PROCEEDINGS OF THE INTERNATIONAL CONFERENCE
ON FIFTH GENERATION COMPUTER SYSTEMS 1992,
edited by ICOT.

Parallel Constraint Solving in Andorra-I

Steve Gregory and Rong Yang

Department of Computer Science
University of Bristol
Bristol BS8 1TR, England

steve/rong@cs.bris.ac.uk

Abstract

The subject of this paper is the integration of two active areas of research: a parallel implementation of a constraint logic programming language. Specifically, we report on some experiments with the and/or-parallel logic programming system Andorra-I extended with support for finite domain constraint solving.

We describe how the language supported by Andorra-I can be extended with finite domain constraints, and show that the computational model underlying Andorra-I is well suited to execute such programs. For example, most constraints are automatically executed eagerly so as to reduce the search space; moreover, they are executed concurrently, using dependent and-parallelism.

We have compared the performance of some constrained search programs on Andorra-I with that of conventional generate-and-test programs. The results show that the use of constraints not only reduces the sequential execution time, but also significantly increases the and-parallel speedup.

1 Introduction

Much of the success of Prolog has been due to its suitability for applications involving search: the language provides a relational notation which is very convenient for expressing non-deterministic problems and it can be implemented with impressive efficiency. However, the search strategy built into Prolog is a rather naive one, which tends to perform an unnecessary amount of search for problems that are stated in a simple manner. To solve realistic search problems in Prolog, it is often necessary to perform additional forward computation in order to reduce the search space to a manageable size. However, since this extra computation must be programmed in Prolog itself, it may be an expensive overhead which partly offsets the speed benefits of the reduced search. Moreover, the resulting program is more opaque and difficult to write than a natural solution in Prolog.

To improve on the search strategy of Prolog while retaining its advantages is the motivation for the development of constraint logic programming (CLP) systems. Most of the CLP languages that have been proposed are based on Prolog, extended with the ability to solve constraints in one or more domains. CLP languages use knowledge specific to their domain to execute certain goals ("constraints") earlier than would be possible in Prolog, thus potentially reducing the search space. Provided that the constraint solving mechanism is implemented efficiently and that the language is simple to use, the search time can be reduced at little cost in either forward computation time or increased program complexity. One type of CLP language, which has proved particularly useful for combinatorial search problems, is that based on finite domains; this is described in a little more detail in Section 2.

There have been many projects in recent years to develop parallel implementations of Prolog. Most of these systems incorporate either or-parallelism, independent and-parallelism, or both. In contrast, the Andorra-I system is an implementation of Prolog that exploits or-parallelism together with dependent and-parallelism, which is the sole form of parallelism exploited in most implementations of concurrent logic programming languages such as Parlog and GHC. Andorra-I has proved effective in obtaining speedups in programs that have potential or-parallelism and those with potential and-parallelism, while in some programs both forms of parallelism can be exploited. Andorra-I, and the Basic Andorra model on which it is based, are described briefly in Section 3.

The subject of this paper is the integration of the above strands of research: a parallel implementation of a constraint logic programming language. Specifically, we report on our experiences with extending the Prolog-like language supported by Andorra-I to support finite domain constraint solving. There are two main reasons why this is of interest:

1. **Language.** To investigate how easily the required language extensions can be supported by the Basic Andorra model.

2. **Performance**. To ensure that the finite domain extensions can be implemented efficiently in Andorra-I and that the efficiency is retained in parallel execution.

Although a prototype or-parallel implementation of the Chip language has been developed [Van Hentenryck 1989b], we are not aware of any previous investigation of and-parallelism with finite domain constraints. By adding these extensions to Andorra-I we can experiment with both forms of parallelism and compare them.

It is particularly interesting to compare the performance of constrained search programs on the Basic Andorra model with that of conventional generate-and-test programs (apart from the expected reduction in overall execution time). The constraint solving represents additional forward computation, so — provided that the constraints can be effectively solved in parallel — we would expect and-parallelism to be increased. At the same time, since the search space is reduced, there may be less scope for or-parallelism. The performance results obtained with Andorra-I confirm these expectations.

The next two sections describe the background to the paper. Section 4 discusses the implementation of finite domain constraints on the Basic Andorra model. It describes in detail the language extensions that we have implemented and the structure of programs that use them. Section 5 presents some results of running constrained search problems on Andorra-I. Section 6 concludes the paper.

2 Finite domain constraints

The idea of adding finite domain constraints to logic programming originated with the work of Van Hentenryck and his colleagues, and was first implemented in the language Chip [Van Hentenryck and Dincbas 1986; Dincbas *et al.* 1988; Van Hentenryck 1989a]. Chip extends Prolog in several ways to handle constraints; the principal extensions relevant to finite domains are outlined below.

2.1 Domain variables

Some variables in a program may be designated *domain variables*, ranging over any specified finite domain. Domain variables appear to the programmer like normal logical variables but are treated differently by unification and by constraints.

2.2 Constraints on finite domains

Goals for certain *constraint* relations behave in a special way when they have domain variables as arguments. For example, if `X` is a domain variable, the goal `X ≤ 5` can be executed by removing from the domain of `X` all items greater than 5. This in turn may reduce the search space that the program explores. A user-defined predicate may be made a constraint by using a '`forward`' or '`lookahead`' declaration, while some primitives (e.g., inequality) have such declarations implicitly. (Unification can have a similar effect: unifying two domain variables reduces the domain of both to the intersection of their original domains, while unifying a domain variable and a constant may fail.)

2.3 Coroutining

Constraints should be executed as early as possible in order to reduce the search space. For example, `X ≤ Y` could be executed as soon as either `X` or `Y` has a value and the other is a domain variable. In general, a coroutining mechanism ensures that control switches to a constraint goal as soon as it can be executed. The simplest such control rule is *forward checking*, used for `forward`-declared constraints, whereby a constraint is executed as soon as its arguments contain at most one domain variable and are otherwise ground. The constraint goal is then effectively executed for each member of its argument's domain and values that cause failure are removed from the domain.

The *lookahead* rule, often used for inequality relations such as '≤', can even execute constraints whose arguments contain more than one domain variable; we shall not consider this further in this paper.

3 The Basic Andorra model

The Basic Andorra model is a computational model for logic programs which exploits both or-parallelism and dependent (stream) and-parallelism. The model works by alternating between two phases:

1. **Determinate phase**. Determinate goals are executed in preference to non-determinate goals. While determinate goals exist they are executed in parallel, giving dependent and-parallelism. (A goal is considered *determinate* if the system can detect that it can match at most one clause.) This phase ends when no determinate goals are available or when some goal fails.

2. **Non-determinate phase**. When no determinate goals remain, one goal — namely, the leftmost one that is not `det_only` (see below) — is selected and a choicepoint created for it. Or-parallelism can be obtained by exploring choicepoints in parallel.

The model and its prototype implementation, Andorra-I, are described in [Santos Costa *et al.* 1991].

Andorra-I supports the Prolog language augmented with a few features specific to the model. For example, `det_only` declarations allow the programmer to specify that goals for some predicate

can only be executed in the determinate phase; if such a goal remains in the non-determinate phase it cannot be used to create a choicepoint, even if it is the leftmost goal. Conversely, `non_det_only` declarations can be used to prevent goals from executing in the determinate phase even if they are determinate.

Performance results for Andorra-I show that the system obtains good speedups from both or-parallelism and and-parallelism. The best and-parallel speedups are obtained for programs that are completely determinate (and therefore have no or-parallelism to exploit). The best or-parallel speedups come from search programs, especially when searching for all solutions.

Unfortunately, very little and-parallel speedup has typically been observed in running standard Prolog search programs on Andorra-I. One reason for this is the sequential bottleneck inherent in the Basic Andorra model: the periods (both during the non-determinate phases and while backtracking) when no and-parallel execution is performed.

This suggests that the key to obtaining greater and-parallel speedup is to increase the "granularity" of the and-parallelism. That is, it is important to minimize the number of choicepoints created and the number of goal failures, relative to the total number of inferences. One way to achieve this in search programs is by the use of constraint satisfaction techniques.

4 Implementing finite domains in Andorra-I

In order to experiment with finite domain constraint solving on Andorra-I, we have defined and implemented finite domains and a few simple primitives to operate on them. Our system defines a new data type, a *domain*, which exists alongside numbers, structures, etc. Domains can only be used as arguments to the domain primitives and have no meaning elsewhere in a program; for example, they cannot be printed. A domain is created with a set of possible values that it may take; eventually it may become instantiated to one of those values, at which time we call it an *instantiated domain*. In contrast with the Chip concept of domain *variables*, a domain instantiated to t is *not* identical to t. We write a domain as a set $\{t_1,\ldots,t_n\}$, where $t_1,\ldots,t_n$ are its current possible values; $\{t\}$ represents an instantiated domain.

Our domains are easier to implement than domain variables because there is no need to change many basic operations of the system such as unification, suspension on variables, etc. At the same time, the efficiency of implementation should be comparable with that of domain variables, while our primitives are still quite convenient to use.

We describe our primitives first and then outline their use and implementation.

4.1 Finite domain primitives

Domains can be created by the primitives `make_domain` and `make_domains`. The latter is potentially more efficient when creating many domains ranging over the same values since the table of values can be shared.

All of the other primitives operate on existing domains; they can only be executed when their first argument is instantiated and will fail if this is not a domain. `domain_var` performs the mapping between a domain and its ultimate value, while `domain_remove` allows the removal of values from a domain. Either of these may cause the domain to be instantiated: the first in a positive way, the second by removing all but one of the values. `domain_guess` is the only non-determinate primitive. The last two, `domain_size` and `domain_values`, may yield different results depending on when they are called and should therefore be used with care.

`make_domain(D,Set)`

Can be executed when `Set` is instantiated to a non-empty list of distinct atomic terms, $[t_1,\ldots,t_n]$. `D`, which should be an unbound variable, is bound to a new domain, $\{t_1,\ldots,t_n\}$.

`make_domains(Ds,Set)`

Can be executed when `Set` is instantiated to a non-empty list of distinct atomic terms, $[t_1,\ldots,t_n]$, and `Ds` is a list of variables. Each variable in `Ds` is bound to a new domain, $\{t_1,\ldots,t_n\}$.

`domain_var(D,Var)`

Unifies `Var` with the *value variable* (a normal logical variable) of domain `D`. Subsequently, if `D` becomes an instantiated domain $\{t\}$, t is unified with `Var`. Alternatively, if `Var` becomes instantiated to t, if t is currently in the domain `D`, `D` becomes an instantiated domain $\{t\}$, otherwise failure occurs.

`domain_remove(D,Value)`

Can be executed when `Value` is ground. If `Value` is not currently in the domain `D`, there is no effect. If `D` is the instantiated domain {`Value`} the primitive fails. Otherwise `Value` is removed from the domain; if only one value, t, remains in the domain `D` becomes instantiated to $\{t\}$.

`domain_guess(D)`

Instantiates `D` non-determinately to one of its possible values. If `D` is the domain $\{t_1,\ldots,t_n\}$, `D` is instantiated successively to $\{t_1\}$, ..., $\{t_n\}$.

Note that `domain_guess(D)` is non-determinate (unless `D` is already instantiated) and can therefore be executed only if there are no determinate goals to execute.

domain_size(D,Size)

`Size` is unified with a positive integer which indicates the number of values currently in domain `D`.

domain_values(D,Values)

`Values` is unified with a list of the values currently in domain `D`.

4.2 Finite domain programming

Like Chip, our aim is to provide the programmer with a language as close as possible to Prolog but with the extensions necessary for constraint programming. However, the "Prolog" language supported by the Basic Andorra model differs in behaviour from that of regular Prolog, and this affects how the language is used. In this section we outline how our primitives can be employed in the context of Prolog on Andorra-I to solve constraint problems.

Program 1 is our solution to the familiar N-queens problem. This program is almost identical to the Chip one on p123 of [Van Hentenryck 1989a], except that the result of the goal `four_queens(Qs)` is a list of domains (which can be converted to a numeric value by `domain_var`). However, it executes differently. The execution order in Chip is the same as in Prolog, repeatedly executing a `domain_guess` goal for one domain followed by a `noattack` goal to remove inconsistent values from the other domains. On Andorra-I the program executes all of the `queens` and `noattack` goals first, since they are determinate, and sets up all '≠' constraints before `domain_guess` is called to non-determinately generate domain values.

```
four_queens(Qs) :-
   Qs = [Q1,Q2,Q3,Q4],
   make_domains(Qs, [1,2,3,4]),
   queens(Qs).

queens([]).
queens([Q|Qs]) :-
   domain_guess(Q),
   noattack(Q, Qs, 1),
   queens(Qs).

noattack(_, [], _).
noattack(Q1, [Q2|Qs], N) :-
   Q1 ≠ Q2,
   Q1 ≠ Q2 - N,
   Q1 ≠ Q2 + N,
   N1 is N + 1,
   noattack(Q1, Qs, N1).
```

Program 1: N-queens

At the end of the first determinate phase, the resolvent contains only the following goals, for `domain_guess` and the inequality predicate '≠', where each of `Q1`, `Q2`, `Q3`, and `Q4` is an uninstantiated domain:

```
domain_guess(Q1),
   Q1 ≠ Q2, Q1 ≠ Q2 - 1, Q1 ≠ Q2 + 1,
   Q1 ≠ Q3, Q1 ≠ Q3 - 2, Q1 ≠ Q3 + 2,
   Q1 ≠ Q4, Q1 ≠ Q4 - 3, Q1 ≠ Q4 + 3,
domain_guess(Q2),
   Q2 ≠ Q3, Q2 ≠ Q3 - 1, Q2 ≠ Q3 + 1,
   Q2 ≠ Q4, Q2 ≠ Q4 - 2, Q2 ≠ Q4 + 2,
domain_guess(Q3),
   Q3 ≠ Q4, Q3 ≠ Q4 - 1, Q3 ≠ Q4 + 1,
domain_guess(Q4).
```

The only goals that can be executed in the non-determinate phase are for `domain_guess`, since the '≠' goals are treated as `det_only` (see Section 3). Selecting the leftmost goal, `domain_guess(Q1)`, `Q1` is instantiated non-determinately to the domain {1} and a new determinate phase begins, in which all nine '≠' goals containing `Q1` can be executed in parallel.

This example illustrates a difference between our language and Chip, which follows from the Basic Andorra model: that the order of goals in a clause is irrelevant. Constraints and generators can appear in any order, but the constraints will always be set up before any non-determinate bindings are made. This is important, since it results in a smaller search space. In order to get the same effect (called "generalized forward checking") in Chip, the structure of the program has to be changed. However, we do have to make sure that constraints can be executed determinately, so that they execute first, whereas constraints need not be determinate in Chip.

The inequality predicate '≠' used above is an example of a constraint that is to be executed by forward checking. Such predicates can be programmed using the primitives of Section 4.1. As an example, Program 2 defines a constraint `plusorminus(X,Y,C)`, which means `X=Y-C` or `X=Y+C`. This can be executed in a forward checking way when either of domains `X` and `Y` is instantiated and the third argument is ground; it then leaves only (at most) the two values `Y-C` and `Y+C` (resp. `X-C` and `X+C`) in the domain of `X` (resp. `Y`).

In Program 2 we use Pandora syntax [Bahgat and Gregory 1989]. The `plusorminus` procedure is a "don't-care procedure" in the style of Parlog: the first clause removes the appropriate values from the domain of `Y` if domain `X` is instantiated, while the second does the converse. This procedure uses the `data` primitive to wait for the domain to be instantiated and the operator ':' to commit to the appropriate clause. A sequential conjunction operator '&' is used in the `pm` procedure, so that the values currently in domain `Y` are found (by a call to `domain_values`) only after the other arguments are instantiated. It then `filters` these values to find which ones must be removed from the domain, and removes them by calling `domain_remove`.

In addition to primitive constraints such as inequality, Chip allows user-defined constraints. These are conventional Prolog procedures augmented with a '`forward`' declaration indicating which

arguments should be ground and which should be domain variables. For example, `plusorminus` is defined [Van Hentenryck 1989a: p134] as follows:

```
forward plusorminus(d,d,g).
plusorminus(X,Y,C) :- X is Y - C.
plusorminus(X,Y,C) :- X is Y + C.
```

The problem with allowing user-defined constraints in Andorra-I is that the procedures may in general be non-determinate and, in any case, a search is required through the elements of a domain. One way to handle such constraints is by transforming the procedure to a determinate, forward checking, equivalent, as we did with `plusorminus` in Program 2. Another way would be to use a "determinate `bagof`" primitive which is currently being implemented in Andorra-I. This is similar to the `bagof` of Prolog but it executes as part of the determinate phase as a new subcomputation, even if it has to create internal choicepoints.

```
mode plusorminus(?, ?, ?).
plusorminus(X, Y, C) <-
   domain_var(X, Xv), data(Xv) :
   pm(Xv, Y, C).
plusorminus(X, Y, C) <-
   domain_var(Y, Yv), data(Yv) :
   pm(Yv, X, C).

mode pm(?, ?, ?).
pm(Xv, Y, C) <-
   Yv1 is Xv - C,
   Yv2 is Xv + C &
   domain_values(Y, Yvs),
   filter(Yvs, Yv1, Yv2, Remove),
   remove_all(Y, Remove).

mode filter(?, ?, ?, ^).
filter([], _, _, []).
filter([V1|Vs], V1, V2, R) <-
   filter(Vs, V1, V2, R).
filter([V2|Vs], V1, V2, R) <-
   filter(Vs, V1, V2, R).
filter([V|Vs], V1, V2, [V|Vs1]) <-
   V \== V1, V \== V2,
   filter(Vs, V1, V2, Vs1).

mode remove_all(?, ?).
remove_all(_, []).
remove_all(D, [V|Vs]) <-
   domain_remove(D, V),
   remove_all(D, Vs).
```

Program 2: Pandora program for the `plusorminus` constraint

The `deleteff` predicate of Chip, which is used to implement the first-fail heuristic, can easily be programmed using another of our primitives, `domain_size`. `deleteff(Best,Ds,Rest)` finds `Best` as the domain in list `Ds` that has the smallest current size; `Rest` contains the remaining elements of `Ds`. Program 3 is a program for N-queens which implements the first-fail heuristic (the `noattack` procedure is the same as in Program 1), and illustrates the general structure of such programs. Note that the "guessing" and "checking" components (the `guess_queens` and `check_queens` procedures) must be separated, though their order is unimportant.

```
four_queens(Qs) :-
   Qs = [Q1,Q2,Q3,Q4],
   make_domains(Qs, [1,2,3,4]),
   guess_queens(Qs),
   check_queens(Qs).

guess_queens([]).
guess_queens([Q|Qs]) :-
   deleteff(Best, [Q|Qs], Rest),
   domain_guess(Best),
   guess_queens(Rest).

check_queens([]).
check_queens([Q|Qs]) :-
   noattack(Q, Qs, 1),
   check_queens(Qs).
```

Program 3: Changes to N-queens to implement first-fail heuristic

The main issue in using `deleteff` in an Andorra-I program is to ensure that it is called at the right time, i.e., immediately before a choicepoint is created. By default, Andorra-I would execute all the `deleteff` goals immediately, since they are determinate. This would just choose the domains to guess in a fixed order. The easiest way to avoid this problem is to declare `deleteff` to be `non_det_only` (see Section 3).

During the first determinate phase, the `check_queens` goal executes to completion, spawning the same inequality ('$\neq$') goals as in Program 1, while `guess_queens([Q1,Q2,Q3,Q4])` reduces to the following:

```
deleteff(Best, [Q1,Q2,Q3,Q4], Rest),
domain_guess(Best),
guess_queens(Rest)
```

Now the leftmost goal, `deleteff`, runs and finds the smallest domain from `[Q1,Q2,Q3,Q4]`. In the next non-determinate phase `domain_guess` is called for the chosen domain, allowing some of the constraints to execute; when no more constraints can be executed, the next `deleteff` goal can execute, and so on.

4.3 Implementation

There are several ways to represent domains and to implement the predicates listed in Section 4.1. The predicates could be implemented by logic programs, provided we design a suitable representation of domains. Two of them, `domain_var` and `domain_remove`, modify the state of a domain but, happily, domains have the property that their size monotonically decreases. This enables us to represent

each domain by a tuple of logical variables, one for each possible domain value; the variable is bound to 0 when the value is removed, or 1 when the domain is instantiated to that value.

Given such a representation, the properties of a domain (e.g., it must not be empty, it cannot be instantiated to a value that has been removed, and so on) must be preserved. One way to do this is for each operation to check the state of the domain before modifying it. This works well in a sequential logic programming system, but is extremely complex to implement correctly in an and-parallel context because of contention by several operations modifying the same domain in parallel. A better method in the presence of and-parallelism is to spawn a process network to maintain the properties of a domain at the time it is created. This technique was described in [Bahgat and Gregory 1989].

Both of the above techniques were used to prototype our domain operations. However, to get more meaningful performance results, we wished to implement them as efficiently as possible, so a lower-level implementation was developed. A domain is represented by a structure containing the following fields:

1. A term (initially an unbound variable) representing the ultimate value of the domain. This term can be accessed by the `domain_var` primitive.

2. A boolean array with one bit for each potential member of the domain.

3. The number of elements currently in the domain. This field is accessed by the `domain_size` primitive.

4. The position of the last element guessed non-determinately.

5. A reference to a table mapping between domain values and positions in the domain.

The key implementation issues concern how to update the domains. Conditional modifications to domains (fields 2, 3, 4) need to be trailed. Fortunately, this can be achieved using the "updatable variables" which are already implemented in Andorra-I and used for many other purposes.

Each domain may be concurrently accessed by many constraints. To implement the required mutual exclusion, the value variable of a domain (field 1) is locked while the domain is modified, using the normal variable locking mechanism of Andorra-I. Each constraint locks only one domain at a time, so there is no danger of deadlock. Starvation is avoided because a domain is locked only when values are to be removed, and the size of domains is finite.

Both the updatable variables and variable locking features of Andorra-I are described in [Santos Costa *et al.* 1991].

5 Performance results

In this section we present some results obtained on the Andorra-I system running on a Sequent Symmetry. Each of the tables gives the results of running a particular program on different problem sizes. The respective columns show:

BT the number of backtrackings,

Time the execution time (in seconds) on one processor,

And-// the and-parallel speedup when run on 10 processors,

Or-// the or-parallel speedup when run on 10 processors.

(The speedup figures are simply the ratio of execution time on one processor to that on 10 processors.)

Table 1 shows the results of a standard Prolog program for N-queens. The structure of this program is similar to that of Program 1, but it makes no use of forward checking: it simply places a queen on each row non-deterministically and tests each time that the resulting configuration is safe with respect to previously-placed queens. The top part of the table gives results of a search for all solutions, while the bottom part shows a search for the first solution.

	N	BT	Time	And-//	Or-//
All solns	4	18	0.22	1.05	1.57
	6	208	2.92	1.11	4.23
	8	3544	54.32	1.17	8.83
	10	75190	1250.41	1.21	9.82
First soln	4	7	0.11	1.00	0.92
	6	46	0.65	1.12	2.41
	8	223	3.27	1.16	3.85
	10	276	3.71	1.16	1.98
	12	873	11.94	1.19	1.51
	16	42865	653.78	1.26	1.10

Table 1: Standard backtracking program for N-queens

Table 1 confirms that the search space and execution time increase dramatically as the problem size increases. It also shows that the or-parallel speedup for the first solution is very variable. This is usual, since an or-parallel search for one solution explores a different part of the search tree than a sequential search, so the backtrack count will differ

from that shown in the **BT** column and indeed will vary between runs. (We give the best or-parallel speedup obtained from several runs.) The consistent results are that a large or-parallel speedup is seen when searching for all solutions, while there is a very small and-parallel speedup in all cases. Both of these increase as the problem size increases. In every case, the or-parallel speedup observed is better than the and-parallel one.

Table 2 gives the same results for the forward checking program (Program 1). As expected, the search space is much reduced. The fact that the total execution time is also much smaller indicates that our implementation of finite domains is efficient enough that the cost of constraint solving pays off. The and-parallel speedup for this program is substantially larger than for the standard backtracking program (though it is still rather small), while the or-parallel speedup is generally less. In contrast to Table 1, for the first-solution search, the and-parallel speedup always exceeds the or-parallel one.

	N	BT	Time	And-//	Or-//
All	4	7	0.09	1.50	1.00
solns	6	41	0.64	1.78	2.91
	8	417	8.34	1.86	6.95
	10	6667	142.54	1.99	9.37
First	4	2	0.05	1.67	1.00
soln	6	8	0.19	1.73	1.46
	8	24	0.52	2.08	2.00
	10	24	0.61	2.18	1.49
	12	54	1.42	2.22	1.34
	16	1833	53.56	2.44	2.12

Table 2: Forward checking program for N-queens

We carried out similar experiments for the graph colouring problem: to colour a graph so that neighbouring nodes have distinct colours, and so that the number of colours used (the chromatic number) is minimized. The programs for this problem perform a depth-first branch-and-bound search by first finding an approximate solution with chromatic number C, then restarting the search with the added constraint that no node can be given a colour greater than C–1; this is repeated until no better solution is found.

Two programs for this problem were tested. The standard backtracking program colours nodes in descending order of degree; each time a node is coloured, each possible colour (from 1 to the current upper bound, C–1) is compared against the colour of each coloured neighbour. The forward checking program uses a domain of size C–1 for the colour of each node; when a node is coloured, the chosen colour is removed from its neighbours' domains. The latter program uses the first-fail heuristic to decide the order in which to colour nodes; when more than one node has the smallest domain, the one with the greatest degree is chosen.

Tables 3 and 4 give the results of our two graph colouring programs run on several randomly generated, constant density, graphs. **N** is the number of nodes and **D** is the density (the probability that any two nodes are connected). **CN** is the chromatic number of the graph. In the top half of each table we keep the size constant and vary the density; below we keep the density constant and vary the size.

N	D	CN	BT	Time	And-//
30	0.1	3	92	8.04	4.62
30	0.3	5	1768	29.20	2.47
30	0.5	7	2891	49.53	2.50
30	0.7	10	5567	81.47	2.23
30	0.9	16	3610	87.72	3.10
10	0.5	4	26	1.26	3.71
20	0.5	6	372	12.86	3.75
30	0.5	7	2891	49.53	2.50
40	0.5	8	256888	1557.59	1.09

Table 3: Standard backtracking program for graph colouring

N	D	CN	BT	Time	And-//
30	0.1	3	1	10.68	3.63
30	0.3	5	2	23.09	5.30
30	0.5	7	3	35.72	6.01
30	0.7	10	5	58.98	6.16
30	0.9	16	1	60.55	6.89
10	0.5	4	1	1.67	3.41
20	0.5	6	1	11.56	5.28
30	0.5	7	3	35.72	6.01
40	0.5	8	9	97.96	6.35

Table 4: Forward checking program for graph colouring

The results show that the use of forward checking dramatically reduces the search space, and also reduces the sequential execution time, especially for larger graphs. Moreover, the and-parallel speedup is much greater for the forward checking program.

6 Conclusions

We have described some extensions to Andorra-I that allow us to experiment with finite domain constraint logic programming in a parallel context. These extensions were implemented with very little effort, thanks to the existing features of the Andorra-I system, such as its coroutining mechanism, updatable variables, variable locking, etc. We have also shown

how easily constraint programs can be written in the Prolog variant supported by Andorra-I. For example, provided that constraints are determinate — a very common case — they are automatically executed "actively", in preference to non-determinate guessing.

Our experiments have confirmed that programs which use constraints are much faster than similar generate-and-test programs, demonstrating that our implementation of forward checking has no substantial overhead.

The results of parallel execution are particularly interesting. Constraint programs exhibit greater and-parallelism than generate-and-test programs, because the extra computation involved in forward checking can be parallelized by solving constraints in parallel. Evidence of this is the difference between Tables 1 and 2, and between Tables 3 and 4. For example, on one processor, forward checking solves the 16-queens problem 12 times faster than standard backtracking, and colours the 40-node graph 16 times faster. On 10 processors, the speed improvement due to forward checking increases to 24 times and 92 times, respectively.

Or-parallelism is usually measured for all-solutions search, mainly because this gives more consistent results than a search for one solution since the whole search tree is explored. The or-parallel speedup for a first-solution search is very variable and depends heavily upon the nature of the or-parallel scheduler built into the system. However, in many combinatorial search problems it is impractical to search for all (or many) solutions, so it is arguably more realistic to measure performance for first-solution search. Our results always give a much smaller or-parallel speedup for the first solution than for all solutions.

For the generate-and-test program of Table 1, the or-parallel speedup does exceed the and-parallel one, which is negligible. However, for the forward checking program of Table 2, the opposite is true. Although the and-parallel speedup in Table 2 is not large, it is enough to tip the balance in favour of exploiting and-parallelism, given a choice.

Finally, we should mention that all of our results concerning and-parallelism are specific to the Basic Andorra model. This is because Andorra-I is the only serious Prolog implementation that features dependent and-parallelism. (It seems unlikely that a system with independent and-parallelism could give similar results, since forward checking involves the solution of constraints that are mutually dependent.) As we noted in Section 3, the Basic Andorra model has a sequential bottleneck with respect to and-parallelism, which is ameliorated by the use of constraint solving. It would be interesting to see whether our results extend to other computational models combining dependent and-parallelism and search. An example of such a model, not yet implemented, is the Extended Andorra model [Warren 1990], which can execute even non-determinate dependent goals in parallel and therefore should not have such a bottleneck.

Acknowledgements

This work would not have been possible without the work of colleagues who implemented various components of the Andorra-I system, in particular Tony Beaumont, Inês Dutra, and Vítor Santos Costa.

We are grateful to Reem Bahgat and the referees for helpful comments on a draft of this paper.

Rong Yang is supported by ESPRIT contract 2471.

References

[Bahgat and Gregory 1989] R. Bahgat and S. Gregory. Pandora: non-deterministic parallel logic programming. In *Proc. 6th Intl. Conf. on Logic Programming* (Lisbon, June). MIT Press, 1989, pp. 471-486.

[Dincbas *et al.* 1988] M. Dincbas, H. Simonis, and P. Van Hentenryck. Solving a cutting-stock problem in constraint logic programming. In *Proc. 5th Intl. Conf. on Logic Programming* (Seattle, August). MIT Press, 1988, pp. 42-58.

[Santos Costa *et al.* 1991] V. Santos Costa, D.H.D. Warren, and R. Yang. The Andorra-I engine: a parallel implementation of the Basic Andorra model. In *Proc. 8th Intl. Conf. on Logic Programming* (Paris, June). MIT Press, 1991.

[Van Hentenryck 1989a] P. Van Hentenryck. *Constraint Satisfaction in Logic Programming*. MIT Press, 1989.

[Van Hentenryck 1989b] P. Van Hentenryck. Parallel constraint satisfaction in logic programming. In *Proc. 6th Intl. Conf. on Logic Programming* (Lisbon, June). MIT Press, 1989, pp. 164-180.

[Van Hentenryck and Dincbas 1986] P. Van Hentenryck and M. Dincbas. Domains in logic programming. In *Proc. AAAI-86* (Philadelphia, August 1986).

[Warren 1990] D.H.D. Warren. The Extended Andorra model with implicit control. *ICLP90 Workshop on Parallel Logic Programming* (Eilat, Israel, June 1990).