IDIOM:
Integrating Dependent and-, Independent and-, and Or-parallelism

Gopal Gupta, Vítor Santos Costa, Rong Yang
Department of Computer Science
University of Bristol
Bristol BS8 1TR, U.K.
{gupta, costa, rong}@compsci.bristol.ac.uk

Manuel V. Hermenegildo
Facultad de Informática
Universidad Politécnica de Madrid
28660-Boadilla del Monte, Madrid, SPAIN
herme@fi.upm.es

Abstract

Independent and-parallelism, dependent and-parallelism and or-parallelism are the three main forms of implicit parallelism present in logic programs. In this paper we present a model, IDIOM, which exploits all three forms of parallelism in a single framework. IDIOM is based on a combination of the Basic Andorra Model and the Extended And-Or Tree Model. Our model supports both Prolog as well as the flat concurrent logic languages. We discuss the issues that arise in combining the three forms of parallelism, and our solutions to them. We also present an implementation scheme, based on binding arrays, for implementing IDIOM.

1. Introduction

Logic programming languages allow considerable freedom in the way programs are executed. This latitude permits one to exploit parallelism implicitly (without the need for programmer intervention) during program execution. Indeed, three main types of parallelism have been identified and successfully exploited in logic programs:

(i). *Independent and-parallelism*: arises when a problem can be subdivided into a number of sub-problems, which can then be solved independently. It is specially found in programs using the "divide-and-conquer" paradigm of programming. An example of a system implementing this form of parallelism is &-Prolog [13].

(ii). *Or parallelism*: arises when a goal unifies with more than one clause heads, and the resulting resolvents can be pursued in parallel. Aurora [14] and Muse [1] are two examples of systems exploiting this form of parallelism.

(iii). *Dependent and-parallelism*: arises when two (or more) goals share common variable(s) which may be used by these goals for mutual cooperation, and possibly to constrain each other's behaviour. In this case, one needs to define the conditions under which the goals can cooperate: in Committed Choice Languages, like Concurrent Prolog [21], GHC [22] or Parlog [2], these conditions are made explicit by means of *guards*; in the Andorra-I system [18] they are implicit.

A system which exploits only one of the 3 forms of parallelism is clearly sub-optimal—it will speed up only those programs which have that form of parallelism present in them. This contradicts our initial premise regarding implicit exploitation of parallelism since now one has to know the forms of parallelism present in one's program, and then execute it in a system which exploits those particular forms of parallelism. Thus, it is of fundamental importance to design a system which can transparently exploit all three forms of parallelism.

We believe that in integrating the diverse forms of parallelism it is sensible to try to reuse the techniques that have been previously developed for systems exploiting individual forms of parallelism. If such an approach is taken then one can be reasonably confident that the final implementation would be efficient, at least for those programs which exploit only one kind of parallelism. For the programs which exploit more than one kind of parallelisms one can still hope that the integrated system would be efficient given that the properties of logic programs which give rise to the three forms of parallelism are largely orthogonal. This principle has indeed been applied before: in Andorra-I, which combines dependent and-parallelism and or-parallelism, using techniques of Aurora [14] and Parlog [3], and also in the AO-WAM, which combines or-parallelism and independent and-parallelism using techniques of &-Prolog [13] and Aurora. However, an implementation of a combination of all three forms of parallelism has never been tried before, and that is the problem we attempt to tackle in this paper.

Our integrated framework, IDIOM (Integrated Dependent- Independent- and Or-parallel Model), is based on the Basic Andorra Model [25, 11] and the Extended And-Or Tree Model of [8, 6]. From the former we borrow the principle of eager execution of determinate goals and or-parallel execution of non-determinate goals; from the latter we borrow techniques for parallel execution of independent (non-determinate) goals, and ways for combining their solutions which avoid recomputation. Since IDIOM is based on the Basic Andorra Model it supports both Prolog like languages as well as (flat) Committed Choice Languages (such as GHC). In this paper, however, our aim is mainly to support Prolog.

The rest of the paper is organised as follows: In section 2 we briefly describe the two models which are the basic components of IDIOM. In section 3 we present IDIOM and discuss the issues that arise due to the interaction between various forms of parallelism. We also discuss the issues involved in designing environment representation techniques for or-parallelism in the presence of (dependent and independent) and-parallelism. Section 4 presents the complete implementation model for IDIOM for a shared memory multiprocessor, describing how its control, work-scheduling, etc., is organised. Our proposed implementation scheme for IDIOM is essentially a combination of Andorra-I, a system for realising the Basic Andorra Model, and AO-WAM, a system for realising the Extended And-Or Tree Model. Section 5 presents our conclusions and summary of our contributions. We assume the reader is in general familiar with the Binding Arrays method [26, 24] and Conditional Graph Expressions (CGEs) [5, 12], and for the latter part of the paper with implementation techniques employed in [8] and [20].

2. Combining Dependent and-, Independent and- and Or Parallelism

As explained above, we base our model on the Basic Andorra Model [25, 11], which exploits dependent and- and or-parallelism, and the Extended And-Or Tree Model [8] which exploits independent and- and or-parallelism. The resulting model, which we call IDIOM, thus exploits all three forms of parallelisms. Since the systems implementing these two models (Andorra-I for Basic Andorra Model and AO-WAM for Extended And-Or Tree model) use an identical implementation technique (Binding Arrays [26, 24]) for implementing or-parallelism (which is common to both) an implementation for IDIOM can be designed by combining Andorra-I and AO-WAM. In the rest of this section we briefly introduce the two component models and their implementations.

2.1. The Basic Andorra Model

Andorra is the name given to a framework proposed by David H. D. Warren for tackling the classic problem of generating all answers to a problem coded as Horn clauses, with minimum number of inferences while performing as many steps in parallel as possible. Essentially, it allows subgoals to execute ahead of their turn ("turn" in the sense of Prolog's depth first search), and in parallel, subject to certain constraints. The first instance of the Andorra framework is the *Basic Andorra Model*, where goals can be executed ahead of their turn in parallel *if they are determinate*, *i.e.*, if at most one clause matches the goal (the determinate phase). If no determinate goals can be found for execution, a choice point is created for one goal (non-determinate phase) and parallel execution of determinate goals along each alternative of the choice point continues. Or-parallelism is obtained by computing along each alternative of the choice point in parallel, while dependent and-parallelism is obtained by having determinate goals execute in parallel. Executing determinate goals (on which other goals may be dependent) eagerly also provides a coroutining effect, which helps in narrowing the search space of the program.

The Basic Andorra Model has been realised by Yang, Costa and Warren in the Andorra-I system [18]. Andorra-I uses binding arrays for environment representation, and goal stacking for supporting eager execution of determinate goals.

2.2. Combining Independent and-Parallelism and Or Parallelism

A number of models have been proposed for combining or- and (independent) and-parallelism in a single framework [17, 8, 27]. One of the most interesting issues in exploiting independent and- and or-parallelism is that the number of inferences performed at run-time can be reduced compared with standard sequential Prolog computation. Consider the following program:

```
   p(1).        p(2).        p(3).
   q(a).        q(b).
?- p(X), q(Y), write(f(X,Y)).
```

Clearly, `p` and `q` can be executed in independent and-parallel. In the presence of or-parallelism, multiple solutions to individual goals (e.g. `p` and `q` above) can be found in or-parallel. Once solutions for `p` and `q` have been found they can be combined via a *cross-product* and then the solution written out. In this way we can avoid executing `q` completely for every solution produced for `p`, contrary to what would be done, for example, by Prolog. Note that for computing the cross-product of solutions one does not have to wait for all solutions for `p` and `q` to be found. Rather, it can be computed incrementally.

We call the above technique *solution sharing*. It has been used in most and-or parallel models and, particularly, in the Extended And-Or Tree Model [8]. Solution sharing introduces a new kind of or-parallelism, in which continuation of the independent and-parallel goals can be executed in parallel for each tuple of the cross-product set.

In the AO-WAM, tuples belonging to cross-product set are represented *symbolically*, i.e., a solution to a goal is denoted by the address of the terminal stack-frame generated during its execution. The multiple environments are represented by means of a suitably extended binding array. Implementation of solution sharing requires that if a processor selects a (symbolic) tuple for further execution of the continuation of the independent and-parallel goals, it first updates its binding array with conditional bindings (recorded in the trail) created during generation of each of the solutions present in the selected tuple. This operation of updating the binding array from the trail during solution sharing is known as *loading*. Loading the BA with conditional bindings made along component solutions of a tuple may also be necessary during a *task switch* from one node to another, if a tuple is encountered along the path.

3. The Computational Model

IDIOM exploits or-parallelism, independent and-parallelism and (determinate) dependent-and parallelism. Like the Extended And-Or Tree Model, IDIOM uses Conditional Graph Expressions (CGEs) [12] (an improvement of EGEs [5]) to express independent and-parallelism, which can be generated by compile time analysis as in [16]. Determinacy properties of goals are found by a determinacy preprocessor as in [19] and the appropriate determinacy code is tagged on to the program.

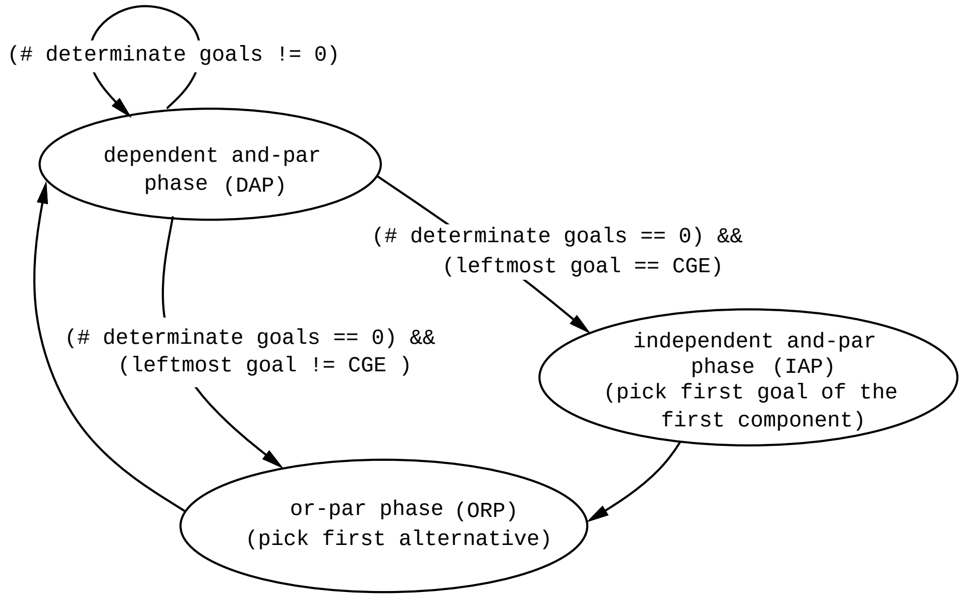

Fig 1: Phases in Parallel Execution

Execution consists of 3 phases: the Dependent And-parallel Phase (DAP), the Independent And-parallel (IAP) phase, and the Or-parallel (ORP) phase. In the DAP phase all goals that can be determinately reduced are evaluated in parallel (including those inside the CGEs) until none are left. The leftmost goal is then examined to see if it is (i) a simple goal; or (ii) a CGE. In case (i) the ORP phase is entered while in case (ii) the IAP phase is entered. In the ORP phase the first alternative to the goal is selected (other alternatives are made available for or-parallel execution), head unification is performed, and the DAP phase is entered again. In the IAP phase, firstly the condition in the CGE is evaluated. If true, the components of the CGE are made available for independent and-parallel processing. In practice, the leftmost component is selected immediately and the other components are made available for parallel execution. The ORP phase will then be entered to process the leftmost goal of the selected component in or-parallel. If the condition in the CGE evaluates to false, the ORP phase is entered immediately to process the leftmost goal in the CGE in or-parallel. When execution of a goal in the ORP or DAP phase leads to success, if this goal was a component of a CGE this is detected and the cross-product is computed incrementally. Execution of the continuation of the CGE then continues in the DAP phase. The control algorithm is abstracted in fig 1.

We will shortly illustrate the IDIOM model through an example (section 3.2), but first let us investigate the interaction of independent and-parallelism (expressed through CGEs) and dependent and-parallelism.

3.1. Conditional Graph Expressions

Conditional Graph Expressions (CGEs) [5, 12] are expressions of the form

$$(\langle condition \rangle \implies goal_1 \ \& \ goal_2 \ \& \ \ldots \ \& \ goal_n)$$

meaning that, if $\langle condition \rangle$ is true, goals $goal_1 \ldots goal_n$ are to be evaluated in parallel, otherwise they are to be evaluated sequentially. The $\langle condition \rangle$ is a conjunction of tests of the form $ground([v_1, \ldots, v_n])$, which checks whether the variables $v_1, \ldots, v_n$ are bound to ground terms, or $indep(v_i, v_j)$, which checks whether the set of variables reachable from v_i and v_j are disjoint. Checking for *groundness* and *independence* involves simple run-time tests, details of which are presented in [5]. In this section we will discuss the interaction of the execution of CGEs with the eager execution of determinate goals.

We assume that compile-time analysis is used to annotate programs with CGEs. Since our actual execution model differs from Prolog, we would prefer compile-time analysis to take this into account in order to exploit maximum independent and-parallelism. However, carrying out such an analysis precisely is quite a complex task in itself. Therefore, we assume that compile-time annotation of CGEs is done with Prolog's operational semantics in mind. This fits well with the fact that we want our user language to be Prolog (so that the users have a simple execution model in mind while writing programs). In addition, this allows us to make use of the compile-time analysis technology that has been developed for Prolog (such as [16]). But it also introduces two new sets of issues arising from:

(i) (eager) evaluation of determinate goals affecting the evaluation of conditions in CGEs.

(ii) (parallel) evaluation of goals in CGEs affecting the (eager) evaluation of determinate goals.

A simple way to tackle the above is to declare the CGEs to be "sensitive" to early bindings, in the sense of Andorra-I [19]. The preprocessor will then insert appropriate control information to guarantee that no goal which can affect these variables is executed early. This is guaranteed to produce the same behaviour (and the same amount of independent and- parallelism) as in Prolog, but at the cost of compromising some of the co-routining. Hence this solution is unattractive. In the next 2 sub-sections we analyze the above issues in more detail and present less restrictive solutions.

3.1.1. Interactions of determinate goals with conditions in CGEs

Due to eager execution of determinate goals certain goals which are independent in Prolog style execution may become dependent in IDIOM, or vice-versa. Hence the condition part of certain CGEs may evaluate to a different value from the one obtained in Prolog execution. As an example, consider the query `?- (indep(X,Y) => p(X) & q(Y)), X = Y.` where `p` and `q` have multiple candidate clauses. If executed left to right, `X` and `Y` would be independent before, and while, `p` and `q` are executed, which allows for their parallel execution. However, if determinate goals are allowed to execute eagerly, the goal `X = Y` would be executed first, while `p` and `q` would be suspended. This would destroy the independence between `p` and `q`.

Rather than placing any restrictions on eager evaluation of determinate goals, we allow unrestricted execution of determinate goals and CGEs. If eager execution of determinate goals renders some independence conditions false, we let this parallelism be lost. In contrast, we can also gain some new parallelism, as the next example shows

```
   g(X,X).        p(X) :- ...         q(X) :- ...
                  p(X) :- ...         q(X) :- ...
?- g(X, Y), ( indep( X, Y ) => p(X) & q(Y) ), Y = c.
```

where `p`, `q` have multiple clauses: In a standard Prolog style execution `indep(X, Y)` would evaluate to false (because `g(X,Y)` aliases `X` to `Y`), while with eager execution of the determinate constraint `Y = c` it would evaluate to true. In fact, if we know at compile-time that some goals are bound to be determinate [4], we can exploit more independent and-parallelism by modifying the compile-time CGE annotator to take the eager execution of these determinate goals into account and thus annotate more goals as CGEs† as suggested in [12].

Note that if independent and-parallelism arises due to groundness conditions or is unconditional, eager determinate execution will have no effect on the parallelism resulting from the corresponding CGE. Experimental data indeed shows [15] that a significant amount of independent and-parallelism indeed arises from groundness conditions, or is unconditional. Considering this fact, and also that we gain new cases of independent and-parallelism due to eager execution of determinate goals, we expect that in practice only very little independent and-parallelism will be lost due to the problems mentioned above.

A problem still remains, however, with unrestricted eager execution of determinate goals and the CGEs. Consider the clause `(true => p(X) & q(Y)), X = Y` (where `true` arises due to a reduction of the condition part of the CGE during compile-time analysis). If `X = Y` is executed first then execution of `p` may influence execution of `q` and vice versa due to the variable aliasing that takes place. We call this the *variable aliasing problem.* We show that in such a situation we can still execute `p` and `q` in independent and-parallel, without performing any redundant work compared to left-to-right execution.

Given a goal `?- p(X), q(X)`, traditionally, one does not execute `p` and `q` in parallel because if `q` starts execution without waiting for bindings created by `p` for `X` then its search space could be inordinately large. To avoid this redundant computation one has to follow the sequential order. Now, going back to the goal `(true => p(X) & q(Y)), X = Y`, if we execute `X = Y` last, `p` and `q` would be independent and, therefore, their parallel execution would not lead to any redundant work being performed. However, if we allow `X = Y` to execute first in accordance with the Andorra principle, our goal would become similar to the earlier goal `?- p(X), q(X)`

† An interesting point is that eager execution of a determinate goal may cancel out the effect of another goal on the result of independence condition evaluation. In the query above if `g(X,Y)` were determinate and occurred to the right of the CGE, its evaluation would cause `indep(X,Y)` to evaluate to false, but executing `Y = c` would restore the independence.

and it would appear that the amount of work done in executing `q` would depend on the order of execution between `p` and `q`. This is true, but still if one lets `p` and `q` execute independently in parallel without any regard for this aliasing, one would still do the same amount of work as in the case where `X = Y` was executed after `p` and `q`, i.e. as in left-to-right execution. In other words, the amount of work *cannot* increase by executing `X = Y` first, compared to left to right execution or pure independent and-parallel execution such as in &-Prolog [13]. However, the implementation would still need to unify the two different values generated for `X` (or `Y`, since the two variables are aliased) by `p` and `q` respectively; this corresponds to the unification that would eventually be performed in executing `X = Y` in left-to-right execution. In our model, the different values for `X` produced by `p` and `q` are unified when the cross-product of their solutions is computed (see section 4.4), following the method in [8].

3.1.2. Interactions of Goals in CGEs with Determinate Goals

Independent execution of two (or more) parallel goals can render goals external to the CGE determinate. This in turn can affect the independence of the components executing in parallel. Consider the following program

```
p(1).              q(3).              r(1, 2) :- l, m, n.
p(2).              q(4).              r(2, 3) :- a, b.
                                      r(3, 4) :- e, f.
```

and the query:

```
?- (true => p(X) & q(Y) ), r(X, Y).
```

During independent and-parallel execution of `p`, its second alternative would render `r` determinate. Executing `r` eagerly in accordance to the Andorra principle would bind `Y` to `3`. This would render `q`, which has perhaps already started execution in parallel with `p`, determinate. Thus, `p` and `q`, which were independent at the beginning of the execution of the CGE, end up influencing each others execution. In this situation, maintaining the consistency of bindings and goal chain becomes extremely complicated and overhead prone.

Consider the same program again, and another scenario: During independent and-parallel execution of `p`, its first alternative renders `r` determinate. Simultaneously, the execution of the first alternative of `q` also renders `r` determinate. However, in both cases the matching clause for `r` is different, and, although the combination of the first alternative of `p` and `q` should lead to failure, this would not be detected in the example above until after the first alternatives of `p` and `q` have completely finished and execution of continuation of the CGE has started. Since the failure is not detected early, any work done in eagerly reducing `r` and then `l, m, n, a, b` etc. would be wasted. This problem is a special case of the problem mentioned in the previous paragraph, in which determinate execution of `r` binds `Y` to a binding inconsistent to that generated by `q`.

A solution to the the above problems of wasted work and loss of independence is to delay *at runtime* the execution of goals which are outside the scope of the CGE and which are made determinate during execution of that CGE until after the alternatives to the independent and-parallel goals which made these goals determinate have been completely executed. Our model indeed adopts this solution. The goals to be delayed in such a manner are recorded in a data structure (in the case of our implementation they are recorded in the *goal trail*, see section 4.4). Before execution of the continuation of the CGE is begun, these recorded goals are placed in the determinate-goal run-queue for determinate execution. However, any goal which is rendered determinate but which lies within the CGE is allowed to execute eagerly in normal fashion. Likewise, those goals which became determinate from other sources are also free to execute eagerly in the normal fashion. We lose some dependent and-parallelism (and some coroutining) due to this slight restriction on the execution of determinate goals to the right of CGE, but any other means of ensuring correct execution would involve too much run-time overhead.

3.2. Example

To illustrate execution in IDIOM we take the following program for finding "cousins at the same generation who have the same genetic attributes" a modification of a program taken from [23]. The assertion `parent(X, Y)` means `Y` is the parent of `X`, `attributes(X, Px)` means `Px` is a list of attributes of `X`, and `set_xion(S1, S2, S3)` represents that set `S3` is the result of intersection of sets `S1` and `S2`.

```
sg(X, X, Pi, Pf ) :- attributes( X, Px ), set_xion( Px, Pi, Pf ).
sg(X, Y, Pi, Pf ) :- X ≠ Y,
         ( indep(X, Y) => (parent(X, Xp), attributes(Xp, Pxp)) &
                          (parent(Y, Yp), attributes(Yp, Pyp)) ),
         set_xion(Pi, Pxp, Pxi), set_xion(Pxi, Pyp, Pii),
         sg(Xp, Yp, Pii, Pf).

set_xion( S1, S2, S3 ) :- set_xion( S1, S2, [ ], S3 ).
set_xion( [ ], _, In, In ).
set_xion( [X|T], S, In, Out ) :- in( X, S, In, IR ),
                                 set_xion( T, S, IR, Out ).
in(_, [ ], In, In).
in(X, [X|_], In, [X|In]).
in(X, [Y|T], In, Out) :- X ≠ Y, in( X, T, In, Out).

parent(fred, frank).
...
...

attributes(fred, [brown-hair, green-eyes, large-build]).
...
...
```

Consider the query:

```
?- sg(fred, john, [brown-hair, green-eyes], Att).
```

where the third argument is a list of attributes common to `fred` and `john`, and we want to find out if they are cousins of the same generation and if so, their common genetic attributes inherited from their common ancestor. The CGE annotator would only annotate the second clause for `sg` which gets annotated as shown in the program.

The goal chain initially consists of the query goal. We check if it is determinate, which indeed it is, since only the second clause matches†. Head unification is then

† A smart preprocessor, such as [19], can look ahead into the body of the clauses for `sg` and from the condition $X \neq Y$ conclude that it is determinate.

performed and the body of the second clause is inserted in the goal-chain. Only the `set_xion(Pi,Pxp, Pxi)` subgoal is determinate (since `Pi` is known) so it is reduced next. The call to `in` from within `set_xion` soon suspends; however, the recursive call to `set_xion` can still continue determinate execution as long as there are list elements available in `Pi`. This will result in a number of calls to `in` all of which would be suspended on `Pxp`. Eventually, no determinate goals are left, and the DAP phase is exited. Since the leftmost subgoal is a CGE, the IAP phase is entered. The independence condition evaluates to true, and parallel execution of the two components is started. The parent goals are executed in or-parallel, and for each alternative of `parent` the corresponding `attributes` goal is executed determinately. As soon as an attributes goal is executed, and a binding for `Pxp` is generated, all the suspended `in` goals are rendered determinate and thus awakened; but due to the *determinate goal* problem mentioned in section 3.1.2 their execution is delayed until the execution of the CGE is over. Rather, the cross-product is computed and a tuple is selected for executing the continuation of the CGE. The determinate execution of the delayed `in` goals can then be started. (The two alternatives each, of the two `parent` goals, give rise to four cross-product tuples, which are pursued in (or-) parallel. The execution of the continuation of the CGE is the same for all tuples.) As soon as the execution of `in` instantiates `Pxi` and causes the first list element to appear in it, the second call to `set_xion` becomes determinate and begins execution. As more and more list elements of `Pxi` are generated by the multiple `in` goals, they are determinately consumed by the second `set_xion`. In the meantime the recursive call to `sg` becomes determinate (since `Xp` and `Yp` are now known) and starts executing. As soon as an element of `Pii` is available, the first `set_xion` call inside recursive call to `sg` becomes determinate and can begin execution. Thus, it's as if a data pipeline has been set up between calls to `set_xion`, calls to `in` and the recursive call to `sg`, which indeed gives rise to dependent and-parallelism. Or-parallelism is exploited in the execution of the parent goals, and independent and-parallelism in the execution of the CGE. Thus, all three forms of parallelism are exploited in the IDIOM-based execution of this program.

4. Implementation of IDIOM

In this section we provide a brief outline of an IDIOM implementation. The IDIOM execution model can be viewed as a combination of AO-WAM and Andorra-I. At the level of or- and independent and-parallelism, execution in IDIOM is similar to AO-WAM, while at the level of dependent and-parallelism execution mimics Andorra-I.

4.1 Environment Representation and Variable Access

A major problem in any logic programming system that incorporates or-parallelism is the management of the multiple environments that may exist simultaneously. A number of techniques have been proposed for handling this problem. A systematic description and analysis of these techniques can be found in [9]. Incorporation of other forms of parallelism such as dependent and independent and-parallelism into the system make this task ever more complicated. In [9] it is shown that BA method [26, 24] is ideally suited for or-parallel implementations on shared memory multiprocessor systems because the two most frequent operations in logic programming systems—task creation and variable access—are performed in it in constant-time. Task switching is a non constant-time operation but it can be optimised by a careful choice of work scheduling algorithm. We use the BA technique in our implementation, suitably modifying it to incorporate independent and dependent and-parallelism.

4.1.1. Extending Binding Arrays for Independent And-parallelism

In the presence of independent and-parallelism the binding-arrays method for the pure or-parallel case must be extended to achieve constant-time access to variables. To see the problem, consider the goals p, (true => q1 & q2), r, where q1 and q2 also exhibit or-parallelism. Suppose further that goal p has been completed. In order to execute goals q1 and q2 in and-parallel, it is necessary to maintain separate binding arrays for them. As a result, the binding-array offsets for any conditionally bound variables that come into existence within these two goals will overlap. Thus, when r is attempted, we are faced with the problem of merging the binding-arrays for q1 and q2 into one composite binding-array or maintaining fragmented binding-arrays.

To solve the above problem, first recall that in the binding-array method [26, 24] an offset-counter is maintained for each branch of the or-parallel tree for assigning offsets to conditional variables. However, offsets to the conditional variables in the and-parallel branches cannot be uniquely assigned, since there is no implicit ordering among them; at run-time a processor can traverse them in any order.

To incorporate independent and-parallelism, we introduce one more level of indirection in the binding arrays. Each processor, in addition to maintaining the binding array, also maintains another array called the *base array.* Each component of a CGE, when it is encounted during execution, is assigned a unique integer id called IAP-id. When a processor encounters a component of the CGE, it stores the offset of the next free location of the binding array in the i-th location of its base array, where i is the IAP-id of the component. The offset-counter is reset to zero. Subsequent conditional variables are bound to the pair $\langle i, v\rangle$, where v is the value of the counter. The counter is incremented every time a conditional variable is bound to this pair. The binding of a conditional variable is dereferenced by double indirection through the base and binding array using this pair [8].

Note that access to variables is constant-time, though the constant is slightly larger compared to the binding-arrays method for pure or-parallelism due to the double indirection. Also note that now the base array must be updated (in addition to the binding array) on a task-switch.

4.1.2. Extending Binding Arrays for Dependent And-parallelism

In the binding arrays method the *value cell* and the *access cell* are not identical [9]. A value cell is the location where the binding of a variable is stored when created (the trail in case of BA method) while the access-cell is the location where the binding of a variable is to be looked up when it is needed during unification (the corresponding cell in the binding array in the BA method)†. If we wish to incorporate dependent and-parallelism then we have to make sure that all processors participating in a dependent and- parallel computation have a *common* access-cell for a given variable so that a binding made by one processor is immediately visible to the other. A very simple way to ensure this, is to have processors work as a

† This is the reason why task-switch time is non constant-time in the BA method since during task-switching the access-cells (in binding arrays) have to be updated with correct bindings from the value cells (stored along the trail).

team, as done in the Andorra-I system [20], where the whole team shares a common binding array during execution. We indeed adopt this concept in IDIOM.

Each IDIOM processor has a binding and base array associated with it, and can be in one of two modes: *master* or *slave*. As a slave, a processor is disallowed to select independent and-or or-parallel work; its task is to help a processor in the master mode to solve determinate goals. However, a slave processor is allowed to dynamically change its state to master mode. A processor in the master mode can have more than one slave processors attached to it, sharing its data-structures, and giving rise to a team. Sharing of binding and base arrays ensures that all processors in a team, participating in the dependent and-parallel work, have a common access-cell for a given variable. Note that or-parallelism is exploited when two master processors select different alternatives from a choice point, while independent and-parallelism is exploited when two master processors select different components of a CGE. Thus, or-parallelism and independent and-parallelism are exploited by *teams* of processors, similar to the way or-parallelism is exploited in Andorra-I.

Note that as a result of sharing, the BA becomes a shared structure and has to be locked when accessed for writing (it need not be locked for reading, since Prolog variables are write-once only). Every time a processor in the team needs to allocate space for a conditional variable in the BA it atomically reads the offset counter and increments it. To avoid excessive locking, the processors can allocate space in the BA in chunks at a time and assign the offsets locally until the chunk is exhausted, as done in [20]. The dereferencing algorithm remains as described in 4.1.1.

4.2. Data Areas and Memory Management

The main data-structures that a IDIOM processor uses are essentially similar to those found in efficient Prolog systems—a local stack, a heap and a trail. In addition, it also uses variable access arrays (binding and base arrays), and work queues for scheduling the different kinds of parallel work available. The local stack is split into a node-stack and a choice-point stack to facilitate flexible task scheduling. Space is recovered from these stacks on backtracking as well as when a determinate goal is successfully reduced. In order to support eager evaluation of determinate goals we propose to use *goal stacking* (similar to Andorra-I) rather than WAM style *environment stacking*. Parallel work created during IDIOM execution is of four kinds: (i) or-parallel work from alternatives of a choice point, (ii) independent and-parallel work arising from CGEs, (iii) dependent and-parallel work arising from determinate goals and (iv) or-parallel work arising from cross-product tuples. There is a queue for each kind of work: we call the 4 queues ORP queue, IAP queue, DAP queue and CP queue, respectively.

Note that only the variable access arrays and the DAP queue, are private to a team, the other data-structures being visible to processors in other teams. Processors in slave mode share the heap, node-stack, choice-point stack, trail, variable access arrays, and work queues of their current master. As mentioned before for the binding arrays, sharing of stacks between members of the same team we use the "chunk" scheme of Andorra-I, in which stacks are divided into chunks which allow each worker to allocate entries independently.

4.3. Goal Chain Management

We propose to use goal stacking to keep track of continuation of the current goal and of the goals remaining to be executed during execution. Hence, we adopt most of the techniques that have been developed for the Andorra-I system, making suitable modifications to support CGEs. During eager determinate execution the goal chain is eagerly explored by slave processors and determinate goals are executed and deleted from the goal chain. To avoid copying the entire goal chain for each alternative during or-parallel forking the links connecting the goals in the goal chain are replaced by variables [20]. The actual link information is stored as bindings of these variables. The advantage is that these bindings can be made conditionally, so that each or- alternative shares a common set of goals but sees a different order between them depending upon its context. As with all conditional bindings, these conditional link assignments are trailed so that during task-switching the correct goal chain can be restored when moving from one node to another.

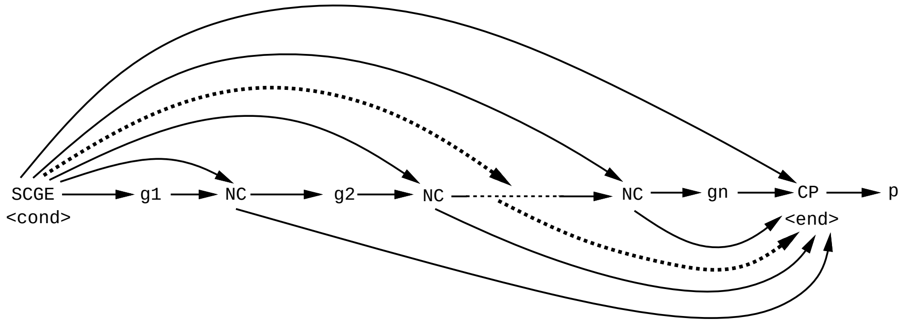

Fig 2: Representing CGE in Goal Chain

The goal chain can support CGEs quite simply. A goal, containing a CGE, of the form `(⟨cond⟩ => g1 & g2 & ...& gn), p` is organised as in fig. 2. The start of a CGE is indicated by SCGE which essentially serves the same role as the "Parcall Frame" of the PWAM [13]. The various components of the CGE are separated by the identifier NC (= Next Component). The end of the CGE is indicated by CP. The beginning of each component is accessible from SCGE, and the end of the CGE (CP) is accessible from the end of each component. These pointers are needed to facilitate independent and-parallel execution of the CGE and computation of the cross-product of solutions of the components of the CGE. When a master processor P1 (of team T1) encounters a CGE it evaluates its condition part (stored with SCGE). If the condition evaluates to false, then NC, CP, etc. are treated as null goals. If the condition evaluates to true, P1 follows the pointers to the various components and puts them in the IAP run-queue.

4.4. Parallel Execution and Task Scheduling

The execution of CGEs in IDIOM is based on the execution of CGEs in the AO-WAM [8]. We therefore first briefly describe how CGEs are executed in the AO-WAM. Essentially, the various components of a CGE are evaluated in parallel. Each component can have more than one solution, which are found in or-parallel. A solution is (symbolically) represented by the address of the terminal stack-frame of that component. Thus, when a processor produces a solution for a component, it first checks if at least one solution has been found for all the other components (a list of solutions is maintained for each component to facilitate this task). If not, it starts working on the component which is still untried. If there are no untried components, but a solution has not been found for some of them (i.e., other processors are working on finding a solution for these components), then it simply backtracks and tries to find another solution for the component it is working on. If at least one solution has been found for all the components, the processor computes a cross-product of the solution sets for other components with the current solution it has just found. It then picks up one of the cross-product tuples, recording the rest in the CP queue for other processors, loads the conditional bindings made in other components in its binding array (in the process recording their IAP-ids in its base array) and continues with the goal after the CGE. Likewise, when a processor steals a cross-product tuple from the CP queue of another processor, it loads binding and base arrays, and continues with the execution of the goal after the CGE. The loading operation can be made efficient by using techniques described in [10].

To adopt the execution of CGEs in IDIOM as described above we have to tackle the problems arising due to dependent and- parallelism described in section 3.1. These problems are solved through the following modifications: (i) To incorporate dependent and-parallelism a component of a CGE is processed by a team, rather than a single processor. This fact can be advantageously used for parallelising the loading operation—the slave processors can be used for loading the conditional bindings from different components of the CGE in parallel. (ii) To solve the variable aliasing problem of section 3.1.1. the operation of loading a BA (performed for solution sharing) is modified. In the AO-WAM the binding installation operation during loading is a simple assignment of the trailed value to the appropriate cell of the BA. In IDIOM the BA cell may already contain a binding (due to aliasing between two independent variables). In such a case, the resident binding and the current binding have to be unified†. If the unification fails, then the team abandons the tuple being loaded, and picks another one. Note, however, that the loading operation performed during task-switching does not need to be changed. (iii) To solve the determinate goal problem described in section 3.1.2 we place pointers to goals to the right of CGE, which are rendered determinate due to execution within the CGE, in a special trail called the *goal trail*. The fact that the goal is to the right of the CGE can be easily determined from the goal chain by comparing IAP-ids. During the loading operation the goal-trail is also traversed and goals found in it are awakened and put in the DAP run-queue.

The execution of dependent and-parallel goals is very similar to Andorra-I. In the DAP phase processors in slave mode keep polling their master's DAP queue. If they find a goal, they load their control and data registers and execute that goal. In the process of execution they may generate some more determinate goals which will be placed in the DAP queue. If the slave processors have to stay idle for some time in the DAP phase, then they abandon their current master and find a new one which has maximum value for the metric $\frac{length(DAP\ \ queue)}{N}$, where N is the number of processors in the team. Note that no significant overhead is involved in changing masters except for the work done in searching for one which has maximum value for the metric. Outside of the DAP phase, the slave processors can also change their mode and help in performing independent and- and or-parallel work. However, if they do not decide to change their mode to master, they help their current master during backtracking and loading operations. During these operations the team has to backtrack/load as a whole, thus the slaves cannot be doing any other work, and hence the master can assign branches in the search tree to the slaves to load or backtrack over.

5. Conclusions

In this paper we presented an integrated model called IDIOM which exploits the three main forms of parallelism—independent and-, (determinate) dependent and-,

† In the context of parallel loading, just mentioned, this also means that processors have to lock the BA cell during binding installation.

and or-parallelism—without any aid from the user. While exploiting dependent and-parallelism we also obtain a reduction in search space because of the attendant co-routining. Exploiting independent and-parallelism leads to a reduction in the number of inferences performed due to solution sharing. We discussed issues that arise from the interaction of different kinds of parallelism, in particular the interaction of CGEs (used for annotating independent and-parallel goals at compile-time) with eager execution of determinate goals, and presented our solutions to them. We also presented a complete implementation scheme for IDIOM, along with main data-structures and control algorithms. We presented an or-parallel environment representation scheme—a modification of the Binding Arrays method—which can accommodate both independent and-parallelism and dependent-and parallelism. We believe that this is the first implementation strategy that attempts to exploit all three main forms of parallelism in a single framework. We also believe that our implementation scheme, which is a combination of the Andorra-I and AO-WAM systems, can be implemented quite efficiently. Other and-or parallel systems such as ACE [7] can also be extended in the manner described in this paper to incorporate dependent and-parallelism.

Acknowledgement

We are grateful to David H.D. Warren, Inês Dutra, Tony Beaumont, Raéd Sindaha and the anonymous referees for their many insightful comments. The research is supported by Esprit project PEPMA and U.K. Science and Engineering Research Council grant GR/F 27420. Vítor Santos Costa would like to thank the Gulbenkian Foundation for supporting his stay at Bristol. Manuel Hermenegildo is also supported in part by CICYT project 361.208.

References

[1] K. Ali, R. Karlsson, "The Muse Or-parallel Prolog Model and its performance," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp 757-776.

[2] K. Clark, S. Gregory, "Parlog: Parallel Programming in Logic," In *TOPLAS*, January 1986. pp 1-49.

[3] J. Crammond, "Scheduling and Variable Assignment in the Parallel Parlog Implementation," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp 642-657.

[4] S. K. Debray, D. S. Warren, "Detection and Optimization of Functional Computations in Prolog," In *Third ICLP*, Lecture Notes in Computer Science, No. 225, Springer-Verlag, pp. 490-505.

[5] D. DeGroot, "Restricted AND-parallelism", *Int'l Conf. on Fifth Generation Computer Systems*, Nov., 1984.

[6] G. Gupta, "And-Or Parallel Execution of Logic Programs: Abstract Machine Design and Implementation," Ph.D. Thesis, UNC Chapel Hill, In preparation.

[7] G. Gupta, M. Hermenegildo, "ACE: And/Or-Parallel Copying-based Implementation of Logic Programs," In *Proc. of ICLP '91 Workshop on Parallel Logic Programming*, June '91.

[8] G. Gupta, B. Jayaraman, "Compiled And-Or Parallel Execution of Logic Programs," In *Proceedings of the North American Conference on Logic Programming '89*, MIT Press, pp. 332-349.

[9] G. Gupta, B. Jayaraman, "On Criteria for Or-Parallel Execution Models of of Logic Programs," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp. 604-623.

[10] G. Gupta, B. Jayaraman, "Optimizing And-Or Parallel Implementations," In *Proceedings of the North American Conference on Logic Programming '90*, MIT Press, pp. 737-756.

[11] S Haridi, "A Logic Programming Language Based on the Andorra Model," In *New Generation Computing*, Vol. 7, No. 2-3, pp. 109-125.

[12] M. V. Hermenegildo, "An Abstract Machine for Restricted And Parallel Execution of Logic Programs", *3rd International Conference on Logic Programming*, London, 1986, pp. 25-39.

[13] M. V. Hermenegildo, K.J. Greene, "&-Prolog and its performance: Exploiting Independent And-Parallelism," In *Proceedings of ICLP '90*, pp. 253-268.

[14] E. Lusk, D. H. D. Warren, S. Haridi, et. al., "The Aurora Or-parallel Prolog System," New Generation Computing, **7**(2,3), 1990, pp. 243-271.

[15] K. Muthukumar, M. Hermenegildo, "The CDG, UDG and MEL methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-Parallelism," In *Proceedings of ICLP '89*, pp. 221-237.

[16] K. Muthukumar, M. Hermenegildo, "Variable Sharing Information through Abstract Interpretation," In *the Journal of Logic Programming*, to appear.

[17] B. Ramkumar, L. V. Kalé, "Compiled Execution of the REDUCE-OR Process Model," In *Proceedings of the North American Conference on Logic Programming '89*, MIT Press, pp. 313-331.

[18] V. Santos Costa, D. H. D. Warren, R. Yang, "Andorra-I: A Parallel Prolog system that transparently exploits both And- and Or-Parallelism," In *Proceedings of Principles & Practice of Parallel Programming*, Apr. '91, pp. 83-93.

[19] V. Santos Costa, D. H. D. Warren, R. Yang, "The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model," In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, 1991, pp. 443-456.

[20] V. Santos Costa, D. H. D. Warren, R. Yang, "The Andorra-I Engine: A parallel implementation of the Basic Andorra model," In *Logic Programming: Proceedings of the 8th International Conference*, MIT Press, pp. 825-839.

[21] E. Shapiro, "Concurrent Prolog : Collected Papers," MIT Press, 1987.

[22] K. Ueda, "Guarded Horn Clauses," Ph.D. Thesis, Tokyo University, Mar. '86.

[23] J. D. Ullman, Principles of Database and Knowledge-base Systems, Computer Science Press, 1989. Volume II.

[24] D. H. D. Warren, "The SRI-model for Or-Parallel execution of Prolog – Abstract Design and Implementation Issues," *1987 IEEE Int. Symp. in Logic Prog.*, San Francisco, pp. 92-102.

[25] David H. D. Warren, "The Andorra model", Presented at Gigalips Project workshop, University of Manchester, March, 1988.

[26] D. S. Warren, "Efficient Prolog Memory Management for Flexible Control Strategies," In *The 1984 Int. Symp. on Logic Prog.*, pp. 198-202.

[27] H. Westphal, P. Robert, J. Chassin, J. Syre, "The PEPSys Model: Combining Backtracking, AND- and OR-parallelism," In *1987 IEEE Int'l Symp. in Logic Prog.*, San Francisco, pp. 436-448.