `b/1` and `c/1` can be safely executed in parallel after the execution of literal `a/2` finishes. When both `b/1` and `c/1` have successfully finished, execution continues with `d/2`.

While this single operator is enough to convert the dependency graph back to a parallel expression in source, the class of dependencies it can express directly (i.e., dependency graphs with a nested fork-join structure) is a subset of that which can possibly appear in a program (Muthukumar et al. 1999). This makes parallelism opportunities to be inevitably lost in cases with a complex enough structure (e.g., that in Figure 1(b)). Likewise, inter-procedural parallelism (i.e., parallel conjunctions which span literals in different predicates) cannot be exploited without program transformation.

In general, several annotations are possible for a given clause, and several annotation algorithms have been proposed so far (Muthukumar et al. 1999; Cabeza 2004) which use the `&/2` operator as the most basic construction to express parallelism between goals. These annotators produce clauses that are parallelized differently. It is in principle possible to statically decide (or, at least, approximate) whether some annotation is better than some other, for example by using the number of goals annotated for parallelism in a clause or, more interestingly, by using information regarding the expected runtime of goals (see, e.g., (Mera et al. 2007; López-García

et al. 1996) and its references). However, finding an optimal solution is a computationally expensive combinatorial problem (Muthukumar et al. 1999) and, in practice, since there are many decisions to take which make the number of possible annotations explode, annotators use heuristics which may be more or less appropriate in concrete cases. These heuristics are part of the difference between these annotators.

As an example, Figure 2 shows two of the possible annotations for our running example. Some other possible parallelizations, as for instance `p :- a(X, Z), b(X) & c(Y), d(Y, Z)`, have been left out of Figure 2, since it would not add anything to the discussion as it would not change the comparison we make in Section 2.2. Some goals appear switched with respect to their order in the sequential clause. This respects the dependencies in Figure 1(b), which reflects a valid notion of parallelism (i.e., if solution order is not important). If additional ordering requirements are needed (due to, e.g., side effects or impurity), these could appear as additional edges in the graph. Note that none of the annotations in Figure 2 fully exploits all parallelism available in Figure 1(b): Figure 2(a) misses the possible parallelism between literals `b/1` and `d/2`, and Figure 2(b) misses the parallelism between literals `b/1` and `c/1`.

One relevant question is which of these two parallelizations is better. Arguably, a meaningful measure of their quality is how long each of them takes to execute (intimately related to term-size computation (López-García and Hermenegildo 1995), which was traditionally used for granularity control). We will term those times T_{fj1} and T_{fj2} for Figures 2(a) and 2(b), respectively. This length depends on the execution times of the goals involved (i.e., T_a, T_b, T_c, T_d), which we assume to be non-zero. T_{fj1} and T_{fj2} are:

$$T_{fj1} = \max(T_a + T_b,\ T_c) + T_d \quad (1)$$

$$T_{fj2} = \max(T_a,\ T_c) + \max(T_b,\ T_d) \quad (2)$$

Comparing the quality of the annotations in Figure 2(a) and Figure 2(b) boils down to finding out whether it is possible to show that $T_{fj1} < T_{fj2}$ or the other way around. It turns out that both execution time expressions are non-comparable, since there are solutions for both orderings, so none of them is definitely better than the other one:

- $T_{fj1} < T_{fj2}$ holds if, for example, $T_a + T_b < T_c$, $T_d < T_b$, and then $T_{fj2} = T_b + T_c$, $T_{fj1} = T_d + T_c$, and
- $T_{fj2} < T_{fj1}$ holds if, for example, $T_c \leq T_a$, $T_d \leq T_b$, and then $T_{fj1} = T_a + T_b + T_d$, $T_{fj2} = T_a + T_b$.

2.2 Parallelization with Finer Goal-Level Operators

As previously explained, some and-parallel systems rely on the use of the fork-join operator (`&/2`) as the most basic construction to exploit parallelism between goals which are independent at run-time, because of the simplicity in which parallel

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

Fig. 3. dep-operator-annotated clause

computations can then be described. In this section, we present an extension of the restricted &-Prolog language introduced in Section 2.1:

Definition 2 (*Unrestricted &-Prolog grammar*)
Let $\overrightarrow{t}$ be a tuple of terms and p a predicte symbol. Then the following grammar defines the set of valid sentences in the unrestricted &-Prolog language:

Program ::= Clause `.` Program | ε
Clause ::= Literal | Literal `:-` Body
Body ::= Literal | Literal `,` Body | Body `->` Body `;` Body | ParExp
ParExp ::= Body `&` Body | Body `&>` Handler | Handler `<&`
Handler ::= Literal
Literal ::= $p(\overrightarrow{t})$

This new language will be referred to as the *unrestricted* &-Prolog language. It extends the parallel expressions used in the restricted &-Prolog language by adding two more basic constructions (Cabeza 2004; Cabeza and Hermenegildo 1996) to schedule goals for parallel executions, `&>/2` and `<&/1`, defined as follows:

Definition 3 (*Publish operator* `&>/2`)
`Goal &> H` schedules goal `Goal` for parallel execution and continues executing the code after `Goal &> H`. `H` is a *handler* which contains (or *points to*) the state of goal `Goal`.

Definition 4 (*Wait operator* `<&/1`)
`H <&` waits for the goal associated with `H` to finish, or executes it if it has not been taken by another thread yet. After that point any bindings made for the output variables of `Goal` are available to the executing thread.

With the previous definitions, the `&/2` operator can be written as:

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

This indicates that any parallelization performed using `&/2` can be made using `&>/2` and `<&/1` without loss of parallelism, i.e., no parallelism is necessarily lost when using `&>/2` and `<&/1`. We will term these operators *dep-operators* henceforth.

Two motivations justify the use of these operators instead of `&/2`. Firstly, their implementation is, in our experience, actually easier to devise and maintain than the monolithic `&/2` (Casas et al. 2008), and, secondly, the dep-operators allow more freedom to the annotator (and to the programmer, if parallel code is written by

`p(X, Y, Z):-`	
	$T_1 = 0$
`c(Y) &> Hc,`	
	$T_2 = T_1$
`a(X, Z),`	
	$T_3 = T_2 + T_a$
`b(X) &> Hb,`	
	$T_4 = T_3$
`Hc <&,`	
	$T_5 = \max(T_4,\ T_1 + T_c)$
`d(Y, Z),`	
	$T_6 = T_5 + T_d$
`Hb <&.`	
	$T_7 = \max(T_6,\ T_3 + T_b)$

Fig. 4. Deduction of execution time for unrestricted parallelization of `p/3`.

hand) to express data dependencies and, therefore, to extract more potential parallelism. We will now illustrate this last point, since the former is out of the scope of this paper.

Figure 3 shows an annotation of our running example using dep-operators. Note that this code allows executing in parallel `a/2` with `c/1`, `b/1` with `c/1`, and `b/1` with `d/2`. As we did in Equations (1) and (2), the execution time of `p/3`, based on that of the individual goals, can be deduced as shown in Figure 4. The clause is at the left and the points in time (with an expression determining their value) are at the right. T_n (with $n \in \{a, b, c, d\}$) denotes the execution time of the respective goals. The primitives `&>`/2 and `<&`/1 themselves are, for simplicity, assumed to take zero time. Then, we can solve T_7, the total time taken by the clause, as a function of the length of the goals:

$$\begin{aligned} T_7 &= \max(T_6,\ T_3 + T_b) \\ &= \max(T_5 + T_d,\ T_2 + T_a + T_b) \\ &= \max(\max(T_4,\ T_1 + T_c) + T_d,\ T_a + T_b) \\ &= \max(\max(T_a,\ T_c) + T_d,\ T_a + T_b) \end{aligned}$$

Thus, the execution time of `p/3` is:

$$T_{dep} = \max(T_a + T_b,\ T_d + \max(T_a,\ T_c)) \tag{3}$$

Although, as presented in Section 2.1, values which make the parallelizations in Figure 2 (Equations (1) and (2)) incomparable are shown, there is no justification that Equation (3) cannot perform worse than any of the previous two. We can see that there is **not** combination of execution times for the sequential goals that can make the *non-jork-join* annotation be worse than either of the *fork-join* ones. Therefore, Equation (3) will never perform worse than Equation (1) or Equation (2), and the *non-fork-join* annotation is, therefore, a better option than any of the other *fork-join* annotations:

- $T_{dep} < T_{fj1}$ holds if, for example, $T_c \leq T_a$, and then $T_{dep} = \max(T_a + T_b, T_a + T_d)$, $T_{fj1} = T_a + T_b + T_d$.
- $T_{dep} < T_{fj2}$ holds if, for example, $T_a < T_c$, $T_c + T_d \leq T_a + T_b$, and then $T_{dep} = T_a + T_b$, $T_{dep} > T_a + T_b$.
- $T_{dep} = T_{fj1}$ holds if, for example, $T_b < T_c$, $T_a + T_b = T_c$, and then $T_{dep} = T_c + T_d = T_{dep}$.
- $T_{dep} = T_{fj2}$ holds if, for example, $T_d \leq T_b$, $T_c \leq T_a$, and then $T_{dep} = T_a + T_b = T_{dep}$.

In addition to these basic operators, other specialized versions can be defined and implemented in order to increase performance by adapting better to some particular cases. In particular, it appears interesting to introduce variants for the very relevant and frequent case of deterministic goals, in which backtracking will not be performed. For this purpose we propose two new operators: `&!>/2` and `<&!/1`. These specialized versions do not perform backtracking and do not prepare the execution data structures to cope with that possibility, which has previously been shown to result in a significant efficiency increase in the underlying machinery (Pontelli et al. 1996).

3 The UOUDG and UUDG Algorithms

In this section we will present two concrete algorithms which generate code annotated for unrestricted independent and-parallelism (as in Figure 3) starting from sequential code. The proposed algorithms process one clause at a time and work on a directed acyclic dependency graph where nodes are associated with a set of one or more body goals in the clause, since it is possible to perform grouping of goals in the outputted parallelized clause. We require that literals which are lexically identical give rise to different nodes, by, e.g., attaching a unique identifier to them. This is necessary in order not to lose information when building sets of nodes.

The idea behind these algorithms is to publish (i.e., to make available) goals for parallel execution as soon as possible and to delay "importing" their bindings (i.e., issuing joins) as much as possible —but always respecting the dependencies in the graph (as in Figure 1(b)). Intuitively, this should maximize the number of goals available for parallel execution and preserve the order of the solutions, if required.

The algorithm that is shown in Figure 5 presents the external interface of the annotation process. The first argument is the dependency graph associated to the clause to be parallelized. The second argument corresponds to a boolean value to check whether an order-preserving annotation has been required or not, in order to decide which more specialized procedures should be called (i.e., both algorithms in Figures 6 and 9). The last argument is some determinacy information of the literals of the clause.

Note that, as mentioned in Section 2.1, we will consider in this paper only unconditional parallelism for simplicity and also because it has shown to be effective as a default strategy in practice. However, the algorithms that we describe can be adapted to deal with conditional parallelism without too much effort.

Algorithm: `UnrestrictedAnnotation`(G, *order*, I_D)
Input: **(1)** A directed acyclic dependency graph $G = (V, E)$.
(2) The boolean value *order*.
(3) Determinacy information I_D for the literals of the clause.
Output: A clause annotated for unrestricted independent and-parallel execution.
begin
  if *order* **then**
    $Exp \leftarrow$ UOUDG(G, I_D);
  else
    $Exp \leftarrow$ UUDG(G, I_D);
  end
  return Exp
end

Fig. 5. Entry point to the annotation algorithms.

In the following, both algorithms in Figures 9 and 6 will denote the dependency graph as a pair $G = (V, E)$, in which G is the name of the graph, V is the set of vertices or nodes and E is the set of edges which represent a binary relation on V. $G|_U$ will denote the subgraph $(U, E|_U)$ of G in which there will be only edges connecting those nodes in U. The concept of set difference is defined as $A \setminus B = \{x \mid x \in A,\ x \notin B\}$. The expression $(x \rightsquigarrow y)_E$ informs that there exists a path from x to y created with edges in E. The auxiliary definition $\texttt{incoming}(\texttt{v}, \texttt{E}) = \{u \mid (u, v) \in E\}$ denotes the set of nodes which are connected to some particular node v.

Finally, in order to keep track of the order of the solutions, we assume that there is a relation $\prec$ on the literals L_i of the body of every clause H :- $L_1, L_2, \ldots, L_{k-1}, L_k$ such that $L_i \prec L_j$ if and only if $i < j$. Additionally, we assume that there is a partial function *pred* which is defined as $pred(L_{i+1}) = L_i$, i.e., the literal at the left of some other literal in a clause. We assume $\prec$ and *pred* are suitably extended, in the straightforward way, to the nodes of the dependency graph. Note also that the graph edges must respect the $\prec$ relation: $(u, v) \in E \Rightarrow u \prec v$, since the graph would have been incorrectly generated otherwise.

3.1 Non Order-Preserving Annotation: the UUDG *Algorithm*

Figure 6 presents an algorithm that parallelizes a clause, represented as an (acyclic) directed dependency graph.

At every iteration step, new nodes in the graph are selected to be published, joined or executed sequentially. Subsequent iterations proceed with a simplified graph in which the literals which have been joined or executed sequentially, together with their outgoing edges, have been removed. The set of goals which have already been published is kept in a separate argument in order to not schedule goals for parallel execution more than once.

In order to not lose parallelism, as a preprocessing stage in each iteration, sequences of goals are collapsed if there exists a path from every node to a successor literal in the clause and if not edges are coming from nodes out of the group. This is

Algorithm: UUDG(G_i, I_D)
Input: **(1)** A directed acyclic graph $G_i = (V_i, E_i)$.
(2) Determinacy information.
Output: A parallelized clause $\texttt{expr}_{G_i}$ in which the order of the solutions in the original clause needs not be preserved.

begin
  $\texttt{expr}_{G_i} \leftarrow (\texttt{true})$;
  $Pub \leftarrow \emptyset$;
  $G = (V, E) \leftarrow G_i$;
  while $V \neq \emptyset$ **do**
    $G \leftarrow \texttt{group\_nonord}(G,\ Pub)$;
    $Indep \leftarrow \{v \mid v \in V,\ \texttt{incoming}(v, E) = \emptyset\}$;
    $Dep \leftarrow \{I_v \mid v \in V,\ I_v = \texttt{incoming}(v, E),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
    if $Dep = \emptyset$ **then**
      $SS \leftarrow \emptyset$;
      $Join \leftarrow V$;
    else
      $SS \leftarrow \{I \mid I \in Dep,\ |I| = \texttt{min\_card}(Dep)\}$;
      $Join \leftarrow s$ s.t. $s \in SS$; `/* s any element from SS */`
    end
    if $(Join \cap (Indep \setminus Pub)) = \emptyset$ **then**
      $Seq \leftarrow \emptyset$;
    else
      $Seq \leftarrow \{v\}$ s.t. $v \in (Join \cap (Indep \setminus Pub))$; `/* v any element */`
    end
    $Fork \leftarrow Indep \setminus (Pub \cup Seq)$;
    $Join \leftarrow Join \setminus Seq$;
    $Pub \leftarrow Pub \cup (\bigcup_{v \in Fork} \texttt{get\_value}(v)) \cup \texttt{get\_value}(\texttt{u})$ s.t. $u \in Seq$;
    $G \leftarrow G|_{(V \setminus Join) \setminus Seq}$;
    $\texttt{expr}_{G_i} \leftarrow (\texttt{expr}_{G_i}, \texttt{gen\_body\_nonord}(Fork, Seq, Join, I_D))$;
  end
  return $\texttt{expr}_{G_i}$;
end

Fig. 6. UUDG annotation algorithm.

performed by running the function `group_nonord`, shown in Figure 7. The auxiliary definition `set_value` adds the literals in S to the node of the graph x.

Two sets are key in each iteration: *Indep*, which contains the *sources* (i.e., all vertices without incoming edges in the current graph, which can therefore be published), and *Dep*, which contains sets of vertices where, for each non-source v which can be reached from sources only, I_v is the set of sources ($I_v \subseteq Indep$) on which v depends. I.e., I_v is the set of vertices to be joined before v can start.

If there are not sets of vertices in *Dep* then all the literals that remain in the graph are independent, and thus they can all be published and joined up. Otherwise, a set of nodes needs to be chosen from *Dep* in order to wait for their result to be ready. The choice of that set is implemented by selecting, among the sets of goals which can be joined at every moment, the one with the lowest cardinality —i.e., we join as few goals as possible, thus postponing the rest of the joins as much as

Algorithm: `group_nonord`(G, Pub)
Input: **(1)** A directed acyclic graph $G = (V, E)$.
(2) A set of goals already forked.
Output: A compacted directed acyclic graph $G_f = (V_f, E_f)$.
begin
  forall $v \in V$ s.t. `get_value`(v) $\not\subseteq Pub$ **and** `incoming`(v, E) $= \emptyset$ **do**
    $Gr \leftarrow \{v\}$;
    $DS \leftarrow \{u \mid u \in V,\ u \notin Gr,\ w \in Gr,\ (w, u) \in E\}$;
    while $DS \neq \emptyset$ **do**
      $Gr' \leftarrow Gr \cup DS$;
      if $(\forall \{v_i, v_j\} \subseteq Gr',\ (v_i \rightsquigarrow v_j)_E\ \vee\ (v_j \rightsquigarrow v_i)_E)$ **and**
        $(\forall e = (v_k, v_l) \in E,\ v_k \notin Gr' \Rightarrow v_l \notin Gr')$ **then**
          $Gr \leftarrow Gr'$;
      else
        break;
      $DS \leftarrow \{u \mid u \in V,\ u \notin Gr,\ w \in Gr,\ (w, u) \in E\}$;
    end
    `set_value`(v, Gr);
    $G \leftarrow G|_{(V \setminus (Gr \setminus \{v\}))}$;
  end
  $G_f \leftarrow G$;
  return G_f;
end

Fig. 7. Nonorder-preserving grouping of nodes.

possible, in order to exploit more parallelism. This is taken care of by the definition `min_card`(S) $= \min(\{|s| \mid s \in S\})$, which returns the size of the smallest set in S.

Note that a random selection from a set is done at two different points. Data regarding, e.g., the relative run-time of goals would allow us to take a more informed decision and therefore precompute a perhaps better scheduling. Since we are not using this information here, we just pick any available goal to join / execute sequentially.

It is possible for a literal to be scheduled to be forked and then immediately joined. In order to detect these situations, which in practice would cause unnecessary overhead, we select (in *Seq*) the literal (only one) to which this applies, and it is not taken into account for the set of *Fork*ed nodes and removed from the set of the *Join*ed nodes.

The UUDG algorithm the continues outputting a parallel expression generated by the function `gen_body_nonord`, shown in Figure 8, composed with the parallelization of a simplified graph, generated by an iterative call. It makes use of the definitions `get_value`, which returns the literals of the original clause associated to the node v, and `seq`, which sequentializes the literals in a set preserving their order in the original clause. Those literals associated to the node in *Seq*, if any, are annotated after all literals in *Fork* have been published for parallel execution, in order to exploit all the detected parallelism.

The function `gen_body_nonord` makes use of determinism information, by using the auxiliary definition `det`, which reports whether the literal associated to a particular node of the graph is deterministic or not, as follows:

Algorithm: `gen_body_nonord`(*Fork*, *Seq*, *Join*, I_D)
Input: **(1)** A set of vertices to be forked.
(2) A set of vertices to be sequentialized.
(3) A set of vertices to be joined.
(4) Determinacy information.
Output: A parallelized sequence of literals *Exp*.
begin
Exp ← (`true`);
ForkDet ← $\{g \mid g \in Fork,\ det(g, I_D)\}$;
ForkNonDet ← $Fork \setminus ForkDet$;
JoinDet ← $\{g \mid g \in Join,\ det(g, I_D)\}$;
JoinNonDet ← $Join \setminus JoinDet$;
forall $v_i \in ForkDet$ **do**
  Exp ← (*Exp*, `seq(get_value(v`$_i$`))` `&!>` H_{v_i});
end
forall $v_i \in ForkNonDet$ **do**
  Exp ← (*Exp*, `seq(get_value(v`$_i$`))` `&>` H_{v_i});
end
if $Seq = \{v\}$ **then**
  Exp ← (*Exp*, `seq(get_value(v))`);
end
forall $v_i \in JoinDet$ **do**
  Exp ← (*Exp*, H_{v_i} `<&!`);
end
forall $v_i \in JoinNonDet$ **do**
  Exp ← (*Exp*, H_{v_i} `<&`);
end
return *Exp*;
end

Fig. 8. Nonorder-preserving generation of a parallel body.

- Since we have the possibility of switching goals around, we try to minimize relaunching goals which are likely to be executed in parallel by forking deterministic goals first.
- Additionally, when a goal is known to have exactly one solution, we can use the specialized versions of the dep-operators (Casas et al. 2008), `&!>/2` and `<&!/1`, which do not need to perform bookkeeping for backtracking (always complex in parallel implementations), and are thus more efficient.

This program information can often be automatically inferred by the abstract interpretation-based determinism analyzer included in CiaoPP (López-García et al. 2005), and is provided as input to the proposed annotators. Alternatively, this information can be stated by the programmer via assertions (Hermenegildo et al. 2005).

We proceed with the total correctness proof of the UUDG algorithm. The total correctness of the `group_nonord` function will be proved first.

Lemma 1
Suppose a particular set of nodes *Gr* to be grouped, obtained by the function

`group_nonord`. Then, no parallelism is lost when nodes in Gr are grouped into the source $v \in Gr$ of the (non-empty) graph $G = (V, E)$.

Proof
Let us prove this by induction in V.

- Base case: if $|V| = 1$ then $E = \emptyset$ and trivially no parallelism can be lost.
- Induction hypothesis: assume that nodes in Gr are mutually dependent.
- Inductive step: let us prove this by contradiction. Assume some new nodes, in DS, are added to Gr. Then, there are three cases in which parallelism may be lost.

  1. Suppose two sources u, w s.t. $\{u, w\} \subseteq Gr$. However, this is not possible because of the definition of DS.
  2. Suppose that $\exists u, w \in Gr$ s.t. $(u \not\rightsquigarrow w)_E \ \wedge \ (w \not\rightsquigarrow u)_E$. Then the possible parallelism between u and w will be lost. However, because of the first condition in the if-structure, $x \in Gr$ if $\forall \{x, y\} \subseteq Gr, \ (x \rightsquigarrow y)_E \ \vee \ (y \rightsquigarrow x)_E$, which means that $\nexists \{x, y\} \in Gr$ s.t. $(x \not\rightsquigarrow y)_E \ \wedge \ (y \not\rightsquigarrow x)_E$, and that leads to a contradiction.
  3. Suppose that $\exists u \in Gr$, $\exists w \notin Gr$ s.t. $\exists (w, u) \in E \vee \exists (u, w) \in E$. Then the possible parallelism between w and the nodes $v_i \in Gr$ s.t. $(w \not\rightsquigarrow v_i)_E \wedge (v_i \not\rightsquigarrow w)_E$ is lost. If $\nexists v_i \in Gr$ s.t. $(w \not\rightsquigarrow v_i)_E \ \wedge \ (v_i \not\rightsquigarrow w)_E$ then, because of the first condition in the if-structure, $w \in Gr$ and that leads to a contradiction. Otherwise,
     (a) If $(w, u) \in E$ then, because of the second condition in the if-structure, $u \notin Gr$, which leads to a contradiction.
     (b) If $(u, w) \in E$ then $\exists (u, v_i) \in E$ s.t. $(w \not\rightsquigarrow v_i)_E \ \wedge \ (v_i \not\rightsquigarrow w)_E$. Because of the first condition in the if-structure, since $w \notin Gr$ then $v_i \notin Gr$, and that leads to a contradiction.

Thus no parallelism is lost. □

Lemma 2 (*Partial Correctness of Algorithm 7*)
The function `group_nonord` is partially correct with respect to the precondition $\{|V| > \emptyset; |E| \geq \emptyset; |Pub| \geq \emptyset\}$ and the postcondition $\{P_1; P_2; P_3\}$ such that:

1. $P_1 \equiv \bigcup_{v \in V_f} \texttt{get\_value}(v) = V$.
2. $P_2 \equiv \bigcap_{v \in V_f} \texttt{get\_value}(v) = \emptyset$.
3. $P_3 \equiv$ no parallelism is lost when constructing G_f.

Proof
Starting with values that make the precondition *true*, for each source $v \in G$ which has not been published yet, Lemma 1 ensures that no parallelism is lost for any of the groups of literals created, and thus no parallelism is lost when building G_f. Moreover, only nodes that have been grouped into a source are removed from the graph, and thus no nodes are missing in G_f. In addition, G is simplified in each iteration, so there will not be repeated nodes in G_f. Therefore, the postcondition is *true* and thus the function `group_nonord` is partially correct. □

Lemma 3 (*Termination of Algorithm 7*)
The function `group_nonord` terminates.

Proof
For the inner loop, we choose as set with strict well-founded ordering $<$ the set of natural numbers $\mathbb{N}$. Let $|V \setminus Gr|$ be the termination expression. By definition, $|V \setminus Gr| \in \mathbb{N}$ whenever the control of the algorithm starts a new iteration. $|V \setminus Gr|$ takes a smaller value in each iteration of the loop with respect to $<$ if $Gr \neq \emptyset$, which implies that $Gr' \neq \emptyset$, and then $DS \neq \emptyset$, which is the exit condition of the loop. Since the outer loop simply consists of a single execution of the inner loop over each vertex of a finite set, the algorithm terminates. □

Theorem 1 (*Total Correctness of Algorithm 7*)
The function `group_nonord` is totally correct.

Proof
Lemma 2 states that the function `group_nonord` is partially correct, and Lemma 3 states that it terminates, so the function `group_nonord` is totally correct. □

Before we proceed with the total correctness proof of the UUDG algorithm, we will introduce the equivalence class of graphs with respect to transitive edges:

Definition 5 (*Equivalence class of graphs w.r.t.* $\equiv_t$)
Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two different dependency graphs. Then, $G_1 \equiv_t G_2 \;\Leftrightarrow\; (V_1 = V_2) \;\wedge\; (\forall e = (a, b) \in E_1,\; e \notin E_2 \;\Rightarrow\; \exists (a \rightsquigarrow b)_{E_2})$.

Lemma 4
In any iteration of the UUDG algorithm over a graph $G = (V, E)$, *Fork*, *Seq* and *Join* are composed only by sources.

Proof
$Join = V$ if $E = \emptyset$, or $Join \in Dep$. Since $\forall X \in Dep,\ X \subseteq Indep \Rightarrow Join \subseteq Indep$.
$Seq = \emptyset$, or $Seq = \{v\}$ s.t. $v \in (Join \cap (Indep \setminus Pub)) \subseteq Join \subseteq Indep$.
$Fork = (Indep \setminus (Pub \cup Seq)) \subseteq Indep$. □

Lemma 5
$G_1 = (V_1, E_1) \equiv_t G_2 = (V_2, E_2) \;\Rightarrow \mathsf{UUDG}(G_1) = \mathsf{UUDG}(G_2)$.

Proof
Proof by contradiction. Suppose that $\mathsf{UUDG}(G_1) \neq \mathsf{UUDG}(G_2)$ when $G_1 \equiv_t G_2$. Then $\texttt{expr}_{G_1} \neq \texttt{expr}_{G_2}$, which means that $\texttt{expr}_{G_1} = (t^1_{\texttt{expr}_{G_1}}, \ldots, t^m_{\texttt{expr}_{G_1}})$, $m \geq 1$, and $\texttt{expr}_{G_2} = (t^1_{\texttt{expr}_{G_2}}, \ldots, t^n_{\texttt{expr}_{G_2}})$, $n \geq 1$, and $\exists i$ s.t. $t^i_{\texttt{expr}_{G_1}} \neq t^i_{\texttt{expr}_{G_2}}$. Thus, for a particular iteration in $\mathsf{UUDG}(G_1)$, either *Fork*, *Seq* or *Join* differs from the respective one in the same iteration of $\mathsf{UUDG}(G_2)$. If $Dep = \emptyset$ then $V_1 = V_2$ and $E_1 = E_2 = \emptyset$. Thus, $\mathsf{UUDG}(G_1) = \mathsf{UUDG}(G_2)$ and that leads to a contradiction. Otherwise, $\exists w \in V_2$ s.t. $(w, b) \in E_2$ and $(a \rightsquigarrow w)_{E_2}$, and then $(a \rightsquigarrow w)_{E_1}$ and $(w \rightsquigarrow b)_{E_1}$. Thus, by definition, the content of *Dep* will be the same for both iterations in $\mathsf{UUDG}(G_1)$ and $\mathsf{UUDG}(G_2)$. Then, *Join* must be the same and, furthermore, *Seq* and *Fork* must also be the same, and that leads to a contradiction. □

This results guarantees that the UUDG algorithm will exploit the same amount of parallelism with two graphs that are in the same equivalence class $\equiv_t$. Now, we will prove the total correctness of the UUDG algorithm:

Lemma 6 (*Partial Correctness of Algorithm 6*)
The UUDG algorithm is partially correct with respect to the precondition $\{|V_i| \geq \emptyset; |E_i| \geq \emptyset\}$ and the postcondition $\{G_{expr} = (V_{expr}, E_{expr}) \equiv_t (V_i, E_i) = G_i\}$ such that:

$$
\begin{aligned}
V_{expr} \quad &= \quad \bigcup_{t \in \texttt{expr}} \texttt{get\_value}(v) \;\text{ s.t. }\; t = (v) \vee t = (H_v \texttt{ <\&!}) \vee t = (H_v \texttt{ <\&}) \\
E_{expr} \quad &= \quad \Big(\bigcup_{f_1 \in \texttt{expr}} \{(a,b) \mid \forall \{a,b\} \subseteq \texttt{get\_value}(v) \;\wedge\; a \prec b\} \Big) \quad \cup \\
&\qquad \Big(\bigcup_{f_2 \in \texttt{expr}} \{(w,v) \mid w = \texttt{last}(\texttt{get\_value}(x)),\; \forall v \in V_i \setminus P^{f_2}\} \Big) \\
&\qquad \text{s.t. } \; f_1 = (v \texttt{ \&> } H_v) \vee f_1 = (v \texttt{ \&!> } H_v) \\
&\qquad \text{and } \; f_2 = (x) \vee f_2 = (H_x \texttt{ <\&}) \vee f_2 = (H_x \texttt{ <\&!})
\end{aligned}
$$

where $\texttt{last}(S) = x$ s.t. $\forall y \in S,\; (y \neq x \;\wedge\; y \prec x)$
and $P^f = \{z \mid \forall y \in \texttt{expr},\; (y = (z) \vee y = (z \texttt{ \&> } H_z) \vee y = (z \texttt{ \&!> } H_z)),\; y \prec f\}$

Proof
The first union of edges in E_{expr} states that there will be an edge connecting each node that is in the same group. Because of Theorem 1, all nodes in the same group are mutually dependent and thus those nodes will just form an equivalent graph to G_i with respect to $\equiv_t$. Because of Lemma 5, the final parallel expression will be the same. In addition, since Theorem 1 states that the function `group_nonord` is totally correct, it is only necessary to prove that, assuming that the precondition is *true*, the postcondition is also *true* when no grouping of nodes is required. Then, the postcondition can be simplified to:

$$
\begin{array}{lcl}
V_{expr} & = & \{v \mid \forall (v) \in \texttt{expr},\ \forall (H_v\ \texttt{<\&}) \in \texttt{expr},\ \forall (H_v\ \texttt{<\&!}) \in \texttt{expr}\} \\
E_{expr} & = & (\bigcup_{f \in \texttt{expr}} \{(w,v) \mid \forall v \in V_i \setminus P^f\}) \\
 & & \text{s.t. } f = (w) \vee f = (H_w\texttt{<\&}) \vee f = (H_w\texttt{<\&!})
\end{array}
$$

Let us prove this now by induction in V.

- Base case: if $|V| = 0$ then $\texttt{expr} = (\texttt{true})$, and thus $V_{expr} = \emptyset$, $E_{expr} = \emptyset$. Therefore, $G_{expr} \equiv_t G_i$.
- Induction hypothesis: assuming that the UUDG algorithm is started with a particular dependency graph G that makes the precondition *true*, the resulting values make the postcondition *true*.
- Inductive step: Let the invariant of the loop be $I = \{G_i \equiv_t (G \cup G_{expr})\}$. We need to prove that the invariant still holds after executing an iteration of the loop.

  — Since G is simplified to $G|_{(V \setminus Join) \setminus Seq}$ at the end of the iteration, V will be $(V \setminus Join) \setminus Seq$. Since `expr` is increased with some nodes to be *Fork*ed, *Seq*uentialized or *Join*ed, V_{expr} will be $V_{expr} \cup \{v \mid (v \in Seq) \vee (v \in Join)\}$, because the annotations (v), $(H_v\ \texttt{<\&})$ and $(H_v\ \texttt{<\&!})$ correspond to those ones for *Seq* and *Join*. Thus, V_{expr} will be $V_{expr} \cup (Join \cup Seq)$. Therefore, $((V \setminus Join) \setminus Seq) \cup (V_{expr} \cup (Join \cup Seq)) = V \cup V_{expr} = V_i$.

  — Since G is simplified to $G|_{(V \setminus Join) \setminus Seq}$ at the end of the iteration, E will be $(E \setminus \{(u,v) \mid (u \in Join) \vee (u \in Seq),\ \forall v \in V\} \setminus \{(u,v) \mid (v \in Join) \vee (v \in Seq),\ \forall u \in V\})$. For Lemma 4, u must be a source and then the new value of E is simplified to $(E \setminus \{(u,v) \mid (u \in Join) \vee (u \in Seq),\ \forall v \in V\})$. Moreover, since v is a dependent node, it can be only a node that has not been published yet, and so E is simplified to:

  $$(E \setminus \{(u,v) \mid (u \in Join) \vee (u \in Seq),\ \forall v \in ((V \setminus Pub) \setminus Fork)\})$$

  Since `expr` is increased with some nodes to be *Fork*ed, *Seq*uentialized or *Join*ed, E_{expr} will be:

  $$E_{expr} \cup (\bigcup_{u \in (Seq \cup Join)} \{(u,v) \mid \forall v \in ((V \setminus Pub) \setminus Fork)\}) =$$
  $$= \{(u,v) \mid (u \in Join) \vee (u \in Seq),\ \forall v \in ((V \setminus Pub) \setminus Fork)\}$$

  Therefore, $E \cup E_{expr} = E_i$.

In addition, $I \wedge \{V = \emptyset\} \Rightarrow \{G_{expr} \equiv_t G_i\}$, which corresponds to the postcondition of the UUDG algorithm, so it is partially correct. □

Lemma 7 (*Termination of Algorithm 6*)
The UUDG algorithm terminates.

Proof
We choose as set with strict well-founded ordering $<$ the set of natural numbers $\mathbb{N}$. Let $|V|$ be the termination expression. By definition, $|V| \in \mathbb{N}$ whenever the control of the algorithm starts a new iteration. $|V|$ takes a smaller value in each iteration of the loop with respect to $<$ if either *Join* or *Seq* is not empty. Since *Join* $= V \neq \emptyset$ when $Dep = \emptyset$ or else $Join = I_v \in Dep$, which is by definition non-empty, the algorithm terminates. □

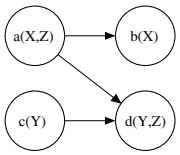
Theorem 2 (*Total Correctness of Algorithm 6*)
The UUDG algorithm is totally correct.

Proof
Lemma 6 states that the UUDG algorithm is partially correct, and Lemma 7 states that it terminates, so the UUDG algorithm is totally correct. □

In the rest of the subsection, we develop some examples that show how the UUDG algorithm works.

Example 1 (*UUDG Annotation*)
In order to illustrate how the UUDG algorithm works, we will study the results that are obtained at each of the iterations for the parallelization process for the predicate `p/3`, introduced in Section 2.1 and whose dependency graph is shown in Figure 1(b).

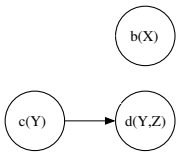
- *First iteration*:



$Indep = \{a, c\}$, $Dep = \{\{a\}, \{a, c\}\}$, $SS = \{a\}$
$Seq = \{a\}$, $Fork = \{c\}$, $Join = \emptyset$
$Pub = \{a, c\}$
Annotation: `p(X,Y,Z) :- c(Y) &> Hc, a(X,Z),`

The first step of the UUDG algorithm consists of grouping literals. However, in this example, no grouping of literals can be performed in any of the iterations. The next step in the algorithm marks the nodes that are sources in the graph (literals `a/2` and `c/1`), in the set *Indep*, and the dependencies of those non-sources which only have incoming edges from sources, in the set *Dep*. In order to exploit all possible parallelism, the least number of literals required to free a dependent literal will be joined. This can be done by choosing the smallest set in *Dep*, which is $\{a\}$ in this case. Thus, literals `a/2` and `c/1` are to be published for parallel execution and only the literal `a/2` needs to be joined. As an optimization, one goal between those scheduled for parallel execution and joined in the same iteration can be sequentially executed (which is what happens with literal `a/2`). After the annotation is done, literals `a/2` and `c/1` are stored in *Pub* and the graph is simplified by removing the node corresponding to `a/2`.

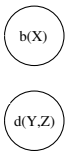
- *Second iteration*:



$Indep = \{b, c\}$, $Dep = \{\{c\}\}$, $SS = \{c\}$
$Seq = \emptyset$, $Fork = \{b\}$, $Join = \{c\}$
$Pub = \{a, b, c\}$
Annotation: `b(X) &> Hb, Hc <&,`

In this iteration, literals `b/1` and `c/1` are sources and only literal `d/2` is a dependent node. Since literal `c/1` was scheduled fr parallel execution in the previous iteration, only literal `b/1` will be published. Since literal `d/2` needs to be freed, literal `c/1` is joined, in the same way as in the previous iteration. After the annotation is done, literal `b/1` is stored in *Pub* and the graph is simplified.

- *Last iteration*:

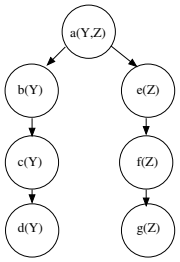


$Indep = \{b, d\}$, $Dep = \emptyset$, $SS = \emptyset$
$Seq = \{d\}$, $Fork = \emptyset$, $Join = \{b\}$
$Pub = \{a, b, c, d\}$
Annotation: `d(Y,Z), Hb <&.`

In this last iteration, as all the nodes are sources, all the literals not published yet (i.e., `d/2`) will be scheduled for parallel execution and the joins of all the remaining literals (i.e., both `b/1` and `d/2`) will be performed, finishing with the unrestricted annotation of the original clause.

Example 2 (*UUDG Annotation with Grouping*)
We will run the UUDG algorithm in the dependency graph shown below. In the second step of the algorithm, grouping of literals will be performed.

- *First iteration*:



$Indep = \{a\}$, $Dep = \{\{a\}, \{a\}\}$, $SS = \{a\}$
$Seq = \{a\}$, $Fork = \emptyset$, $Join = \emptyset$
$Pub = \{a\}$
Annotation: `a(Y,Z),`

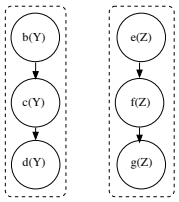
In the first step of the algorithm, literal `a/2` is executed sequentially, since the rest of the literals in the clause depend on it. Thus, literal `a/2` is removed from the graph.

Algorithm: UOUDG(G_i, I_D)
Input: **(1)** A directed acyclic graph $G_i = (V_i, E_i)$.
(2) Determinacy information.
Output: A clause parallelized $\texttt{expr}_{G_i}$ in *unrestricted and* fashion in which the order of the solutions in the original clause is preserved.

begin
  $\texttt{expr}_{G_i} \leftarrow (\texttt{true})$;
  $Pub \leftarrow \emptyset$;
  $G = (V, E) \leftarrow G_i$;
  while $V \neq \emptyset$ **do**
    $G \leftarrow \texttt{group\_ord}(G,\ Pub)$;
    $Indep \leftarrow \{v \mid v \in V,\ \texttt{incoming}(v, E) = \emptyset\}$;
    $Dep \leftarrow \{(v, I_v) \mid v \in V,\ I_v = \texttt{incoming}(v, E),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
    if $Dep = \emptyset$ **then**
      $(pvt,\ Join) \leftarrow (u, V)$ s.t. $\forall(w \in (V \setminus \{u\}))\ .\ w \prec u$;
    else
      $(pvt,\ Join) \leftarrow (u, S)$ s.t. $(u, S) \in Dep \wedge \forall((w, D) \in (Dep \setminus \{(u, S)\}))\ .\ u \prec w$;
    end
    $Seq \leftarrow \{v \mid v \in\ (Indep \setminus Pub),\ v \rightarrow pvt \in E,\ v = pred(pvt)\}$;
    $Fork \leftarrow \{v \mid v \in\ (Indep \setminus Pub),\ v \prec pvt\} \setminus Seq$;
    $Join \leftarrow Join \setminus Seq$;
    $Pub \leftarrow Pub \cup (\bigcup_{v \in Fork} \texttt{get\_value}(v)) \cup \texttt{get\_value}(\texttt{u})$ s.t. $u \in Seq$;
    $G \leftarrow\ G|_{(V \setminus Join) \setminus Seq}$;
    $\texttt{expr}_{G_i} \leftarrow (\texttt{expr}_{G_i}, \texttt{gen\_body\_ord}(Fork,\ Seq,\ Join,\ I_D))$;
  end
  return $\texttt{expr}_{G_i}$;
end

Fig. 9. UOUDG annotation algorithm.

- *Second (and last) iteration*:



$Indep = \{(b, c, d), (e, f, g)\}$, $Dep = \emptyset$, $SS = \emptyset$
$Seq = (e, f, g)$, $Fork = \{(b, c, d)\}$, $Join = \{(b, c, d)\}$
$Pub = \{a, b, c, d, e, f, g\}$
Annotation:
```
(b(Y),c(Y),d(Y)) &> H,
(e(Z),f(Z),g(Z)),
H <&.
```

In this iteration, literals `b/1`, `c/1` and `d/1` are grouped, and literals `e/1`, `f/1` and `g/1` are also grouped. Thus, the graph is reduced to one that only has two nodes, which are both sources and then they can be executed in parallel. Note that, as in Example 1, all the parallelism is exploited due to the fact that mutually dependent literals are grouped.

3.2 Order-Preserving Annotation: the UOUDG Algorithm

The UOUDG algorithm, presented in Figure 9, follows the same idea underlying the UUDG algorithm introduced in Figure 6: publish early and join late. However, the UOUDG algorithm has less freedom to publish goals, since the order of solutions needs now to be preserved. This is done by respecting the relative order of literals in the original clause, through the use of the relation $\prec$ and the partial function *pred*.

As a previous step in each iteration of the algorithm, the function `group_ord`, which is shown in Figure 10, is called in order to group nodes in a similar fashion as the function `group_nonord` in Figure 7. However, for this case the grouping of literals is done in such a way that the order of the literals in the original clause is always preserved.

An important element in the algorithm is *pvt*, the *pivot* vertex, which will be used in order to decide which nodes are to be joined, taking into account that we do not want to change the order of solutions. If there are not nodes in *Dep*, then all the remaining literals are already independent and we can join up to the rightmost literal in the clause. Otherwise, we select the leftmost node among those which have dependencies which can be fulfilled in one step. These dependencies are readily available in *Dep*. Note that as we select the leftmost node among those which can be joined, we are delaying as much as possible joining nodes —or, alternatively, we are performing in every step only the joins which are needed to continue one more step. This is aimed at maximizing the number of parallel goals being executed at any moment.

The UOUDG uses the function `gen_body_ord`, which is shown in Figure 11, to output a parallelized clause. The function `gen_body_ord`, as well as the function `gen_body_nonord` in Figure 8, makes use of the auxiliary function `get_value`, in addition to some determinism information, by using the auxiliary definition `det`, in order to decide whether the optimized versions of the operators `&>/2` (i.e., `&!>/2`) and `<&/1` (i.e., `<&!/1`) are to be used when a literal is known to be deterministic.

Furthermore, we proceed with the total correctness proof of the UOUDG algorithm. The total correctness of the `group_ord` function will be proved first.

Lemma 8 (*Partial Correctness of Algorithm 10*)
The function `group_nonord` is partially correct with respect to the precondition $\{|V| > \emptyset;\ |E| \geq \emptyset;\ |Pub| \geq \emptyset\}$ and the postcondition $\{P_1; P_2; P_3; P_4\}$ such that:

1. $P_1 \equiv \bigcup_{v \in V_f} \texttt{get\_value}(v) = V$.
2. $P_2 \equiv \bigcap_{v \in V_f} \texttt{get\_value}(v) = \emptyset$.
3. $P_3 \equiv$ no parallelism is lost when constructing G_f.
4. $P_4 \equiv \forall v \in V_f,\ \forall u \in \texttt{get\_value}(v),\ ((u = v)\ \vee\ (\mathit{pred}(u) \in \texttt{get\_value}(v)))$.

Proof
It can be proved in a similar way as Lemma 2. Note that the only difference in the postcondition is the new statement P_4, which demands that all nodes to be grouped

Algorithm: `group_ord`(G, Pub)
Input: **(1)** A directed acyclic graph $G = (V, E)$.
(2) A set of goals already forked.
Output: A compacted directed acyclic graph $G_f = (V_f, E_f)$ which preserves the order of literals in the grouping.

begin
  forall $v \in V$ s.t. `get_value`$(v) \not\subseteq Pub$ **and** `incoming`$(v, E) = \emptyset$ **do**
    $Gr \leftarrow \{v\}$;
    $DS \leftarrow \{u \mid u \in V,\ u \notin Gr,\ w \in Gr,\ (w, u) \in E\}$;
    while $DS \neq \emptyset$ **do**
      $Gr' \leftarrow Gr \cup DS$;
      if $(\forall u \in Gr',\ (u = v) \vee (pred(u) \in Gr'))$ **and**
        $(\forall \{v_i, v_j\} \subseteq Gr',\ (v_i \rightsquigarrow v_j)_E \ \vee \ (v_j \rightsquigarrow v_i)_E)$ **and**
        $(\forall e = (v_k, v_l) \in E,\ v_k \notin Gr' \Rightarrow v_l \notin Gr')$ **then**
          $Gr \leftarrow Gr'$;
      else
        break;
      $DS \leftarrow \{u \mid u \in V,\ u \notin Gr,\ w \in Gr,\ (w, u) \in E\}$;
    end
    `set_value`$(v,\ Gr)$;
    $G \leftarrow G|_{(V \setminus (Gr \setminus \{v\}))}$;
  end
  $G_f \leftarrow G$;
  return G_f;
end

Fig. 10. Order-preserving grouping of nodes.

must be consecutive, in order to preserve the order of the solutions. That condition will be always *true* because of the first condition of the if-structure. □

Lemma 9 (*Termination of Algorithm 10*)
The function `group_ord` terminates.

Proof
Same proof as in Lemma 3. □

Theorem 3 (*Total Correctness of Algorithm 10*)
The function `group_ord` is totally correct.

Proof
Lemma 8 states that the function `group_ord` is partially correct, and Lemma 9 states that it terminates, so the function `group_ord` is totally correct. □

The following definition introduces the concept of equivalence class of graphs with respect to a notion of order of literals:

Algorithm: `gen_body_ord`(*Fork*, *Seq*, *Join*, I_D)
Input: **(1)** A set of vertices to be forked.
(2) A set of vertices to be sequentialized.
(3) A set of vertices to be joined.
(4) Determinacy information.
Output: An unrestricted parallelized sequence of literals *Exp*.
begin
  Exp ← (`true`);
  forall $v_i \in Fork$ **do**
    if $det(v_i, I_D)$ **then**
      Exp ← (*Exp*, `seq(get_value(v`$_i$`)) &!>` H_{v_i});
    else
      Exp ← (*Exp*, `seq(get_value(v`$_i$`)) &>` H_{v_i});
    end
  end
  if $Seq = \{v\}$ **then**
    Exp ← (*Exp*, `seq(get_value(v))`);
  end
  forall $v_i \in Join$ **do**
    if $det(v_i, I_D)$ **then**
      Exp ← (*Exp*, H_{v_i} `<&!`);
    else
      Exp ← (*Exp*, H_{v_i} `<&`);
    end
  end
  return *Exp*;
end

Fig. 11. Order-preserving generation of a parallel body.

Definition 6 (*Equivalence class of graphs w.r.t.* $\equiv_o$)
Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$ be two different dependency graphs. Then, $G_1 \equiv_o G_2 \Leftrightarrow V_1 = V_2 \wedge (\forall v \in V_1,\ ((v,w) \in E_1 \Rightarrow ((v,w) \in E_2 \wedge (\forall x \in V_2 \text{ s.t. } w \prec x,\ (v,x) \in E_2))))$.

Lemma 10
$G_1 = (V_1, E_1) \equiv_o G_2 = (V_2, E_2) \Rightarrow \mathsf{UOUDG}(G_1) = \mathsf{UOUDG}(G_2)$.

Proof
Proof by contradiction. Suppose that $\mathsf{UOUDG}(G_1) \neq \mathsf{UOUDG}(G_2)$ when $G_1 \equiv_o G_2$. Then $\mathtt{expr}_{G_1} \neq \mathtt{expr}_{G_2}$, which means that $\mathtt{expr}_{G_1} = (t^1_{\mathtt{expr}_{G_1}}, \ldots, t^m_{\mathtt{expr}_{G_1}})$, $m \geq 1$, and $\mathtt{expr}_{G_2} = (t^1_{\mathtt{expr}_{G_2}}, \ldots, t^n_{\mathtt{expr}_{G_2}})$, $n \geq 1$, and $\exists i$ s.t. $t^i_{\mathtt{expr}_{G_1}} \neq t^i_{\mathtt{expr}_{G_2}}$. Thus, for a particular iteration in $\mathsf{UOUDG}(G_1)$, either *Fork*, *Seq* or *Join* differs from the respective one in the same iteration of $\mathsf{UOUDG}(G_2)$. If $Dep = \emptyset$ then $V_1 = V_2$ and $E_1 = E_2 = \emptyset$. Thus, $\mathsf{UOUDG}(G_1) = \mathsf{UOUDG}(G_2)$ and that leads to a contradiction. Otherwise, *pvt* and *Join* will be the same in both $\mathsf{UOUDG}(G_1)$ and $\mathsf{UOUDG}(G_2)$ because *pvt* represents the first dependent node with respect to $\prec$, and *Join* its dependencies which precede it. *Fork* and *Seq* will also be the same since they are dealing only with predecessors to the *pvt*, and that leads to a contradiction. □

We will proceed now with the total correctness proof of the UOUDG algorithm.

Lemma 11 (*Partial Correctness of Algorithm 9*)
The UOUDG algorithm is partially correct with respect to the same precondition as in Lemma 6, and postcondition $\{G_{expr} = (V_{expr}, E_{expr}) \equiv_o (V_i, E_i) = G_i\}$, where V_{expr} and E_{expr} have the same value as in Lemma 6.

Proof
This lemma can be proved in a similar way as Lemma 6. Because of the results of Theorem 3 and Lemma 10, the postcondition can also be simplified. The inductive part of the proof is similar as that one done in Lemma 6, with the loop invariant $I = \{G_i \equiv_o (G \cup G_{expr})\}$. □

Lemma 12 (*Termination of Algorithm 9*)
The UOUDG algorithm terminates.

Proof
It can be proved in a similar fashion as Lemma 7. □

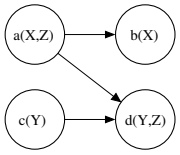
Theorem 4 (*Total Correctness of Algorithm 9*)
The UOUDG algorithm is totally correct.

Proof
Lemma 11 states that the UOUDG algorithm is partially correct, and Lemma 12 states that it terminates, so the UOUDG algorithm is totally correct. □

In the rest of the subsection, we develop an example that sketches how the UOUDG algorithm works.

Example 3 (*UOUDG Annotation*)
In order to illustrate how the UOUDG algorithm works, in a similar fashion as in Example 1, we will go through the different iterations in the parallelization process of the predicate `p/3` (dependency graph shown in Figure 1(b)).

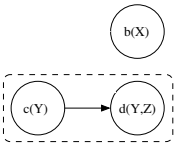
- *First iteration*:



$Indep = \{a, c\}$, $Dep = \{(b, \{a\}), (d, \{a, c\})\}$, $pvt = b$
$Seq = \{a\}$, $Fork = \emptyset$, $Join = \emptyset$
$Pub = \{a\}$
Annotation: `p(X,Y,Z) :- a(X,Z),`

In the first algorithm step, both literals `a/2` and `c/1` are candidates for parallel execution (they are in *Indep*). Also, since literals `b/1` and `d/2` only have dependencies with nodes in *Indep*, both literals will be stored in *Dep*, with their respective set of dependencies. Since literal `b/1` is the one in *Dep* with fewer dependencies, it is chosen as pivot, and literal `a/2` marked to be joined in this iteration of the algorithm. Moreover, although literal `c/1` is in *Indep*, only literal `a/2` can be marked to be published for parallel execution, since the order of the literals in the initial clause must be preserved, and literal `b/1` has not been published yet. However, as literal `a/2` is the only one to be published and must be joined too, then it is simply selected to be sequentially executed. As a final step, literal `a/2` is stored in *Pub* and the dependency graph simplified by removing the node corresponding to the literal `a/2`. Note how this annotation has less freedom than the UUDG annotation in Example 1, always respecting the dependencies implicit in the graph.

- *Second (and last) iteration*:



$Indep = \{b, (c, d)\}$, $Dep = \{\}$, $pvt = (c, d)$
$Seq = \{(c, d)\}$, $Fork = \{b\}$, $Join = \{b\}$
$Pub = \{a, b, c, d\}$
Annotation: `b(X) &> Hb, ( c(Y), d(Y,Z) ), Hb <&.`

In this iteration, both literals `b/1` and `c/1` are sources. In this case, literals `c/1` and `d/2` are compacted into a single node in the graph. Thus, all the nodes in the graph are sources and can be scheduled for parallel execution.
Once this iteration finishes, the initial clause is unrestrictedly parallelized and the order of the literals in the initial clause, given by the operator `&>/2`, preserved.
Finally, note that more parallelism is exploited (in fact, all the possible parallelism) with the UUDG annotation in Example 1 than with the UOUDG annotation, since the order of the literals in the clause does not have to be preserved.

4 Performance Evaluation

The proposed annotation algorithms have been integrated into the Ciao/CiaoPP system (Hermenegildo et al. 2005). Information gathered by the analyzers on variable sharing, groundness, and freeness is used to determine goal independence, using the libraries available in CiaoPP. Determinism is used in the annotators as described previously.

As execution platform we have used a high level implementation of the proposed parallelism primitives (Casas et al. 2008), which we have developed as an extension of the Ciao system (Bueno et al. 2006). This implementation is an evolution and simplification of (Hermenegildo and Greene 1991) which is based on raising

AIAKL	An abstract interpreter for the AKL language.
FFT	An implementation of the Fast Fourier transform.
FibFun	A version of the Fibonacci program written in functional notation.
Hamming	A program to compute the first N Hamming numbers.
Hanoi	A program to compute the movements to solve the well-known puzzle, as proposed in (Muthukumar et al. 1999).
Takeuchi	Computes the Takeuchi function.
WMS2	A scheduler assigning a number of workers to a series of jobs.

Table 1. *Benchmark programs*

the level of certain components to the level of the source language and keeping only some selected operations (related to thread handling, locking, etc.) at a lower level. This approach does not eliminate altogether modifications to the abstract machine, but it greatly simplifies them. Although the actual underlying parallel implementation is beyond the scope of this paper, we would like to mention that the results obtained are quite reasonable given the simplicity of our implementation approach and encourage us to work further on the optimization of our high-level implementation.

4.1 Evaluation

It should be noted however that the dep-operators do not assume any particular architecture: while our current implementation and all the performance results were obtained on a multicore machine, the techniques presented can be also applied in distributed memory machines —and in fact, the first prototype implementation of the dep-operators (Cabeza and Hermenegildo 1996; Cabeza 2004) was actually made on a distributed environment.

We have evaluated the impact of the different annotations on the execution time by running a series of benchmarks (briefly described in Table 1) in parallel. Table 2 shows the speedups obtained *with respect to the sequential execution* (i.e., they are *actual* speedups, which is the reason why some speedups start below 1 for, e.g., one thread), when using from 1 to 8 threads.

The machine we used is a Sun UltraSparc T2000 (a *Niagara*) with 8 4-thread cores. In the performance results shown in Table 2, we did not use more than 8 cores since in that case, and due to access to shared units, speedups are sublinear even for completely independent tasks.

The *fork-join* annotators we chose to compare with are MEL (Muthukumar et al. 1999) (which preserves goal order and tries to maximize the length of the parallel expressions) and UDG (Cabeza 2004) (which can reorder goals). The MEL algorithm tries to find longest parallel expression by proceeds backwards from the last literal in order to find a hard dependency between two literals and hence the expression can be split into two different ones. The UDG algorithm assumes that all the dependence conditions that cannot be completely determined statically are false, in order to eliminate the overhead of evaluating run-time checks in the parallel expression. However, the limitation of these fork-join annotation algorithms relies on the use

Benchmark	**Annotator**	**Number of threads**							
		1	2	3	4	5	6	7	8
AIAKL	UMEL	0.97	0.97	0.98	0.98	0.98	0.98	0.98	0.98
	UOUDG	0.97	1.55	1.48	1.49	1.49	1.49	1.49	1.49
	UDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
	UUDG	0.97	1.77	1.66	1.67	1.67	1.67	1.67	1.67
FFT	UMEL	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UOUDG	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UDG	0.98	1.76	2.14	2.71	2.82	2.99	3.08	3.37
	UUDG	0.98	1.82	2.31	3.01	3.12	3.26	3.39	3.63
FibFun	UMEL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UOUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
	UDG	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UUDG	0.99	1.95	2.89	3.84	4.78	5.71	6.63	7.57
Hamming	UMEL	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	UOUDG	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
	UDG	0.93	1.13	1.52	1.52	1.52	1.52	1.52	1.52
	UUDG	0.93	1.15	1.64	1.64	1.64	1.64	1.64	1.64
Hanoi	UMEL	0.89	0.98	0.98	0.97	0.97	0.98	0.98	0.99
	UOUDG	0.89	1.70	2.39	2.81	3.20	3.69	4.00	4.19
	UDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
	UUDG	0.89	1.72	2.43	3.32	3.77	4.17	4.41	4.67
Takeuchi	UMEL	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UOUDG	0.88	1.62	2.17	2.64	2.67	2.67	2.67	2.67
	UDG	0.88	1.61	2.16	2.62	2.63	2.63	2.63	2.63
	UUDG	0.88	1.62	2.39	3.33	4.04	4.47	5.19	5.72
WMS2	UMEL	0.85	0.81	0.81	0.81	0.81	0.81	0.81	0.81
	UOUDG	0.99	1.09	1.09	1.09	1.09	1.09	1.09	1.09
	UDG	0.99	1.01	1.01	1.01	1.01	1.01	1.01	1.01
	UUDG	0.99	1.10	1.10	1.10	1.10	1.10	1.10	1.10

Table 2. *Speedups for several benchmarks and annotators.*

of the fork-join operator, which is a rigid operator in the sense that it does not allow to execute some other literals until the execution of both parallel goals has finished, and furthermore this can produce the lack of some of the parallelism that is implicitly represented in the dependency graph, as seen in Section 2. MEL can add runtime checks to decide dynamically whether to execute or not in parallel. In order to make the annotation unconditional (as the rest of the annotators we are dealing with), we simply removed the conditional parallelism in the places where it was not being exploited. This is why it appears in Table 2 under the name *UMEL*.

All the benchmarks executed were parallelized automatically by CiaoPP, starting from their sequential code. Since UOUDG and UUDG can improve the results of fork-join annotators only when the code to parallelize has at least a certain level of complexity, not all benchmarks with (independent) parallelism can benefit from using the dep-operators. Additionally, comparing speedups obtained with programs parallelized using order-preserving and non-order-preserving annotators is not completely meaningful.

Note that, in this paper, we are not focusing on the speedups themselves. Al-

though of utmost practical interest, raw speed is very connected with the implementation of the underlying parallel abstract machine, and improvements on it can be expected to uniformly affect all parallelized programs. Rather, our main focus of attention is in the *comparison* among the speedups obtained using different annotators.

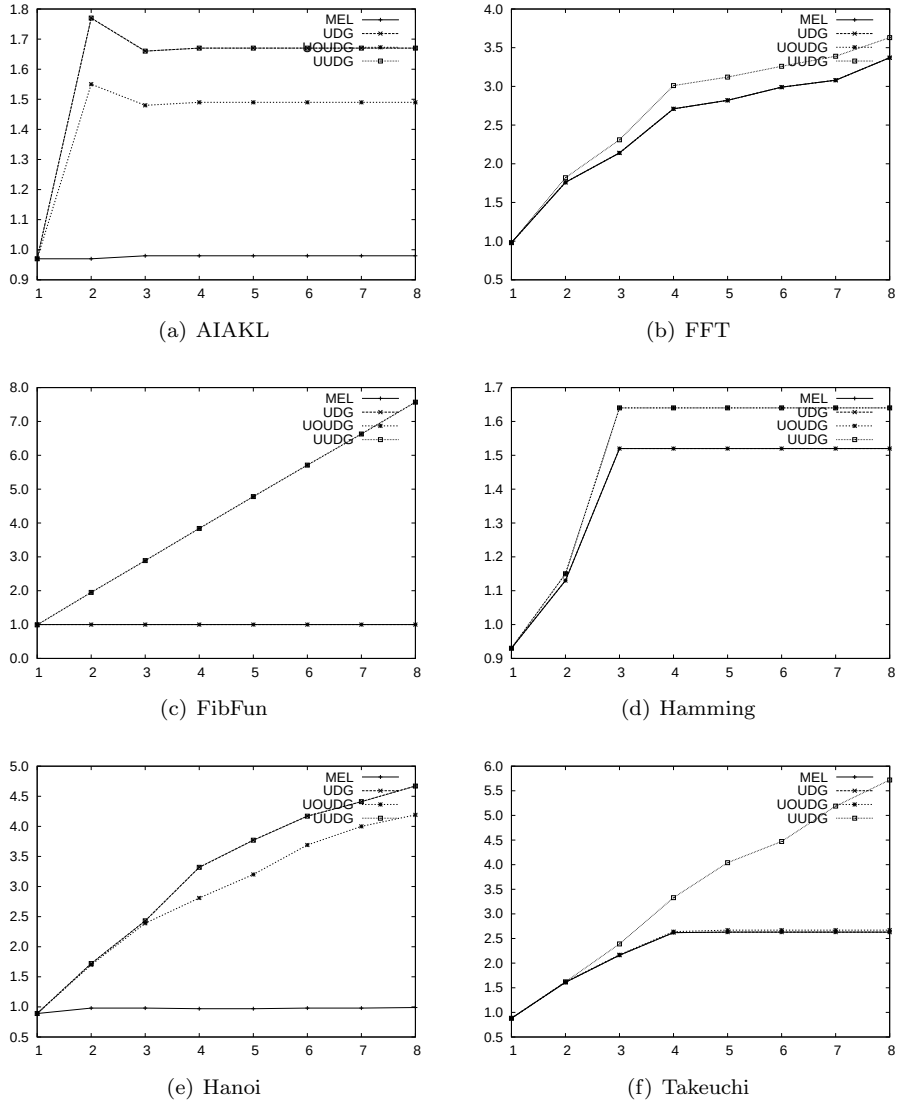
A first examination of the experimental results in Table 2, and also in Figure 12 allows inferring that in no case is UUDG worse than any other annotator, and in no case is UOUDG worse than (U)MEL. They should therefore be the *annotators of choice* if available. Besides, there are cases where UOUDG is better than UDG, and the other way around, which is in accordance with the non-comparable nature of these two algorithms.

Among the cases in which a better speedup is obtained by some of the U(O)UDG annotators, improvements range between "no improvement" (because no benefit is obtained for some particular cases and combinations of annotators) to an increase of 757% in speedup, with several other stages in between. Also, it is worth pointing out that the speedup does not stabilize in any benchmark (at least in a sizable amount) as the number of threads increases; moreover, in some cases the difference in speedup between the restricted and the unrestricted versions grows substantially with the number of threads. This can (clearly) be seen in, e.g., Figure 12(f).

Finally, we would like to comment specially on three benchmarks. **FibFun** is the result of parallelizing a definition of the Fibonacci numbers written using the functional notation capabilities of Ciao (Casas et al. 2006). Because of the order in which code is generated in the (automatic) translation into Prolog, the result is only parallelizable by UOUDG and UUDG, hence the speedup obtained in this case. The case of **Hanoi** is also interesting, as it is the first example in (Muthukumar et al. 1999): in the arena of order-preserving parallelizers, UOUDG can extract more parallelism than MEL for this benchmark. Lastly, the **Takeuchi** benchmark has a relatively small loop which only allows parallelizing with a simple `&/2`. However, by unrolling one iteration in the loop the resulting body has dependencies which are complex enough to take advantage of the increased flexibility of the dep-operator annotators.

5 Conclusions

We have proposed two annotation algorithms which perform a source-to-source transformation of a logic program into an unrestricted independent and-parallel version of itself. Both algorithms rely on the use of more basic high-level primitives than the fork-join operator, and differ on whether the order of the solutions in the original program must be preserved or not. We have implemented the proposed algorithms in the CiaoPP system, which infers automatically groundness, sharing, and determinacy information, used to simplify the initial dependency graph. The results of the experiments performed show that, although the parallelization provided by the new annotation algorithms is the same in quite a few of the traditional parallel benchmarks, in our experiments it is never worse and in some cases it is significantly better. This supports the observations made based on the expected



(a) AIAKL (b) FFT

(c) FibFun (d) Hamming

(e) Hanoi (f) Takeuchi

Fig. 12. Speedups obtained with different annotations for **AIAKL**, **FFT**, **FibFun**, **Hamming**, **Hanoi** and **Takeuchi**.

performance of the annotations. We have also noticed that the benefits are larger for programs with high numbers of goals in their clauses, since their more complex graphs make the ability to exploit unrestricted parallelism more relevant.

Acknowledgments: This work was funded in part by Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS*, by Ministry of Industry (MIN) PROFIT project FIT-350400-2006-44 *GGCC*, by Madrid Regional Government (CM) project S-0505/TIC/0407 *PROMESAS*, and by IST pro-

gram of the European Commission FP6 FET project IST-15905 *MOBIUS*. Manuel Hermenegildo and Amadeo Casas were also funded in part by the Prince of Asturias Chair in Information Science and Technology at UNM.

References

Ali, K. A. M. and Karlsson, R. 1990. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*. MIT Press, 757–776.

Bueno, F., Cabeza, D., Carro, M., Hermenegildo, M., López-García, P., and (Eds.), G. P. 2006. The Ciao System. Ref. Manual (v1.13). Tech. rep., C. S. School (UPM). Available at `http://www.ciaohome.org`.

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

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

Butler, R., Lusk, E. L., Olson, R., and Overbeek, R. A. 1986. Anlwam: A Parallel Implementation of the Warren Abstract Machine. Internal report, Argonne National Laboratory, Argonne, Il 60439.

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

Cabeza, D. and Hermenegildo, M. 1996. Implementing Distributed Concurrent Constraint Execution in the CIAO System. In *Proc. of the AGP'96 Joint Conference on Declarative Programming*. 67–78.

Casas, A., Cabeza, D., and Hermenegildo, M. 2006. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *The 8th International Symposium on Functional and Logic Programming (FLOPS'06)*. Fuji Susono (Japan).

Casas, A., Carro, M., and Hermenegildo, M. 2008. Towards a High-Level Implementation of Execution Primitives for Non-restricted, Independent And-parallelism. In *10th International Symposium on Practical Aspects of Declarative Languages (PADL'08)*, D. Warren and P. Hudak, Eds. LNCS, vol. 4902. Springer-Verlag.

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

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

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

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

Hermenegildo, M. and CLIP Group, T. 1994. Some Methodological Issues in the Design of CIAO - A Generic, Parallel Concurrent Constraint System. In *Proc. of the*

1994 ICOT/NSF Workshop on Parallel and Concurrent Programming (1994), E. Tick, Ed. U. of Oregon.

Hermenegildo, M. and Greene, K. 1991. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing 9,* 3,4, 233–257.

Hermenegildo, M., Puebla, G., Bueno, F., and López-García, P. 2005. Integrated Program Debugging, Verification, and Optimization Using Abstract Interpretation (and The Ciao System Preprocessor). *Science of Computer Programming 58,* 1–2 (October), 115–140.

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

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

Janson, S. 1994. Akl. a multiparadigm programming language. Ph.D. thesis, Uppsala University.

Kalé, L. V. 1987. Parallel Execution of Logic Programs: the REDUCE-OR Process Model. In *Fourth International Conference on Logic Programming.* Melbourne, Australia, MIT Press, 616–632.

Karp, A. and Babb, R. 1988. A Comparison of 12 Parallel Fortran Dialects. *IEEE Software*.

López-García, P., Bueno, F., and Hermenegildo, M. 2005. Determinacy Analysis for Logic Programs Using Mode and Type Information. In *Proceedings of the 14th International Symposium on Logic-based Program Synthesis and Transformation (LOPSTR'04).* Number 3573 in LNCS. Springer-Verlag, 19–35.

López-García, P. and Hermenegildo, M. 1995. Efficient Term Size Computation for Granularity Control. In *International Conference on Logic Programming.* MIT Press, Cambridge, MA, Cambridge, MA, 647–661.

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

Lusk et al., E. 1990. The Aurora Or-Parallel Prolog System. *New Generation Computing 7,* 2,3.

Mera, E., López-García, P., Puebla, G., Carro, M., and Hermenegildo, M. 2007. Combining Static Analysis and Profiling for Estimating Execution Times. In *Ninth International Symposium on Practical Aspects of Declarative Languages.* Number 4354 in LNCS. Springer-Verlag, 140–154.

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

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

Pontelli, E., Gupta, G., Tang, D., Carro, M., and Hermenegildo, M. 1996. Improving the Efficiency of Nondeterministic And–parallel Systems. *The Computer Languages Journal 22,* 2/3 (July), 115–142.

Santos-Costa, V., Warren, D., and Yang, R. 1991. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *Proceedings of the 3rd. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming.* ACM, 83–93. SIGPLAN Notices vol 26(7), July 1991.

Santos-Costa, V. M. 1993. Compile-time analysis for the parallel execution of logic programs in andorra-i. Ph.D. thesis, University of Bristol.

Sarkar, V. 1990. Instruction Reordering for Fork-Join Parallelism. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation.* Vol. 25. 322–336.

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