The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism

K. Muthukumar
MCC and The University of Texas at Austin, C.S. Dept
Austin, TX 78712, USA
muthu@cs.utexas.edu

M. V. Hermenegildo
Universidad Politécnica de Madrid (UPM)
Facultad de Informática
28660-Boadilla del Monte, Madrid - Spain
herme@fi.upm.es *or* herme@cs.utexas.edu

Abstract

There has been significant interest in parallel execution models for logic programs which exploit Independent And-Parallelism (IAP). In these models, it is necessary to determine which goals are independent and therefore eligible for parallel execution and which goals have to wait for which others during execution. Although this can be done at run-time, it can imply a very heavy overhead. In this paper, we present three algorithms for automatic compile-time parallelization of logic programs using IAP. This is done by converting a clause into a graph-based computational form and then transforming this graph into linear expressions based on &-Prolog, a language for IAP. We also present an algorithm which, given a clause, determines if there is any loss of parallelism due to linearization, for the case in which only unconditional parallelism is desired. Finally, the performance of these annotation algorithms is discussed for some benchmark programs.

1 Introduction

Parallel execution (Or- and And-parallelism [4]) is starting to prove itself as an effective way of achieving improved performance in logic programming systems. In particular, there has been significant interest (e.g. see [6], [16], [4], [3], [10], [12], [19], etc.) in parallel execution models for logic programs which exploit "independent and-parallelism," where, only goals which

don't share any variables at run-time[2] are run in parallel. These models thus have the very desirable characteristics of offering performance improvements through the use of And-parallelism, while at the same time preserving the conventional "don't know" semantics of logic programs and the computational complexity expected by the programmer, as shown in [9].[3] However, in these models it is necessary to determine which goals are independent and therefore eligible for parallel execution. Although this can be done at run-time [12, 4], it can imply a significant overhead. In this paper we are interested in performing as much of the work as possible at compile-time. Chang [3] proposed an approach which generated a single graph for a clause from a worst case analysis thus being somewhat limited, given the global analysis technology used. DeGroot [6] proposed a way of representing a fixed set of execution graphs in an expression generated at compile-time, choosing among them at run-time through some checks. An important issue in such a framework is the type of parallelizing expressions allowed, because it determines the flexibility available to the compiler and the complexity of the run-time system. In the context of the RAP-WAM model[10] more flexible graph expressions (which can represent arbitrary execution graphs and conditions – including often needed conjunctive checks) were proposed in the form of a "language", &-Prolog [15], which subsumes full Prolog. In &-Prolog, graph expressions are built using if-then-else, synchronization and dependence-checking primitives. &-Prolog will be used in this paper for the sake of concreteness and because of the convenience of its Prolog-compatible syntax which makes it possible to describe the parallelization techniques as a series of rewritings of the original Prolog program. However, the parallelization techniques that will be proposed are applicable to any model using annotation-based parallelization, such as, for example Kale's [16].

In general, the task of parallelizing a given program through compile-time analysis can be *conceptually* viewed as comprising two steps:

1. a local or global analysis of the program in order to gather as much information as possible regarding the terms to which program variables will be bound and

2. given that information, a rewriting of the program into another one which contains expressions which will cause the parallel execution of some goals, perhaps under certain run-time conditions.

The main topic of this paper is the second point above, and, in particular, the generation of &-Prolog parallel expressions. It is assumed throughout the paper that any binding information which might have been gathered through global analysis [2, 5, 13] is available and will be used to improve the expressions generated.

It is worth noting that while arbitrary graphs can be implemented in &-Prolog with the use of `wait` primitives [14], it is of practical interest for

[2]Or are "non-strictly independent," see [9].

[3]Another way of achieving this is by running only determinate goals in and-parallel fashion [1]. This very interesting approach is complementary to Independent And-parallelism, providing an efficient method for running dependent goals which are determinate in parallel.

efficiency reasons to restrict the expressions generated to *linear expressions*, i.e. parenthesized expressions with no `wait` built-ins, such as the &-Prolog examples presented in [15]. This restriction (basically corresponding to a conditional "fork and join" paradigm) was first proposed by DeGroot [6] and the type of And-parallelism thus generated is called "restricted." Guidelines for constructing correct annotations at compile-time were first proposed in [10]. Other theoretical results which are of direct importance in this process are presented in [9]. DeGroot [7] proposed a technique for generating graph expressions using a very simple heuristic, although the expressions generated tend to be rather large, with a significant number of checks, and with no provision for conjunctions of checks. Jacobs and Langen [11] describe a framework for compiling logic programs to an extension of DeGroot's graph expressions equivalent to that introduced in [10]. They propose two rules (SPLIT and IF rules) for transforming a dependency graph (such as those used in [12, 4, 3]) into graph expressions. Their paper sets interesting groundwork by describing such rules, but no algorithm or set of heuristics is given that would suggest how and when to use such rules in a parallelization process and it therefore doesn't represent a complete algorithm for our purposes.

In this paper we we propose a) three complete algorithms for compiling (rewriting) Prolog clauses into &-Prolog clauses containing parallel execution expressions, and b) an algorithm which determines if a given Prolog clause can be compiled into an &-Prolog parallel expression without loss of parallelism for the case in which only unconditional parallelism is desired. Essentially, the algorithms in the first point above involve heuristics which seek to maximize the amount of parallelism while, at the same time, minimizing the overhead associated with such parallelism. The rest of the paper proceeds as follows (proofs are omitted for the sake of brevity and they can be found in [15]): in section 2 we first deal with the important problem of characterizing in which cases a clause can be compiled into *linear* &-Prolog parallel expressions without loss of parallelism, for the case in which only unconditional parallelism is desired.[4] Section 3 then presents the three heuristic-based algorithms for compilation of logic programs into &-Prolog parallel expressions. Section 4 presents some results from the implementation of these algorithms. Finally, section 5 presents our conclusions.

2 Loss of parallelism in the conversion to linear expressions

In this section, we present an algorithm to determine if a given Prolog clause can be compiled into an &-Prolog parallel expression which achieves *Maximal Efficient Independent And-Parallelism*, MEIAP[9]. Basically, MEIAP stipulates that while trying to execute as many goals in parallel as possible it is ensured that

- Dependent goals never execute concurrently,

[4] See [15] for an algorithm which deals with the case in which run-time tests are present.

- Dependent goals never execute out of order, i.e. they execute in a left-to-right order and
- "Intelligent failure" (termination of sibling independent goals) is enforced.

These conditions allow the execution of goals in parallel while guaranteeing important correctness and complexity ("no-speedown") properties, as shown in [9]. Note that, in order to achieve MEIAP, a goal should be initiated as soon as all dependent goals to its left have finished executing.

In the next section, we introduce Conditional Dependency Graphs(CDG) and describe their underlying execution model. Similar dependency graphs have been used in various forms by other researchers in the area [12, 11, 4, 3].

2.1 Conditional Dependency Graphs

A CDG is a directed acyclic graph where the vertices are subgoals and each edge is labeled by a condition. The CDG associated with a clause C has a vertex for each subgoal in the body of C and an edge from subgoal A to subgoal B (denoted by the tuple (A,B)) if A is to the "left" of B. The condition labeling edge (A,B) is the one that needs to be satisfied so that subgoals A and B are *independent* of each other so that they can be executed in parallel.[5] As shown therein, given two goals a sufficient condition for their independence[6] can be formed as a conjunction of the following tests:

- `ground(X)` for each program variable `X` that occurs in both A and B
- `indep(X,Y)` for every pair of variables `X` and `Y` such that `X` occurs in A but not in B and `Y` occurs in B but not in A.

As an example, consider the clause `h(X,Y) :- a(X), b(Y), c(X,Y)`. The CDG for the body of this clause is shown in figure 3(a).

In the CDG execution model, the following two step cycles are performed repeatedly until all subgoals have been initiated. A cycle should start as soon as a subgoal finishes.

- `Edge Removal`: Remove every edge whose origin has finished executing. If a condition labeling an edge holds, remove that edge.
- `Subgoal Initiation`: Initiate all subgoals with no incoming edges.

It is clear that this model achieves MEIAP. Essentially, this is also the underlying model for And-Parallelism described in [12]. In general, though this model achieves MEIAP, the overhead associated with checking each and every edge and vertex in every cycle may be unacceptable and can hinder speed-up. Instead, we propose to compile the CDG into &-Prolog parallel expressions at compile-time.

Note that if a clause has N subgoals in its body, there will be *at most* N(N-1)/2 (= 1+2+ ... +(N-1)) edges in its CDG. Some of the possible edges

[5] The correctness of these conditions has been shown in [9].

[6] This includes non-strict independence [9] which uses identical conditions, although additional global analysis is required.

may not be in the CDG because the subgoals connected by these edges have been shown to be independent by virtue of the groundness and independence information obtained from a prior global analysis of the program.

An edge (A,B) may have the label `false`. In this case, compile-time analysis has shown that subgoal B is dependent on A and hence it can be initiated *only after* A's completion. In our representation, we do not label such edges at all i.e. unlabeled edges in our representation of CDG actually stand for edges with the label `false`. If all the edges of a CDG are unlabeled, then we call it an *Unconditional Dependency Graph* (UDG).

2.2 Necessary and Sufficient conditions for converting a UDG into a linear expression without loss of parallelism

In this section, we give necessary and sufficient conditions for UDGs so that they can be compiled into linear expressions (which use the & operator for parallelism) without loss of parallelism.[7] Later, we use these conditions to design algorithms to compile CDGs and UDGs into &-Prolog parallel expressions without loss of parallelism (Sections 3.1 and 3.2). Without loss of generality we consider only those UDGs whose set of edges is closed under *transitivity* i.e. if the UDG has edges (A,B) and (B,C), then it also has an edge (A,C).

In the following paragraph, we describe informally a recursive algorithm which checks if a UDG can be compiled into a linear expression without loss of parallelism. The basic idea behind the algorithm is as follows: consider subgoals in this UDG which can be executed in parallel. These correspond to vertices in the UDG which have in-degree = 0. Let these belong to a set P (step 1). Find out how the remaining vertices in the UDG (the set Q) are related to vertices in P (step 2). We investigate whether they can be executed in parallel with, or they should sequentially follow, a given subgoal in P. We find that they have to satisfy certain conditions (Lemmas 1 and 3). If not, the given UDG cannot be compiled into a linear expression without loss of parallelism. If they do satisfy these conditions, then some edges from the UDG are removed, Q is partitioned into subsets and it is recursively checked if the induced sub-UDGs satisfy these conditions (step 7). If that is the case, then the given UDG can be compiled into a linear expression without loss of parallelism.

Let the UDG = (V, E) where V is the set of vertices and E is the set of edges.

1. Let the vertices that have in-degree = 0 in the current graph belong to the set $P = \{p_1, \ldots, p_m\}$.

2. Consider the set $Q = V - P$. For each $q_i \in Q$, form the nonempty set $\mathcal{E}(q_i) = \{p_j | (p_j \in P) \wedge ((p_j, q_i) \in E)\}$. Let $S = \{\mathcal{E}(q_i) | q_i \in Q\} = \{S_1, \ldots, S_n\}$.

3. **Lemma 1** *The given UDG can be compiled into a linear expression without loss of parallelism, only if, for each S_i, S_j in S, either*

[7] Throughout this paper, we consider loss of parallelism that is caused only by *spurious* dependencies i.e two subgoals are sequentially executed even though they are independent.

- $S_i \cap S_j = \emptyset$, *or*
- *one must be a subset of the other.*

4. Now, we are going to (partially) compile the given UDG into a linear expression which would consist of the vertices in V, a parallel operator "&" and a sequential operator "," i.e. (A & B & C) means that A,B,C can be run in parallel, (A,B) means that A and B are executed sequentially in that order.

   For $R \in S$, define $\mathcal{F}(R) = \{q_i | \forall p_j \in P(p_j \in R \Leftrightarrow (p_j, q_i) \in E)\}$ i.e. $\mathcal{F}(R)$ is the set of all vertices in Q that must wait *only* for *all* vertices in R to finish executing before their execution can be initiated. Basically, the subexpression for each $S_i \in S$ and $\mathcal{F}(S_i)$, should satisfy the following conditions:

   - There should be & operators between all the elements of S_i so that they can be run in parallel.
   - The subexpressions involving elements of $\mathcal{F}(S_i)$ should sequentially follow the subexpressions involving elements of S_i and $\mathcal{F}(S_j)$ for each $S_j \subset S_i$

   Also, the following lemma holds for UDGs which are *closed under transitivity.*

   Lemma 2 *For each non-intersecting pair of sets S_i, S_j in S, there are no edges between a vertex in $\mathcal{F}(S_i)$ and a vertex in $\mathcal{F}(S_j)$.*

   Hence, the vertices in S_i can be executed in parallel with the vertices in S_j.

5. **Lemma 3** *Each S_i, S_j such that $S_i \subset S_j$, should satisfy the following condition:*

$$\forall uv((u \neq v \wedge u \in \mathcal{F}(S_i) \wedge v \in \mathcal{F}(S_j)) \Rightarrow (u, v) \in E)$$

   Else, the given UDG cannot be compiled into a linear expression without loss of parallelism.

6. In step 4, we saw how, in the linear expression,

   - vertices in P are related to each other (they can execute in parallel)
   - vertices in $\mathcal{F}(S_i)$ are related to vertices in S_i (vertices in $\mathcal{F}(S_i)$ have to sequentially follow vertices only in S_i and not in $P - S_i$)

   Now, we have to investigate the relationship among vertices in each $\mathcal{F}(S_i)$. For this we do the following: If $\mathcal{F}(S_i)$ contains only one element, then the subexpression for it is simply that element itself. Else, go to step 1 with the UDG formed by vertices in $\mathcal{F}(S_i)$

```
function udg_compilable_WLOP(V,E): boolean
/* WLOP = Without Loss Of Parallelism */
begin
  compute P,Q and S (steps 1 and 2); /* S = {S_1,...,S_n} */
  If the condition in lemma 1 is not satisfied then return false;
  If the condition in lemma 3 is not satisfied then return false;
  Answer := true;
  i := 1;
  Repeat
     V1 := F(S_i);
     E1 := edges between vertices in V1;
     Answer := Answer AND udg_compilable_WLOP(V1,E1);
     i := i+1;
  until (Answer = false) OR (i > n)
  return Answer;
end.
```

Figure 1: Algorithm for checking UDG parallelism

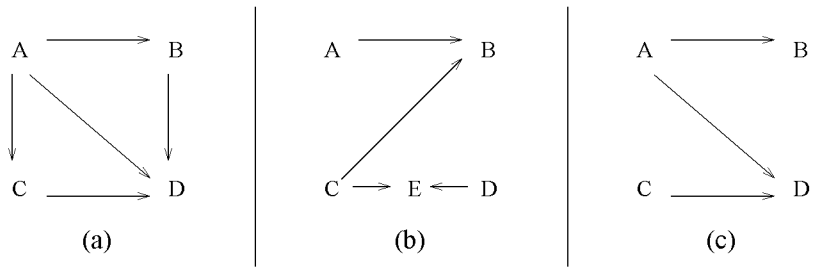

Figure 2: Example UDGs

2.3 Algorithm for checking UDG parallelism

Figure 1 contains a formal description of the algorithm in pseudo-pascal form. It returns the answer *true* if the given UDG can be compiled into a linear expression without loss of parallelism and the answer *false* otherwise. The correctness of this algorithm follows from the above lemmas.

2.4 Examples to illustrate UDG algorithm

This section illustrates the above algorithm with the aid of three examples. The first is a UDG that can be compiled into a linear expression without loss of parallelism; the second and third UDGs do not have this property.

• The UDG for the first example is shown in figure 2(a).
Steps 1 and 2: Initially, $V = \{A, B, C, D\}$ and $P = \{A\}$. Hence, $Q = V - P = \{B, C, D\}$. $\mathcal{E}(B) = \mathcal{E}(C) = \mathcal{E}(D) = \{A\}$. Hence $S = \{\{A\}\}$.
Steps 3 through 7: Since S is a singleton set, conditions in lemmas 1, 2 and

3 are trivially satisfied. $\mathcal{F}(\{A\}) = \{B, C, D\}$. Hence, the linear expression should be `A, sub-expression for B, C and D`. So we remove A from the vertex set and the edges (A,B), (A,C) and (A,D) from the edge set. The UDG has now three vertices B, C and D and two edges (B,D) and (C,D). We set $V = \{B, C, D\}$ and go to step 1.
Steps 1 and 2: $P = \{B, C\}$. Hence, $Q = \{D\}$. $\mathcal{E}(D) = \{B, C\}$. Hence, $S = \{\{B, C\}\}$.
Steps 3 through 7: $S = \{\{B, C\}\}$. Again, since S is a singleton set, conditions in lemmas 1, 2 and 3 are trivially satisfied. $\mathcal{F}(\{B, C\}) = \{D\}$. So, the linear expression for this subgraph is `(B & C), D.` and the linear expression for the given UDG is `A, (B & C), D.`

• The UDG for the second example is in figure 2(b).
Steps 1 and 2: $V = \{A, B, C, D, E\}$ and $P = \{A, C, D\}$. Hence, $Q = \{B, E\}$. $\mathcal{E}(B) = \{A, C\}$. $\mathcal{E}(E) = \{C, D\}$. Hence $S = \{\{A, C\}, \{C, D\}\}$.
Step 3: S has two elements, $\{A, C\}, \{C, D\}$ which do not satisfy the condition in lemma 1. Hence, this UDG cannot be compiled into a linear expression without loss of parallelism.

• The UDG for the third example is in figure 2(c).
Steps 1 and 2: $V = \{A, B, C, D\}$ and $P = \{A, C\}$. Hence, $Q = \{B, D\}$. $\mathcal{E}(B) = \{A\}$. $\mathcal{E}(D) = \{A, C\}$. Hence, $S = \{\{A\}, \{A, C\}\}$.
Steps 3 through 6: S satisfies the condition in lemma 1. $\mathcal{F}(\{A\}) = \{B, D\}$ and $\mathcal{F}(\{A, C\}) = \{D\}$. Since $\{A\} \subset \{A, C\}$, we check for the condition in lemma 3. $B \in \mathcal{F}(\{A\})$ and $D \in \mathcal{F}(\{A, C\})$, but (B, D) is not an edge in the given graph. So this condition is violated. Hence, this UDG cannot be compiled into a linear expression without loss of parallelism.

3 Algorithms for compiling Prolog clauses into &-Prolog parallel expressions

In this section, we describe three algorithms for compilation of Prolog clauses into &-Prolog parallel expressions. These clauses are assumed to be free of disjunctions.

For simplicity, the descriptions of these algorithms do not consider whether a given subgoal is a prolog builtin or whether it has any side-effect. A practical implementation of these algorithms would, of course, have to deal with these issues. However, they would essentially follow the steps described in these algorithms.

For all the three algorithms, we start with the given clause and construct the CDG for it. This CDG is then simplified using (a) the results of an abstract interpreter, or (b) user provided input or output mode information for non-builtin predicates, or (c) input or output mode information for builtin predicates.

• *The CDG algorithm:* this algorithm is closely related to the algorithm presented in the previous section. It seeks to maximize the amount of parallelism available in a clause, without being concerned about the size of the resultant &-Prolog expression. In achieving this objective, it may switch the positions of independent goals i.e. if A and B are two independent subgoals

and if A occurred to the left of B in the Prolog clause, in the compiled &-Prolog clause, A may be to the right of B in a sub-expression. Also, this algorithm uses IF-THEN-ELSE constructs in addition to CGEs [15] in the resultant &-Prolog clauses.

• *The UDG algorithm:* this algorithm is essentially the same as the CDG algorithm, except that only unconditional parallelism is exploited, i.e., only goals which can be determined to be independent at compile-time are run in parallel. The motivation for this algorithm is that groundness and independence checks are very expensive and contribute a significant overhead to the achieved and-parallelism. Thus, no run-time groundness or independence checks are generated.

• *The MEL algorithm:* this algorithm creates *only* CGEs in its expressions to achieve parallelism. In addition, it preserves the left-to-right order of subgoals in its expressions. Within these constraints, it seeks to maximize the number of goals to be run in parallel within a CGE. The results from the implementation of this algorithm were reported in [18].

3.1 CDG Algorithm

In this algorithm, we transform the simplified CDG into an &-Prolog parallel clause, by using, during intermediate stages, hybrid expressions which consist of CDGs and &-Prolog parallel expressions.

We start with $G :=$ the CDG for the given clause C and the boolean formula $B := true$. We seek to find an &-Prolog expression D corresponding to the graph G.

1. Let the vertices that have in-degree = 0 in G belong to the set $P = \{p_1, \ldots, p_m\}$.
2. Consider the *atomic* conditions ($\neq false$) on the edges going out of vertices in P. Let these belong to the set Q.
   - If $Q = \emptyset$, then let G' be the CDG obtained by removing from G those vertices which are in P and the edges coming out of such vertices.
     If G' is an empty graph at this point, then $D := (p_1 \& \ldots \& p_m)$.
     If G' is a UDG at this point, use the algorithm in section 3.2 to convert G to a linear expression D.
     Else, $D := (p_1 \& \ldots \& p_m), D'$, where D' is the &-Prolog parallel expression corresponding to $G := G'$ and $B := true$.
   - If $Q \neq \emptyset$, then take the conjuncts of the boolean combinations of the conditions in Q, simplify them and put them into a set R. Remove from R those combinations which are identically equal to $false$.
3. Let $R = \{r_1, \ldots, r_n\}$. Then, $D := \mathcal{S}(r_1 \rightarrow D_1; \ldots; r_n \rightarrow D_n)$ where, D_i is the &-Prolog parallel expression corresponding to $G_i := \mathcal{U}(G, r_i, P)$ and $B_i := B \wedge r_i$ and the function $\mathcal{S}$ is defined in step 6.
4. *The update function* $\mathcal{U}$: Given a CDG G, a conjunct of atomic conditions C and a set of vertices P, $G' = \mathcal{U}(G, C, P)$ is computed as follows:

- Initially, $G' := G$.
- If $C = true$, go to next step. Else, for each conjunct c in C, G' is changed as follows:

  If $c = ground(X)$, then if an edge has the label $ground(X) \wedge other\_conjuncts$ or $indep(X,Y) \wedge other\_conjuncts$ or $indep(W,X) \wedge other\_conjuncts$, its label is changed to $other\_conjuncts$.

  If $c = indep(X,Y)$, then consider the edges coming out of vertices in P. For each such edge which has the label $indep(X,Y) \wedge other\_conjuncts$ or $indep(Y,X) \wedge other\_conjuncts$, its label is changed to $other\_conjuncts$.

  If $c = \neg ground(X)$, then if an edge coming out of a vertex in P has the label $ground(X) \wedge other\_conjuncts$, its label is changed to $false$.

  If $c = \neg indep(X,Y)$, then if an edge coming out of a vertex in P has the label $indep(X,Y) \wedge other\_conjuncts$ or $indep(Y,X) \wedge other\_conjuncts$ or $ground(X) \wedge other\_conjuncts$, its label is changed to $false$.
- Remove each edge whose label is $true$ and remove the label from each edge whose label is $false$.

5. *The simplification function* $\mathcal{S}$: $\mathcal{S}(r_1 \rightarrow D_1; \ldots; r_n \rightarrow D_n)$ is defined as follows, where $R = \{r_1, \ldots, r_n\}$:
   - If $R = \{r_{i_1}, \ldots, r_{i_M}\} \cup \{r_{j_1}, \ldots, r_{j_N}\}$, where

     $$r_{i_k} = cond \wedge s_{i_k} \;\; 1 \leq k \leq M, \quad r_{j_k} = \neg cond \wedge s_{j_k} \;\; 1 \leq k \leq N$$

     then

     $$\mathcal{S}(r_1 \rightarrow D_1; \ldots; r_n \rightarrow D_n) := (cond \rightarrow D1; D2)$$

     where

     $$D1 := \mathcal{S}(s_{i_1} \rightarrow D_{i_1}; \ldots; s_{i_M} \rightarrow D_{i_M})$$
     $$D2 := \mathcal{S}(s_{j_1} \rightarrow D_{j_1}; \ldots; s_{j_N} \rightarrow D_{j_N})$$
   - If the conditions in R cannot be decomposed in the manner described above, they are all atomic, and so $\mathcal{S}(E) := E$.

3.1.1 Example to illustrate CDG algorithm

Consider the clause `h(X,Y) :- a(X), b(Y), c(X,Y).` The CDG G for the body of this clause is shown in figure 3(a). Here `gX` and `iXY` are abbreviations for `ground(X)` and `indep(X,Y)` respectively. Initially $B := true$. The goal is to find the &-Prolog parallel expression D for this body.

- *Steps 1 and 2*: $P = \{a(X)\}$ and $Q = \{gX, iXY\}$.
- *Steps 3 and 4*: $R = \{gX, \neg gX \wedge iXY, \neg gX \wedge \neg iXY\}$. All the elements of R are logically consistent with $B = true$.

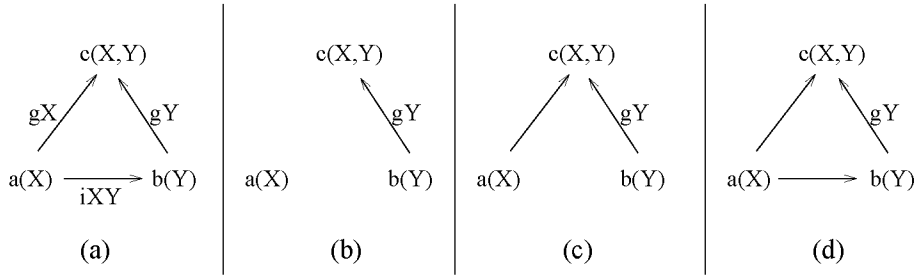

Figure 3: CDGs for the example

- *Step 5*: R is not empty and so

$$D = \mathcal{S}(gX \rightarrow D_1; (\neg gX \wedge iXY) \rightarrow D_2; (\neg gX \wedge \neg iXY) \rightarrow D_3)$$

where, D_1, D_2 and D_3 are respectively the &-Prolog parallel expressions for $(\mathcal{U}(G, gX, \{a(X)\}), gX)$, $(\mathcal{U}(G, \neg gX \wedge iXY, \{a(X)\}), \neg gX \wedge iXY)$ and $(\mathcal{U}(G, \neg gX \wedge \neg iXY, \{a(X)\}), \neg gX \wedge \neg iXY)$. The updated CDGs corresponding to these conditions are in figures 3 (b), (c) and (d) respectively. This simplifies to $D = (gX \rightarrow D_1; (iXY \rightarrow D_2; D_3))$.

Further execution of this algorithm leads us to

```
D  =  (  gX ->  ( gY -> a(X) & b(Y) & c(X,Y)
                ;       a(X) & (b(Y), c(X,Y)))
      ;  iXY -> ( gY -> (a(X), c(X,Y)) & b(Y)
                ;       (a(X) & b(Y)), c(X,Y))
      ;  a(X),  ( gY -> (b(Y) & c(X,Y))
                ;       (b(Y), c(X,Y)))
      ).
```

3.2 UDG Algorithm

We start with the given clause C and construct the simplified CDG for it. This CDG is then converted to a UDG by converting the labels of all its edges to `false`.

The rest of the algorithm is essentially similar to the one in Section 2.2. Let the UDG = (V,E) where V is the set of vertices and E is the set of edges. A brief description of the algorithm is as follows (the variables used here are the same as the ones used in Section 2.2):

1. Construct the sets $P, Q, S = \{S_1, \ldots, S_n\}$ and $\mathcal{F}(S_i), i = 1, \ldots, n$.
2. Recursively compute the &-Prolog expressions for the subgraphs induced by $\mathcal{F}(S_i)$.
3. The linear expression for the given UDG is then built by introducing & operators between all elements of S_i (so that they can be run in parallel) and by placing the subexpressions for $\mathcal{F}(S_i)$ after the subexpressions for S_i and for each $\mathcal{F}(S_j)$ such that $S_j \subset S_i$.

3.3 MEL Algorithm

Let $C = H :- B_1, \ldots, B_n$. Define $\mathcal{I}(B_i)$, which describes the *groundness* and *independence* information at the point to the immediate left of B_i, as follows:

- $ground(X) \in \mathcal{I}(B_i)$ iff X is a program variable in the clause C and X is known to be ground at this point.
- $indep(X, Y) \in \mathcal{I}(B_i)$ iff X and Y are program variables in C and they are known to be independent at this point.

As explained before, $\mathcal{I}$ can be computed from a combination of abstract interpretation, mode information for builtin predicates and user-supplied mode information for non-builtin predicates.

Start with $B := B_1, \ldots, B_n$. B is a sequence of Prolog literals in the body of the clause C.

1. Let $B = B_1, \ldots, B_q$. Find the largest p such that
   - there is a program variable X that occurs in B_p and
   - the first occurrence of X in C is in B_p i.e. X does not occur in H or in $B_i, 1 \leq i < p$ and
   - X occurs in some $B_i, p < i \leq q$

   If there is no such p, then set $p := 0$, $B1 := null$ and $B2 := B$. Else, set $B1 := B_1, \ldots, B_p$ and $B2 := B_{p+1}, \ldots, B_q$.
2. Let G be the set of variables X such that X occurs in some $B_s, B_t, p < s < t \leq q$. Let

   $$I := \{(X, Y) | X \; is \; in \; B_s, Y \; is \; in \; B_t, p < s < t \leq q, X \notin G, Y \notin G\}$$

   i.e. G represents the set of program variables that should be ground and I represents the set of pairs of program variables that should be independent so that the subgoals $B_{p+1}, \ldots B_q$ can be run in parallel.
3. Remove from G all elements X such that $ground(X) \in \mathcal{I}(B_{p+1})$ and from I all elements (X, Y) such that $indep(X, Y) \in \mathcal{I}(B_{p+1})$.
4. Therefore, the &-Prolog parallel expression corresponding to $B2$ is

   $$D2 := (((\bigwedge_{X \in G} ground(X)) \wedge (\bigwedge_{(X,Y) \in I} indep(X, Y))) \Rightarrow B_{p+1} \& \ldots \& B_q)$$

5. If $B1 = null$, then $D := D2$. Else, $D := D1, D2$ where $D1$ is the &-Prolog parallel expression corresponding to $B1$.

As an example, let $C = a(P,Q) :- b(P,Q), c(P,R), d(P), e(Q,R)$. Here R occurs first in the literal $c(P,R)$. So C can be compiled into the following &-Prolog parallel expression: `a(P,Q) :- (ground(P)` $\Rightarrow$ `b(P,Q) & c(P,R)), ((indep(P,Q)` $\wedge$ `indep(P,R))` $\Rightarrow$ `d(P) & e(Q,R))`. Note that, the first CGE does not have the condition $indep(Q, R)$ since this condition is automatically satisfied by virtue of the fact that R is a first occurrence in the literal $c(P, R)$.

Bench	# p goals	# g. cks	# g.c. suc	# i. cks	# i.c. suc
fib	176	0	0	0	0
hanoi	126	0	0	0	0
matrix	146	0	0	8	8
qsort	200	0	0	0	0
consist	792	0	0	0	0
deriv	174	0	0	87	87
tak	1059	0	0	0	0
boyer	502	170	2	2	2
occur	506	252	252	279	279

Table 1: Performance of MEL annotator: number of processes and checks

Bench	# p goals	# g. cks	# g.c. suc	# i. cks	# i.c. suc
fib	176	0	0	0	0
hanoi	126	0	0	0	0
matrix	146	0	0	8	8
qsort	200	0	0	0	0
consist	800	0	0	0	0
deriv	222	0	0	87	87
tak	1059	0	0	0	0
boyer	751	423	6	584	335
occur	506	252	252	279	279

Table 2: Performance of CDG annotator: number of processes and checks

4 Some Performance Results

Although an extensive analysis of the performance of the algorithms presented is a subject for further study, we herein report some preliminary results from the implementation of these algorithms. The three annotation algorithms presented have been implemented and incorporated into the compiler of the &-Prolog system [8]. A compiler switch determines which parallelizer will be used. The results of our implementation of the abstract interpretation-based global analysis presented in [13, 18] are conditionally made available to the annotators via another compiler switch. Programs are analyzed, parallelized, compiled into PWAM code [10, 8] and run on the &-Prolog system. The results are summarized in tables 1, 2, and 3 which show for each benchmark and each parallelization algorithm the number of parallel goals generated at run-time, the total number of groundness checks, the number of groundness checks which succeeded, the number of independence checks, and the number of independence checks which succeeded.

Several conclusions can be arrived at from the data in these tables. Barring *deriv* and *boyer*, MEL and CDG seem to achieve the same parallelism for all other benchmarks. This is because the latter benchmarks mostly have clauses in which only two *consecutive* atoms in the body present opportunities for parallelism. On the other hand, in *deriv* and *boyer*, there are clauses in which atoms which are not adjacent in the clause can legally be parallelized. This type of parallelism can be detected only by graph-based

Bench	# p goals
fib	176
hanoi	126
matrix	130
qsort	200
consist	800
deriv	50
tak	1059
boyer	747
occur	2

Table 3: Performance of UDG annotator: number of processes

analysis, such as done by the CDG and the UDG methods. All in all, CDG appears to be better than MEL. The tradeoff between CDG and UDG is less clear. UDG loses significant parallelism in `matrix`, `boyer`, and `deriv` due to the fact that some checks are still needed in the clauses which are the main source of parallelism. On the other hand, in both `matrix` and `deriv` the checks always succeed. This suggests that more accurate global analysis will perhaps make UDG more attractive, especially considering that the lack of run-time tests ensures that no speedown will occur with respect to sequential execution, a property that does not hold in general for the annotated programs generated by CDG.

An important issue in automatic parallelization, although beyond the scope of this paper, is the inference of task granularity information and the application of such information to optimizing the annotation process. Information regarding the granularity of builtin predicates is used in the implementations of the algorithms presented herein. Inferring granularity information for user predicates is a rather involved but definitely very interesting topic in itself. Approaches to performing such an analysis have been described for example in [17]. These references also provide some results on the performance increase which can be obtained due to granularity analysis.

5 Conclusions

We have presented three algorithms for compilation of logic programs into linear &-Prolog parallel expressions and illustrated them with examples. We have also described an algorithm to determine if a UDG can be compiled into a linear &-Prolog parallel expression without loss of parallelism. Finally, we have reported on preliminary performance results from the implementation of the three algorithms. The results are encouraging, showing that automatic parallelization of logic programs, although it can probably never do as good a job as an experienced user can nevertheless find parallelism in many cases and produce speedups. The results give a certain performance edge to the CDG algorithm over MEL, due to its ability to rearrange the order of goals when they are independent. It is also clear from the simulations that UDG is the safest algorithm, in the sense that it ensures no speedown, although at the sacrifice of maximum potential parallelism.

References

[1] P. Brand, S. Haridi, and D.H.D. Warren. Andorra Prolog—The Language and Application in Distributed Simulation. In *International Conference on Fifth Generation Computer Systems.* Tokyo, November 1988.

[2] M. Bruynooghe. A Framework for the Abstract Interpretation of Logic Programs. Technical Report CW62, Department of Computer Science, Katholieke Universiteit Leuven, October 1987.

[3] 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.

[4] 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.

[5] S. K. Debray and D. S. Warren. Automatic Mode Inference for Prolog Programs. *Journal of Logic Programming*, pages 207–229, September 1988.

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

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

[8] M. Hermenegildo and K. Greene. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.

[9] M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.

[10] M. V. Hermenegildo. *An Abstract Machine Based Execution Model for Computer Architecture Design and Efficient Implementation of Logic Programs in Parallel.* PhD thesis, U. of Texas at Austin, August 1986.

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

[12] 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. University of Washington, MIT Press, August 1988.

[13] K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.

[14] K. Muthukumar and M. Hermenegildo. Efficient Methods for Supporting Side Effects in Independent And-parallelism and Their Backtracking Semantics. Technical Report ACA-ST-031-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, January 1989.

[15] K. Muthukumar and M. Hermenegildo. Methods for Automatic Compile-time Parallelization of Logic Programs using Independent/Restricted And-parallelism. Technical Report ACA-ST-233-89, Microelectronics and Computer Technology Corporation (MCC), Austin, TX 78759, March 1989.

[16] B. Ramkumar and L. V. Kale. Compiled Execution of the Reduce-OR Process Model on Multiprocessors. In *1989 North American Conference on Logic Programming*, pages 313–331. MIT Press, October 1989.

[17] E. Tick. Compile-Time Granularity Analysis of Parallel Logic Programming Languages. In *Int. Conf. on FGCS*. Tokyo, November 1988.

[18] R. Warren, M. Hermenegildo, and S. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*. MIT Press, August 1988.

[19] W. Winsborough and A. Waern. Transparent and- parallelism in the presence of shared free variables. In *Fifth International Conference and Symposium on Logic Programming*, pages 749–764, 1988.