Towards Independent And-Parallelism in CLP

María García de la Banda[1], Francisco Bueno[2] and Manuel Hermenegildo[2]

[1] Monash University, Clayton 3168 VIC, Australia
[2] Universidad Politécnica de Madrid, 28660-Boadilla del Monte, Spain

Abstract. In this paper we propose a complete scheme for automatic exploitation of independent and-parallelism in CLP programs. We first discuss the new problems involved because of the different properties of the independence notions applicable to CLP. We then show how independence can be derived from a number of standard analysis domains for CLP. Finally, we perform a preliminary evaluation of the efficiency, accuracy, and effectiveness of the approach by implementing a parallelizing compiler for CLP based on the proposed ideas and applying it on a number of CLP benchmarks.

1 Introduction

Independent-and parallelism refers to the parallel execution of "independent" goals in the body of a clause. Its purpose is to achieve increased speed while maintaining correctness and efficiency w.r.t. sequential programs. Given the promising performance gains achieved using this type of parallelism in LP [11, 18, 2], it seems natural to explore the possibility of exploiting such parallelism in CLP programs. However, although both the and-parallel model and the notion of independence have already been extended to the CLP context [6], no practical parallelizing compilers have been reported so far. Even for concurrent constraint languages, designed for this purpose, to the best of our knowledge there have been no parallel implementations reported which perform real constraint solving.[3]

In this paper we study the new issues which need to be addressed in order to develop a parallelizing compiler for CLP programs. We first redefine the clause parallelization process for CLP programs, which as we show requires some significant modifications w.r.t. the traditional parallelization process used in LP. We then show how independence can be derived from the information inferred by a number of standard analysis domains for CLP. Finally, we provide empirical results from a prototype implementation of a parallelizing compiler based on two different notions of independence. It is important to note that the technology

[3] Partial exceptions are Andorra-I [17] and AKL [14], but their constraint solving capabilities are comparatively limited, they are based on execution models which are quite different from the CLP scheme, and they rely to a large extent on run-time technology for detecting parallelism.

necessary for parallel LP has taken almost a decade to mature to the point of producing significant speedups. Therefore, our empirical results cannot be expected to match such maturity level. More realistic benchmarks, faster run-time independence tests, and a more robust and versatile CLP parallel system are still needed. However, we feel we do provide a major step forward in that direction.

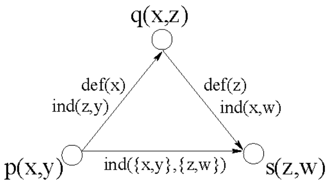
2 The automatic parallelization process in LP

The aim of the parallelization process is to detect the program parts that can always be run in parallel (are independent), those that must be run sequentially (are dependent), and, for those parts whose parallelism depends on unknown characteristics of the input data, to introduce the fewest run-time checks possible. Among the several different approaches proposed for LP, a particularly effective one seems to be that defined in [2, 1]. In this section we briefly summarize this approach. Its extension to CLP will be discussed in the following sections.

The parallelization process is performed as follows. Firstly, if required by the user, the program is analyzed using one or more *global analyzers*, aimed at inferring useful information for detecting independence. Then, an *annotator* performs a source-to-source transformation of the program in which eligible clauses are annotated with parallel expressions. This source–to–source transformation is referred to as the *annotation process*.

The annotation process is in turn divided into three subtasks. The first one is to *identify the dependencies* between each two literals in a clause and to generate the conditions which ensure their independence (we will only consider parallelization at the literal level). The left-to-right precedence relation for the literals in each clause h:-$b_1 \cdots b_n$ is represented using a conditional, directed, acyclic graph (CDG) in which each node i represents the literal b_i, every two literals b_i, b_j, $i < j$, are connected by an edge e_{ij} from i to j, and each edge e_{ij} is adorned with a set of tests T_{ij}. The set T_{ij} is defined as $\{def(D_{ij}), ind(I_i, I_j)\}$, where $D_{ij} = vars(b_i) \cap vars(b_j)$ and $I_k = vars(b_k) \setminus D_{ij}, k \in [i, j]$; the test $def(D_{ij})$ is true when D_{ij} is definite, and the test $ind(I_i, I_j)$ is true when I_i and I_j are independent for the particular independence notion considered. We will assume that the independence notion is symmetric.

Example 1.
The following figure provides the CDG associated to the clause:
`h(x,y,w):-p(x,y),q(x,z),s(z,w).`
For readability, in an abuse of notation, the singleton $\{x\}$ is represented as the element x. □



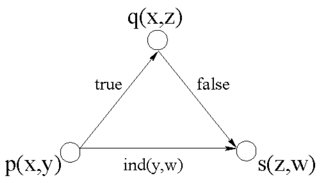
The run-time execution of the tests attached to the CDG edges can be expensive [2]. Therefore, the second task in the annotation process is to *simplify the dependencies*, reducing the sets of tests T_{ij} by means of the information inferred by the anayzers for the i-th program point. This improvement is based on identifying tests in T_{ij} which are ensured to either fail or succeed w.r.t. such information: if a test is guaranteed to succeed, it can be reduced to true; if a test is guaranteed to fail, the entire set of tests can be reduced to false.

As presented in [2], both the set of tests T_{ij} and the compile-time information associated to each program point i in each clause C, can be translated into a *domain of interpretation* DI for definiteness and independence: a subset of the propositional[4] logic theory, such that each element κ of DI defined over the variables in C is a set of formulae (interpreted as their conjunction) from a language suitable for expressing both the relevant information and the tests. The domain is augmented with axioms which hold for the notion of independence used, in such a way that the simplification task can be viewed as propositional theorem proving in this theory (although specific efficient algorithms can be used in practice). The accuracy and the size (the number of atomic formulae for simple facts) of each $\kappa \in DI$ depend on the kind of analysis performed.

Example 2. Consider the *strict independence* notion, in which terms b_1 and b_2 are said to be strictly independent for substitution θ iff $\text{vars}(b_1\theta) \cap \text{vars}(b_2\theta) = \emptyset$. Let $ind(I_i, I_j)$ be satisfied for substitution θ iff I_i and I_j are strictly independent for θ. Let $def(D)$ be satisfied for substitution θ iff $vars(D\theta) = \emptyset$. It is easy to prove that, in strict independence, the sets $\{def(D), ind(I_i, I_j)\}$ and $\{def(x) | x \in D\} \cup \{ind(x, y) | x \in I_i, y \in I_j\}$ are equivalent. I.e., the conditions defined over sets of variables can be "split" into conditions on single variables. Therefore, the domain DI can be based on a language containing only predicates of the form $def(x)$ and $ind(y, z)$, $\{x, y, z\} \subseteq vars(C)$, with the following simple axioms:

$$\{def(x) \rightarrow ind(x, y) | \{x, y\} \subseteq vars(C)\} \cup \{ind(x, x) \rightarrow def(x) | x \in vars(C)\}.$$

Parallelization methods based on such "split" domains will be referred to as *split methods.* Consider the clause `h(x,y,w) :- p(x,y),q(x,z),s(z,w)` whose CDG was shown in the previous example. Assume that, according to the information provided by some global analyzer, `x` is definite when `h(x,y,w)` is called. Then, for all program points previous to the execution of `q(x,z)`, the following information is available: {`def(x), ¬def(z), ind(y,z)`}. Thus, both tests labeling the edge from `p(x,y)` to `q(x,z)` are known to be true. Furthermore, at least one test labeling the edge from `q(x,z)` to `s(z,w)` is known to be false. Thus, the set can be simplified to *false*.



Finally, test `ind({x,y},{z,w})` labeling the edge from `p(x,y)` to `s(z,w)` is split into {`ind(x,z),ind(x,w),ind(y,z),ind(y,w)`} and simplified to `ind(y,w)`. □

As mentioned before, in the approach described, program parallelization is conceived as a source-to-source transformation into a suitable parallel language. The third and last task in the annotation process is concerned with expressing in such a language the conditional parallelism contained in the CDG. Consider a

[4] Although our syntax resembles first order formulae, clause variables can be regarded as constants, and a simple mapping into a propositional language can be done.

language in which parenthesized expressions are built using a fork / join operator, which is normally represented by "`&/2`", in addition to the sequential connective "`,`" (such expressions are said to be *linear*). Then, the aim of this last task is to apply a particular strategy *to obtain a (quasi-)optimal linear parallel expression* among all the possibilities represented by the simplified CDG, hopefully further optimizing the number of tests. Many different strategies for obtaining a linear parallel expression from a CDG can be defined [1]. A strategy is correct if the resulting linear parallel expression ensures that if the two linear subexpressions E_1 and E_2 are executed in parallel in store π, then E_1 and E_2 are independent for π, for the particular notion of independence considered.

Example 3. Let $lit(E)$ represent the set of literals in the linear expression E. When considering strict independence, linear expressions E_1 and E_2 are independent w.r.t. θ if $\forall b_i \in lit(E_1)$ and $\forall b_j \in lit(E_2)$, the tests associated to the edge connecting b_i and b_j are true w.r.t. θ. This can easily be ensured at run–time by, for example, using an if–then–else expression. Consider the simplified CDG shown in the previous example. It is possible to build different correct linear parallel expressions such as:
`h(x,y,w):- (p(x,y) & q(x,z)), s(z,w)` and
`h(x,y,w):- ind(y,w)->p(x,y)&(q(x,z),s(z,w));p(x,y),q(x,z),s(z,w)` □

3 Independence in CLP

Independence refers to the conditions that the run-time behavior of the goals to be run in parallel must satisfy to guarantee correctness and efficiency w.r.t. the sequential execution. The above parallelization process has been proved correct, implemented, and evaluated for the particular case of strict independence in LP languages [2, 1]. However, correctness can in fact be proved for all independence notions which, as strict independence, are *a-priori*, i.e., tests exist that can be executed prior to the goals, and *grouping*, i.e., if goal g_1 is independent of goals g_2 and g_3 for store π, then g_1 is also independent of the goal (g_2, g_3) for π.

The most general a-priori notion in CLP is defined in [6] as follows. Let $def\_vars(c)$ denote the set of definite variables in the constraint c (i.e., the set of variables uniquely defined by c). Let $\bar{x}$ denote a sequence of distinct variables. $\exists_{-\bar{x}}\phi$ denotes the existential closure of the formula ϕ except for the variables $\bar{x}$. Goals $g_1(\bar{x})$ and $g_2(\bar{y})$ are *projection independent* for constraint store π iff $(\bar{x} \cap \bar{y} \subseteq def\_vars(\pi))$ *and* $(\exists_{-\bar{x}}\pi \wedge \exists_{-\bar{y}}\pi \rightarrow \exists_{-\bar{y}\cup\bar{x}}\pi)$. Intuitively, the notion states that all shared variables must be definite, and that any constraint c in π defined over variables of both $\bar{x}$ and $\bar{y}$ is "irrelevant".[5] The same definition can also be applied to terms and constraints without any change.

Example 4. The goals $p(x)$ and $q(z)$ are independent and can be executed in parallel if the state of the store just before their execution is $\pi \equiv \{x > y, z > y\}$ since $\exists_{\{x\}}\pi = \exists_{\{z\}}\pi = \exists_{\{x,z\}}\pi = true$. However, they would be dependent for $\pi \equiv \{x > y, y > z\}$ since $\exists_{\{x\}}\pi = \exists_{\{z\}}\pi = true$ but $\exists_{\{x,z\}}\pi = \{x > z\}$. □

[5] That is, it is entailed by the conjunction of the constraints over $\bar{x}$ and the constraints over $\bar{y}$. Note that $(\exists_{-\bar{x}}\pi \wedge \exists_{-\bar{y}}\pi \leftarrow \exists_{-\bar{y}\cup\bar{x}}\pi)$ is always satisfied.

Unfortunately, projection is an expensive operation. A pragmatic solution, proposed in [6], is to simplify the run-time tests by checking if the variables involved are “linked” through the constraints in the store, thus sacrificing accuracy in favor of simplicity. Let Π denote the sequence of constraints in the store, $link_\Pi(x,y)$ holds if $\exists c \in \Pi$ s.t. $\{x,y\} \subseteq (vars(c) \setminus def\_vars(\Pi))$. The relation $links_\Pi(x,y)$ is the transitive closure of $link_\Pi(x,y)$. Finally, *links* is lifted to set of variables by defining $Links_\Pi(\bar{x},\bar{y})$ iff $\exists x \in \bar{x}, y \in \bar{y}$ such that $links_\Pi(x,y)$ holds. We then have that $g_1(\bar{x})$ and $g_2(\bar{y})$ are link independent for Π if $\neg Links_\Pi(\bar{x},\bar{y})$. Note that it is possible to further simplify $link_\Pi(x,y)$ by not taking knowledge of definite variables into account. This is simpler to detect for solvers which do not propagate definiteness. Run-time tests based on both this simpler version (*link independence*) and the original projection independence will be evaluated in our experiments. Note that while link independence is a grouping notion for any CLP language, projection independence is not.

Example 5. Consider goals `p(x)`, `q(y)`, and `s(z)`, and equation $x = y + z$. While $ind(x,y)$ and $ind(x,z)$ hold for projection independence, $ind(x,\{y,z\})$ does not. Thus, `p(x)` is independent of `q(y)` and of `s(z)`, but not of `q(y),s(z)`. □

4 The annotation process for non-grouping notions

Non-grouping independence notions considerably affect the annotation process. While the first subtask of the annotation process (identifying dependencies) is not affected, the other two subtasks become somewhat more complex.

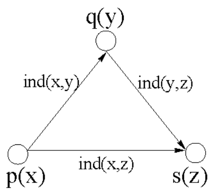
The first problem appears during simplification of dependencies: the sets $\{ind(I_i,I_j)\}$ and $\{ind(x,y) | x \in I_i, y \in I_j\}$ are no longer equivalent. This happened in the previous example, where $\{ind(x,[y,z])\}$ was shown to be different from $\{ind(x,y), ind(x,z)\}$ since, for example, the latter succeeds but the former fails for the linear equation $x = y + z$. As a consequence, the domain DI becomes more involved: it is now based on the language containing predicates of the form $def(x)$ and $ind(I_1,I_2)$, $\{x\} \cup I_1 \cup I_2 \subseteq vars(C)$, with the following axioms:

1. $\{def(x) \rightarrow ind(x, vars(C)) \mid x \in vars(C)\}$
2. $\{ind(I_1,I_2) \rightarrow def(I) \mid I \in subseteq \in (I_1 \cap I_2), I_1 \cup I_2 \subseteq vars(C)\}$
3. $\{ind(I_1,I_2) \rightarrow ind(I_{11},I_{21}) | I_{11} \subseteq I_1, I_{21} \subseteq I_2, I_1 \cup I_2 \subseteq vars(C)\}$
4. $\{ind(I, I_1 \cup I_2), ind(I_1,I_2) \rightarrow ind(I \cup I_1, I_2) \mid I \cup I_1 \cup I_2 \subseteq vars(C)\}$

The first and second axioms correspond to those defined for strict independence. The third states the independence of subsets of variables of an already proved independence test. The last one allows us to prove an independence test from the conjunction of other independence tests which are known to hold. In combination with the third axiom it can also be used as a heuristic: if $ind(I, I_1 \cup I_2)$ holds then $ind(I \cup I_1, I_2) \leftrightarrow ind(I_1,I_2)$, and we can replace $ind(I \cup I_1, I_2)$ by $ind(I_1,I_2)$. Definite and unconstrained variables can be eliminated using this heuristic. Parallelization methods based on this complex domain will be referred to as *non-split methods*. The parallel program obtained using a non-split method can always be executed using a grouping independence notion.

The non-grouping characteristics of the independence notion also affect the creation of the independence tests appearing in a linear parallel expression.

Example 6. Consider the clause `h(x,y,z):- p(x),q(y),s(z)` whose associated CDG is shown in the following figure. In strict independence, the three literals can be run in parallel in store π if all tests labeling the associated edges are satisfied in π. The following linear expression ensures this condition:
`(ind(x,y),ind(x,z),ind(y,z)) -> p(x)& q(y)& s(z) ; p(x),q(y),s(z)`
Unfortunately, the above expression is not correct for non-grouping independence notions. Consider projection independence and the constraint store $\pi \equiv \{x = y + z\}$. Though all three tests are satisfied for π, `p(x)` is not independent of `q(y)&s(z)` for π, as we saw in the previous example. Thus, they cannot be executed in parallel. □



As illustrated above, for a non-grouping notion, the conjunction of the tests labeling the edges involved is not enough for ensuring correctness. The solution is to directly apply the correctness condition: a linear parallel expression is correct if for any subexpression of the form E_1 `&` E_2, the test $\{def(D), ind(I_1, I_2)\}$, where $D = vars(E_1) \cap vars(E_2)$ and $I_k = vars(E_k) \setminus D, k \in [1, 2]$, is ensured to be satisfied immediately before their parallel execution.

Since the simplified tests are not used during the construction of the linear parallel expression, it is natural to question the usefulness of the simplification of dependencies subtask. Indeed, when considering non-grouping notions, the only aim of the second subtask is to detect edges whose label can be reduced to $false$. If so, the literals connected are known to be dependent and cannot be run in parallel. It is in the third subtask, after the parallel expression has been built, that the (full) simplification of dependencies should be performed.

Example 7. For the CDG in the above example, the independence test required to obtain a correct linear expression is {`ind(x,`{`y,z`}`), ind(y,z)`}. If the store $\pi \equiv \{x = y + z\}$ occurs in the program, and the compile-time analysis is able to infer that ¬`ind(x,`{`y,z`}`)` holds, then the condition is reduced to $false$ and no attempt is made to construct a parallel expression with all three goals. Otherwise the tests will have to be executed at run-time to determine independence. □

5 Global Analysis-Based Test Simplification

Compile-time information is usually obtained via global analysis, generally based on the abstract interpretation technique [4, 13]. In this section we study how definiteness and independence can be inferred by some analysis domains proposed for CLP programs: the Def[6] [5], Free [7], and FD domains [5].

Def approximates *definiteness* information. An abstract constraint AC is of the form (D, R), where the set of variables D approximates the definite variables,

[6] This domain is a variant of the *Prop* domain [3] for CLP, with efficient abstract functions specific for the framework of the PLAI analyzer used in our experiments.

and each element $(x, SS) \in R$ approximates definite dependencies. In particular, the variable x is known to be definite if all variables in a set S of SS are also definite. Consider an abstract constraint $AC_i = (D, R) \in \mathsf{Def}$ for program point i of a clause C. The contents of the corresponding $\kappa_i \in DI$ are as follows:

- $def(D)$
- $def(S) \rightarrow def(x)$ if $(x, SS) \in R, S \in SS$
- $ind(x, S) \rightarrow def(x)$ if $(x, SS) \in R, S \in SS$

The first rule states that all variables in D are known be definite. The second rule represents the definiteness dependencies approximated by each $(x, SS) \in R$. The third rule states that the dependencies approximated by R are definite: x and S can only be independent if x is definite. This rule can be used to (a) execute $def(\{x\})$ at run-time instead of $ind(\{x\}, S)$ (definiteness tests are faster and, thanks to the above axiom, they are known to be equivalent) and (b) with the aid of non-definiteness information, obtain $\neg ind(\{x\}, S)$, confirming a dependency.

Example 8. Consider a clause C such that $vars(C) = \{x, y, z, v, w\}$ and an abstract constraint $AC = (\{x\}, \{(z, \{\{w, v\}\})\})$. The corresponding κ will be: $\{def(x),\ def(\{w, v\}) \rightarrow def(z), ind(z, \{w, v\}) \rightarrow def(z)\}$. □

Free approximates *freeness* information. An abstract constraint AC is a set of set of variables approximating *possible* dependencies. In particular, if x does not appear in any element of AC, then x is known to be unconstrained; and if $\{x\}$ is not an element of AC, then x is possibly constrained but still free to take any value within its particular constraint type (referred to as $free$ variables). Consider an abstract constraint $AC_i \in \mathsf{Free}$ for program point i of a clause C. Let $(AC_i)^*$ denote its closure under union. Then $\kappa_i \in DI$ contains:

- $\neg def(x)$ if $x \in vars(C)$, $\{x\} \notin AC_i$
- $ind(I_1, I_2)$ if $\forall I_1' \subseteq I_1, I_1' \neq \emptyset, \forall I_2' \subseteq I_2, I_2' \neq \emptyset : (I_1' \cup I_2') \notin (AC_i)^*$

The first rule states that unconstrained and $free$ variables cannot be definite. The second is an extension of Prop. 3.1 of [7] defining the conditions under which I_1 and I_2 ae independent w.r.t. AC_i.[7] Note that unconstrained variables are independent of any other set of variables.

Example 9. Consider a clause C such that $vars(C) = \{x, y, z, v, w\}$ and an abstract constraint $AC = \{\{x\}, \{z\}, \{v\}, \{z, w\}\}$. The corresponding κ will be: $\{\neg def(y), \neg def(w),\ ind(x, w), ind(v, w),\ ind(y, \{x, z, v, w\})\}$. □

The aim of the FD domain is to improve the efficiency of the Free analyzer by explicitly separating the definite variables and specializing the abstract operations to make use of this particular information. The $\kappa \in DI$ corresponding to an abstract constraint in FD is obtained by conjoining the result of translating its definiteness and freeness components.

[7] The closure is needed due to the solved form in Free which eliminates any set in AC which can be obtained by the union of other sets in AC.

6 Experimental Results

Our experimental study evaluates (a) the efficiency and usefulness of the analyzers when parallelizing CLP programs, (b) the trade-off between the complexity and usefulness of the split and non-split methods (c) the efficiency and accuracy of the projection and link independence notions, and (d) the amount of a-priori and-parallelism detected by the method. To perform the evaluation we have implemented a parallelizing compiler based on the proposed ideas and incorporated it into the *CIAO* system [10]. The abstract machine includes native support for ***attributed variables*** [12] (used for implementating the constraint solvers), as well as for parallelism and concurrency (it is a derivation of the &-Prolog PWAM [11]). The system also includes the PLAI analysis framework [16] and several algorithms for obtaining the linear parallel expression associated to a given CDG. In the experiments we will use the **MEL** algorithm (see [1]).

Bench.	AgV	MV	Cl	Ls	Ps	Rec	Gs
ackerman	2.33	4	9	30	1	100	4
amp, amp2	4.62	18	45	69	16	37	20
amp3	5.03	26	60	103	24	37	33
bridge	3.72	12	18	33	6	66	17
circuit	3.39	10	18	25	8	62	10
dnf	2.34	7	32	40	3	100	14
fib	1.33	4	3	4	1	100	3
ladder	3.69	10	13	28	9	33	13
laplace1	4.00	12	4	4	2	100	4
laplace3	4.75	15	4	7	2	100	3
matmul1	2.50	5	6	5	3	100	3
mg, mggnd	3.00	6	2	3	1	100	2
mg-extend	3.90	8	10	16	6	33	9
mining	2.63	18	43	78	21	52	27
nombre	2.62	11	64	91	10	0	15
ode2, ode3	3.67	7	6	9	5	20	5
ode4	4.17	9	6	12	5	20	7
pic	5.71	9	7	23	7	0	9
power	3.14	19	42	75	18	50	24
rkf45a	7.30	26	97	236	41	24	63

Table 1. Benchmark Profile

6.1 Benchmarks

The set of benchmarks used includes programs in the set of examples of the clp($\Re$) distribution (fib, mg, dnf, and laplace), programs designed for PrologIII (nombre, mining, and power) which have been translated into clp($\Re$), and others designed for clp($\Re$) and used, for example, in the evaluation of several optimizations performed for the clp($\Re$) compiler. Table 1 provides information regarding the (reachable code of the) benchmarks, useful for interpreting the analysis and annotation performance results:[8] average (AgV) and maximum (MV) number of variables in each clause; total number of clauses (Cl), of body literals (Ls), and of predicates analyzed (Ps); percentage of predicates which are recursive (Rec), and total number of different goals solved in analyzing the program (Gs), i.e., of syntactically different calls. The benchmarks have not been normalized.

[8] Benchmarks in the same row refer to different queries with identical reachable code. Benchmarks with the same name in different rows refer to queries with different reachable code.

Benchmark	Analysis Times			Annotation Times							
				Split				Non-Split			
	Def	Free	FD	Lo	Def	Free	FD	Lo	Def	Free	FD
ackerman	0.04	0.12	0.10	0.03	0.05	0.05	0.05	0.04	0.05	0.06	0.06
amp	0.66	4.87	5.62	0.30	0.35	0.57	0.54	0.24	0.30	0.33	0.37
amp2	0.66	4.87	5.62	0.30	0.35	0.57	0.54	0.24	0.30	0.33	0.37
amp3	0.94	6.33	6.86	0.66	1.05	1.55	1.64	0.47	0.71	0.66	0.90
bridge	0.28	2.66	1.28	0.10	0.14	0.19	0.18	0.10	0.13	0.18	0.17
circuit	0.27	10.75	1.73	0.09	0.12	0.12	0.13	0.08	0.11	0.10	0.13
dnf	0.41	2.37	1.53	0.14	0.19	0.23	0.23	0.13	0.19	0.22	0.24
fib1	0.02	0.05	0.05	0.02	0.03	0.03	0.03	0.02	0.03	0.03	0.03
fib2	0.04	0.06	0.07	0.03	0.03	0.03	0.04	0.02	0.03	0.03	0.04
fib3	0.03	0.04	0.06	0.02	0.03	0.03	0.03	0.02	0.03	0.03	0.04
ladder	0.11	0.30	0.38	0.14	0.20	0.21	0.25	0.14	0.19	0.18	0.21
laplace1	0.03	0.00	0.05	0.02	0.02	0.00	0.03	0.02	0.02	0.00	0.02
laplace3	0.09	5.06	5.14	0.06	0.09	0.11	0.14	0.03	0.07	0.05	0.08
mmatrix	0.02	0.07	0.05	0.03	0.03	0.04	0.03	0.03	0.03	0.03	0.04
matmul1	0.03	0.08	0.12	0.03	0.03	0.03	0.03	0.03	0.03	0.03	0.03
mg-extend	0.98	0.31	1.77	0.06	0.09	0.11	0.10	0.06	0.08	0.10	0.10
mg	0.05	0.05	0.11	0.02	0.03	0.03	0.03	0.02	0.02	0.02	0.03
mggnd	0.01	0.05	0.03	0.02	0.02	0.03	0.02	0.02	0.02	0.02	0.02
mining	0.27	3.54	3.63	0.28	0.42	0.64	0.73	0.26	0.37	0.42	0.51
nombre	0.62	5.85	4.17	0.78	1.04	1.31	1.30	0.64	0.92	1.01	1.16
ode2	0.06	0.11	0.17	0.04	0.05	0.05	0.06	0.04	0.06	0.05	0.06
ode3	0.09	0.07	0.21	0.04	0.05	0.05	0.06	0.04	0.05	0.04	0.06
ode4	0.14	0.14	0.30	0.06	0.09	0.09	0.12	0.06	0.09	0.08	0.11
pic	0.04	0.12	0.11	0.22	0.16	0.23	0.18	0.17	0.16	0.17	0.17
power	0.45	5.43	1.99	0.39	0.62	0.83	0.94	0.29	0.47	0.50	0.69
rkf45a	335.14	61.00	482.69	2.06	27.81	3.45	22.15	1.20	15.19	1.52	17.76
Average 1	13.13	4.57	20.15	0.23	1.27	0.42	1.14	0.17	0.76	0.25	0.90
Average 2	0.25	2.22	1.65	0.16	0.21	0.30	0.30	0.13	0.18	0.20	0.23

Table 2. Compilation Efficiency Results

6.2 Efficiency

Table 2 presents the analysis and annotation times (for both the split and non-split methods) in seconds (SparcStation 10, one processor, SICStus 2.1 #5, native code). The times are the average out of ten executions. The last two rows show the average time for each analyzer and annotator with and without considering the results for rkf45a, respectively. The column Lo provides the results for the parallelization with information provided by a simple local analysis.

The analysis times are quite reasonable: 4-10 seconds for the bigger benchmarks. There are two exceptions: laplace1 and rkf45a. The analysis of rkf45a takes between 1 and 8 minutes, depending on the analyzer. In part, this is due to the high number of different calling patterns analyzed: 5 predicates have 10-15 different calling patterns, and one has 25. Even then, most analysis time is spent analyzing a single clause containing 12 atoms and 27 variables related by many definite and possible dependencies. In particular, the Def abstractions for this clause keep track of up to 593 sets of variables and 3166 variables, per abstraction. The other exception is the behavior of the Free analyzer for laplace1, which did not finish after one hour due to the high number of different calling patterns generated by the analysis. This is solved in laplace3 by performing a simple normalization which reduces the number of variables in the literal and thus the number of calling patterns. Thus, to be practical, the analyses should include a widening step (perhaps switching selectively to a special, compact definition of "top") and a tighter control of the number of calling patterns allowed.

Regarding the annotators, we observe that both methods are quite fast, the non-split method behaving almost consistently better, specially for the complex cases. This could seem surprising since, in the non-split method, further simplifications have to be performed for the final CGEs. However, the high number of tests obtained after splitting decreases the efficiency of the split method. We can conclude that, although conceptually more complex, the non-split method is usually faster.

6.3 Effectiveness: static tests

One way to measure the accuracy and effectiveness of the analysis information is to count the number of parallel expressions (or CGEs) annotated, the number of these which are unconditional (i.e., do not require run-time tests), and the number of definiteness and independence tests in the remaining CGEs. These numbers give an idea of the overhead introduced in the program. The results for the non-split method are shown in the upper part of Table 3. Then, the benchmarks for which the results obtained with the split method are different to those obtained with the non-split method are shown in the lower part of Table 3. For clarity, we only show those numbers which differ from the column corresponding to the annotation performed with the information provided by the FD analyzer, the rest appearing blank. Benchmarks that all analyzers determine to be sequential (mg, mggnd, ode2, ode3, and ode4) do not appear in the table.

In general, a lower number of CGEs and a higher number of unconditional CGEs indicate a better parallelization. It usually means that CGEs whose tests are going to fail have been detected and eliminated. This reasoning is valid for all benchmarks but ladder. In ladder the better information inferred by FD allows the annotator to change its strategy, obtaining better (because unconditional) parallel expressions. We can conclude that the higher information content provided by FD produces the best results, showing advantage in almost half of the benchmarks. Furthermore, many benchmarks present unconditional parallelism: the non-split method with the information provided by FD accurately detects that all a-priori and-parallelism in bridge, fib1, mmatrix, matmul1, laplace1, and mg-extend, is unconditional. Also, all analyzers successfully detect that mg, mgnd, ode2, ode3, and ode4 do not have any a-priori parallelism, FD being also capable of adding nombre to this list. Finally, though conceptually different, the non-split and split methods provide the same results for all but six benchmarks. In nombre the split method for Free is able to simplify 29 independence

Benchmark	CGEs: Tot/Unc				Conds: def/indep			
	Lo	Def	Free	FD	Lo	Def	Free	FD
ackerman				1/0	2/1		2/1	1/0
amp				3/0	2/	2/		1/3
amp2				3/0	2/	2/		1/3
amp3	6/	6/		5/0	5/21	5/	2/21	3/8
bridge	/0		/0	3/3	3/		3/	0/0
circuit	3/	3/		2/0	1/13	3/	0/9	2/1
dnf	/0		/0	14/12	/30		/30	0/2
fib1	/0	/0	/0	1/1	1/1	/1	1/	0/0
fib2				1/0				1/1
fib3				1/0	/1	/1		1/0
ladder	7/0	7/1	7/1	8/4	/34	5/8	/29	3/9
laplace1	/0			1/1	2/1			0/0
laplace3				1/0				2/1
mmatrix	/0	/0	/0	2/2	2/8	/2	2/2	0/0
matmul1	2/0	2/0	/0	1/1	2/8	1/1	1/1	0/0
mg-extend	/0		/0	1/1	/2		/2	0/0
mining	/0	/1	/0	4/2	5/8	/5	5/5	2/4
nombre	5/	5/	5/	0/0	/111	12/11	/65	0/0
pic	4/0	4/	/0	3/2	7/12	1/5	6/8	0/2
power				5/1	/46	/46		3/42
rkf45a	5/	5/		1/0	10/85	17/22		2/2
fib1	1/	1/	1/	0/0	1/1	/1	1/	0/0
fib2	1/	1/	1/	0/0	1/1	1/1	1/1	0/0
fib3	1/	1/	1/	0/0	1/1	1/1	1/	0/0
nombre	5/	5/	5/	0/0	/111	12/11	/47	0/0
power	5/	5/	5/	3/1	3/46	3/46	3/18	2/3
rkf45a	5/	5/		1/0	10/85	17/34		2/2

Table 3. Parallel Expressions / Conditional Checks

tests more than the non-split method. In fib1, fib2, fib3, and power, CGEs are eliminated for FD due to independence tests which are known to fail for any grouping notion (split method), but might succeed for a non-grouping one (non-split method). Finally, the results of the simplification function depend on the order in which the tests to be simplified are considered (the simplified set of tests is not always minimal). This only affects the parallelization of rkf45a with Def.

6.4 Link vs. Projection Independence

In this section we study the overhead created by the definiteness and independence tests, and compare the accuracy of the link and projection independence versions. Thus, we only consider benchmarks whose parallelized versions have tests. Unfortunately, we have not been able to execute some of these benchmarks (rkf45a, mining and power) in our parallel system, due to precision problems. For the rest, tables 4 and 5 show the results of the execution on one processor

Benchmark	Definite		Link Independence					Projection Independence				
	S	F	S	F	AS	AF	O	S	F	AS	AF	O
ackerman	0	1187	0	0	0	0	0.00	0	0	0	0	0.00
amp	0	1	10	3	1	0	0.62	10	3	1	0	4.77
amp2	0	1	10	5	1	0	0.67	11	4	1	0	5.87
amp3	0	7	48	98	0	1	0.17	48	98	0	1	0.63
circuit	0	14	1	7	0	0	0.12	1	7	0	0	3.40
dnf	0	0	115	0	1	0	0.33	115	0	1	0	3.00
fib1	0	0	0	609	0	1	0.39	609	0	1	0	0.71
fib2	0	1503	0	0	0	0	0.00	0	0	0	0	0.00
fib3	0	1503	0	0	0	0	0.00	0	0	0	0	0.00
ladder	0	154	50	0	2	0	0.00	50	0	2	0	0.00
laplace3	0	3	0	0	0	0	0.00	0	0	0	0	0.00
mmatrix	0	0	182	0	2	0	0.47	182	0	2	0	3.53
matmul1	0	25	5	0	1	0	0.00	5	0	1	0	0.00
nombre	0	448	0	0	0	0	0.00	0	0	0	0	0.00
pic	0	3	3	0	1	0	0.00	3	0	1	0	0.01

Table 4. Dynamic Results for Def

Benchmark	Definite		Link Independence					Projection Independence				
	S	F	S	F	AS	AF	O	S	F	AS	AF	O
ackerman	0	1187	0	0	0	0	0.00	0	0	0	0	0.00
amp	0	1	10	3	1	0	0.46	10	3	1	0	4.77
amp2	0	1	10	5	1	0	0.40	11	4	1	0	5.47
amp3	0	6	48	98	0	1	0.17	48	98	0	1	0.63
circuit	0	13	1	7	0	0	0.10	1	7	0	0	3.45
dnf	0	0	115	0	1	0	0.33	115	0	1	0	3.00
fib2	0	1503	0	0	0	0	0.00	0	0	0	0	0.00
fib3	0	1503	0	0	0	0	0.00	0	0	0	0	0.00
ladder	0	153	0	1	0	1	0.04	0	1	0	1	0.01
laplace3	0	3	0	0	0	0	0.00	0	0	0	0	0.00
pic	0	0	3	0	1	0	0.00	3	0	1	0	0.00

Table 5. Dynamic Results for FD

of the benchmarks parallelized with the non-split method, using the information provided by the Def and FD analyzers, respectively. The last ten columns in each table show the number of times that the tests have succeeded (S) and failed (F), the number of tests which have always succeeded (AS) and always failed (AF), and the overhead w.r.t. the execution of the original sequential program (TestTime/SeqTime), for the link and projection independence tests. For the definiteness tests only the number of successes and failures are shown, their overhead being negligible (it ranges from 0.0002 for ladder to 0.0028 for fib3).

Several conclusions can be extracted from the tables. First, although there exist cases, like fib1, in which projection independence detects parallelism which link independence fails to detect, this is not a common case. Second, as implemented, the independence tests introduce too much overhead, specially when using projection independence. The implementation of the tests is still very naive. In particular, the projection independence test can be improved significantly by, for example, performing a link test while doing the projection, so that success can be detected in an amount of time similar to that of the link test. In any case, significant effort must be devoted to implementing these tests efficiently.

Finally, given the cost of the independence tests and the number of them which always succeed or fail, we can conclude that more accurate information is needed. The domain (LSign) recently defined for CLP [15] approximates information about possible interaction between linear arithmetic constraints. A preliminary implementation performed at Monash University shows very promising accuracy, and will hopefully help in further simplifing the number of tests.

6.5 Effectiveness: speedup tests

The ultimate way of evaluating the effectiveness of the annotators is by measuring the speedup achieved, i.e., the ratio of the parallel execution time of the program to that of the sequential program. Since we are interested in the quality of the parallelization process, and not in the characteristics of a particular run-time system, this should ideally be done in a controlled environment. To this end, we have performed a number of preliminary experiments using the simulation tool IDRA [8], which was already shown to match actual speedups in several LP systems. This tool takes as input a real execution trace file of a parallel program run on the *CIAO* system (i.e., an encoded description of the events that occurred during such execution) and the time for its sequential execution, and computes the achievable speedup for any number of processors. The results presented in the following table show the speed-ups obtained parallelizing the benchmarks with the non-split method, using the information provided by the FD analyzer, and running the programs with the link independence tests. Only benchmarks with at least a parallel expression in the parallelized version have been considered. The column labeled "@2" provides the speedup on two processors and the column labeled "Max" the maximum possible speedup for the input data used.

Bench.	Max	@2
ackerman	1.00	1.00
fib2	0.99	0.99
fib3	0.99	0.99
laplace3	1.00	1.00

Bench.	Max	@2
bridge	1.02	1.02
fib1	80.84	1.99
laplace1	1.97	1.97
mmatrix	37.83	1.94
matmul1	3.53	1.71

Bench.	Max	@2
dnf	0.91	0.91
ladder	0.97	0.97
pic	1.97	1.50
dnf	0.94	0.94
ladder	1.16	1.16
pic	1.92	1.50

Bench.	Max	@2
amp	0.57	0.57
amp2	0.57	0.57
amp3	0.88	0.88
circuit	0.92	0.92
amp	0.97	0.97
amp2	0.97	0.97
amp3	1.00	1.00
circuit	1.00	1.00

The first four benchmarks are the only programs without a-priori parallelism whose parallelization actually contains conditional parallel expressions (mg, mggnd, nombre, ode2, ode3, and ode4 where already detected as sequential during the parallelization process). Even then, since the tests introduced by the parallelization are only definiteness tests, the overhead is negligible. In the next five benchmarks the compile-time information has been capable of determining for all tests whether they are going to succeed or fail, thus only obtaining unconditional parallelism. As a result, no slow-downs are obtained and most benchmarks get quite good speed-ups (which, as in many other benchmarks, depend on the size of the input data, which is generally small in the benchmarks used).

The next three benchmarks contain one or two conditional parallel expressions. The associated overhead in this case is higher (e.g., for dnf). This suggests perhaps eliminating those parallel expressions which contain tests. Since the overhead introduced by definiteness tests has proved negligible, only independence tests would need to be removed. The next three rows in the table show the results of applying this idea to the three previous benchmarks: although the achievable speed-up in pic is reduced w.r.t. the previous version it still has a speedup of 1.5, and the approach succeeds in eliminating the slow-down for the other two benchmarks.

The final four benchmarks have several independence tests which sometimes succeed and sometimes fail. The result is a considerable slow-down. However, after eliminating all parallel expressions containing independence tests, the results are shown in the last 4 rows of the table: no slow-downs occur, and since none of the benchmarks had very useful parallelism, the effect is quite satisfactory.

7 Conclusions and Future Work

We conclude from this preliminary evaluation that the parallelization process can be quite efficient (using widening) and relatively effective, specially considering the genericity of the domains used. Very often sequential programs and programs containing unconditional parallelism were statically identified as such. Surprisingly, the conceptually more complex non-split annotation method is faster in practice due to the reduced number of tests in the simplification. Regarding link and projection independence, although the latter can detect parallelism more accurately, it seems that in practice this rarely happens. While definiteness tests are very efficient our independence tests for these notions introduce significant overhead, which reduces the effectiveness of the parallelization. Overall, it appears best to use the non-split method, the most accurate domain, and, in order to avoid slow-downs while independence tests remain unoptimized, reduce such tests to false. Clearly, there is still quite a bit of room for improvement. The speedup results are significant when compared to standard compiler optimizations, but they are certainly not as good as those obtained for LP programs (e.g., [2]), and those obtained by or-parallelism for CLP programs which perform intensive search [9]. Identified avenues for future research include considering more realistic CLP benchmarks solving larger problems and combining both LP and CLP, studying better suited domains such as perhaps LSign [15], applying specialization, improving run-time test performance, controlling granularity, allowing the exploitation of a-posteriori parallelism, and parallelizing at finer grain levels than the goal level.

References

1. F. Bueno, M. García de la Banda, and M. Hermenegildo. A Comparative Study of Methods for Automatic Compile-time Parallelization of Logic Programs. In *Parallel Symbolic Computation*, pages 63–73. World Scientific Publishing, 1994.
2. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Logic Programming Symposium*, pages 320–336. MIT Press, 1994.
3. A. Cortesi, G. File, and W. Winsborough. Prop Revisited: Propositional Formulas as Abstract Domains for Groundness Analysis. In *Sixth IEEE Symposium on Logic in Computer Science*, pages 222–327, 1991. IEEE Computer Society.
4. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
5. M. García de la Banda and M. Hermenegildo. A Practical Approach to the Global Analysis of Constraint Logic Programs. In *International Logic Programming Symposium*, pages 437–455. MIT Press, 1993.
6. M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *ILPS'93*, pages 130–146. MIT Press, 1993.
7. V. Dumortier, G. Janssens, M. Bruynooghe, and M. Codish. Freeness Analysis in the Presence of Numerical Constraints. In *Tenth International Conference on Logic Programming*, pages 100–115. MIT Press, 1993.
8. M.J. Fernández, M. Carro, and M. Hermenegildo. Idra (ideal resource allocation): Computing ideal speedups in parallel logic programming. In *Proceedings of EuroPar'96*, LNCS. Springer-Verlag, 1996.
9. P. Van Hentenryck. Parallel Constraint Satisfaction in Logic Programming. In *International Conference on Logic Programming*, pages 165–180. MIT Press, 1989.
10. M. Hermenegildo, F. Bueno, M. García de la Banda, and G. Puebla. The CIAO Multi-Dialect Compiler and System. In *Proc. of the ILPS'95 Workshop on Visions for the Future of Logic Programming*, 1995.
11. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
12. C. Holzbaur. Metastructures vs. Attributed Variables in the Context of Extensible Unification. In *International Symposium on Programming Language Implementation and Logic Programming*, pages 260–268. LNCS631, Springer Verlag, 1992.
13. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 13/20:503–581, 1994.
14. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *Int'l Logic Programming Symp.*, pages 167–183. MIT Press, 1991.
15. K. Marriott and P. Stuckey. Approximating Interaction Between Linear Arithmetic Constraints. In *Int'l Logic Prog. Symp.*, pages 571–585. MIT Press, 1994.
16. K. Muthukumar and M. Hermenegildo. Compile-time Derivation of Variable Dependency Using Abstract Interpretation. *Journal of Logic Programming*, 13(2 and 3):315–347, 1992.
17. V. Santos-Costa, D.H.D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-parallelism. In *ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming.* ACM, 1990.
18. K. Shen. Exploiting Dependent And-Parallelism in Prolog: The Dynamic, Dependent And-Parallel Scheme. In *Joint Int'l. Conference and Symposium on Logic Programming* MIT Press, 1992.