Proceedings of PASCO'94

A Comparative Study of Methods for Automatic Compile–time Parallelization of Logic Programs *

(Extended Abstract)

F. Bueno M. García de la Banda M. Hermenegildo

{bueno,maria,herme}@fi.upm.es
Facultad de Informática
Universidad Politécnica de Madrid (UPM)
28660-Boadilla del Monte, Madrid – Spain

Abstract: This paper presents a study of the effectiveness of three different algorithms for the parallelization of logic programs based on compile–time detection of independence among goals. The algorithms are embedded in a complete parallelizing compiler, which incorporates different abstract interpretation–based program analyses. The complete system shows the task of automatic program parallelization to be practical. The trade–offs involved in using each of the algorithms in this task are studied experimentally, weaknesses of these identified, and possible improvements discussed.

Keywords: Automatic Parallelization, Logic Programming.

1 Introduction

The parallel execution of a set of goals is ensured to be correct and efficient w.r.t. the sequential execution if the goals are proved to be independent [12]. This is the basis of the Independent And-Parallelism (IAP) model (see e.g. [6, 7, 13]). Thus, the aim of automatic parallelization in this model is to detect the independence of sets of goals and to choose the best parallelization among all existing possibilities.

This work can be done at run–time. However, checking independence of goals in the resolvent at run–time is not always straightforward. In general, it implies performing (not inexpensive) checks which guarantee the sufficient conditions for independence, prior to the execution of the goals involved, and also considering all possible combinations of goals in all resolvents, in order to choose the best parallelization possible [6, 15]. An interesting way of reducing both sources of overheads is to mark at compile–time selected program literals for parallel execution and, when independence can not be determined statically (via program analysis) [4], generate parallelization tests which will minimize the checking overhead for the goals arising from such literals [7, 11, 8, 16, 14]. In this context, the parallelization process can be seen as a source to source transformation, which has been called *annotation*.

For this purpose, three different algorithms — called *annotators*, were presented in [16]: **MEL**, **CDG**, and **UDG**. Each of them implements a strategy for automatic parallelization based on a different heuristic. Since then, the algorithms have been extended and enhanced in various ways. We report on these extensions and present a new and exhaustive comparative study of the three different strategies from the point of view of their effectiveness in the task they were designed for.

*This work was funded in part by ESPRIT project 6707 "*ParForce*" and by CICYT project TIC93-0976.

2 Background

Although the algorithms for automatic parallelization can be viewed (and defined) independently of the concept of independence whose sufficient conditions will allow the parallelization, in our study we have instantiated this concept to the particular case of Strict Independence [6]. In this section we will briefly introduce this notion and the sufficient conditions associated to it, and the role and management of such conditions in the process of automatic parallelization.

2.1 Parallelization tests based on Strict Independence

Two goals are strictly independent [12] for a substitution iff (after applying the substitution) they do not share any variable. Let $g_1, \ldots, g_n$ be the literals to be parallelized. Following the above ideas, we must generate at compile–time a condition *i_cond* which, when evaluated at run–time, guarantees that the goals which are instantiations of such literals are strictly independent.

Consider the set of conditions which includes "true", "false", or any set, interpreted as a conjunction, of one or more of the tests $ground(x)$, $indep(x, y)$, where x and y can be goals, variables, or terms in general. Let $ground(x)$ be true when x is ground and false otherwise. Let $indep(x, y)$ be true when x and y do not share variables and false otherwise. If $g_1, \ldots, g_n$ are considered in isolation, rather than as part of a program, an example of a correct *i_cond* is $\{ground(x) | \forall x \in SVG\} \cup \{indep(x, y) | \forall (x, y) \in SVI\}$, where SVG is the set of variables x occuring in at least two elements of $\{g_1, \ldots, g_n\}$, and SVI the set of pairs of variables, each one occuring in a different element of $\{g_1, \ldots, g_n\}$ ($SVG \cap SVI = \emptyset$).

It is easy to see that in general a groundness check is less expensive than an independence check, and thus a condition, such as the one given, where some independence checks are replaced by groundness checks is obviously preferable.

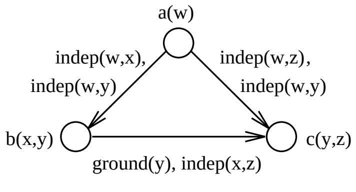
If the above condition is satisfied the literals are strictly independent for any possible substitution, thus ensuring that the goals resulting from the instantiations of such literals will also be strictly independent. However, when considering the literals involved as part of a clause and within a program, the test can be simplified since strict independence then only needs to be ensured for those substitutions which can appear in that program. This fundamental observation is clearly instrumental when using the results of abstract interpretation–based global analysis in the process of automatic parallelization.

2.2 Identifying and Simplifying Dependencies

The annotation process is divided into two subtasks. The first one is concerned with identifying the dependencies between each two goals in a clause and the minimum number of tests for ensuring their independence, based on the sufficient conditions applicable. The second one is concerned with the core of the annotation process, namely to apply a particular strategy in order to obtain an optimal (under such a strategy) parallel expression among all the possibilities detected in the previous step, hopefully further optimizing the number of tests. The first task is common to all the annotators, and is briefly summarized in the following. Note, however, that simplification is also applicable in the second task, once an expression has been built by a particular algorithm.

The dependencies between goals can be represented in the form of a dependency graph. Informally, a dependency graph is a directed acyclic graph where each node represents a goal and each edge represents in some way the dependency between the connected goals. A conditional dependency graph (CDG) is one in which the edges are adorned with sufficient conditions. If those conditions are satisfied, the dependency does not hold. In an unconditional dependency graph (UDG) dependencies always hold, i.e. conditions are always "false."

Example 1 *Consider a clause body with literals* `a(w)`, `b(x,y)`, `c(y,z)`. *The left–to–right precedence relation for the goals in the body of the clause can be represented using a directed, acyclic graph in which we associate with each edge which connects a pair of literals the sufficient condition for their strict independence, thus resulting in the following dependency graph:*



The conditions labeling the edges can be simplified by using compile–time information provided by an analyzer (or the user). For any clause C, the information actually known at every program point i in C can be expressed in what we call a *domain of interpretation GI*: a subset of the first order logic theory, such that each element κ of GI defined over the variables in C is a set of formulae (interpreted as their conjunction) which contains relevant information for the notion of independence the checks are built on. The simplification process [1] is then based on identifying tests which are ensured to either fail or succeed w.r.t. this information: if a test is ensured to succeed, it can be eliminated, and edges possibly removed; if it is ensured to fail, it can be reduced to false, yielding an unconditional edge.

3 Building Parallel Expressions

The vehicle for expressing and implementing independent and–parallelism used in this study will be the &-Prolog language. This language is essentially Prolog, with the addition of the parallel conjunction operator "`&`" (used in place of "`,`" –comma– when goals are to be executed concurrently), a set of parallelism–related builtins, which includes the groundness and independence tests already described, and a number of synchronization primitives which allow expressing both restricted and non–restricted parallelism. Combining these primitives with the normal Prolog constructs, such as "`->`" (if-then-else), users can conditionally trigger parallel execution of goals. For syntactic convenience an additional construct is also provided: the *Conditional Graph Expression (CGE)*. A CGE has the general form (*i_cond* `=>` $goal_1$ `&` $goal_2$ `& ... &` $goal_N$) and represents an if-then-else where the else part is a copy of the then part replacing "`&`" with "`,`". &-Prolog if-then-else expressions and CGEs can be nested in order to create richer execution graphs.

The annotation task is thus performed as source to source transformations of the (&-Prolog) program in which each clause is annotated with parallel expressions and conditions which encode the notion of independence used.

Example 2 *Different (but not all) possible CGEs for the goals* `a(w), b(x,y), c(y,z)` *of Example 1 would be:*

```
a(w), ( ground(y),indep(x,z) =>
        b(x,y) & c(y,z) )

indep(w,x),indep(w,z),indep(x,z),
  ground(y) => a(w) & b(x,y) & c(y,z)

indep(w,[x,y]) -> a(w) & b(x,y), c(y,z)
   ; a(w), ( ground(y),indep(x,z) =>
             b(x,y) & c(y,z) )
```

Given a clause, several different annotations are possible. Different heuristic algorithms implement different strategies to select among all possible parallel expressions for a given clause.

3.1 The MEL Algorithm

This algorithm is based on a heuristic which tries to find points in the body of a clause where it can be split into different expressions. One such point occurs where a new variable appears. Consider a goal which has the first occurrence of a variable in a clause, and this variable is used as an argument of another goal to the right of the first one. The condition in Strict IAP which must hold for two goals which share variables establishes that these variables must be ground; obviously this is not the case for the previously mentioned goals, and thus this is a point where it is not appropriate to annotate a parallel expression.

The algorithm proceeds in this manner from right to left, i.e. from the last goal of the body to the neck of the clause. The clause body is then broken into two at the points where shared new variables appear, and a parallel expression (a CGE) built for the right part of the sequence split. In proceeding backwards the underlying intention is to allow capturing the longest parallel expressions possible, since goals are generally expected to be more instantiated – and therefore more likely independent – towards the end of the clause. A similar heuristic would proceed forwards but splitting the body at the second occurrences of new variables.

Example 3 *Consider a clause* `h(X):- p(X,Y), q(X,Z),r(X),s(Y,Z)`. *Its body can be compiled (under the conditions for strict independence) to the following &-Prolog parallel expression:*

```
ground(X) => p(X,Y) & q(X,Z),
(indep(X,Y),indep(X,Z) => r(X) & s(Y,Z)).
```

Note that the body is split at `q(X,Z)` *and not at* `p(X,Y)`, *the largest expression being achieved in this way. Also, the first CGE should have the condition* `indep(Y,Z)` *but it does not since this condition is automatically satisfied by virtue of the fact that* `Z` *is a new variable.*

As originally described, the algorithm is instantiated to the particular case of strict independence. Nonetheless, we propose to define its main heuristic in terms of a CDG and thus make it independent of the particular notion of independence for which the conditions are constructed. Let a CDG be built for each clause in the program being annotated. The **MEL** algorithm can then be defined as finding edges in the CDG labeled with "false" and partitioning the clause body at these points (see [2] for a detailed description).

3.2 The UDG Algorithm

This algorithm starts with a graph $G(V, E)$ which is a UDG, i.e. all dependencies are unconditional. This graph is the result of making all conditional dependencies in a general graph hard dependencies. The algorithm seeks to maximize the amount of parallelism possible under these dependencies, i.e. the maximum parallelism achievable with no run-time tests. This is achieved if for any two goals for which a dependency is not present, they are annotated to be run in parallel — thus, no loss of parallelization opportunities occurs. For this, the transitive dependency relations among goals, represented by the graph edges, are considered, and conditions upon these established. Because the UDG is transitively closed, it holds that $\forall x, y \in V \cdot x\ dep^\star\ y \Leftrightarrow (y, x) \in E$.

The **UDG** algorithm works as follows [16]. It starts with the set of independent goals $I = \{p \in V \mid \forall x \in V \neg\exists(x, p) \in E\}$, those which do not have incoming edges. A set of partitions $PP = \{P \in 2^I \mid \forall p \in P \cdot \exists x \in V \cdot x\ dep^\star\ p\}$ is then built, so that there is at least one goal in $V - I$ for each of these partitions P which depends on all elements of P. These goals are grouped together so that $\forall P_i \in PP \cdot Q_i = \{x \in V \mid \forall p \in P_i \cdot x\ dep^\star\ p\}$. In this context, no loss of parallelism can occur when converting the graph into a linear (parallel) expression, if and only if $\forall P_1 P_2 \in PP$, $P_1 \cap P_2 = \emptyset \vee P_1 \cap P_2 = P_1 \vee P_1 \cap P_2 = P_2$, and $P_1 \cap P_2 = P_1 \Rightarrow \forall q_1 \in Q_1, q_2 \in Q_2 \cdot q_2\ dep^\star\ q_1$

Under such conditions, a corresponding parallel expression is constructed from a set of rules which guarantee that, for a transitively closed graph, no loss of parallelism occurs using such expression. However, note that from the definition of the partitions in PP, it always holds that $\forall P_1 P_2 \in PP \cdot P_1 \neq P_2$ and either:

(1) $P_1 \cap P_2 = \emptyset$, or

(2) $P_1 \cap P_2 = P_1$, or $P_1 \cap P_2 = P_2$, or

(3) $P_1 \cap P_2 = P$ s.t. $P \neq \emptyset, P \neq P_1, P \neq P_2$

and the **UDG** algorithm relies on either the first case or a special sub–case of the second one. In order to extend the algorithm to deal with all possible cases, different possible graph linearizations have to be considered. However, each possible extension implies a loss of parallelism.

We have considered extending the algorithm to deal with all of the second case and with the third case [2]. In particular, for case (2) there are two other sub–cases, in addition to the one which implies no loss of parallelism. For one of them there are two possible parallel expressions, and three for the other one; thus, we have five other possibilities. Let $P_1 = \{p_1\}$, $P_2 = \{p_1, p_2\}$, $Q_1 = \{q_{11}, q_{12}\}$ and $Q_2 = \{q_2\}$ be the minimum sets that fulfill the conditions in case (2) which allow building expressions for all of the five sub–cases. These will be as follows:

b.1 $(p_1, q_{11}, q_{12})\&p_2, q_2$ c.1 $(p_1, q_{11}, q_{12})\&p_2, q_2$

b.2 $p_1\&p_2, (q_{11}, q_{12})\&q_2$ c.2 $p_1\&p_2, q_{11}\&q_{12}, q_2$

c.3 $(p_1, q_{12})\&p_2, q_{11}\&q_2$

The most natural extension will be that of cases b.1 and c.1, because they apply the same intuition behind "sub–case a" (the original strategy) to the cases where the conditions of this one do not hold. This extension leads to the algorithm we have used for our study. Nonetheless, when the execution efficiency of the parallel expressions obtained is considered, it turns out that the extension may be more profitable if made in another direction.

This can be seen with a simple experiment. Consider assigning to each of the goals above an upper bound on its granularity and computing all possible combinations of granularities of all the goals. Then, the efficiency of each of the five possible parallel expressions can be computed. In Table 1 the percentage of combinations where each parallel expression behaves best is shown. The percentage includes the situations where all the expressions behave the same, and thus the total percentage can add up to more than 100%.

Max.Goal	% Best Case				
Grain	b.1	b.2	c.1	c.2	c.3
1	0	100	0	100	0
2	18	100	12	100	12
3	22	97	15	96	12
4	23	95	16	93	11
5	24	94	16	92	11
6	24	93	17	91	10
7	24	92	17	90	10
8	24	92	17	89	10
9	24	91	17	88	10
10	24	91	17	88	9

Table 1: Performance test for parallel expressions

From the table, it is clear that the best parallelization strategy corresponds to the second option in both cases (b.2 and c.2). This is due to the fact that this strategy performs a better load balancing of parallel tasks with goals which are already balanced (i.e. have almost the same granularity, as with maximum grain of 1 or 2) or for which the differences in grain size are not high. When a bigger difference is allowed (increasing the maximum permitted goal grain size) the average efficiency of this strategy decreases somewhat, while the strategy of b.1 and c.1 — that used to extend the algorithm — progressively behaves better, but in any case the asymptotic values seem to stabilize.

It is worth noting that this result points out the importance of having granularity information on the goals being annotated, so that the annotators could attempt a load balancing of expressions. Granularity information is also useful in controlling parallel execution (see [10]). Unfortunately, having good measures for the granularity of goals is a difficult task. In their absence, the strategy of cases b.2 and c.2 should be pursued.

3.3 The CDG Algorithm

This algorithm is quite close to the previous one, except that in this case the conditional dependencies present in a CDG $G(V, E)$ are used. To do this, all possible states of computation which can occur w.r.t. the conditions present in the graph are considered, and the body goals annotated with the best parallel expressions achievable under these conditions. The algorithm starts with the same set I of independent goals as above [16]. The main difference resides in that goals depending unconditionally on goals in I are not coupled to them (i.e. the closure of the dependency relation upon each P_i and corresponding Q_i in the **UDG** algorithm is not performed). On the contrary, the **CDG** algorithm focuses on the conditional dependencies present in the graph.

Consider the set $D = V - I$ of dependent goals. The sets of conditions other than "false" in labels of edges between goals in I and goals in D, $IConds = \{label((p,x)) \neq false \mid (p,x) \in E, p \in I, x \in D\}$, and in labels of edges among goals in D, $DConds = \{label((x,y)) \neq false \mid (x,y) \in E, x,y \in D\}$ are built. The algorithm proceeds by incrementally constructing the parallel expression exp as follows, let $I = \{p_1, \ldots, p_n\}$:

(1) if $D = \emptyset$ then exp is $p_1 \& \ldots \& p_n$

(2) if $D \neq \emptyset$, $DConds = IConds = \emptyset$ then exp is built using **UDG**

(3) if $D \neq \emptyset$, $DConds \neq \emptyset$, $IConds = \emptyset$ then exp is $p_1 \& \ldots \& p_n, exp_1$, where exp_1 is recursively computed for $G(V - I, E_1)$ with $E_1 = E - \{(p,x) \in E \mid p \in I\}$

(4) if $D \neq \emptyset$, $DConds \neq \emptyset$, $IConds \neq \emptyset$ then exp is constructed from the boolean combinations of the elements of $IConds$

For each boolean combination C the graph $G(V, E)$ is updated as if the conditions in C held, that is, all conditions in labels of edges of E which are implied by elements of C are deleted and all labels with conditions which are incompatible with some element of C rewritten into "false." Note that an edge can be removed if its label becomes void (i.e. "true"). In [16] an algorithm to perform this updating in the particular case of Strict IAP is presented. The parallel expressions resulting from recursively applying the **CDG** algorithm after this updating are annotated as if-then-elses and combined in a simplified form.

Note that, in case (3) of the algorithm, an unconditional parallel expression is built for elements in I followed sequentially by another expression recursively computed for the rest of the goals. No consideration is made regarding the unconditional dependencies which can occur from other goals to goals in I. The **UDG** algorithm, on the other hand, does this, and groups goals depending unconditionally on those of I together and with those on which they depend, building a different expression for the different groups of goals made. An extension of the **CDG** algorithm in this direction will extract from the CDG the subgraph of unconditional dependencies on goals of I (for the case mentioned above) and behave as **UDG**, returning an updated graph to the original algorithm. This extension will allow a one–to–one correspondence between both algorithms, the expressions constructed by each of them being the same, modulo the conditions present in conditional expressions.

Example 4 *Consider the clause* `h:- p(X),q(Y), r(X),s(X,Y)`. *There are unconditional dependencies for* `r(X)` *on* `p(X)` *and* `s(X,Y)` *on* `p(X)` *and* `q(Y)`, *and a dependency (labeled* `ground(X)`*) for* `s(X,Y)` *on* `r(X)`. *The original* **CDG** *algorithm will annotate its body as:*

```
p(X)&q(Y), (ground(X) => r(X) & s(X,Y)).
```

UDG *regards the dependency as unconditional:*

```
(p(X), r(X)) & q(Y), s(X,Y).
```

While its equivalent conditional expression, which **CDG** *will yield if suitable modified, is:*

```
ground(X) -> (p(X),r(X)) & (q(Y),s(X,Y))
           ; (p(X),r(X)) & q(Y), s(X,Y).
```

its worst case subexpression being that of **UDG**.

Further extensions of the **CDG** algorithm in the same direction can be done [2]. The heuristic of considering all combinations of conditions can be replaced by the **UDG** strategy of partitioning the graph into strongly dependent groups of goals. In this case the effect would be that the parallel expressions being annotated will look like conditional expressions nested inside unconditional ones. This turns out to be a different algorithm than **CDG**: it results in the same **UDG** algorithm performing not only on unconditional edges of the graph but also on conditional ones, annotating however such conditional edges with the required conditions.

4 Experimental Results

The **MEL**, **CDG**, and **UDG** algorithms have been integrated in the &-Prolog system [11] parallelizing compiler. Compiler switches determine whether or not code will be parallelized and, if so, through which type of analysis and annotator. In our experiments, a number of analyses have been used, both local and global. Since the focus of this paper is on the annotation algorithms, we only present herein results for the cases of local analysis and the most powerful of the global analyses available. A study of the performance of different analyzers can be found in [1].

Bench.	Cl	AvG	G	CGE	AvC	C
aiakl	7	3.00	5	2	3.50	5
ann	65	3.32	6	26	2.62	6
bid	18	2.78	5	8	2.50	4
boyer	10	3.60	6	2	2.00	2
browse	9	2.89	5	5	2.20	3
deriv	5	3.20	4	4	2.00	2
fib	1	6.00	6	1	2.00	2
grammar	4	2.50	3	4	2.25	3
hanoiapp	1	6.00	6	1	4.00	4
mmatrix	3	2.33	3	2	2.00	2
occur	3	3.00	4	2	2.00	2
progeom	6	3.00	5	3	3.00	4
qplan	47	4.00	9	31	2.68	5
qsortapp	2	3.50	4	1	4.00	4
query	2	4.50	6	2	2.00	2
rdtok	46	3.43	8	0	0.00	0
read	37	4.14	7	2	2.00	2
serialize	6	3.17	4	2	3.00	3
tak	2	5.00	7	1	4.00	4
tictactoe	37	4.24	48	5	2.20	3
warplan	26	3.69	10	16	2.56	5
zebra	2	10.50	19	3	3.33	6

Table 2: Benchmark Profile

A relatively wide range of programs (available by FTP at *clip.dia.fi.upm.es*) has been used as benchmarks. Due to lack of space, they are not discussed here. Instead, Table 2 gives (in our view) more insight into their complexity useful for the interpretation of the results. For each benchmark, columns show the number of clauses for which an annotation is considered[1] ("Cl"), the average and maximum number of goals in these clauses ("AvG" and "G"), the number of CGEs whose creation is attempted by the annotators ("CGE"), and the average and maximum number of goals in such CGEs ("AvC" and "C"). The rationale behind the three last columns lays in the treatment of builtins and side–effects. Annotators share a common subprocess which partitions the clause body at the points where these occur, so that annotation is not concerned with them (in general, side–effects cannot be allowed to execute freely in parallel with other goals).

[1]Facts and clauses involved in query preparation are not considered.

4.1 Annotation Efficiency

Table 3 presents the results in terms of annotation times in seconds (SparcStation 10, one processor, SICStus 2.1, native code). It shows for each annotator the average time out of ten executions in two different situations: with information given by a local analysis ("local" in the tables), and with that provided by a global analysis ("global") based on a combination of the Sharing+Freeness and ASub abstract domains [5].

Bench.	local			global		
prog.	MEL	CDG	UDG	MEL	CDG	UDG
aiakl	0.26	0.26	0.24	0.37	0.36	0.36
ann	1.55	1.55	1.43	7.60	7.60	7.53
bid	0.39	0.39	0.36	0.48	0.45	0.46
boyer	0.34	0.31	0.31	0.68	0.66	0.64
browse	0.53	0.46	0.45	0.63	0.56	0.55
deriv	0.20	0.18	0.18	0.27	0.26	0.25
fib	0.13	0.11	0.11	0.15	0.15	0.14
grammar	0.17	0.15	0.15	0.21	0.20	0.20
hanoiapp	0.18	0.18	0.16	0.22	0.20	0.20
mmatrix	0.21	0.19	0.19	0.22	0.21	0.20
occur	0.26	0.25	0.24	0.28	0.27	0.26
progeom	0.20	0.19	0.18	0.25	0.24	0.24
qplan	1.59	1.67	1.35	3.63	3.43	3.43
qsortapp	0.17	0.16	0.16	0.19	0.18	0.18
query	0.26	0.23	0.23	0.29	0.27	0.28
rdtok	0.87	0.79	0.80	1.87	1.82	1.84
read	0.90	0.82	0.82	2.02	1.99	2.01
serialize	0.22	0.20	0.20	0.41	0.38	0.39
tak	0.17	0.15	0.15	0.23	0.21	0.21
tictactoe	0.90	0.81	0.81	2.08	2.03	2.02
warplan	0.54	0.54	0.51	2.89	2.86	2.77
zebra	2.08	300	0.57	4.96	4.65	4.64

Table 3: Annotation Efficiency

4.2 Performance of CGEs and Tests

One way to measure the effectiveness of the annotators is to count the number of CGEs which actually result in parallelism and to study the overhead introduced in the program by the tests generated. For this purpose we have measured the total number of checks which occur in the annotated programs ("T" in the tables), the number of these which are not checked in the execution of the program ("N"), and, for the rest, the number of them which always succeed ("S"), always fail ("F"), or sometimes succeed and sometimes fail ("SF"). Also, the number of times the checks have succeeded ("TS") or failed ("TF") during execution, and the number of parallel expressions

Bench.	Info	Ann	ground/indep							E
			T	N	S	F	SF	TS	TF	
aiakl	local	mel	0/10	0/0	0/10	0/0	0/0	0/10	0/0	2
		cdg	4/42	2/38	2/4	0/0	0/0	2/4	0/0	2
		udg	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0
	global	all	0/0	0/0	0/0	0/0	0/0	0/0	0/0	2
bid	local	mel	7/12	2/0	4/12	1/0	0/0	17/44	1/0	27
		cdg	10/19	5/7	4/12	1/0	0/0	17/44	1/0	27
		udg	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0
	global	all	0/0	0/0	0/0	0/0	0/0	0/0	0/0	27
progeom	local	mel	2/2	0/0	1/2	1/0	0/0	13/220	13/0	110
		cdg	2/2	0/0	1/2	1/0	0/0	13/220	13/0	110
		udg	0/0	0/0	0/0	0/0	0/0	0/0	0/0	0
	global	all	0/0	0/0	0/0	0/0	0/0	0/0	0/0	110

Table 4: No Checks with "global"

Bench.	Info	Ann	ground/indep							E
			T	N	S	F	SF	TS	TF	
deriv	local	mel/cdg	4/16	0/ 0	4/ 16	0/ 0	0/ 0	538/ 2152	0/ 0	538
	global	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	538
mmatrix	local	mel/cdg	2/ 8	0/0	2/8	0/0	0/0	182/728	0/ 0	182
	global	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	182
occur	local	mel/cdg	2/ 5	0/0	2/5	0/0	0/0	252/279	0/ 0	252
	global	all	0/ 1	0/1	0/0	0/0	0/0	0/0	0/ 0	252
qsortapp	local	mel/cdg	0/ 1	0/0	0/1	0/0	0/0	0/250	0/ 0	250
	global	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	250
query	local	mel/cdg	1/ 4	0/0	0/4	1/0	0/0	0/4	2/ 0	1
	global	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
read	local	mel/cdg	1/ 6	0/0	1/6	0/0	0/0	1/6	0/ 0	1
	global	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
serialize	local	mel/cdg	0/ 4	0/0	0/4	0/0	0/0	0/36	0/ 0	9
	global	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	9
tictactoe	local	mel/cdg	10/ 3	0/0	10/3	0/0	0/0	29796/5176	0/ 0	11124
	global	all	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	11124
all	local	udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	0

Table 5: MEL=CDG — Identical Code in "global"

which have been effectively run in parallel as a result ("E"). We have parallelized our benchmarks using the three annotators in the two different situations of analysis already mentioned. The results for each benchmark and each of the situations are shown in tables 4, 5, 6, 7, and 8.

Bench.	E
fib	986
grammar	0
rdtok	0
tak	2372

Table 6: No Checks — Identical Code

Table 6 shows programs for which the parallelized result is identical in all cases. Table 4 gives the figures for the programs for which all algorithms give the same result for "global" but different otherwise. In the programs of Table 5, **MEL** and **CDG** yield the same result (not **UDG**), but it is different for "global" and "local." The same happens in Table 7, but the result of **UDG** is the same with "global" and "local." The rest of the programs appear in Table 8.

4.3 Speedup Results

An arguably better way of measuring the effectiveness of the annotators is to measure the speedup achieved: the ratio of the parallel execution time of the program (ideally for an unbounded number of processors) to that of the sequential program. In order to concentrate on the available parallelism itself, without the limitations imposed by a fixed number of physical processors, a novel evaluation environment, called **IDRA**, has been defined in [9]. **IDRA** takes as input a *real* execution trace file of a parallel program and the time for its sequential execution, and computes the achievable speedup for any number of

Bench.	Info	Ann	ground/indep							E
			T	N	S	F	SF	TS	TF	
boyer	local	mel/cdg	4/ 2	0/1	3/1	1/0	0/0	42/14	38348/ 0	14
	global	mel/cdg	4/ 0	0/0	3/0	1/0	0/0	42/0	38348/ 0	14
	all	udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	0
browse	local	mel/cdg	3/ 7	0/2	1/4	2/0	0/1	60/16300	25/ 20	4105
	global	mel/cdg	2/ 2	0/0	0/1	2/0	0/1	0/4105	25/ 20	4105
	all	udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	0

Table 7: CDG/MEL Identical Code

Bench.	Info	Ann	ground/indep							E
			T	N	S	F	SF	TS	TF	
ann	local	mel	14/ 36	3/19	3/12	5/1	3/4	168/183	207/ 93	99
		cdg	22/ 46	6/29	5/12	8/1	3/4	180/183	297/ 93	99
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	0
	global	mel	6/ 14	0/3	0/6	3/1	3/4	75/111	138/ 93	99
		cdg	12/ 18	2/8	1/5	6/1	3/4	81/105	228/ 93	99
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	0
hanoiapp	local	mel	2/ 1	0/0	2/1	0/0	0/0	510/255	0/ 0	255
		cdg	5/ 1	2/1	3/0	0/0	0/0	765/0	0/ 0	255
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	0
	global	mel	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	255
		cdg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	255
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	255
qplan	local	mel	13/ 57	9/47	3/10	1/0	0/0	6/12	3/ 0	7
		cdg	16/ 84	12/74	3/10	1/0	0/0	6/12	3/ 0	7
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	0
	global	mel	2/ 1	2/1	0/0	0/0	0/0	0/0	0/ 0	7
		cdg	2/ 1	2/1	0/0	0/0	0/0	0/0	0/ 0	7
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	7
warplan	local	mel	14/ 11	3/3	6/8	2/0	3/0	105/47	50/ 0	66
		cdg	28/ 15	13/9	8/5	3/1	4/0	113/45	58/ 4	66
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	6
	global	mel	14/ 7	3/1	6/6	2/0	3/0	105/33	50/ 0	66
		cdg	28/ 10	13/6	8/3	3/1	4/0	113/29	58/ 4	66
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	6
zebra	local	mel	0/ 250	0/247	0/2	0/1	0/0	0/112	0/ 56	1
		cdg	1/ 4835	0/4729	1/96	0/10	0/0	56/3346	0/ 420	1
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
	global	mel	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
		cdg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1
		udg	0/ 0	0/0	0/0	0/0	0/0	0/0	0/ 0	1

Table 8: Other Programs

processors. Trace files are encoded descriptions of the events occurred during the execution of a program. Since &-Prolog generates all possible parallel tasks of a parallel program, regardless of the number of processors in the system, all possible execution graphs, with their *exact* execution times, can be constructed from this data. The results have been shown to be very good approximations to the *best possible* parallel execution [9]. Though ideal, match closely the actual speedups obtained in the &-Prolog system for the number of processors available.

The results for a representative subset of the benchmarks used are presented in figures 1, 2, and 3. For each benchmark and situation of analysis, a diagram with speedup curves obtained with **IDRA** is shown. Each curve represents the speedup achievable for the parallelized version of the program obtained with one annotator.

5 Discussion

Annotation times are fairly acceptable for all annotators. **MEL** and **CDG** usually take the same time, with a slight difference favoring **CDG** for simpler programs. On the contrary, **MEL** takes less time for complex programs, like zebra. Note that complexity here is measured as the number of literals in clauses: the higher the number of literals, the more linearizations of the clause graph are possible and this dominates the complexity

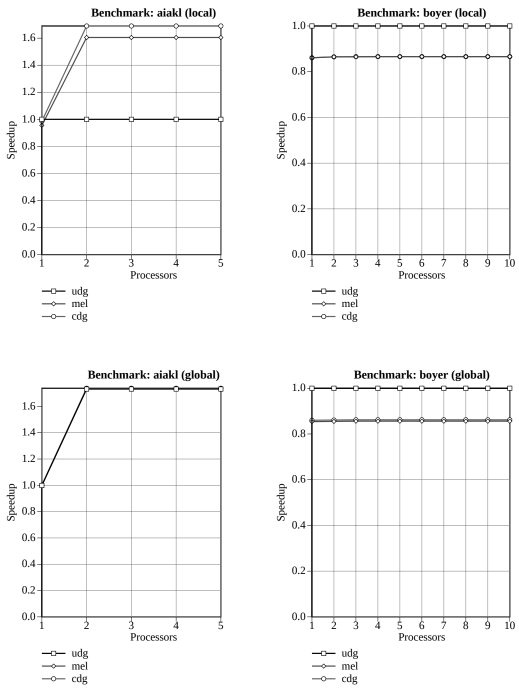


Figure 1: Dynamic Tests — Part I

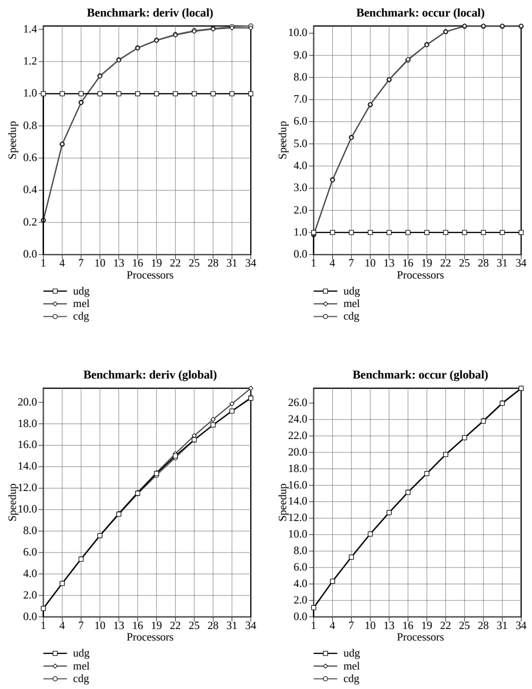


Figure 2: Dynamic Tests — Part II

of **CDG**, as it tries to consider all possible alternatives. **UDG** takes much less than the other two without information (from global analysis), because in this case it can rarely find any opportunities for parallelization. When having information, it takes as much as the other two.

Regarding the parallelized programs resulting from annotation, we identify several classes of programs. Two purely sequential programs and two (simple) parallel programs appear in Table 6, the simplest cases. The annotators are successful at detecting such sequentiality and do not generate any parallel expression. In the case of simple parallel programs, where independence of goals can be inferred even with a local analysis of the clauses, all the annotators are able to exploit this (unconditional) parallelism, with no checks.

Programs whose parallelization is more complex appear in Table 5. **MEL** and **CDG**, as well as **UDG** (when having good information) are able to extract parallelism to a great extent. This is shown by the fact that none of the checks ever fail at execution time. For **MEL** and **CDG** the annotated code is exactly the same, and thus the same parallelism is exploited. The worst case is that of **UDG**, which cannot exploit any parallelism without information.[2] When having information, its annotated code is also identical to that of the other two: all annotators are able to extract the same amount of parallelism without the need for checks.

For more complex programs, like those of Table 4, the differences in the behavior of **MEL** and **CDG** are more apparent. Once again, for these programs the three annotators behave the same when having good global information, and extract the same parallelism as when not having such information, but without checks. Without information, though, annotators are forced to place some checks to be executed at run–time. In the case of **CDG**, it turns out that most of these checks are not actually executed at run–time because many of the possible parallel expressions annotated by **CDG** are not used in the execution of the program. Nonetheless, note that in the case of aiakl, the expression exploited has many less checks than the corresponding one annotated by **MEL** (for the same goals in the program): 2 ground checks and 4 indep checks against 10 indep checks. This is due to the graph linearization performed by **CDG**, taking all possibilities into account. If-then-elses built by **CDG** can be viewed as an "indexing" over the possible parallel expressions, based on some checks. In aiakl, this indexing is able to lead to the parallel ex-

[2] This can actually be observed in all tables, except for the cases of warplan and zebra; the parallelism exploited in these cases is marginal, and with granularity analysis it would be avoided.

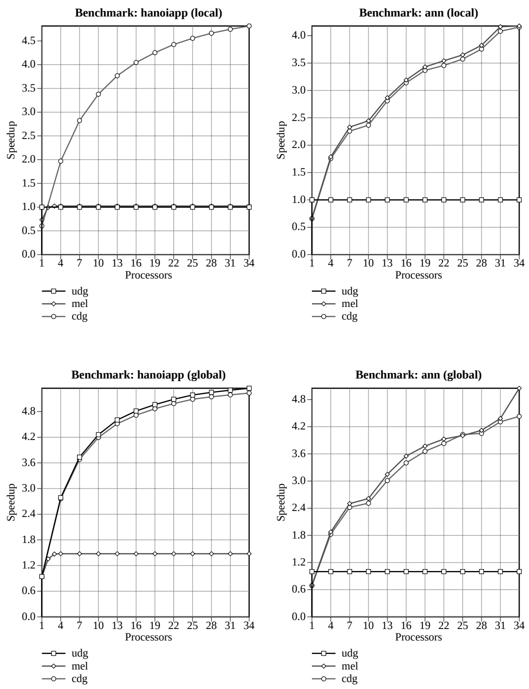


Figure 3: Dynamic Tests — Part III

pressions with less effort than that required by **MEL**, which simply puts conditions at certain points in the clause.

In Table 7 there are two programs which are harder to parallelize. **UDG** cannot extract parallelism, because there is no unconditional parallelism. **MEL** and **CDG** extract the same amount of conditional parallelism, but for both algorithms the number of checks is less when having information. In fact, little parallelism is obtained. In the case of boyer, significant parallelism can be exploited but only using the concept of non–strict independence [12, 3]; in browse, although a good number of goals are executed in parallel, a critical part of the algorithm is still sequential.

Programs in Table 8 deserve more discussion. The first thing to be noticed is that in some cases **UDG** is not able to extract parallelism even with information — this happens for ann, and for warplan and zebra, in which the parallelism extracted is marginal. On the contrary, for hanoiapp and qplan the same parallelism as the other two annotators is extracted by **UDG**. Considering the high complexity of qplan, the analysis turns out to be quite effective.

Regarding **MEL** and **CDG**, it has to be noted that in most programs of Table 8 the overhead in number of checks of **CDG** is high. Although in some cases (e.g. qplan) it happens (as it happened in aiakl or bid) that these extra checks (and the corresponding expressions) are discarded at execution time, in other cases they yield some overhead also at execution time. This is the case for ann, as can be seen in Figure 3, where speedups for **CDG** are always lower than for **MEL**. The same happens also for warplan.

An interesting case is that of hanoiapp. Its speedup curves (also in Figure 3) illustrate a case where **CDG** achieves good speedups while **MEL** shows very little speedup. **MEL** correctly but inefficiently parallelizes a call to hanoi and a call to append, while **CDG** parallelizes a call to hanoi with a sequence composed of the other call to hanoi and a call to append. **MEL** needs an indep check, while **CDG** uses instead a ground check, which is much less expensive.

In general, though, the differences in speedups are not significant. Exceptions are hanoiapp, as discussed, and programs with very little parallelism, as in aiakl (Figure 1). In this case, as in hanoiapp, **CDG** does better than **MEL** due to its ability to annotate different possibilities for the same clause body. In this program only one body with two parallel expressions is parallelized, and since very little speedup is achieved, the different annotations of the two algorithms are more relevant. For other programs with good speedups, as those in Figure 2, this does not happen.

6 Conclusions

We have studied the effectiveness of three algorithms for parallelization of logic programs using *strict independence* by comparing a number of measures. The algorithms have been implemented and incorporated in a complete parallelizing compiler. This compiler also includes a number of program analyzers based on well–known approximation domains. The complete system proves the task of automatic program parallelization feasible. Performance of the annotators at this task, in terms of the time taken in annotating the programs, shows them to be practical, although **CDG** in some cases (e.g. zebra) requires some improvement.

The comparison study shows that **MEL** and **CDG** give very similar results in practice. Despite this, each one of them has its advantages and disadvantages. **CDG** appears to be better when not having information if the programs are simple, or for more complex programs, if good analyses are available. In the latter case, **CDG** can be able to extract more sophisticated parallelism than **MEL**. On the contrary, for complicated programs for which the analysis is not accurate enough (or no analysis is available), **CDG** can cause significant overhead, and thus **MEL** is a reasonable alternative. To avoid slow–downs caused by too much dependency checking overhead, there is always the option of using **UDG**. However, **UDG** is not effective when not having good analyses which yield accurate information.

Several improvements are possible in order to obtain performance beyond our results. The discussed extensions of **UDG** and **CDG** [2] should also be compared in practice. Providing the algorithms with granularity information and allowing them to perform a load balancing of annotated goals based on this information can be of great importance. Also, **CDG** can be enhanced with heuristics to determine best and/or most probable alternatives, in order to reduce its overheads.

References

[1] F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. TR CLIP7/93.0, UPM, October 1993.

[2] F. Bueno, M. García de la Banda, and M. Hermenegildo. The MEL, UDG, and CDG Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-Parallelism: A Comparative Study. TR CLIP3/94.0, UPM, Jan. 1994.

[3] D. Cabeza and M. Hermenegildo. Towards Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *International Static Analysis Symposium*, Namur, Belgium, September 1994. To appear.

[4] J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.

[5] M. Codish, A. Mulkers, M. Bruynooghe, M. García de la Banda, and M. Hermenegildo. Improving Abstract Interpretations by Combining Domains. In *Proc. ACM SIGPLAN Symposium on Partial Evaluation and Semantics Based Program Manipulation*, pages 194–206. ACM, June 1993.

[6] J. S. Conery. *The And/Or Process Model for Parallel Interpretation of Logic Programs.* PhD thesis, The University of California At Irvine, 1983. Technical Report 204.

[7] D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.

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

[9] M.J. Fernández, M. Carro, and M. Hermenegildo. IDRA (IDeal Resource Allocation): A Tool for Computing Ideal Speedups. In *ICLP'94 Workshop on Parallel and Data Parallel Execution of Logic Programs*, 1994.

[10] P. López García, M. Hermenegildo, and S.K. Debray. Towards Granularity Based Control of Parallelism in Logic Programs. In *Proc. of PASCO'94*. World Scientific Publishing Company, September 1994.

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

[12] 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*, 1994. To appear.

[13] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, University of Texas at Austin, August 1986. Dept. of Computer Science TR-86-20.

[14] D. Jacobs and A. Langen. Compilation of Logic Programs for Restricted And-Parallelism. In *European Symposium on Programming*, pages 284–297, 1988.

[15] Y. J. Lin and V. Kumar. AND-Parallel Execution of Logic Programs on a Shared Memory Multiprocessor: A Summary of Results. In *Fifth International Conference and Symposium on Logic Programming*, pages 1123–1141. MIT Press, August 1988.

[16] K. Muthukumar and M. Hermenegildo. 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*, pages 221–237. MIT Press, June 1990.