(To be presented at IFIP Congress '89)

PARALLELISM IN LOGIC PROGRAMMING

Kazunori UEDA

Institute for New Generation Computer Technology
4-28, Mita 1-chome, Minato-ku, Tokyo 108, Japan

This paper discusses two aspects of parallelism in logic programming: parallelism as a computational formalism (often referred to as concurrency) and the implications of parallelism with regard to performance. Two alternatives for a parallel logic programming system are compared in detail. One allows programmers to describe processes and communication using concurrent logic languages, and the other attempts to exploit the parallelism of ordinary logic programs.

1. INTRODUCTION

Since the early days of logic programming, its affinity to parallelism has often been pointed out and studied [21][3][7]. There have been two major directions in research on parallelism in logic programming over the last ten years. One emerged from the process interpretation of logic programs introduced in the late 1970s [12], and led to the design and (possibly parallel) implementation of a variety of concurrent logic programming languages amenable to process interpretation [4][34][5][37]. These languages aim at the description of systems of processes and not directly at the description of search problems. Control is an integral part of the languages, and users program concurrent execution.

The other direction aims at the parallel execution of pure logic or Prolog programs that involve searching. Programmers may specify control only for guiding execution. OR-parallelism was exploited first [2][26][8][43], and AND-parallelism has been incorporated also [9][16][45][15].

There has been a long history of controversy between the proponents of these two directions. Proponents of concurrent logic programming languages claim that pure logic programs are not expressive enough to describe efficient parallel algorithms or to make effective use of the computational power of parallel computers. Proponents of the parallel execution of ordinary logic programs claim that concurrent logic languages are not logic languages because they fundamentally lack completeness in the sense of theorem proving, and that programming concurrency is too difficult for ordinary programmers. Section 2 compares these two directions from a semantical point of view and clarifies how their frameworks are different. Section 3 discusses how these directions can be used for writing efficient programs. Section 4 concludes the paper by proposing how to reconcile these directions.

2. PARALLELISM AS A COMPUTATIONAL FORMALISM

2.1 Logic as a Programming Language

Logic programming was born from theorem proving with Horn-clause logic. Kowalski [21] showed the procedural interpretation of Horn-clause logic, in which predicates are interpreted as procedures that can compute bindings (the values of variables) as the result of computation. The capability of computing values seems to be the minimum requirement for a framework to be called a programming language.

Since then, there have been two directions of research in logic programming. One is its extension and enhancement within the framework of theorem proving. Various alternative data domains to Herbrand universes have been considered for constraint satisfaction problems. Studies on control structures have yielded such techniques as coroutining, OR- and AND-parallelism, and forward checking. The inclusion of negation and other extensions of Horn-clause logic have been investigated also.

The other direction is the attempt to demonstrate the viability of logic programming for diverse aspects of programming. The most important is the design of a general programming language. Prolog is the first such language, but its generality rests more or less on impure constructs such as cuts, side-effects, and meta-logical predicates. The redesigning of these features is still in progress [42][24]. Other aspects of programming we may wish to consider include systems and meta programming, concurrent programming and programming in the large, which motivated studies on meta-interpreters, perpetual processes and modularization, respectively. These paradigms may be put into practice simply by developing new interpretations of logic or new programming techniques, or they may be put into practice only by adding new language constructs. In the latter case, we should give clean semantics to the constructs.

2.2 Process Interpretation of Logic Programs

In the late 1970s, much research was conducted on the coroutining (pseudo-parallel) execution of logic programs [3][30][14][19], while Kahn [20] had shown a network of communicating processes to be a simple and elegant framework of concurrent programming. These two lines of research were put together by van Emden and de Lucena Filho [12], who

introduced process interpretation of logic programs and initiated the use of logic for concurrent programming with communicating processes.

By a process we mean a unit of computation that may run in parallel with other processes and communicate with them. In their interpretation, each process sequentially executes goals in its own stack. Goals belonging to different processes may share variables, which may be used as communication channels. In general, it is the attention to communication that characterizes concurrent programs. The semantics of processes should therefore describe the *process* or *history*, rather than the result, of computation. The result of computation is not necessarily important, and accordingly, processes need not terminate.

In spite of the proposal of process interpretation, pure Horn-clause logic could not be used immediately for concurrent programming because it was not clear what to do with its ability to compute multiple solutions. We still had to design a concrete programming language in which to describe communicating processes. The first concrete concurrent logic programming language was Relational Language [4]. It introduced Dijkstra's concept of the guard [10] into logic programming for the first time. This made Relational Language capable of describing don't-care nondeterministic processes. The subsequent concurrent logic languages attempted to refine existing ones or to enhance their expressive power. These languages include Concurrent Prolog [34], PARLOG [5], Guarded Horn Clauses (GHC) [37][38], Flat Concurrent Prolog [35] and Oc [18]. A survey and a genealogy of these languages can be found in [36] and [31], respectively.

Here, we introduce GHC without guard goals as a process description language. This subset of GHC is essentially equivalent to Oc, which is the simplest of the concurrent logic languages.

A program is a set of guarded clauses of the following form:

$$h \texttt{ :- | } B$$

where h is an atom(ic formula) called a head and B is a multiset of atoms called goals. Each clause represents a rewrite rule of a goal. The commitment operator '`|`' divides a clause into a guard (left-hand side) and a body (right-hand side). The guard specifies the condition for rewriting, and h in particular specifies the template of a goal to be rewritten. The body specifies the multiset of goals that replaces the old goal.

The execution of a program begins with the initial multiset of goals specified by a goal clause of the following form:

$$\texttt{:- } B$$

Goals in B rewrite themselves in parallel. Let g be a goal in B. Then

(1) if there are a clause h `:- |` B' and a substitution θ such that $g \equiv h\theta$ (that is, g and h are unifiable without instantiating g), then g is replaced by $B'\theta$, and

(2) if g is of the form $t_1 = t_2$ and t_1 and t_2 are unifiable with a most general unifier (mgu) θ, then g is deleted and θ is applied to the rest of the goals.

Unlike the original process interpretation in [12], we no longer have any notion of sequential execution. A process is just an entity that observes and generates substitutions. A substitution, which is a finite set of bindings between variables and their values, models a piece of information transferred between processes. A process is realized by a multiset of goals which reduce themselves into other goals repeatedly using guarded clauses. Interprocess communication is done by unification. Unification executed in a body is for generating a substitution, and unification executed in a guard is for observing a substitution. A process is an abstract entity for our understanding of a computation; what multiset of goals should be regarded as a process depends entirely on our interpretation of the computation.

To transfer information, its sender and its receiver(s) must share a variable. Synchronization accompanying communication is realized by Rule (1), which allows a goal to rewrite itself only after it is sufficiently instantiated. Rule (1) controls the direction of information flow and is the only means of control in the language. The application of an mgu in Rule (2) need not be done as an atomic action. The information represented by the mgu need only be published eventually [33].

Control in a concurrent logic language is not like control in an ordinary logic language. While control in an ordinary logic language is for efficiency and is independent of logic, control in a concurrent logic language determines the direction of communication and hence is a far more essential construct of the language. Remembering Kowalski's equation Algorithm = Logic + Control [22], we see that concurrent logic languages are for describing concurrent algorithms.

Unification enables quite flexible interprocess communication; this is a unique feature of concurrent logic languages. It can be used both for one-directional communication such as pipelining and for two-directional communication such as messages that require replies. Another feature of concurrent logic languages is that, unlike concurrent procedural languages, communication channels (streams) are not part of the language constructs but are represented and operated as ordinary lists. This contributes much to the simplicity of the languages. The following is a program implementing stack objects:

```
stack([push(X)|S],D     ) :- | stack(S,[X|D]).
stack([pop(X) |S],[Y|D1]) :- |
   X=Y, stack(S,D1).
stack([],         D     ) :- | true.
```

To use a stack, we first generate a process `stack(S,[])` and instantiate `S` to a list of requests. For example, if the goals `S=[push(5)|S1]`, `S1=[push(6)|S2]`, `S2=[pop(X)|S3]` and `S3=[pop(Y)|S4]` are executed (in any order), `X` and `Y` will be instantiated to `6` and `5`, respectively.

Note that Rule (1) expresses don't-care nondeterminism in the choice of a rewrite rule, due to which concurrent logic programming languages are not complete when viewed as a theorem prover of Horn-clause logic.

2.3 The Frameworks of Ordinary and Concurrent Logic Languages

This section compares the frameworks of ordinary logic lan-

guages with the ability to generate multiple answer substitutions (don't-know nondeterminism) and concurrent logic languages with don't-care nondeterminism. Some concurrent logic languages feature don't-know nondeterminism also [46][32]. Those languages are considered more similar to ordinary logic languages than to other concurrent logic languages from the viewpoint given below.

The fundamental difference between the frameworks of ordinary and concurrent logic languages lies in the way in which the result of computation is observed and the material for computation is provided. Whether explicitly by input/output primitives or implicitly by the system, transfer of information to and from the outside world *must* be done in any computational system. Transfer of information will ultimately be done by some operational means; the question is how it should be modeled in declarative languages.

In a word, the difference is that a concurrent logic programming system is an open system while an ordinary logic programming system is a closed system. In concurrent logic programming, input/output is formulated as interprocess communication. The advantages of this formulation are that no special operations need be introduced and that a program has full control over input/output. The outside world (more specifically, each peripheral device) is modeled as a system-defined process that observes and generates substitutions. This process is assumed to run in parallel with user-defined processes. A concurrent logic programming system should provide a means to establish communication channels between system and user processes. In sum, the outside world *participates in* the execution ('proof' in logic programming terms) of a program which proceeds with real time. This is why a concurrent logic programming system can be called an open system.

In contrast, the framework of ordinary logic programming is that of theorem provers. The outside world *observes* proofs at the meta-level, the level of the system that searches for proofs. This observation is considered to be done using some special mechanism inaccessible from within a program. We often wish to observe different answer substitutions of a goal clause obtained from different, independent proofs, but this is enabled only by *simulating* don't-know nondeterminism. Don't-care nondeterminism can be directly implemented on an actual computer, but don't-know nondeterminism must be simulated.

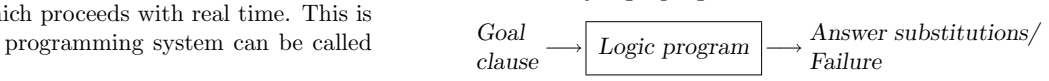
Why do concurrent logic languages lack the ability to generate multiple answer substitutions? It is often claimed that this is to avoid complex mechanisms such as distributed backtracking between communicating processes. However, the more fundamental reason is the incompatibility of the ability to generate multiple answer substitutions and the ability to communicate with the outside world. If we allowed multiple proofs beginning with a multiset of goals, that many processes corresponding to the outside world would have to be created. If we were *reasoning about* the interaction of processes with the outside world, we could consider multiple possible outside worlds. However, the purpose of concurrent logic languages is to describe the actual interaction of processes with the real outside world. The real outside world can participate in only one of the proofs, the one that actually happens.

To sum up, concurrent logic languages and ordinary logic languages have quite different purposes. Concurrent logic languages aim at the description of efficient concurrent systems, and ordinary logic languages aim at the high-level description of problem solving. Which of the frameworks should be used depends on whether the interaction with the outside world is important or not in the problem to be programmed.

Finally, we note that although concurrent logic languages are generally suitable for describing systems of processes, not all of them are suitable as they are for systems programming such as the writing of operating systems. An operating system must be able to safely execute user programs that may not be cooperative with the operating system, whether inadvertently or deliberately. Concurrent logic languages proposed so far took different approaches to this requirement. Flat Concurrent Prolog enabled systems programming by adopting larger atomic operations. That is, it made indivisible the two aspects of resolution, rewriting and unification, while PARLOG and GHC separated them. Instead, PARLOG featured an additional construct called 'metacall'. KL1 [1], the kernel language for the Multi-PSI machine based on GHC, took an approach similar to PARLOG's to write an operating system PIMOS [1].

2.4 The Semantics of Ordinary and Concurrent Logic Programs

Let us compare ordinary and concurrent logic programs in terms of their semantics. First we examine the typical view of an ordinary logic program:



The question is: what is *the* result of the computation, or what is *the* output of the system? Each computed answer substitution could be called *a* result of don't-know nondeterministic computation, but this view is valid only within the simulated world of don't-know nondeterminism.

Let us suppose that *the* result is the set of all computed answer substitutions. What we would like to consider next is the way to pass that result to the subsequent computation, because without this facility, we cannot write a program that collects and processes (compares, for example) the solutions of some goal despite the search capability of logic programming. One possibility is to represent each substitution implicitly as the value of a variable or a term of interest and to represent the set of substitutions as a list of such values. Another possibility is to explicitly represent each substitution as a first-class object, namely an association list.

Whatever representation may be used, putting answer substitutions together into a single data structure requires meta-level operations, operations at the level implementing exhaustive search. For this reason, most Prolog systems provide all-solutions predicates such as `setof` of DEC-10 Prolog, all of which adopt implicit representation.

The semantics of existing all-solutions predicates is, however, by no means clear, as Naish [28] pointed out. For example, they can be used for defining the extralogical predicate

`var`. The major source of this and other problems seems to be the improper treatment of variables in the goal (for which exhaustive search is performed) and the result. First, it is considered problematic to allow an uninstantiated variable not appearing free in the all-solutions goal (called a local variable) to appear in the result [28][39], because what a local variable represents is quite model-dependent. The result with local variables cannot provide a model-independent notion of the number of solutions. Furthermore, with local variables, the universal closure of a successful all-solutions goal (with the computed result list) can be logically wrong [28].

Non-local variables, namely variables appearing free in the all-solutions goal, are less problematic if we disallow them to be instantiated during the exhaustive search. For example, the DEC-10 Prolog goal `setof(X,perm([A,B,C],X),S)` for generating permutations will return the binding `S = [[A,B,C],[A,C,B],...,[C,B,A]]`. Although a goal `perm([A,B,C],X)` may subsume infinitely many elements of the least Herbrand model generated from the definition of `perm`, it has only six results (say $x_1, \ldots, x_6$) for which the universal closures $\forall$(`perm([A,B,C],`x_i`)`) are logical consequences of the definition of `perm`. A problem still remains when we want the result to be a set rather than a bag, because in this case, the values of non-local variables given from outside may affect the number of different solutions. However, this problem can be easily avoided by distinguishing between non-local variables which will not be instantiated and those which will be instantiated, and by indicating the occurrences of the former by constant symbols. If the result can be a bag, non-local variables can always be left as they are.

Now our goal in the implicit representation approach is to restrict the use of all-solutions predicates to safe cases where local variables do not occur in the result and non-local ones are not instantiated. One possible approach is static checking. Ueda [39] proposes a compilation technique from an exhaustive search program into a deterministic program. The technique is based on dataflow analysis, and a program amenable to compilation is safe in the above sense. The technique was developed for a class of programs manipulating ground data, but it will allow generalization. A problem is that the dataflow of Prolog programs that make use of logical variables as blackboards does not allow simple static analysis. How to analyze and compile such programs has yet to be studied.

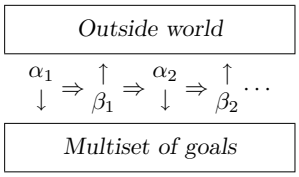
Another solution to the treatment of variables in exhaustive search might be to represent both the goal and the result by ground terms, using constant symbols to indicate the occurrences of variables. In this scheme, the result can be represented either implicitly (by values) or explicitly (by substitutions). However, the ground representation may be too powerful in that it tells not only what variables are bound but what variables are *not* bound; such information is otherwise accessible only using extralogical predicates. Moreover, the change of representation does not solve all the conceptual problems of the all-solutions predicates. The explicit manipulation of substitutions may cause inefficiency also.

Next, let us consider the semantics of concurrent logic languages. Most of the proposed formal semantics of concurrent logic languages [32][23][25][13][27][41] try to capture the possible behavior of a process. Let us take the semantics of GHC in [41] as an example.

The purpose of the semantics in [41] is to capture the abstract behavior of a process or a multiset of goals by paying attention to external communication.

Communication with a process B is modeled as a finite sequence $\langle \alpha_1, \beta_1 \rangle \langle \alpha_2, \beta_2 \rangle \ldots \langle \alpha_n, \beta_n \rangle$ of *transactions*. A (normal) transaction, denoted $\langle \alpha, \beta \rangle$, is the act of providing a process with a possibly empty input substitution α and obtaining an observable output substitution β. The first input substitution α_1 is given through the variables in var(B) (the set of variables occurring in B), which we call the *interface* of B. The corresponding output substitution β_1 is considered a response to α_1, and hence must be such that $B\alpha_1\beta_1 \not\equiv B\alpha_1$. An output substitution is also called a *partial answer substitution*. The size of a transaction depends on when the outside world observes an output substitution. The substitution β_1 need not represent atomic information, nor need it represent maximal information returned in response to α_1.

After the first transaction, B will be instantiated to $B\alpha_1\beta_1$, and the second transaction $\langle \alpha_2, \beta_2 \rangle$ will be made through the interface var($B\alpha_1\beta_1$). Our view of a process can be illustrated as follows:



The point is that the outside world may determine α_2 depending on β_1. What characterizes an interactive program is that *the input to the program may depend on the output from the program*, and we must be able to model the causality among communicated data.

The semantics of a multiset B of goals running under a program $\mathcal{P}$, denoted $[\![B]\!]_{\mathcal{P}}$, is modeled as the set of all possible finite sequences of transactions with B. Besides the modeling of behavior, our semantics is different from the semantics of ordinary logic programs in two points. First, a meaningful semantics can be given to a program that does not terminate but is still useful. Each element of $[\![B]\!]_{\mathcal{P}}$ represents a possible finite sequence of transactions with the process B which itself may be non-terminating. Second, our semantics deals with the anomalous behavior of a process such as the failure of unification and no response to an input substitution. This is necessary because we want to distinguish between a process that will always return an output substitution β_n in response to α_n and a process that sometimes does so and sometimes fails or returns nothing. Two kinds of anomalous behavior are modeled. First, providing a process with α_n may cause the failure of some unification goal, and this is denoted as $\langle \alpha_n, \top \rangle$. Second, given α_n, a process may become inactive without generating any observable output substitution. The inactivity may be caused in three ways: by reduction to an empty multiset of goals, by reduction to a multiset of goals that does not allow further reduction, and by falling into an infinite computation that does not yield any observable output substitution. These cases are denoted as

$\langle \alpha_n, \bot_{succeed} \rangle$, $\langle \alpha_n, \bot_{deadlock} \rangle$, and $\langle \alpha_n, \bot_{divergence} \rangle$, respectively, or simply as $\langle \alpha_n, \bot \rangle$ when the distinction is unnecessary. Note that the above semantics is a starting point of our semantics research. Its properties and relationship with other semantic models proposed in the context of dataflow languages, CCS, and CSP should be studied in detail.

Although the above semantics models the behavioral or operational aspects of a GHC program, it is still related to the original framework of logic programming. That is, if $\langle \alpha_1, \beta_1 \rangle \langle \alpha_2, \beta_2 \rangle \ldots \langle \alpha_n, \bot_{success} \rangle \in [\![B]\!]_{\mathcal{P}}$, then the universal closure $\forall(B\alpha_1\beta_1\alpha_2\beta_2 \ldots \alpha_n)$ is a logical consequence of the declarative reading of $\mathcal{P}$. Thus the operational semantics of GHC is sound as a theorem prover, and the declarative reading of a program provides us with the static properties of a process.

2.5 Other Aspects

This section discusses the implications of ordinary and concurrent logic languages with regard to modularity and programming. Another important aspect, performance, will be discussed in detail in Section 3.

Modularity: It is crucial for a large program to be composable from small building blocks or modules. In this respect, concurrent logic languages support process-oriented modularization with no extra cost. A process can be used as a building block of a larger process, and the output of a program can be easily directed to another process running in parallel. A problem with ordinary logic languages is that pure versions of those languages are inadequate for writing large programs. A large program may have some fragments in which a pure ordinary logic language enables elegant description, but its overall structure cannot dispense with operational notions. Pure ordinary logic languages, with the aid of negation-as-failure or modal logic, may be useful for building large databases or knowledge-bases using hierarchical and other modularization schemes. However, such databases should still be managed by a language with control.

Programming: Meta-level constructs such as all-solution predicates, `var`, `assert` and `retract` in ordinary logic languages will certainly make programming in the mundane sense easier. Without them, we would have to lower the level of programming by programming exhaustive search, by simulating unification using, say, association lists, and by maintaining databases explicitly. However, the use of meta-level constructs complicates the semantics. The situation is even worse when parallel execution is taken into account, because many of the meta-level predicates are sensitive to *how* programs are executed, while parallel execution does not guarantee the total order of primitive operations. For this reason, GHC did not inherit any extralogical features in Prolog (except for commitment, a cleaner version of cut). This decision proved to be very useful for discouraging the use of extralogical notions and encouraging better programming from the logic programming point of view.

2.6 Alternative to Concurrent Logic Languages

A problem with concurrent logic languages perceived by many researchers is that its control constructs, although semantically fundamental, have nothing to do with logic.

Then can we represent control in logic? If we are reasoning about the behavior of a concurrent system, we can describe the whole system as an *ordinary* logic program, and its execution will infer what behavior can happen. There might be various ways to represent the result, including a serialized trace and a program in an appropriate concurrent language. However, the control constructs of concurrent logic languages are for controlling the actual execution of programs. They could still be specified in logic if we could define an appropriate meta-level at which non-first-class objects (like substitutions and events) can be reified (that is, made first-class).

2.7 Summary

We have seen that ordinary and concurrent logic languages are designed for different purposes. We could argue that concurrent logic languages are at a lower level than ordinary logic languages because control is essential. However, the control of concurrent logic languages is for correctness and not for efficiency; it is for guiding computation in the correct direction. The presence or absence of control in this sense is more a matter of formalism than a matter of the level of abstraction. Control for efficiency must be considered separately, as is the case with ordinary logic programming.

Which family of languages is more suitable depends on what should be elegantly described. Ordinary logic languages will be appropriate for describing fragments of a program in which communication is not made or need not be specified. Concurrent logic languages will be appropriate for describing communication.

3. PARALLELISM FOR PERFORMANCE

This section discusses another aspect of parallelism, parallelism for the faster execution of a program.

3.1 Parallelism in Programming and Parallelism in Implementation

First, we note that parallelism in programming (concurrency) and parallelism in implementation and execution are *independent* notions. Parallelism may well be uncovered from a program not written in a concurrent language. Conversely, a concurrent language may well be used for writing a program to be run on a sequential computer if it allows natural description.

The granularity (of parallelism) of a language and the granularity of an implementation are also independent. For instance, GHC is an inherently parallel language; it is designed so that programmers cannot express unnecessary sequentiality. However, a GHC process need not always be implemented as a process in the ordinary sense. It is very important to exploit sequentiality from concurrent programs and thus to eliminate the overhead of interprocess communication and process spawning. Ueda and Furukawa [41] propose the use of program transformation for fusing communicating processes. Abstract interpretation will be useful for analyzing dataflow and compiling control.

Processes can be used for storage as well as computation, because their behavior can be history sensitive. This means

they can be used as building blocks of mutable data structures and databases that allow concurrent access. We have to develop quite different optimization techniques for processes used for storage rather than computation.

3.2 Parallelism and Algorithms

There are in general two ways to obtain good performance: parallel execution and the adoption of good algorithms. Both approaches have been studied in ordinary logic programming. For parallelism, Disz et al. [11] reported that OR-parallel execution can attain substantial speedup. For algorithms, the study started with the coroutining execution of generate-and-test programs, in which each constraint is checked passively when all its arguments have been determined. Van Hentenryck and Dincbas [17] proposed active constraint checking (called forward checking), in which constraints are used for reducing the number of possible values of uninstantiated variables. The effect of forward checking they demonstrated reminded us that we should consider algorithms before resorting to parallelism.

Then, is parallelism unnecessary for solving search problems? The answer is no. In general, a better algorithm designed for sequential execution tends to have less parallelism, because to reduce computation often requires access to nonlocal information. However, in search problems, a good algorithm may still use backtracking. In that event, we can easily attain parallel speedup by exploiting OR-parallelism, and this is actually the case in van Hentenryck's and Dincbas's method.

One way to exploit the OR-parallelism of ordinary logic programs is to write an OR-parallel implementation in a low-level concurrent language. Another way is to compile search programs into a high-level concurrent language like concurrent logic languages, as we discussed in Section 2.4. The advantages of the former approach are that better efficiency will be attained with sophisticated implementation and that any ordinary logic program can be processed. The advantages of the latter approach is that implementation is much easier and that the result of search can gracefully be passed to the subsequent stage written in the same language as the target language of the search programs.

The viability of the latter approach depends on whether exhaustive search using *good* algorithms can be compiled into efficient concurrent logic programs. Our first step was to show that the AND-sequential execution of a class of ordinary logic programs can be compiled [39], and the second step was to show that coroutining execution can be analyzed and compiled as well [40]. Both techniques compile the OR-parallelism of pure Prolog into the AND-parallelism of concurrent logic languages, and the sequential or coroutining execution of conjunctive goals into continuation processing. The essence of the techniques is to analyze and delay output unification so that multiple binding environments need not be created. The AND-parallelism in object programs is *independent* AND-parallelism, which can be most easily exploited.

Recently, we proposed a compilation technique of logic programs with finite domains that realizes forward checking [47]. The technique uses a source language similar to the one in [17]. The domain (set of possible values) of a variable, represented as a bit vector, is reduced by the active evaluation of constraints. The main task of the compilation is the derivation of a domain reducer from the constraints in a source program. First, the 'test' predicates describing constraints are partially evaluated to obtain a conjunction of primitive constraint goals. Second, primitive constraints such as equality and inequality are compiled into predicates for reducing the domain of a variable. Then, a predicate is constructed whose call reduces the domains of (some of the) variables upon determination of the value of some variable. The domain reducer obtained from the above procedure is called from the problem-independent main program in which it is checked whether the domain of any variable is reduced to an empty set or to a singleton, and whether there is a variable whose value is yet to be determined.

We have ascertained that the object program of the n-queens problem, if optimized, outperforms the object programs using our previous techniques and even the n-queens program using layered-streams [29]. The speedup fully reflected the reduction of the search tree. Advantages of our technique are that object programs obtained use no special primitives and that they have independent AND-parallelism.

Various types of constraints appear in search problems. For example, the n-queens problem has a lot of weak constraints (inequality) that make up a regular structure. The cryptoarithmetic problem has a rather small number of strong constraints (equality). We observed that the effect of forward checking can be drastic when constraints are strong but may not be so drastic when constraints are weak. For example, generalized forward checking [17] reduced the size of the search trees to 70% (8-queens) and to 58% (12-queens) compared with the trees formed with passive constraint checking. This means that parallelism is still important for this problem.

A disadvantage of our technique is that, being based on static analysis, it is not very flexible. However, it will be possible to move part of the analysis to run time without significant loss of efficiency. The use of a high-level target language makes this kind of experiment easy.

Ordinary logic programming enables concise description of OR-parallel search for *feasible* solutions. However, search problems of another kind look for the *best* (or approximately best) solutions instead of feasible solutions. In this case, the paths of a search tree should communicate so that computation is concentrated on promising paths. Processes exploring different paths must evaluate their current work occasionally and communicate the results to know if they are exploring promising paths. If they evaluate themselves and communicate too infrequently, they may explore unimportant paths for a long time. If they evaluate and communicate too frequently, they can avoid unnecessary computation but will spend too much time for evaluation and communication. This means that in parallel search with communication, the optimal frequency of communication depends on the properties of the underlying hardware.

3.3 Programming versus Uncovering Parallelism

There are two alternative ways to improve performance using parallelism: one is to program parallelism and the other

is to uncover parallelism.

It is true that parallelism can be uncovered. For example, it is quite easy to uncover the OR-parallelism of search problems written in ordinary logic languages. If parallelism is not a difficult issue and can be fully exploited by language implementors only, ordinary programmers need not be bothered with parallelism.

Systems under this hypothesis may work well on small- or middle-scale parallel computers. However, we conjecture that programming parallelism is more important in the long run. Parallelism, we feel, is too difficult to be considered by a small number of people working on specific areas of computer science. There are so many things to be considered and many people should be involved. The current practice of sequential computation owes much to many fruitful results on sequential algorithms. It seems unlikely that we can make effective use of parallel computers without accumulating good parallel algorithms for a variety of problems. A naive parallel algorithm may well be inferior to a good sequential algorithm. Furthermore, good sequential algorithms may well be hard to parallelize, because the cost of communication will not have been considered in designing them.

Concurrent logic programming systems try to let people program parallelism as easily as possible by providing them with a simple and abstract framework of parallel computation. We found that writing programs using processes is rather easy. Writing *efficient* parallel programs is not easy, but this is partly because we are inexperienced in taking the cost and the locality of communication into account. Communication is the most important aspect in designing parallel algorithms and is worth much more study. Realistic parallel computation models with which to evaluate parallel algorithms are badly needed.

4. CONCLUSIONS

Two alternatives for a parallel logic programming system, one using an ordinary logic language and the other using a concurrent logic language, have been compared from the semantics and performance points of view. There are several proposals for unifying these two families of languages; Clark and Gregory [6] propose a hybrid language approach and Haridi and Brand [15] propose a unified language called Andorra. However, we believe that the combination of these languages should be made very carefully and only when a well-defined and semantically clear interface can be defined between them. The viewpoints discussed in Section 2.4 will help graceful combination.

Current research on the parallel implementation of these families of languages seems to have different scopes. ICOT intends a concurrent logic language KL1 to be the kernel language of large-scale, non-shared-memory parallel computers in the future, and takes the approach of exposing the locality of computation and parallelism. The Gigalips project plans to implement Andorra Prolog on virtual shared-memory multiprocessors [44]; this seems to be based on the principle that locality as well as parallelism should be considered only at a very low level.

These approaches are not necessarily incompatible; individual technicalities developed could be combined in the future. The semantic gap between hardware and applications seems to be widening in pursuit of performance and functionality, making the connection of these two ends less straightforward. This means that layers of abstraction should be provided between these two ends, because a method or a technique should generally be considered and put into practice at the highest possible layer for the sake of simplicity and generality. Ordinary logic languages will serve as one of the high-level layers for applications in which parallelism can be hidden, and concurrent logic languages will serve as a lower-level layer.

ACKNOWLEDGMENTS

I am indebted to Koichi Furukawa, Ryuzo Hasegawa, Makoto Yokoo (currently at NTT), and the members of ICOT First and Fourth Research Laboratories for valuable discussions.

REFERENCES

[1] Chikayama, T., Sato, H. and Miyazaki, T. Overview of the Parallel Inference Machine Operating System (PIMOS). In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 230–251.

[2] Ciepielewski, A. and Haridi, S. A Formal Model for OR-Parallel Execution of Logic Programs. In *Proc. IFIP '83*, Elsevier Science Publishers B. V., 1983, pp. 299–305.

[3] Clark, K. L. and McCabe, F. G. IC-PROLOG—Language Features. In *Logic Programming*, Clark, K. and Tärnlund, S.-Å. (eds.), Academic Press, 1982, pp. 253–266.

[4] Clark, K. L. and Gregory, S. A Relational Language for Parallel Programming. In *Proc. ACM Conf. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 171–178.

[5] Clark, K. L. and Gregory, S. PARLOG: Parallel Programming in Logic. *ACM. Trans. Prog. Lang. Syst.*, Vol. 8, No. 1 (1986), pp. 1–49.

[6] Clark, K. L. and Gregory, S. PARLOG and PROLOG United. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 927–961.

[7] Conery, J. S. and Kibler D. F. Parallel Interpretation of Logic Programs. In *Proc. 1981 Symp. on Functional Programming Languages and Computer Architecture*, ACM, 1981, pp. 163–170.

[8] Conery, J. S. Binding Environments for Parallel Logic Programs in Non-Shared Memory Multiprocessors. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987, pp. 457–467.

[9] DeGroot, D. Restricted AND-Parallelism. In *Proc. Int. Conf. on FGCS'84*, ICOT, 1984, pp. 471–478.

[10] Dijkstra, E. W. Guarded Commands, Nondeterminacy and Formal Derivation of Programs. *Comm. ACM*, Vol. 18, No. 8 (1975), pp. 453–457.

[11] Disz, T., Lusk, E. and Overbeek, R. Experiments with OR-Parallel Logic Programs. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 576–600.

[12] van Emden, M. H. and de Lucena Filho, G. J. Predicate Logic as a Language for Parallel Programming. In *Logic*

Programming, Clark, K. L. and Tärnlund, S.-Å. (eds.), Academic Press, 1982, pp. 189–198.

[13] Gerth, R., Codish, M., Lichtenstein, Y. and Shapiro, E. Fully Abstract Denotational Semantics for Flat Concurrent Prolog. In *Proc. Third Annual Conf. on Logic in Computer Science*, IEEE, 1988, pp. 320–335.

[14] Hansson, A., Haridi, S. and Tärnlund, S.-Å. Properties of a Logic Programming Language. In *Logic Programming*, Clark, K. L. and Tärnlund, S.-Å. (eds.), Academic Press, 1982, pp. 267–280.

[15] Haridi, S. and Brand, P. Andorra Prolog: An Integration of Prolog and Committed Choice Languages. In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 745–754.

[16] Helmenegildo, M. V. Relating Goal Scheduling, Precedence, and Memory Management in AND-Parallel Execution of Logic Programs. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 556–575.

[17] van Hentenryck, P. and Dincbas, M. Forward Checking in Logic Programming. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 229–256.

[18] Hirata, M. Letter to the editor. *Sigplan Notices*, Vol. 21, No. 5 (1986), pp. 16–17.

[19] Hogger, C. J. Concurrent Logic Programming. In *Logic Programming*, Clark, K. L. and Tärnlund, S.-Å. (eds.), Academic Press, 1982, pp. 199–211.

[20] Kahn, G. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP '74*, North-Holland, 1974, pp. 471–475.

[21] Kowalski, R. Predicate Logic as Programming Language. In *Proc. IFIP '74*, North-Holland, 1974, pp. 569–574.

[22] Kowalski, R. Algorithm = Logic + Control. *Comm. ACM*, Vol. 22, No. 7 (1979), pp. 424–436.

[23] Levi, G. and Palamidessi, C. An Approach to the Declarative Semantics of Synchronization in Logic Languages. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 877–893.

[24] Lloyd, J. W. Directions for Meta-Programming. In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 609–617.

[25] Maher, M. J. Logic Semantics for a Class of Committed-Choice Programs. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 858–876.

[26] Masuzawa, H., Kumon, K., Itashiki, A., Satoh, K. and Sohma, Y. "Kabu-wake" Parallel Inference Mechanism and Its Evaluation. In *1986 Proc. FJCC*, IEEE, 1986, pp. 955–962.

[27] Murakami, M. A Declarative Semantics of Parallel Logic Programs with Perpetual Processes. In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 374–381.

[28] Naish, L. All Solutions Predicates in Prolog. In *Proc. 1985 Symp. on Logic Programming*, IEEE, 1985, pp. 73–77.

[29] Okumura, A. and Matsumoto, Y. Parallel Programming with Layered Streams. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987, pp. 224–231.

[30] Pereira, L. M. and Monteiro, L. F. The Semantics of Parallelism and Coroutining in Logic Programming. In *Mathematical Logic in Computer Science*, Dömölki, B. and Gergely, T. (eds.), North-Holland, 1981, pp. 611–657.

[31] Ringwood, G. A. Parlog86 and the Dining Logicians. *Comm. ACM*, Vol. 31, No. 1 (1988), pp. 10–25.

[32] Saraswat, V. A. Partial Correctness Semantics for CP[↓,|,&]. In *Proc. Fifth Conf. on Foundations of Software Technology and Theoretical Computer Science*, LNCS 206, Springer-Verlag, 1985, pp. 347–368.

[33] Saraswat, V. A. Concurrent Constraint Programming Languages. Ph. D. Thesis, CMU, 1989.

[34] Shapiro, E. Y. A Subset of Concurrent Prolog and Its Interpreter. Tech. Report TR-003, ICOT, 1983.

[35] Shapiro, E. Y. Concurrent Prolog: A Progress Report. *Computer*, Vol. 19, No. 8 (1986), pp. 44–58.

[36] Takeuchi, A. and Furukawa, K. Parallel Logic Programming Languages. In *Proc. Third Int. Conf. on Logic Programming*, LNCS 225, Springer-Verlag, 1986, pp. 242–254.

[37] Ueda, K. Guarded Horn Clauses. In *Logic Programming '85*, LNCS 221, Springer-Verlag, 1986, pp. 168-179.

[38] Ueda, K. Guarded Horn Clauses: A Parallel Logic Programming Language with the Concept of a Guard. In *Programming of Future Generation Computers*, Nivat, M. and Fuchi, K. (eds.), North-Holland, 1988, pp. 441–456.

[39] Ueda, K. Making Exhaustive Search Programs Deterministic. *New Generation Computing*, Vol. 5, No. 1 (1987), pp. 29–44.

[40] Ueda, K. Making Exhaustive Search Programs Deterministic, Part II. In *Proc. 4th Int. Conf. on Logic Programming*, MIT Press, 1987, pp. 356–375.

[41] Ueda, K. and Furukawa, K. Transformation Rules for GHC Programs. In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 582–591.

[42] Warren, D. S. Database Updates in Pure Prolog. In *Proc. Int. Conf. on FGCS'84*, ICOT, 1984, pp. 244–253.

[43] Warren, D. H. D. The SRI Model for Or-Parallel Execution of Prolog—Abstract Design and Implementation Issues. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987, pp. 92–102.

[44] Warren, D. H. D. and Haridi, S. The Data Diffusion Machine—A Scalable Shared Virtual Memory Multiprocessor. In *Proc. Int. Conf. on FGCS'88*, ICOT, 1988, pp. 943–952.

[45] Westphal, H., Robert, P., Chassin, J. and Syre, J.-C. The PEPSys Model: Combining Backtracking, AND- and OR-parallelism. In *Proc. 1987 Symp. on Logic Programming*, IEEE, 1987, pp. 436–448.

[46] Yang, R. and Aiso, H. P-Prolog: A Parallel Logic Language Based on Exclusive Relation. In *Proc. Third Int. Conf. on Logic Programming*, LNCS 225, Springer-Verlag, 1986, pp. 255–269.

[47] Yokoo, M. and Ueda, K. Solving Constraint Satisfaction Problems in Concurrent Logic Programming. ICOT Tech. Memorandum, ICOT, 1988 (in Japanese).