Annotation Algorithms for Unrestricted Independent And-Parallelism in Logic Programs

July 2007

A. Casas
M. Carro
M. Hermenegildo

TR Number CLIP 5/2007.0

Technical Report Number: CLIP 5/2007.0
June, 2007

Authors

A. Casas
amadeo@cs.unm.edu
Department of Computer Science and Electrical and Computer
Engineering U. of New Mexico (UNM)

M. Carro
mcarro@fi.upm.es
Universidad Politécnica de Madrid
Boadilla del Monte E-28660, Spain.

M. Hermenegildo
herme@fi.upm.es,herme@unm.edu
Technical University of Madrid
Boadilla del Monte E-28660, Spain, and
Departments of Computer Science and Electrical and Computer
Engineering U. of New Mexico (UNM)

Keywords

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

Acknowledgements

This work was funded in part by IST programme of the European Commission, FET project FP6 IST-15905 *MOBIUS*, by Ministry of Education and Science (MEC) project TIN2005-09207-C03 *MERIT-COMVERS* and CAM project S-0505/TIC/0407 *PROMESAS*. Manuel Hermenegildo is also funded in part by the Prince of Asturias Chair in Information Science and Technology at UNM.

Abstract

We present two new algorithms which perform source-to-source transformations aimed at exploiting goal-level, restricted independent and-parallelism. They rely on annotating the code with execution primitives which are simpler and more flexible than the well-known `&/2` parallel execution operator. This 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 algorithms we present differ on whether the order of the solutions obtained is preserved or not and on the use of determinism information. Finally, we compare the performance obtained by our approach with that of previous annotation algorithms and show that we can obtain relevant improvements.

Resumen

En este trabajo presentamos dos nuevos algoritmos que realizan transformaciones de programa fuente a fuente encaminadas a explotar el paralelismo conjuntivo independiente restringido. Se basan en anotar el código con primitivas de ejecución que son más simples y flexibles que el conocido operator de ejecución paralela `&/2`. Ello posibilita la generación de mejores expresiones paralelas al exponer más posibilidades de paralelismo entre los literales de una cláusula de lo que es posible con `&/2`. Los algoritmos que presentamos se diferencian en si preservan o no el orden de las soluciones con respecto al correspondiente programa secuencial y en el uso de información de determinismo. Finalmente, comparamos la eficiencia obtenida usando nuestro enfoque con la de algoritmos de anotación previos y mostramos que podemos obtener mejoras relevantes.

Contents

1 Introduction

Parallelism capabilities are becoming ubiquitous thanks to the widespread use of multi-core processors. Indeed, most current laptops contain two processors (capable of running four threads) and single-chip, 8-processor servers are now in wide use. The trend is that the number of on-chip processors will double with each processor generation. The difficulty in automatically exploiting these capabilities has renewed research interest in language tools to simplify the task of producing parallel programs. Declarative languages and, among them, logic programming languages can potentially facilitate exploitation of parallelism. The higher level of abstraction of declarative languages allows writing programs that are closer to the problem. Such higher-level encodings do not need to incorporate lower-level implementation decisions that often obscure the parallelism intrinsic in the problem. The notion of control in declarative languages provides more flexibility to arrange the order in which some operations can be evaluated without affecting the semantics of the original program. Additionally, their relatively simple semantics makes it possible to detect more accurately the existence of dependencies between operations and hence automatic extraction of parallelism is potentially easier than in imperative languages.

Because of this potential, automatic parallelization has received significant attention in logic programming [8]. Two main forms of parallelism have been studied in the execution of logic programs. *Or-parallelism* stems when the alternatives created by non-deterministic goals can be explored simultaneously, in order to reduce the time taken to traverse their (possible large) search space. Some relevant or-parallelism systems are Aurora [15] and MUSE [1]. *And-parallelism* executes in parallel different conjunctive goals in clauses (or in the resolvent). While or-parallelism exploits only parallelism when there is search involved, and-parallelism is found in more code schemes, divide-and-conquer and map-style algorithms being classic representatives. Examples of systems that have exploited and-parallelism are ROPM [13], AO-WAM [3], DDAS [20], AKL [12], Andorra-I [19] and &-Prolog [9]. Additionally, some systems such as ACE [7] and Andorra [19] exploit certain combinations of both and- and or-parallelism. In this paper, we will concentrate on and-parallelism.

A correct parallelization has been defined as one that preserves some key properties, typically correctness and no-slowdown, during and-parallel execution [11]. The preservation of these properties is ensured by executing in parallel goals which meet some 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 example absence of competition for binding variables 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.[1]

One of the better understood conditions which ensure that goals meet the efficiency and correctness criteria for parallelization is strict independence [11], which entails the absence of shared variables at runtime. It should be noted that some proposals exploit and-parallelism between goals which are dependent, but on which other restrictions are imposed which ensure no-slowdown and correctness. Examples of such restrictions are determinism and non-failure [11] (determinism is exploited for example in the Basic Andorra Model [19]) and absence of conflicts due to the binding of shared variables (as in non-strict independent and-parallelism [11]). Another 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.

[1]Note however that this does not affect our presentation, as we will deal with dependencies in a generic way.

One particularly successful approach to automatically parallelizing a logic program uses three different stages [2]. The first one detects data (and control) dependencies between pairs of literals in the original program. A dependency graph (see Figure 1 in Section 2 as an example) is built to capture this information. Nodes in the graph correspond to literals in the body of the clause and edges represent dependencies between them. Edges are labeled with the associated dependency conditions (which may be trivially *true* or *false* —we will not represent those edges labeled with *true*). The second stage performs (global) analysis [8, 2] to gather information regarding, e.g., variable aliasing, groundness, side effects, etc. in order to remove edges from the dependency graph or to simplify the conditions labeling these edges, if they cannot be evaluated statically to completion. Labeled edges will result in run-time checks if conditional parallel expressions are allowed. Alternatively, unresolved dependencies can be 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. Finally, the third stage transforms the original program into a parallel version by *annotating* it with parallel execution operators using the information gathered by the analyzers. This annotation should respect the dependencies found in the original program while, at the same time, exploiting as much parallelism as possible.

Several annotation algorithms have been proposed so far [17, 4] that are based on the use of the `&/2` (conjunctive *fork-join*) operator as the basic construction to express parallelism between goals. The need to adapt the dependency graph to a (nested) fork-join structure makes parallelism opportunities to be inevitably lost in cases with a complex enough structure (e.g., that in Figure 1). Likewise, inter-procedural parallelism (i.e., parallelism which spans literals in different predicates) cannot be exploited without program transformation.

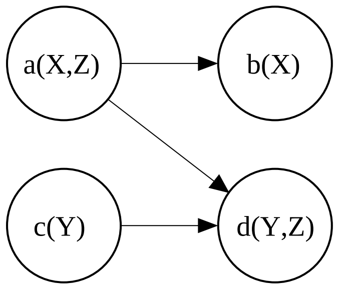


Figure 1: Dependency graph for `p/3`.

This annotation process is the focus of this paper. We will present and evaluate new annotation algorithms which use as target parallelism primitives which can express richer dependencies than those which can be expressed using `&/2`. Our hope is that since the transformed programs will contain in some cases more parallelism, using the proposed approach we will be able to obtain better speedups for such cases.

2 Background and Motivation

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

```
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

Figure 2: *Fork-Join* annotations for `p/3` (Section 2).

2.1 *Fork-Join*-Style Parallelization

We will use as running example the following clause:

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

and will assume that the dependencies detected between the literals in the predicate are as shown in Figure 1: an arrow between two nodes means that the goal in the origin has to be completed before the goal in the end. We will assume that all the dependencies are unconditional —i.e., conditional dependencies are assumed to be always false. This avoids potentially costly run-time checks in the parallelized code, at the expense of having fewer opportunities for parallelism.

Conjunctive parallel execution is usually denoted using the parallel operator `&/2` instead of the sequential comma (','). The former binds more tightly than the latter. An expression of the form `a, b & c, d` means that the literals `b` and `c` can be safely executed in parallel after the execution of the literal `a` finishes. When both `b` and `c` have successfully finished, execution continues with `d`.

This single operator is enough to parallelize many programs, but the class of dependencies it can express directly (called μ-graphs in [17]) is a subset of the dependencies that can possibly appear in a program. It, therefore, falls short in some cases —for example, for the clause above. In general, several annotations are possible for a given program. As an example, Figure 2 shows two annotations for the sample clause.[2] Note that some goals appear switched w.r.t. their order in the sequential clause, but changing the order of pure goals in a clause does not change its logical meaning, and if goals are strictly independent no slowdown need happen. If additional ordering requirements are needed (due to, e.g., side effects or impurity), these can appear as additional dependencies in the graph.

Note that none of the annotations in Figure 2 fully exploits all parallelism available in Figure 1: Figure 2(a) misses the parallelism between `b(X)` and `d(Y, Z)`, and Figure 2(b) misses the parallelism between `b(X)` and `c(Y)`.

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; we will term those times T_{fj1} and T_{fj2} for Figures 2(a) and 2(b), respectively. This length ultimately depends on the execution times of their goals (i.e., T_a, T_b, T_c, T_d), which we assume to be non-zero. Expressions for T_{fj1} and T_{fj2} follow:

$$T_{fj1} = \max(T_a + T_b,\ T_c) + T_d \qquad (1)$$
$$T_{fj2} = \max(T_a,\ T_c) + \max(T_b,\ T_d) \qquad (2)$$

The question whether the annotation in Figure 2(a) is or not better than that of Figure 2(b) boils down to finding out whether it is possible that $T_{fj1} < T_{fj2}$ or the other way around. It

[2]The parallelization `p :- a(X, Z), b(X) & c(Y), d(Y, Z)` has been left out of Figure 2. However, this does not add anything to the discussion as it does not change the comparison we make in Section 2.2.

turns out that they are non-comparable. In fact:

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

Deciding at compile time which annotation is to be preferred needs further information regarding the expected runtime of goals (see, e.g., [16] and its references). While it is possible to use rankings to decide whether some annotation is better than some other, for example using the number of goals annotated for parallelism in a clause, finding this optimality is computationally expensive [17] —there are many decisions to make which make the number of annotations explode. In practice, annotators use heuristics which may be more or less appropriate in concrete cases.

Additional differences between annotators come from whether run-time tests for independence are allowed and from whether they preserve (or not) the goal order with respect to the original, sequential program: some annotators include independence conditions in the output, in order to decide dynamically whether to execute or not in parallel, and some annotators switch around the order of goals in a clause.

2.2 Parallelization with Finer Goal-Level Operators

It has been observed [5, 4] that more basic constructions can be used to represent and-parallelism by using two operators, `&>/2` and `<&/1`, defined as follows:

Definition 1 `G &> H` *schedules the goal* `G` *for parallel execution and continues executing the code after* `G &>H`*.* `H` *is a* handler *which contains (or* points to*) the state of goal* `G`*.*

Definition 2 `H <&` *waits for the goal associated to* `H` *to finish. After that point the bindings made for the output variables of* `G` *are available to the executing thread.*

With the previous definitions, the operator `&/2` 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` —or, that 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`, and, secondly, the dep-operators allow more freedom to the annotator (and to the programmer, if parallel code is written by hand) to express data dependencies and, therefore, to extract more potential parallelism. We will now illustrate the latter remark (the former is out of the scope of this paper).

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

Figure 3: dep-operator-annotated clause

```
Algorithm: unrestricted_annotation($G$, $order$, $\mathrm{I}_D$)
Input  : (1) An acyclic dependency graph $G = (V, E)$. (2) The boolean value order. (3)
         Determinacy information $\mathrm{I}_D$ for the literals of the clause.
Output: A clause annotated for parallel execution.
begin
    if $order$ then
        $Exp \leftarrow$ UOUDG($G$, $\emptyset$);
    else
        $Exp \leftarrow$ UUDG($G$, $\emptyset$, $\mathrm{I}_D$);
    end
    return $Exp$;
end
```

Algorithm 1: Entry point to the annotation algorithms.

Figure 3 shows an annotation of the running example using dep-operators. Note that this code allows executing in parallel `a/2` with `c/1`, `b/2` with `c/1`, and `b/1` with `d/2`. The execution time of `p/3`, based on that of the individual goals, is:

See Appendix A for a deduction.

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

If we compare expression (3) with expressions (1) and (2), it turns out that:

- It is possible that $T_{dep} < T_{fj1}$, $T_{dep} < T_{fj2}$, $T_{dep} = T_{fj1}$, and $T_{dep} = T_{fj2}$ (possibly with different lengths for every goal in each case).
- It is **not** possible that $T_{dep} > T_{fj1}$ or that $T_{dep} > T_{fj2}$.

See Appendix B for a proof.

This means that the annotation in Figure 3 cannot be worse than those of Figure 2, and can perform better in some cases. It is, therefore, a better option than any of the others.

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. 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 [18]. However, while in our benchmarks these operators are used, for the sake of clarity only the general versions will appear throughout the discussion.

3 The UOUDG and UUDG Algorithms

In this section we will present two concrete algorithms to generate code annotated for parallelism (as in Figure 3) starting with sequential code while respecting dependencies between literals (as in Figure 1). Algorithm 1 shows the external interface of these algorithms: it checks whether an order-preserving annotation has been required (or not), and then more specialized procedures are called.

```
Algorithm: UOUDG($G$, $Pub$)
Input  : (1) A directed acyclic graph $G = (V, E)$. (2) A set of goals already forked. (3)
         Determinacy information.
Output: An unrestricted parallelized clause in which the order of the solutions in the
         original clause is preserved.
begin
    if $|V| = 0$ then return (true);
    else
        $Indep \leftarrow V \setminus \{v \mid e \in E,\ e = u \rightarrow v\}$;
        $Dep \leftarrow \{(v, I) \mid v \in V,\ I_v = \mathtt{incoming}(v),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
        if $|Dep| = 0$ then
            $Joinable \leftarrow V$;
        else
            $Joinable \leftarrow S$ s.t. $(u, S) \in Dep \wedge \forall((w, D) \in (Dep \setminus S))\ .\ L_u \prec L_w$;
        end
        $Indep \leftarrow Indep \setminus Pub$;
        $Fork' \leftarrow \mathtt{consecutive\_vertices}(Indep, u)$;
        $And \leftarrow \mathtt{last\_common\_vertices}(Fork', Joinable)$;
        $Fork \leftarrow Fork' \setminus And$; $ToJoin \leftarrow Joinable \setminus And$;
        $Pub \leftarrow Pub \cup Fork \cup And$; $G \leftarrow G - And - ToJoin$;
        return ($\mathtt{gen\_body}(Fork, And, ToJoin)$, UOUDG($G$, $Pub$));
    end
end
```

Algorithm 2: UOUDG Annotation Algorithm.

As mentioned in Section 2.1 we will consider in this paper only unconditional parallelism, both for simplicity and because it has been experimentally found to be a good compromise between the degree of parallelism uncovered and the execution time of independence tests. However, the algorithms that we describe can be adapted to deal with conditional parallelism without too much effort.

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 a join) as much as possible —but always respecting the dependencies in the graph. Intuitively, this should maximize the number of goals available for parallel execution.

The algorithm processes a clause at a time, and its input consists of three arguments. The first one is an (acyclic) directed dependency graph $G = (V, E)$, in which the vertices V correspond to the literals L_i of the clause H `:-` L_1, ..., L_n. Each clause induces an order $\prec$ between the literals such that for $L_i \prec L_j$ iff $i < j$. There exists an edge between L_i and L_j in E if $ind(L_i, L_j, \lambda(i), \lambda(j)) = false$ (i.e., the literals L_i and L_j are dependent), where ind is the notion of independence and $\lambda(\overline{n+1})$ is the vector of (abstract) data dependency states. This information is obtained, in our case, from global data-flow analysis [2]. The second argument is a boolean value that indicates whether the order of the solutions must be preserved or not. Finally, determinacy information pertaining to the literals L_i of the clause is given as third argument, to be used when necessary.

```
Algorithm: gen_body(Fork, And, Join)
Input  : (1) A set of vertices to be forked. (2) A set of vertices to be and-parallelized.
         (3) A set of vertices to be joined.
Output: An expression Exp.
begin
    Exp ← (true);
    forall $v_i \in Fork$ do
        Exp ← (Exp, $v_i$ &> $H_{v_i}$);
    end
    Let And be $\{v_1, \ldots, v_n\}$;
    Exp ← (Exp, ($v_1$ & (... & $v_n$)));
    forall $v_i \in Join$ do
        Exp ← (Exp, $H_{v_i}$ <&);
    end
    return Exp;
end
```

Algorithm 3: gen_body function without determinacy information.

3.1 Order-Preserving Annotation: the UOUDG Algorithm

Algorithm 2 parallelizes a clause represented as an (acyclic) directed dependency graph preserving the order of the solutions by respecting the relative order of literals in the original clause. At every recursion step, new nodes (i.e., literals) in the graph are annotated to be published (i.e., forked off) and joined, proceeding in the following iterations with a simplified graph in which the joined vertices and their outgoing edges have been removed. The set of goals which have already been published is kept in the second argument.

Two sets are key in each iteration: *Indep* contains the *sources* (i.e., all vertices in the graph without incoming edges, and which can therefore be published), and *Dep*, which for each non-source vertex v which can only be reached from vertices in *Indep* contains the set of vertices in *Indep* which reach v. It is, therefore, a set of tuples, where the dependent vertex v and the set of vertices which corresponds to the literals that must be joined to fulfill its dependencies are stored.

Algorithm 2 then proceeds by partitioning the vertices of the graph in order to decide which ones are to be published and which ones are to be joined. The sources (in *Indep*) that have not been already published can be parallelized. To select which nodes have to be published while preserving the solution order, only vertices consecutive to the leftmost vertex u of the graph are to be published. Similarly, the vertices of any of the sets in *Dep* could in principle be joined. However, as the order of the solutions in the original clause must be preserved, only those vertices on which the leftmost vertex u depends are to be joined. Note that, in order to maximize the degree of parallelism, joining the rest of the vertices that are dependencies of the other tuples in *Dep* will be delayed. As a further simplification, nodes that are to be published and joined in the same step of the algorithm can be annotated directly with `&/2`. These are stored in the *And* variable.

The fact that the vertex corresponding to the first literal in the original clause is always a source and that the dependency graph is directed and acyclic implies that at least one literal will be joined in each iteration, simplifying the graph for the following iteration and thus freeing

```
Algorithm: UUDG($G$, $Pub$, $I_D$)
Input  : (1) A directed acyclic graph $G = (V, E)$. (2) A set of goals already forked. (3)
         Determinacy information.
Output: An unrestricted parallelized clause in which the order of the solutions in the
         original clause need not be preserved.
begin
    if $|V| = 0$ then return (true);
    else
        $Indep \leftarrow V \setminus \{v \mid e \in E,\ e = u \rightarrow v\}$;
        $Dep \leftarrow \{I_v \mid v \in V,\ I_v = \texttt{incoming}(v),\ I_v \neq \emptyset,\ I_v \subseteq Indep\}$;
        if $|Dep| = 0$ then
            $Joinable \leftarrow V$;
        else
            $SS \leftarrow \{I_v \mid I_v \in Dep,\ |I_v| = \texttt{min\_card}(Dep)\}$;
            $Joinable \leftarrow$ Pick randomly any element from $SS$;
        end
        $Indep \leftarrow Indep \setminus Pub$; $Fork \leftarrow Indep \setminus Joinable$;
        $And \leftarrow Indep \setminus Fork$; $ToJoin \leftarrow Joinable \setminus And$;
        $Pub \leftarrow Pub \cup Fork \cup And$; $G \leftarrow G - And - ToJoin$;
        return (gen_body_det($Fork$, $And$, $ToJoin$, $I_D$), UUDG($G$, $Pub$, $I_D$));
    end
end
```

Algorithm 4: UUDG Annotation Algorithm.

some of the dependent vertices, and implying that the algorithm terminates.

Algorithm 2 (as well as Algorithm 4) uses some auxiliary definitions which we include below:

`consecutive_vertices(S, v)`	=	set of vertices belonging to S that correspond to original clause literals consecutive to the one associated to v.
`incoming(v)`	=	$\{u \mid u \rightarrow v \in E\}$.
`last_common_vertices(`S_1, S_2`)`	=	set of latest vertices common to those S_1 and S_2, according to the order $\prec$.
`min_card(S)`	=	$\min(\{\lvert s\rvert \mid s \in S\})$.

Finally, Algorithm 2 uses Algorithm 3 in order to return an expression in which the literals passed as arguments are annotated for parallel execution.

3.2 Non Order-Preserving Annotation: the UUDG Algorithm

Algorithm 4 follows the same idea underlying Algorithm 2: publish early and join late. However, it has more freedom to publish goals, since the order of solutions does not need to be preserved. This is implemented by selecting among the sets of goals which can be joined at every moment the one with lower cardinality —i.e., we join the least number of goals in the current iteration, thus postponing the rest of the joins as much as possible, in order to exploit as much parallelism as possible. This selection does not take into account the order in the original, sequential clause of the goals that can be started after this join.

```
Algorithm: gen_body_det($Fork$, $And$, $Join$, $I_D$)
Input  : (1) A set of vertices to be forked. (2) A set of vertices to be and-parallelized.
         (3) A set of vertices to be joined. (4) Determinacy information.
Output: An expression $Exp$.
begin
    $Exp \leftarrow$ (true);
    Sort $Fork$ and $And$ from non-deterministic goals to deterministic ones;
    forall $v_i \in Fork$ do
        if $det(v_i)$ then $Exp \leftarrow (Exp,\ v_i$ &>! $H_{v_i})$;
        else $Exp \leftarrow (Exp,\ v_i$ &> $H_{v_i})$;
    end
    Let $And$ be $\{v_1,\ \ldots,\ v_n\}$;
    if $det(v_1) \wedge \ldots \wedge det(v_n)$ then $Exp \leftarrow (Exp,\ (v_1$ &! $(\ldots$ &! $v_n)))$;
    else $Exp \leftarrow (Exp,\ (v_1$ & $(\ldots$ & $v_n)))$;
    forall $v_i \in Join$ do
        if $det(v_i)$ then $Exp \leftarrow (Exp,\ H_{v_i}$ <&!$)$;
        else $Exp \leftarrow (Exp,\ H_{v_i}$ <&$)$;
    end
    return $Exp$;
end
```

Algorithm 5: `gen_body_det` function with determinacy information.

Finally, Algorithm 4 uses Algorithm 5 (a version of Algorithm 3) to output a parallelized clause. The difference lies in the use that Algorithm 5 makes of determinism information:

- Since we already 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 specialized versions of the dep-operators which do not need to keep bookkeeping for backtracking (always complex in parallel implementations), and are thus more efficient.

This program information can be automatically inferred by the abstract interpretation-based determinism analyzer in CiaoPP [14], and is provided as input to the proposed annotators. Alternatively, this information can be stated by the programmer via assertions [10].

3.3 A Comparison of the Annotation Algorithms

As mentioned in Section 2.1, `&/2`-based annotators transform non-μ-graphs into μ-graphs, in order to be able to fully parallelize them with the `&/2` operator, by adding extra dependencies between the nodes. These annotators choose to parallelize goals that are sources in the graph. By definition, those goals that are parallelized via the `&/2` operator must all be joined at the same time. Algorithm 4 is able in each iteration to detect and mark all the sources in the graph to be published (in case they were not published before). Moreover, it produces the minimal number of joins necessary to free a dependent node, postponing as many joins as possible to following iterations. The UUDG annotator proposed exploits at least as much parallelism as any fork-join annotator, extracting more parallelism when some of the joins can be postponed, i.e., when parallelizing a non-μ-graph.

On the other hand, Algorithm 2 preserves the order of the solutions in the initial clause that is to be parallelized by disallowing annotations in which the goals are switched around w.r.t. the order $\prec$ in the sequential clause. Some annotators for parallelism which are based on `&/2` (e.g., MEL and URLP) share this characteristic, while others (e.g., UDG) do not. In our particular case, Algorithm 2 can exploit as much parallelism as any `&/2` annotator in its class can (e.g., MEL, if we disregard the conditional part of the CGEs or URLP), and more if the dependency graph is not a μ-graph.

The restriction on goal reordering makes it impossible to exploit potential (independent) parallelism that goal reordering can uncover. Therefore, and in principle, reordering annotators have an advantage over non-reordering ones, and comparing results belonging to both kinds must be made with caution, as it can be misleading. However, if we still want to draw a comparison based on potential execution times, Figure 4 presents a lattice that shows the relationship between annotators.

FJ-(No-)Order represents the general class of `&/2` annotators which respect (or not) the order of solutions. The need for the order relation to be "less or equal than" instead of "less than" is clear as some clauses (e.g., those with only two independent goals) will be parallelized in an equivalent way (also, at the limit, clauses with not parallelism whatsoever must be left untouched by any annotator).

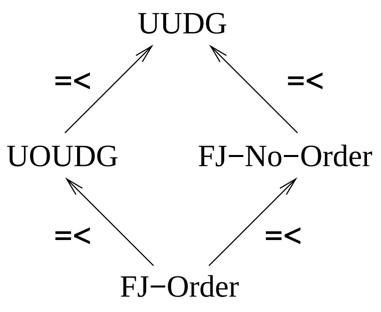


Figure 4: Comparing annotators.

4 Performance Evaluation

Our annotation algorithms have been integrated in the Ciao/CiaoPP system [10]. Information gathered by the analyzers on variable sharing, groundness, and freeness is used to determine goal independence. Determinism is used in the annotators as described previously.

As execution platform we have used a high level implementation of the proposed parallelism primitives, which we have developed as an extension of the Ciao system. This implementation is an evolution of [9] and is based on raising the implementation 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. The actual underlying parallel implementation is beyond the scope of this paper. 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 multiprocessor machine, the techniques presented can be also applied in distributed memory machines —and in fact, the first prototype implementation of the dep-operators [5, 4] 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 in parallel. Table 1 shows the speedups obtained *with respect to the sequential execution* when using from 1 to 8 threads. The machine we used is a Sun UltraSparc

Benchmark	**Annotator**	**Number of processors**							
		1	2	3	4	5	6	7	8
AIAKL	MEL	0.90	0.94	0.95	0.95	0.95	0.95	0.95	0.95
	UOUDG	0.90	1.40	1.36	1.37	1.37	1.37	1.37	1.37
	UDG	0.90	1.65	1.56	1.59	1.59	1.59	1.59	1.60
	UUDG	0.90	1.65	1.56	1.59	1.59	1.59	1.59	1.60
FFT	MEL	0.94	1.66	2.00	2.58	2.65	2.78	2.87	3.06
	UOUDG	0.94	1.66	2.00	2.58	2.65	2.78	2.87	3.06
	UDG	0.94	1.66	2.00	2.58	2.65	2.78	2.87	3.06
	UUDG	0.94	1.70	2.03	2.59	2.74	2.92	3.04	3.20
FibFun	MEL	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UOUDG	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UDG	1.00	1.00	1.00	1.00	1.00	1.00	1.00	1.00
	UUDG	0.98	1.93	2.87	3.82	4.75	5.68	6.60	7.52
Hamming	MEL	0.79	1.04	1.29	1.29	1.29	1.29	1.29	1.29
	UOUDG	0.79	1.06	1.34	1.34	1.34	1.34	1.34	1.34
	UDG	0.79	1.04	1.29	1.29	1.29	1.29	1.29	1.29
	UUDG	0.79	1.06	1.34	1.34	1.34	1.34	1.34	1.34
Hanoi	MEL	0.80	0.93	0.94	0.93	0.93	0.93	0.94	0.94
	UOUDG	0.80	1.48	1.99	2.49	2.94	3.21	3.41	3.72
	UDG	0.84	1.56	2.20	2.71	3.16	3.52	3.92	4.10
	UUDG	0.84	1.56	2.20	2.71	3.16	3.52	3.92	4.10
Takeuchi	MEL	0.87	1.51	2.13	2.57	2.58	2.58	2.58	2.58
	UOUDG	0.87	1.53	2.16	2.59	2.61	2.61	2.61	2.61
	UDG	0.87	1.51	2.13	2.57	2.58	2.58	2.58	2.58
	UUDG	0.87	1.53	2.28	3.30	3.94	4.36	5.16	5.75
WMS2	MEL	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 1: Speedups for several benchmarks with MEL, UDG, UOUDG, and UUDG.

T2000 (a *Niagara*) with 8 4-thread processors.[3] The *fork-join* annotators we chose to compare with are MEL (which preserves goal order and tries to maximize the length of the parallel expressions) and UDG (which can reorder goals). 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 1 under the name *UMEL*.

The test programs we used are the following:

[3]We did not use more than 8 processors since in that case, and due to data contention and access to shared processor units, we have observed speedups to be sublinear (and difficult to predict) even for completely independent tasks.

AIAKL	an abstract interpreter for the AKL language.
FFT	an implementation of the Fast Fourier transform.
FibFun	a version of **Fib** written in functional notation.
Hamming	computes the first N Hamming numbers, i.e., natural numbers which are multiples of 2, 3, and 5.
Hanoi	solves a version of the well-known puzzle.
Takeuchi	computes the Takeuchi function.
WMS2	schedules a number of workers in a series of jobs.

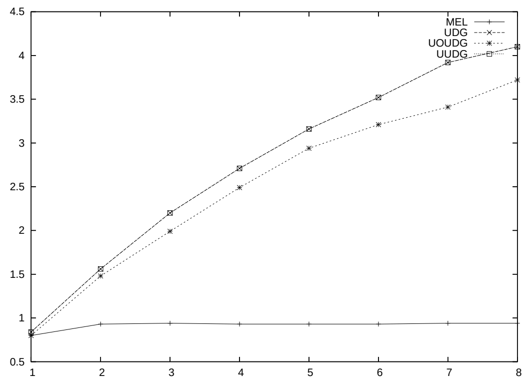
All the benchmarks executed were parallelized automatically by CiaoPP, starting from their sequential code. Since UOUDG and UUDG can improve the results of fork-join annotators only when the code to parallelize has at least a certain complexity, not all benchmarks with (independent) parallelism can benefit 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. Although of utmost practical interest, raw speed is mainly connected with the implementation of the underlying parallel abstract machine, and improvements on it should uniformly affect all parallelized programs. Rather, our main focus of attention is in the *comparison* among the speedups obtained using different annotators. In any case note that the speedups reported are *actual* speedups, i.e., computed w.r.t. the speed of the *sequential* version on one processor (hence the speedups are sometimes below 1 for one processor of the parallelized versions).

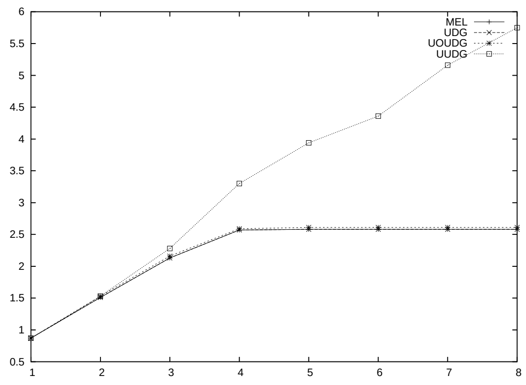
A first examination of the experimental results in Table 1 supports the discussion in Section 3.3: in no case UUDG is worse than any other annotator, and in no case UOUDG is worse than (U)MEL. They should therefore be the *annotators of choice* if available. Besides, there are cases where UOUDG is better than UDG, and the other way around, which is again in accordance with the non-comparable points in Figure 4.

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 752% in speedup, with several other points in between. Also, it is worth pointing out that the gain in speedup does not diminish in any benchmark (at least in a sizable amount) as the number of processors increases; moreover, in some cases the difference in speedup grows substantially with the number of processors. This can (clearly) be seen in, e.g., Figure 5(b).

Finally, we want 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 [6]. The (automatic) translation of the code into Prolog is only parallelizable by UUDG, hence the speedup obtained in this case. The case of **Hanoi** is also interesting, as it is the first example in [17]: in the arena of order-preserving parallelizers, UOUDG can extract more parallelism than MEL for this benchmark. Last, the **Takeuchi** benchmark has a relatively small loop which only allows parallelizing with a simple `&/2`. However, by unrolling one iteration the resulting body has dependencies which are complex enough to take advantage of the increased flexibility of the dep-operator annotators.



(a) Hanoi



(b) Takeuchi

Figure 5: Speedups with different annotations for Hanoi and Takeuchi.

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 within the CiaoPP system, which infers automatically groundness, sharing, and determinism 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, it is never worse and in some cases it is significantly better. This supports the observations made based on the expected 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 non-restricted parallelism more relevant.

References

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

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

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

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

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

6. A. Casas, D. Cabeza, and M. Hermenegildo. A Syntactic Approach to Combining Functional Notation, Lazy Evaluation and Higher-Order in LP Systems. In *FLOPS'06*, Fuji Susono (Japan), April 2006.

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

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

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

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

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

12. Sverker Janson. *AKL. A Multiparadigm Programming Language.* PhD thesis, Uppsala University, 1994.

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

14. P. López-García, F. Bueno, and M. Hermenegildo. 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, pages 19–35. Springer-Verlag, August 2005.

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

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

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

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

19. V. Santos-Costa, D.H.D. Warren, and R. Yang. 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*, pages 83–93. ACM, April 1991. SIGPLAN Notices vol 26(7), July 1991.

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

This section is **not** part of the paper. It is only intended to help the reviewers check some claims stated in the paper.

A Minimum Time to Execute a Parallel Expression

The bound given in Equation (3) can be deduced as follows.

```
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 <&,
                                $T_5 = \max(T_3,\ T_1 + T_c)$
    d(Y, Z),
                                $T_6 = T_5 + T_d$
    Hb <&.
                                $T_7 = \max(T_6,\ T_3 + T_b)$
```

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 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:

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

B Comparison Between Parallelizations

Although values which make the parallelizations in Figure 2 (Equations (1) and (2)) incomparable are shown in the text, there is no justification that Equation (3) cannot be smaller than any of the previous two. We show here how we reached to that conclusion.

Let us consider the predicate `p/3` of Section 2.1, whose dependency graph is shown in Figure 1. The execution time expressions for the two parallelizations of `p/3`, given in Figure 2(a) and Figure 2(b), are presented in Equation (1) and Equation (2). These two equations can be implemented using constraint logic programming as follows:

```
%% Tfj1 = max(a + b, c) + d
tfj1(A, B, C, D, T):-
        positive([A,B,C,D,T]),
        AB .=. A + B,
        max(AB, C, MaxABC),
        T .=. D + MaxABC.

%% Tfj2 = max(a,c) + max(b, d)
tfj2(A, B, C, D, T):-
        positive([A,B,C,D,T]),
        max(A, C, MAC),
        max(B, D, MBD),
        T .=. MAC + MBD.
```

```
positive([]).
positive([X|Xs]):-
        X .>. 0,
        positive(Xs).


max(X, Y, X):- X .>=. Y.
max(X, Y, Y):- X .<. Y.
```

The parallelization resulting from the execution of the UUDG annotator is in Figure 3 and the expression which gives execution time appears in Equation (3). This equation can again be implemented as follows:

```
%% Tdep = max(a+b, d + max(a,c))
tdep(A, B, C, D, T):-
        positive([A,B,C,D,T]),
        AB .=. A + B,
        max(A, C, MaxAC),
        DAC .=. D + MaxAC,
        max(AB, DAC, T).
```

Each of the following queries corresponds to the answer for a precise question:

- Is any of the *fork-join* parallelization really better than the other?

```
?- tfj1(A,B,C,D,T1),                      ?- tfj1(A,B,C,D,T1),
   tfj2(A,B,C,D,T2),                         tfj2(A,B,C,D,T2),
   T1 .<. T2.                                T2 .<. T1.
D.>.0, A.>.0,                             C.>.0, D.>.0,
T2 .=<. A+2*B,                            C  .=<. A,
T2 .>. A+B+D,                             D  .=<. B,
T2 .=. B+C,                               T2 .=. A+B,
T1 .=. A+B+D ?                            T1 .=. A+B+D ?

yes                                       yes
```

There are solutions for both orderings, so none of them is definitely better than the other one.

- Can any of the *fork-join* annotations be handled better than the annotation with dep-operators?

```
?- tfj1(A,B,C,D,T1), tdep(A,B,C,D,T2), T1 .<. T2.

no
?- tfj2(A,B,C,D,T1), tdep(A,B,C,D,T2), T1 .<. T2.

no
```

The answer is `no` in both cases – no combination of execution times for the sequential goals can make UUDG be worse than either *fj1* or *fj2*.