Invited Paper at EUROPAR'97, August 1997, LNCS, Springer-Verlag

Automatic Parallelization of Irregular and Pointer-Based Computations: Perspectives from Logic and Constraint Programming

Manuel Hermenegildo

School of Computer Science
Technical University of Madrid (UPM), Spain
herme@fi.upm.es http://www.clip.dia.fi.upm.es/~herme

Abstract. Irregular computations pose some of the most interesting and challenging problems in automatic parallelization. Irregularity appears in certain kinds of numerical problems and is pervasive in symbolic applications. Such computations often use dynamic data structures which make heavy use of pointers. This complicates all the steps of a parallelizing compiler, from independence detection to task partitioning and placement. In the past decade there has been significant progress in the development of parallelizing compilers for logic programming and, more recently, constraint programming. The typical applications of these paradigms frequently involve irregular computations, which arguably makes the techniques used in these compilers potentially interesting. In this paper we introduce in a tutorial way some of the problems faced by parallelizing compilers for logic and constraint programs. These include the need for inter-procedural pointer aliasing analysis for independence detection and having to manage speculative and irregular computations through task granularity control and dynamic task allocation. We also provide pointers to some of the progress made in these areas. In the associated talk we demonstrate representatives of several generations of these parallelizing compilers.

Keywords: Automatic Parallelization, Irregular Computations, Speculation, Pointer Aliasing Analysis, Task Granularity Control, Global Analysis, Abstract Interpretation.

1 Introduction

Some very significant progress has been made in parallelizing compilers for regular, numerical computations, generally based on the FORTRAN language (see, e.g., [3]). This research has resulted in well known concepts and techniques including a well understood notion of independence (based on the Bernstein conditions or, for example, more recent notions of "semantic independence" [4]), sophisticated syntactic loop transformations, transformations based on polytope models, extensive work on partitioning and placement, etc. On the other hand, the applicability of these techniques has remained comparatively limited for irregular or symbolic computations, and still few practical systems deal with parallelization across procedure calls. Also, the techniques used often rely on the relative cleanliness of FORTRAN as a programming language and additional work is needed in order to extend them to other mainstream languages like C or C++. These languages include features such as dynamic, recursive data structures and pointer manipulation which complicate the detection of independence among statements or procedure calls and much current work is aimed at developing the related independence analyses. An important example is pointer aliasing analysis (see, e.g., [40] and its references).

We argue that, despite the apparent differences among imperative, functional, logic, constraint, and object oriented languages, the fundamental issues being tackled are quite similar. Thus, we believe that progress towards more effective parallelizing compilers for all programming paradigms can be sped up by cross fertilization of the results obtained in different paradigms. It is with this thought in mind that we present in the following a brief overview of some of the problems which appear in the area of automatic parallelization of logic and constraint programs. We also provide pointers to the some of the solutions and achievements of the area.

2 Logic and Constraint Programming

Due to space limitations, we will present only a brief overview of logic and constraint programming, specifically tailored to the objective of our presentation (the reader is referred for example to [42,30] for details). We warn the reader that this cannot in any way be considered a fair introduction to the topic, since we completely overlook aspects of logic and constraint programming which are widely perceived as important. These include the declarative nature and the logical semantics: programs in these languages are often not only the coding of an algorithm, but also a logical statement of a problem, which is very close to a specification. In the following we take a fully operational view – the same one that the parallelizing compiler takes.

The basic "statements" of a constraint logic program are *constraints*. Constraints relate (logical) *variables*. Such variables can be *free*, or they can be *constrained* to a certain value or set of values. For example, the statement `X=Y+Z` establishes that the given constraint must hold among those variables (we assume for example that the variables range over floating point numbers). Such constraints are kept in the *store*. Assume `Y` and `Z` have a "known" value at the time of executing this constraint (for example, the store contains `Y=2` and `Z=3`). Then, the operational semantics of such a constraint is very similar to that in any other language: the statement implies an addition (2+3) and an "assignment" of the result (5) to `X`. This can also be seen as *telling* (posting) the constraint `X=5`. Assume instead that such values are not known. Then executing the statement involves placing the constraint in the store for later solution if/when another constraint is executed. Sequences of constraints are separated by commas. Assume again an empty initial store and the sequence of constraints "`Y=2, X=Y+Z`". After executing this sequence the store would contain "`Y=2, X=2+T1, Z=T1`". Here, we are making the assumption that sequences of constraints execute sequentially in the order in which they appear and that the store is always kept as "fully solved" as possible and in a normalized form –see [30] for details.

Constraint logic programming also provides a method for *procedure abstraction*. For example, code segment *(a)* below:

`foo(Z,X) :- Y=2,`
`            X=Y+Z.` *(a)*
	`main :- foo(K,W),`
	`        K = 3,`
	`        write(W).` *(b)*

defines a two-argument procedure `foo`. A procedure defines a local dynamic invocation context in the usual way, i.e., upon entering the procedure `Y` is a new local variable while `X` and `Z` are formal parameters. The calling regime is not unlike "call by reference" (see the discussion later about logical variables being essentially pointers). For example, the effect of calling `foo(3,W)` is that upon return `W=5` is added to the calling context. Note that the procedure is syntactically not very different from what one would write in a functional or imperative language, and its behavior is essentially the same for calls such as `foo(3,W)`. However, the complete operational behavior of the constraint programming procedure is richer because it allows other "calling modes." For example, a call to `foo(K,5)` succeeds and upon return `K=3` is added to the calling context. Furthermore, a call to `foo(K,W)` also succeeds and upon return the constraint `W=2+K` is added to the calling context. In some ways, the statements and procedures in constraint programs can be seen as "reversible" versions of their syntactic counterparts in conventional languages. Note that also the declarative meaning of such programs is richer because it defines a complete logical *relation* (rather than a function) among its arguments. Procedure calls can appear in the bodies of procedures interspersed with constraints. For example, code segment *(b)* above would produce "5" on the standard output.

Procedures can have multiple definitions, which represent different *alternatives*. Establishing a somewhat inaccurate parallel with conventional languages, a set of procedure definitions can be seen as an "undoable" form of case statement or conditional. When such a procedure is entered it is said to create a *choice*. Such alternatives are tried in the textual order in which they appear in the program, i.e., the first definition of a procedure is tried first and, if that results in a *failure*, then the next one is tried (again, we follow the default execution strategy used in most practical constraint programming languages). A failure occurs when a constraint is executed which makes the store unsolvable (i.e., it is incompatible with the current state of the store). This is not unlike the case of a test evaluating to *false* in a conditional. When a failure occurs, the system *backtracks* to the last choice left behind and tries the next alternative in that choice. For example, the following program:

`main :- bar(K,W),`
`        K > 2,`
`        write(W).`
	`bar(X,Y) :- X < 0, Y = -10.`
	`bar(X,Y) :- X >= 0, Y = 10.`

prints "10". The first alternative of `bar` is tried first, resulting in `W=-10` and `K < 0`, but executing `K > 2` produces a failure since the store now has no solution. After trying the second alternative of `bar`, `K > 2` succeeds (the store is then `K > 2, W = 10`) and the program terminates after printing the value of `W`.

The following, slightly more interesting example defining the Fibonacci relation illustrates the use of recursion:

`fib(0, 0).`
`fib(1, 1).`
	`fib(N, F1+F2) :- N>1, F1>=0, F2>=0,`
	`                  fib(N-1, F1),`
	`                  fib(N-2, F2).`

(where some syntactic sugar is used). Calling `fib(8,Y)` establishes `Y=21`, and calling `fib(X,21)` establishes `X=8`. Calling `fib(X,Y)` produces as *alternatives* the constraints `(X=0, Y=0)`, `(X=1, Y=1)`, `(X=2, Y=1)`, etc.

In the previous examples we have been using a certain *constraint system*: essentially, equalities and inequalities involving linear arithmetic expressions over the (pseudo-)real numbers. In many cases the operations of constraint programs can be compiled directly into standard machine operations. However, in others (when actual constraint solving is involved) a *constraint solving algorithm* needs to be applied. Thus, the definition of each constraint system must include a decidable and (hopefully) efficient "solver." Practical languages typically include several constraint systems.

A particularly interesting constraint system present in almost all constraint languages is that of "equality relations over data structures" (i.e., finite trees). This is generally referred to as the *Herbrand domain* (and is the "working domain" of the Prolog language). For example, the following program (note that variable identifiers start with upper case while constants and data structure descriptors –functors– start with lower case):

```
main :- X = f(Y,Z),          W = g(K),
        Y = a,               X = f(a,g(b)).
        W = Z,
```

first builds (dynamically) a new two-argument structure whose constructor symbol is `f` (in other words, a tree whose root node is `f` and which has two open branches). The variables `Y` and `Z` are *pointers* to the two arguments of the structure. The statement `Y = a` "binds" the first argument of the structure to the constant `a` (i.e., at this time `X` points to `f(a,Z)`). The following statement *aliases* the pointers `W` and `Z` (e.g., `W` points to `Z`). Therefore, the result of the statement `W = g(K)` is to "bind" the second argument of the structure to `g(K)` (and as a result `X` now points to `f(a,g(K))`). The last statement finally binds `K` to the constant `b`. This last statement illustrates how open arguments inside a structure can also be accessed by traversing the structure using a process not unlike the "pattern matching" available in modern functional programming languages (except that it is again a "reversible" version of it). The algorithm capable of solving all such equality constraints over data structures is *unification*. One of the nice characteristics of this constraint system is that there exist very efficient algorithms for performing unification.[1] As mentioned before, Prolog, one of the most popular logic programming languages, is essentially a constraint logic programming language which uses exclusively the Herbrand domain. It is no surprise that Prolog is considered very well suited for the easy manipulation of data structures with pointers.[2]

[1] Furthermore, there are also very successful compilation techniques which (specially if global analysis of the program is performed) can translate sequences of operations such as those in the program above into a number of machine instructions that is essentially the same as if a lower-level language had been used to express the same data structure and pointer creation and binding operations. The reader is referred to [43] for an overview of progress in such compilation techniques.

[2] Modern logic and constraint programming languages have many other features, such as support for higher order and meta programming, module and object systems,

3 Parallelization of Constraint Logic Programs

One of the main theses of this paper is that logic programming and constraint programming languages offer a particularly interesting case study for the area of automatic parallelization. On one hand, these programming paradigms pose significant challenges to the parallelization task, which relate closely to the more difficult challenges faced in imperative language parallelization. Such challenges include highly irregular computations and dynamic control flow (due to the symbolic nature of many of their applications), non-trivial notions of (semantic) independence, the presence of dynamically allocated, complex data structures containing pointers, and having to deal with speculation.

On the other hand, due to their high-level nature these languages also facilitate the study of parallelization issues. As we have seen, logical variables are actually a quite "well behaved" version of pointers, in the sense that no castings or pointer arithmetic (other than array indexing) is allowed. Thus, pointers in these languages are not unlike those allowed, for example, in "clean" versions of C. In addition, similarly to functional languages, logic and constraint languages allow coding in a way which expresses the desired algorithm in a way that reflects more directly the structure of the problem. This makes the parallelism available in the problem more accessible to the compiler. The relatively clean semantics of these languages also makes it comparatively easy to use formal methods and prove the transformations performed by the parallelizing compiler both correct and efficient.[3] Quite significant progress has been made in the past decade in the area of automatic program parallelization for logic programs and some of the challenges have been tackled quite effectively. In the following touch upon a few of them (see, for example, [11] for an overview of the area).

Where the Parallelism can be Found: There are several types of parallelism which are traditionally exploited in logic and constraint programs. For example, in applications involving extensive *search* the choices represented by alternative procedure definitions are often "deep." I.e., a number of steps are typically executed before a failure implies exploring an alternative definition. In this case different processors can execute simultaneously the different procedure definitions (i.e., the different branches of this *search space*). The resulting parallelism is called *or-parallelism*. An alternative strategy is to parallelize the statements and/or procedure calls in procedure bodies, in the same way as in more traditional languages.[4] This kind of parallelism is referred to as *and-parallelism*. A typical

aggregation procedures, different sets of libraries, etc. with interesting implications on the automatic parallelization process. However, space limitations prevent us from considering these additional issues.

[3] Functional programming is another paradigm which also facilitates exploitation of parallelism. However, it can be argued that the lack of certain features, such as pointers and backtracking, while making the parallelization problem easier, also precludes studying some interesting problems.

[4] In fact, at a finer level of granularity, also *parts* of body statements can be executed in parallel. However, for simplicity, and without loss of generality, we assume parallelization at the *goal level*, meaning that the units scheduled will be body statements and procedure calls. Note also that the concurrency expressed by *concurrent logic*

example of and-parallelism is the parallel execution of the two recursive calls in the definition of the Fibonacci relation given before. Because and-parallelism corresponds to the traditional parallelism exploited in loop parallelization, divide and conquer algorithms, etc., we will concentrate our discussion on it. Also, and-parallelism is the only kind of parallelism that can be exploited in applications where choices are "shallow" (i.e., they correspond more closely to standard conditionals).

Correctness and Efficiency of the Parallelization: As in any other programming paradigm, the objective of the parallelizing compiler is to uncover as much as possible of the available parallelism, while guaranteeing that the correct results are computed (*correctness*) and that other observable characteristics of the program, such as execution time, are improved (*speedup*) or, at the minimum, preserved (*no-slowdown*) – *efficiency*. For comparison, consider the following segments of programs in *(a)* a traditional imperative language, *(b)* a (strict) functional language, and *(c)* a constraint logic programming language (we assume that the values of `W` and `Z` are initialized to some value before execution of these statements):

s_1	`Y := W+2;`	`(+ (+ W 2)`	`Y = W+2,`
s_2	`X := Y+Z;`	`Z)`	`X = Y+Z,`
	(a)	*(b)*	*(c)*

For simplicity, we will reason about the correctness and efficiency of parallelism using the instrumental technique of considering reorderings (interleavings). Statements s_1 and s_2 in *(a)* are generally considered to be *dependent* because reversing their order would yield an *incorrect* result, i.e., it violates the *correctness* condition above (this is an example of a *flow-dependency*).[5] A slightly different, but closely related situation occurs in *(b)*: reversing the order of function application would result in a run-time error (one of the arguments to a function would be missing). Interestingly, reversing the order of statements s_1 and s_2 in *(c)* does yield the correct result. In fact, this is an instance of a more general rule: if no side effects are involved, reordering statements does not affect correctness in a constraint logic program. As another example, consider the following program (which uses only the Herbrand domain, i.e., it is a Prolog program, and which we will call program *(d)*):

`main:-`
s_1 `p(X),`
s_2 `q(X),`
`write(X).`
	`p(X) :- X=a.`
	
	`q(X) :- X=b,` *`large computation.`*
	`q(X) :- X=a.`

Note that, again, reversing statements s_1 and s_2 produces the same result (`X=a`).

programming languages express is between and-tasks. See [28] for an extended discussion on this topic. Interesting models for exploiting and-parallelism at a finer level of granularity are, for example, [41,31].

[5] To complete the discussion above, note that output-dependencies do not appear in functional or logic and constraint programs because single assignment is generally used – we consider this a minor point of difference since one of the standard techniques for parallelizing imperative programs is to perform a transformation to a single assignment program before performing the parallelization.

The fact that (at least in pure segments of programs) the order of statements in constraint logic programming does not affect the result[6] led in early models to the proposal of execution strategies where parallelism was exploited "fully" (i.e., all statements were eligible for parallelization). However, the problem is that such parallelization often violates the principle of efficiency: for a finite number of processors, the parallelized program can be arbitrarily slower than the sequential program, even under ideal assumptions regarding run-time overheads. For instance, in the last example, reversing the order of the calls to `p` and `q` in the body of `main` implies that the call `q(X)` (`X` at this point is free, i.e., a pointer to an empty cell) will first enter its first alternative, performing the large computation. Upon return of `q` (with `X` pointing to the constant `b`) the call to `p` will *fail* and the system will backtrack to the second alternative of `q`, after which `p` will succeed with `X=a`. On the other hand the sequential execution would terminate in two or three steps, without performing the large computation. The fundamental observation is that, in the sequential execution, `p` *affects* `q`, in the sense that it *prunes* (limits) its choices. Executing `q` before executing `p` results in performing *speculative choices* with respect to the sequential execution. Note that this is in fact very related to executing conditionals in parallel (or ahead of time) in traditional languages (note that `q` above could also be (loosely) written as "`q(X) :- if X=b then` *`large computation`* `else if X=a then true else fail.`").

Something very similar occurs in case *(c)* above: while execution of the two constraints in the original order involves two additions and two assignments (the same of operations as those of the imperative or functional programs), executing them in reversed order involves first adding an equation to the system, corresponding to statement s_2, and then solving it against s_1, which is more expensive. The obvious conclusion is that, in general, arbitrary parallelization does not guarantee that the *two* conditions above are met.

Notions of Independence: Contrary to early beliefs held in the field, most work in the last decade has considered that violating the efficiency condition is as much a "sign of dependence" among statements as violating the correctness condition. As a result, novel notions of independence have been developed which capture these two issues of correctness and efficiency at the same time: independent statements as those whose run-time behavior, if parallelized, produces the same results as their sequential execution and an increase (or, at least, no decrease) in performance. As seen before, dealing with correctness is a matter of correctly sequencing side-effects (plus low-level issues, of course, such as locking). The techniques developed to this end are interesting, but, due to space limitations, we will concentrate on the arguably more interesting issue of guaranteeing efficiency. To separate issues better, we will discuss the issue under the assumption of ideal run-time conditions, i.e., no task creation and scheduling overheads (we will deal with overheads later). Note that, even under these ideal conditions, the statements in *(c)* and *(d)* are clearly *dependent*.

[6] Note that in practical languages, however, termination characteristics may change, but termination can actually also be seen as an extreme effect of the other problem to be discussed: efficiency.

A fundamental question then is how to guarantee independence (without having to actually run the statements, as suggested by the definition). A fundamental result in this context is the fact that, if only the Herbrand constraint system is used (as in the Prolog language), a statement or procedure call, `q`, *cannot be affected* by another, `p`, unless there are free pointers (pointers to empty structure fields) from the run-time data structures passed to `q` from the data structures passed to `p`. This condition is called *strict independence* [16,25].[7] For example, in the following program:

`main :- X=f(K,g(K)),`
`        Y=a,`
`        Z=g(L),`
`        W=h(b,L),`
	`p(X,Y),`
	`q(Y,Z),`
	`r(W).`

`p` and `q` are *strictly independent*, because `X` and `Z` point to data structures which do not point to each other, and, even though `Y` is a shared pointer, it points to a fixed value, which `p` cannot change (note again that we are dealing with single assignment languages). As a result, the execution of `p` cannot affect `q` in any way and they can be safely run in parallel (and, again assuming no run-time overheads, no-slowdown is guaranteed). Furthermore, no locking or copying of the intervening data structures is required (which helps bring the implementation closer to the ideal situation). Similarly, `q` and `r` are not strictly independent, because there is a pointer in common (`L`) among the data structures they have access to.

Unfortunately, the compiler cannot always determine independence by simply looking at one procedure, as above. For example, in the program *(a)* below:

`main :- t(X,Y),`
`        p(X),       (a)`
`        q(Y).`
	`main :- t(X,Y),`
	`        ( indep(X,Y)              (b)`
	`          ->          p(X) & q(Y)`
	`          ;           p(X), q(Y) ).`

it can determine that `p` and `q` are not (strictly) independent of `t`, since, upon entering the body of the procedure, `X`, `Y`, and `Z` are free pointers which are shared with `t`. On the other hand, after execution of `t` the situation is unknown since perhaps the structures created by `t` (and pointed to by `X` and `Y`) have no free pointers to each other. Unfortunately, in order to determine this for sure a global (inter-procedural) analysis of the program must be performed. An alternative is to compile in a *run-time test* just after the execution of `t`. This has the undesirable side-effect that then the no-slowdown property does not automatically hold, because of the overhead involved in the test, but it is still potentially useful. The compilation of such a test can be seen as a source to source transformation of the program as shown in program *(b)* above (where, following the &-Prolog notation, "`&`" represents parallel execution, and `(`*a* `->` *b* `;` *c*`)` is Prolog's syntax for "(if *a* then *b* else *c*)"). Furthermore, perhaps the global analysis can determine that in fact the operations that `t` performs on `X` and `Y` do not affect the execution of `p` and `q`. This kind of independence is called *non-strict independence* [26]. It cannot

[7] To be completely precise, in order to avoid any speculation, some non-failing conditions are also required of the goals executed in parallel, but we knowingly overlook this issue to simplify the discussion.

be determined in general *a priori* (i.e., by inspecting the state of the computation prior to executing `t`, `p`, and `q`) and thus necessarily requires a global analysis of the program. However, it very interesting because it appears often in programs which manipulate "open" data structures (difference lists, dictionaries, etc.).

An even more interesting case occurs if other constraint systems are used in addition to or in place of the Herbrand domain. Consider for example two procedure calls `p(X),q(Y)` and assume *(a)* that the store contains only `(X>Z,Y>Z)`. Assume, alternatively, that the store contains `(X>Z,Z>Y)` *(b)*. The simple pointer aliasing reasoning implied by the definition of strict independence does not apply directly. However, `p` cannot in any way affect `q` in case *(a)*, while this could be possible in case *(b)*, i.e., two calls are clearly independent in case *(a)* while they are (potentially) dependent in case *(b)*.

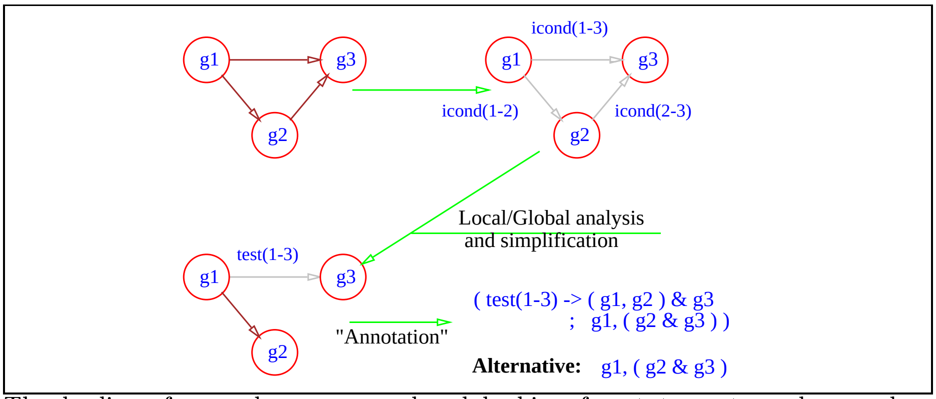
Notions of independence which apply to general constraint programming (and can thus deal with the situation above) have been proposed recently [21]. For example, two goals `p` and `q` are independent if all constraints posed during the execution of `q` are consistent with the output constraints of `p`.[8] The following is a sufficient condition for the previous definition but which only needs to look at the state of the store prior to the execution of the calls to be parallelized (for example, using run-time tests which explore the store c). Assuming the calls are $p(\bar{x})$ and $q(\bar{y})$: $(\bar{x} \cap \bar{y} \subseteq def(c))$ *and* $(\exists_{-\bar{x}} c \wedge \exists_{-\bar{y}} c \rightarrow\ \exists_{-\bar{y} \cup \bar{x}} c)$ where $\bar{x}$ is the set of arguments of `p`, $def(c)$ is the set of variables constrained to a unique value in c, and $\exists_{-\bar{x}}$ represents the projection of the store on those variables (the notion of projection is predefined for each constraint system). In the example above, for $c = \{\mathtt{X} > \mathtt{Z}, \mathtt{Y} > \mathtt{Z}\}$ we have $\exists_{-\{\mathtt{X}\}} c = \exists_{-\{\mathtt{Y}\}} c = \exists_{-\{\mathtt{X},\mathtt{Y}\}} c = true$ and therefore `p` and `q` are independent. For $c = \{\mathtt{X} > \mathtt{Z}, \mathtt{Z} > \mathtt{Y}\}$ we have $\exists_{-\{\mathtt{X}\}} c = \exists_{-\{\mathtt{Y}\}} c = true$ while $\exists_{\{\mathtt{X},\mathtt{Y}\}} c = \mathtt{X} > \mathtt{Y}$ and therefore `p` and `q` are not independent.

Other notions of independence proposed are based on "determinacy" (i.e., lack of choices) [39]: two computations that have no choices (i.e., "do not backtrack") are independent (provided, as before, that they can be guaranteed not to fail). Note that this is in general also captured by the notion of constraint independence given above.

The Parallelization Process: Experiments have shown that parallelization using only local analysis and generating run-time tests results in an excessive amount of overhead that severely limits speedups (see [8] for a recent comparison of actual speedups obtained by several parallelization methods). On the other hand it has also been observed that there exist programs that obtain better speedups if a limited amount of run-time checking of independence is used than if only static decisions are made. Thus, a parallelization methodology is generally used which can accommodate both static analysis and run-time checking.

One of the more widely used approaches is illustrated in the following figure (representing the parallelization of "$\mathtt{g}_1\mathtt{(\ldots)},\ \mathtt{g}_2\mathtt{(\ldots)},\ \mathtt{g}_3\mathtt{(\ldots)}$") [24,27,7]:

[8] This actually implies a better result even for Prolog programs since its projection on the Herbrand domain is a strict generalization of previous notions of non-strict independence. E.g., the sequence `p(X), q(X)` can be parallelized if `p` is defined for example as `p(a)` and `q` is defined as `q(a)`.



The bodies of procedures are explored looking for statements and procedure calls which are candidates for parallelization. As in many other parallelizers, a dependency graph is first built which in principle reflects the total ordering of statements and calls given by the sequential semantics. To control the complexity of the process these graphs are limited to one body of one procedure (if the body is too long, the body can be partitioned in segments, but this does not happen often in constraint logic programs). Each edge in the graph is then labeled with the independence condition (the run-time check) that would guarantee independence of the statements or calls joined by the edge. A global analysis of the program then tries to prove these conditions statically true or false. If a condition is proved to be true the corresponding edge in the dependency graph is eliminated. If proved false, then an unconditional edge (i.e., a static dependency) is left. Still, in other edges conditions may remain (possibly simplified). The annotation process then encodes the resulting graph in the target parallel language (a variant of the source language). The techniques proposed for performing this process depend on many factors including whether the target language allows arbitrary parallelism or just fork-join structures and whether run-time independence tests are allowed or not. As an example, the figure above presents two possible encodings in &-Prolog of the (schematic) dependency graph obtained after analysis. The parallel expressions generated in this case use only fork-join structures, one with run-time checks and the other one without them. Interesting techniques have been developed for compilation of *conditional* non-planar dependency graphs into fork-join structures, in addition to other, non graph-based techniques [17,35,7].

The global analysis required to simplify the conditional graphs has to perform, among other tasks, inter-procedural pointer analyses, not unlike those proposed for clean versions of C or C++. Early proposals based on traditional data flow analysis techniques pointed in the right direction but proved imprecise [10]. The presence of recursion and dynamic data structures has fueled the development of quite sophisticated, incremental inter-procedural analyzers based on abstract interpretation [12]. This has required the development of efficient analysis algorithms as well as abstract domains for accurately and efficiently keeping track of sharing patterns and pointer aliasing in recursive data structures [8,29,34,36]. These analyses have been applied to the detection of both strict and non-strict independence [8,9]. Analyses have been developed also to derive other important properties beyond variable instantiation states such as determinism [39], non-failure [13], and number of answers [6].

Dealing with Irregularity and Speculation – Dynamic Solutions: The preceding discussion has on purpose avoided the issue of run-time overheads. The obvious practical implication of the existence of overheads (task creation, scheduling, data movement, etc.) is that even if a task is known to be independent, its parallel execution may still render a slow-down. This can happen if the task does not represent a sufficient amount of computation with respect to the overheads incurred in its parallelization. In the case of constraint logic programming the problem is compounded by the fact that, because of the symbolic nature of the applications typically coded, the number of tasks generated at run-time (as well as the computational cost and dynamic memory demands of each such task) depend on run-time parameters.

Two main approaches have been explored in order to overcome these problems. The first one is to combine dynamic task allocation policies with compilation techniques (abstract machines) which reduce as much as possible the overhead involved in the parallel execution of tasks. The best results have been obtained by performing low level "micro-task" scheduling, independently of the operating system threads, and generally based on distributed "task stealing" approaches. Micro tasks are often represented simply by two pointers, one pointing to the procedure call or statement and another to the relevant invocation record. Interesting techniques have also been proposed for parallel dynamic memory management. These techniques efficiently support, for example, efficient memory recovery during parallel backtracking search. Some interesting examples of these dynamic scheduling and memory management techniques are presented in [22,24,37] for and-parallelism and in [33,1,18] for or-parallelism, where also quite interesting techniques for controlling speculation have been developed.

Dealing with Irregularity and Speculation – Static Solutions: While the dynamic techniques mentioned above have proven sufficient for obtaining speedups in previous generations of shared memory multiprocessors (paradigmatic examples are the Sequent Balance and Symmetry series), current trends point towards larger multiprocessors but with less uniform shared memory access times. Controlling in some way the granularity (execution time and space) of the tasks to be executed in parallel can be a useful optimization in such machines, and is in any case a necessity when parallelizing for machines with slower interconnections. This includes, for example, networks of workstations or the Internet. The problem is challenging because the tasks being parallelized are often procedure calls whose computational cost greatly depends on dynamic characteristics of the input data. One of the solutions currently used is to derive at compile time complexity cost functions which give *upper and lower bounds* on task execution time as a function of certain measures of input data [14,15,32]. Interestingly, this analysis makes use of some techniques developed in the context of imperative program parallelization, such as the Omega test [38]. Programs are transformed at compile-time into semantically equivalent counterparts but which automatically control granularity at run-time based on such functions. Performance improvements have been shown to result from the incorporation of this type of grain size control, specially for systems with medium to large parallel execution overheads.

4 Conclusions: Towards Cross-Fertilization

As a result of the work outlined in previous sections, quite robust, publicly available compilers and run-time systems have been available for some time now, generally for Prolog, which automatically exploit parallelism in complex applications. Such systems have been shown to provide speedups over the state of the art sequential implementations. The speed and robustness of these compilers has also been instrumental in demonstrating that abstract interpretation provides a very adequate framework for developing provably correct, powerful, and efficient global analyzers and, consequently, parallelizers [44]. More recently, techniques and practical tools have also been developed for the analysis of general constraint logic programs [20] as well as for their parallelization [19]. Prototypes incorporating the granularity control techniques mentioned above are also starting to be available. However, much work still remains to be done in these areas, and we believe there may be good opportunity at this time for increased transference of techniques across programming paradigms.

It can be argued that particularly strong progress has been made in the context of (constraint) logic programming in inter-procedural analysis of programs with dynamic data structures and pointers, in parallelization using conditional dependency graphs (and possibly generating run-time independence tests), in the definition of the advanced notions of independence that are needed in the presence of speculative computations or languages which include constraints, in the development of efficient task representation techniques and dynamic scheduling algorithms to deal with irregularity and speculation, and in the static inference of task cost functions for controlling granularity.

On the other hand, the techniques developed in the area of constraint logic program parallelization are certainly weaker than those developed in the context of numerical computing for regular problems. For example, logic programming parallelizers can discover the parallelism in complex recursive traversals of data structures, but do not handle well traversals that are based on integer (i.e., array subscript) arithmetic, for which much work exists in the area of imperative languages. Also, while current parallel constraint logic programming systems are reasonably good at dealing with tasks with dynamic costs, the techniques currently used are again comparatively weaker for the static case than the partitioning and placement algorithms used in imperative program parallelization [5,23]. Ideally, a parallelizing compiler should perform good partitioning and placement for any kind of architecture, using static techniques when possible and dynamic techniques when unavoidable. It thus appears that it would be quite interesting to merge the complementary work done in these areas by the different communities.

Constraint logic programming extends the high-level programming paradigm that logic programming offers in symbolic applications to numerical domains. We believe it offers a natural platform in which to study the combination of the parallelization techniques used in the numerical and symbolic programming fields. Independently of the convenience of using constraint programming languages directly (as is being done with significant commercial success in difficult problem areas such as scheduling or resource allocation), we also believe that many features of these languages, such as the use of constraints ("reversible statements") or the embedded search capabilities, will slowly make their way into the designs of mainstream languages. In the same way, other features of symbolic languages (such as dynamic data structure creation and garbage collection, or bytecode compilation) have already made it into widely used languages such as Java. Current proposals in this direction include ILOG (a commercially successful library which which extends C++ and Java with constraint handling capabilities) and [2], an imperative language with search capabilities.

References

1. K. A. M. Ali and R. Karlsson. The Muse Or-Parallel Prolog Model and its Performance. In *1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.
2. K. Apt and A. Shaerf. Search and Imperative Programming. In *POPL'97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 67–79, Paris, France, January 1997. ACM.
3. D. Bacon, S. Graham, and O. Sharp. Compiler Transformations for High-Performance Computing. *Computing Surveys*, 26(4):345–420, December 1994.
4. E. Best and C. Lengauer. Semantic Independence. *Science of Computer Programming*, 13:23–50, 1990.
5. J. Bevemyr, T. Lindgren, and H. Millroth. Reform Prolog: the language and its implementation. In *Proc. 10th Intl. Conf. Logic Programming*, Cambridge, Mass., 1993. MIT Press.
6. C. Braem, B. Le Charlier, S. Modart, and P. Van Hentenryck. Cardinality analysis of prolog. In *Proc. International Symposium on Logic Programming*, pages 457–471, Ithaca, NY, November 1994. MIT Press.
7. 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 Company, September 1994.
8. F. Bueno, M. García de la Banda, and M. Hermenegildo. Effectiveness of Global Analysis in Strict Independence-Based Automatic Program Parallelization. In *International Symposium on Logic Programming*, pages 320–336. MIT Press, November 1994.
9. D. Cabeza and M. Hermenegildo. Extracting Non-strict Independent And-parallelism Using Sharing and Freeness Information. In *1994 International Static Analysis Symposium*, number 864 in LNCS, pages 297–313, Namur, Belgium, September 1994. Springer-Verlag.
10. J.-H. Chang, A. M. Despain, and D. Degroot. And-Parallelism of Logic Programs Based on Static Data Dependency Analysis. In *Compcon Spring '85*, pages 218–225, February 1985.
11. J. Chassin and P. Codognet. Parallel Logic Programming Systems. *Computing Surveys*, 26(3):295–336, September 1994.
12. P. Cousot and R. Cousot. Abstract Interpretation: a Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *Fourth ACM Symposium on Principles of Programming Languages*, pages 238–252, 1977.
13. S. K. Debray, P. López García, and M. Hermenegildo. Non-Failure Analysis for Logic Programs. In *1997 International Conference on Logic Programming*, Leuven, Belgium, June 1997. MIT Press, Cambridge, MA.

14. S. K. Debray, N.-W. Lin, and M. Hermenegildo. Task Granularity Analysis in Logic Programs. In *Proc. of the 1990 ACM Conf. on Programming Language Design and Implementation*, pages 174–188. ACM Press, June 1990.
15. S.K. Debray and N.W. Lin. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
16. D. DeGroot. Restricted AND-Parallelism. In *International Conference on Fifth Generation Computer Systems*, pages 471–478. Tokyo, November 1984.
17. D. DeGroot. A Technique for Compiling Execution Graph Expressions for Restricted AND-parallelism in Logic Programs. In *Int'l Supercomputing Conference*, pages 80–89, Athens, 1987. Springer Verlag.
18. European Computer Research Center. *Eclipse User's Guide*, 1993.
19. M. García de la Banda, F. Bueno, and M. Hermenegildo. Towards Independent And-Parallelism in CLP. In *Programming Languages: Implementation, Logics, and Programs*, number 1140 in LNCS, pages 77–91, Aachen, Germany, September 1996. Springer-Verlag.
20. M. García de la Banda, M. Hermenegildo, M. Bruynooghe, V. Dumortier, G. Janssens, and W. Simoens. Global Analysis of Constraint Logic Programs. *ACM Transactions on Programming Languages and Systems*, 18(5):564–615, 1996.
21. M. García de la Banda, M. Hermenegildo, and K. Marriott. Independence in Constraint Logic Programs. In *1993 International Logic Programming Symposium*, pages 130–146. MIT Press, Cambridge, MA, October 1993.
22. M. Hermenegildo. An Abstract Machine for Restricted AND-parallel Execution of Logic Programs. In *Third International Conference on Logic Programming*, number 225 in Lecture Notes in Computer Science, pages 25–40. Imperial College, Springer-Verlag, July 1986.
23. M. Hermenegildo and M. Carro. Relating Data–Parallelism and And–Parallelism in Logic Programs. In *Proceedings of EURO–PAR'95*, number 966 in LNCS, pages 27–42. Springer-Verlag, August 1995.
24. M. Hermenegildo and K. Greene. The &-Prolog System: Exploiting Independent And-Parallelism. *New Generation Computing*, 9(3,4):233–257, 1991.
25. M. Hermenegildo and F. Rossi. On the Correctness and Efficiency of Independent And-Parallelism in Logic Programs. In *1989 North American Conference on Logic Programming*, pages 369–390. MIT Press, October 1989.
26. M. Hermenegildo and F. Rossi. Non-Strict Independent And-Parallelism. In *1990 International Conference on Logic Programming*, pages 237–252. MIT Press, June 1990.
27. M. Hermenegildo and F. Rossi. Strict and Non-Strict Independent And-Parallelism in Logic Programs: Correctness, Efficiency, and Compile-Time Conditions. *Journal of Logic Programming*, 22(1):1–45, 1995.
28. M. Hermenegildo and the CLIP group. Some Methodological Issues in the Design of CIAO - A Generic, Parallel, Concurrent Constraint System. In *Principles and Practice of Constraint Programming*, number 874 in LNCS, pages 123–133. Springer-Verlag, May 1994.
29. D. Jacobs and A. Langen. Accurate and Efficient Approximation of Variable Aliasing in Logic Programs. In *1989 North American Conference on Logic Programming*. MIT Press, October 1989.
30. J. Jaffar and M.J. Maher. Constraint Logic Programming: A Survey. *Journal of Logic Programming*, 19/20:503–581, 1994.
31. S. Janson and S. Haridi. Programming Paradigms of the Andorra Kernel Language. In *1991 International Logic Programming Symposium*, pages 167–183. MIT Press, 1991.

32. P. López García, M. Hermenegildo, and S. K. Debray. A Methodology for Granularity Based Control of Parallelism in Logic Programs. *Journal of Symbolic Computation, Special Issue on Parallel Symbolic Computation*, 22:715–734, 1996.
33. E. Lusk et. al. The Aurora Or-Parallel Prolog System. *New Generation Computing*, 7(2,3), 1990.
34. K. Muthukumar and M. Hermenegildo. Determination of Variable Dependence Information at Compile-Time Through Abstract Interpretation. In *1989 North American Conference on Logic Programming*, pages 166–189. MIT Press, October 1989.
35. K. Muthukumar and M. Hermenegildo. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Int'l. Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.
36. K. Muthukumar and M. Hermenegildo. Combined Determination of Sharing and Freeness of Program Variables Through Abstract Interpretation. In *1991 International Conference on Logic Programming*, pages 49–63. MIT Press, June 1991.
37. E. Pontelli, G. Gupta, and M. Hermenegildo. &ACE: A High-Performance Parallel Prolog System. In *International Parallel Processing Symposium*, pages 564–572. IEEE Computer Society Technical Committee on Parallel Processing, IEEE Computer Society, April 1995.
38. W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 8:102–114, August 1992.
39. V. Santos-Costa, D.H.D. Warren, and R. Yang. The Andorra-I Preprocessor: Supporting Full Prolog on the Basic Andorra Model. In *1991 International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.
40. M. Shapiro and S. Horwitz. Fast and Accurate Flow-Insensitive Points-To Analysis. In *POPL'97: 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–14, Paris, France, January 1997. ACM.
41. K. Shen. Overview of DASWAM: Exploitation of Dependent And-parallelism. *Journal of Logic Programming*, 29(1–3):245–293, November 1996.
42. L. Sterling and E.Y. Shapiro. *The Art of Prolog*. MIT Press, Cambridge MA, 1986.
43. P. Van Roy. 1983-1993: The Wonder Years of Sequential Prolog Implementation. *Journal of Logic Programming*, 19/20:385–441, 1994.
44. R. Warren, M. Hermenegildo, and S. K. Debray. On the Practicality of Global Flow Analysis of Logic Programs. In *Fifth International Conference and Symposium on Logic Programming*, pages 684–699. MIT Press, August 1988.

Acknowledgments: The author wishes to thank D. Padua, E. Pontelli, J. Mariño, F. Bueno, D. Cabeza, M. Carro, M. García de la Banda, P. López, and G. Puebla for their comments on previous drafts of this paper.

Demo Information: During the invited talk some of the capabilities of two (publicly available) parallelizing compilers are demonstrated. These are the &-Prolog system parallelizer [44,24,8] (which parallelizes standard Prolog programs) and the CIAO system parallelizer [19] (a more recent system which parallelizes constraint programs), both developed by our group, in collaboration with others.