POTENTIALITY OF PARALLELISM IN LOGIC

Franz Kurfeß*

TR-91-055

Abstract

The processing of knowledge is becoming a major area of applications for computer systems. In contrast to data processing, the current stronghold of computer use, where well-structured data are manipulated through well-defined algorithms, the treatment of knowledge requires more intricate representation schemes as well as refined methods to manipulate the represented information. Among the many candidates proposed for representing and processing knowledge, logic has a number of important advantages, although it also suffers from some drawbacks. One of the advantages is the availability of a strong formal background with a large assortment of techniques for dealing with the representation and processing of knowledge. A considerable disadvantage so far is the amount and complexity of computation required to perform even simple tasks in the area of logic. One promising approach to overcome this problem is the use of parallel processing techniques, enabling an ensemble of processing elements to cooperate in the solution of a problem. The goal of this paper is to investigate the combination of parallelism and logic.

*Preprint from: Wrightson, G and Fronhöfer, B.; Parallelization in Inference Systems. Springer Verlag
1 Parallelism in Inference Systems

The best-known instance of an inference mechanism in computer science is the use of PROLOG as logic programming language. As a consequence, most approaches to combine parallelism and inference systems are based on PROLOG or a derivative thereof as language, and evaluation mechanisms based on the resolution calculus. Among these approaches, OR-parallelism and AND-parallelism are almost exclusively used, either isolated or a combination of both. In the following, we will show that there is quite a variety of other categories of parallelism, sometimes with distinct advantages over AND-/OR-parallelism. Our investigation will concentrate on Horn clause logic and first order predicate logic, and not include extensions or specializations like temporal logic, modal logic, higher order logic, fuzzy logic, probabilistic reasoning, etc.

For most of our investigations, we will also assume that a general-purpose parallel architecture will be used as execution vehicle. Such an architecture may provide inadequate support for certain basic operations in the evaluation of logic programs (selection of clause heads, unification), but experience with dedicated machines leads to the conclusion that the gain in performance in general-purpose architectures is overwhelming compared to special-purpose ones. We will not restrict ourselves, however, to a certain processing mode, architecture or topology; in particular, we want to overcome the severe implications imposed by the inherently sequential execution mode of a stack-based Warren Abstract Machine WAM [Warren, 1983] and resolution as underlying calculus. This will give us the freedom to investigate different computational models, especially with respect to their suitability for parallel evaluation.

2 Categories of Parallelism

This section gives an overview of various categories of parallelism which can be identified in inference systems. From a conceptual point of view, parallelism can be introduced by two ways: first by identifying independent parts in the program to be evaluated, and executing these parts separately. It is also possible, however, to view the program to be evaluated as data, which are transformed by certain operations according to a particular inference mechanism, and apply some of these operations in parallel to the whole, or parts of the original program.

Table 1 shows an overview of the categories of parallelism, arranged according to the granularity and the components of a logic program. It identifies the particular data structures and operations applied in a category.

The notation used in Table 1 is based on viewing a logic program as a collection of clauses, possibly organized into modules (or objects). The clauses consist of literals, arranged as head and tail. A literal is identified through a predicate, and may have arbitrarily complex terms composed from functions, constants, and variables. Further notational details depending on the particular category of parallelism will be given at the appropriate place.
<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>$P$</td>
<td>logical program or formula</td>
</tr>
<tr>
<td>${P_1, \ldots, P_m}$</td>
<td>distinct programs</td>
</tr>
<tr>
<td>$\mathcal{P} = {M_1, \ldots, M_m}$</td>
<td>program composed of modules</td>
</tr>
<tr>
<td>$\mathcal{P} = {\mathcal{P}_1, \ldots, \mathcal{P}_s}$</td>
<td>spanning sets in a program</td>
</tr>
<tr>
<td>$\bar{P}$</td>
<td>modified program</td>
</tr>
<tr>
<td>$C$</td>
<td>clause</td>
</tr>
<tr>
<td>$L$</td>
<td>literal</td>
</tr>
<tr>
<td>${R_1, \ldots, R_r}$</td>
<td>routes in a program</td>
</tr>
<tr>
<td>$t = {st_1, \ldots, st_s}$,</td>
<td>term, composed of subterms</td>
</tr>
<tr>
<td>$(t, t')$</td>
<td>pair of terms to be unified</td>
</tr>
<tr>
<td>${f^1, f^2, f^3, \ldots}$</td>
<td>occurrences of one function</td>
</tr>
<tr>
<td>$f(t_1, \ldots, t_m)$,</td>
<td>function with arbitrary terms as arguments</td>
</tr>
<tr>
<td>$(g_1, \ldots, g_m)$,</td>
<td>ground terms</td>
</tr>
<tr>
<td>${\text{INF}_1, \ldots, \text{INF}_n}$</td>
<td>different inference mechanisms</td>
</tr>
<tr>
<td>$\text{INF}$</td>
<td>modified inference mechanism</td>
</tr>
<tr>
<td>${\text{RED}_1, \ldots, \text{RED}_r}$</td>
<td>reduction transformations</td>
</tr>
<tr>
<td>${\text{UNIF}_1, \ldots, \text{UNIF}_r}$</td>
<td>unification mechanisms</td>
</tr>
</tbody>
</table>

**Multitasking** Independent programs can be evaluated at the same time. This concept is well-known from traditional operating systems and does not involve major problems, at least from our point of view. Difficulties which may arise are only on the operating system level, e.g. through the utilization of shared resources, load balancing, built-in predicates, and should be solved on that level without intervention from the inference mechanism or the user. Multitasking may be of interest for the evaluation of logic programs from different users on a shared, large multiprocessor machine.

**Competition** The evaluation of a logic program can be done in a number of different ways, usually described in the form of a calculus. The idea of competition is to apply different evaluation mechanisms to one and the same program. Whereas in principle this results in redundant computation, different calculi are well suited for different classes of programs [Ertel, 1990, Ertel, 1991] (see also the contribution of W. Ertel in this volume). Often it is not even necessary to use different calculi: changing the strategy to traverse the search space can make a tremendous difference and is often the source of claims to superlinear speedup in the parallel evaluation of logic programs. To illustrate this point, consider the way PROLOG builds the proof tree for a program, with its depth-first, left-to-right strategy. In the worst case, it gets stuck in an infinite branch in the left part of the proof tree, whereas a solution would be found easily on the other side of the tree. Changing the search strategy, either from depth-first to breadth-first or from left-to-right to right-to-left, would lead
<table>
<thead>
<tr>
<th>Level</th>
<th>Data Structures</th>
<th>Operations</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Formula</td>
<td>( { \mathcal{P}_1, \ldots, \mathcal{P}_m } )</td>
<td>INF</td>
<td>Multitasking</td>
</tr>
<tr>
<td>( \mathcal{P} )</td>
<td>{INF, \ldots, INF_n}</td>
<td>Competition</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{P} )</td>
<td>{INF, \ldots, INF_n}</td>
<td>Cooperation</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{P} )</td>
<td>{INF_{Pr1}, \ldots, INF_{Prn}}</td>
<td>Precision</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{P} = { \mathcal{M}_1, \ldots, \mathcal{M}_m } )</td>
<td>INF</td>
<td>Modularity</td>
<td></td>
</tr>
<tr>
<td>( \tilde{\mathcal{P}} )</td>
<td>{RED_1, \ldots, RED_r}</td>
<td>Reductions</td>
<td></td>
</tr>
<tr>
<td>( \mathcal{P} = { \mathcal{S}\mathcal{P}_1, \ldots, \mathcal{S}\mathcal{P}_s } )</td>
<td>INF_{ConnectionMethod}</td>
<td>Recursion</td>
<td></td>
</tr>
<tr>
<td>Clause</td>
<td>{\mathcal{L}, \mathcal{C}_1, \ldots, \mathcal{C}_s}</td>
<td>INF_{Resolution}</td>
<td>OR-Parallelism</td>
</tr>
<tr>
<td>Literal</td>
<td>{\mathcal{R}_1, \ldots, \mathcal{R}_r}</td>
<td>UNIF</td>
<td>Routes</td>
</tr>
<tr>
<td>{\mathcal{L}_1, \ldots, \mathcal{L}_n; \mathcal{C}_1, \ldots, \mathcal{C}_n}</td>
<td>Resolution Steps</td>
<td>AND-Parallelism</td>
<td></td>
</tr>
<tr>
<td>{\mathcal{L}_1, \ldots, \mathcal{L}_n; \mathcal{C}_1, \ldots, \mathcal{C}_n}</td>
<td>Pipelined Resolution Steps</td>
<td>Pipelining-Parallelism</td>
<td></td>
</tr>
<tr>
<td>Term</td>
<td>{(t_1, t'_1), \ldots, (t_n, t'_n)}</td>
<td>UNIF</td>
<td>Term Parallelism</td>
</tr>
<tr>
<td>( (t_1, t'_1) )</td>
<td>{UNIF, \ldots, UNIF}</td>
<td>Unification</td>
<td></td>
</tr>
<tr>
<td>( (t_1, t'_1) )</td>
<td>{UNIF, \ldots, UNIF}</td>
<td>Unification</td>
<td></td>
</tr>
<tr>
<td>( (t_1, t'_1) )</td>
<td>Cycles</td>
<td>Occur Check</td>
<td></td>
</tr>
<tr>
<td>( t = {st_1, \ldots, st_a}, )</td>
<td>Unification</td>
<td>Separability</td>
<td></td>
</tr>
<tr>
<td>( t' = {st'_1, \ldots, st'_b} )</td>
<td>Function Evaluation</td>
<td>Function Call</td>
<td></td>
</tr>
<tr>
<td>{f^1, f^2, f^3, \ldots}</td>
<td>Decomposition of</td>
<td>Forwarding</td>
<td></td>
</tr>
<tr>
<td>( f(t_1, \ldots, t_m), )</td>
<td>Unification</td>
<td>Bottom Up</td>
<td></td>
</tr>
<tr>
<td>( f(t'_1, \ldots, t'_m) )</td>
<td>Composition of</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( (g_1, \ldots, g_m), )</td>
<td>Unification</td>
<td></td>
<td></td>
</tr>
<tr>
<td>( (g'_1, \ldots, g'_m) )</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Atom</td>
<td>arrays</td>
<td>SIMD Operation</td>
<td>Data Parallelism</td>
</tr>
<tr>
<td></td>
<td>lists</td>
<td>Incremental</td>
<td>Streams</td>
</tr>
<tr>
<td></td>
<td>e.g. feature maps</td>
<td>Evaluation</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>e.g. Search for Global Minimum</td>
<td>Subsymbolic</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Representation</td>
</tr>
</tbody>
</table>

Table 1: Categories of parallelism in logic
to a rapid detection of the solution. On the other hand, PROLOG programs written
by experienced PROLOG programmers usually rely on knowledge about the under-
lying execution mechanism, and changing it may have disastrous results. The use
of competition is better suited for areas like automated theorem proving where the
formulation of a program is derived from the specification of the problem to be solved
and not so much through the description of the solution process with logical means.

**Cooperation** The cooperation category is based on the exchange of useful infor-
mation between different inference mechanisms working on the same program, and
thus can be viewed as a counterpart to the previously described competition approach
[Fronhöfer and Kurfeß, 1987]. Useful information can consist of intermediate results,
e.g. in the form of lemmata, or a partitioning of the search space into sections treated
by the different mechanisms, or meta-level information, e.g. the (estimated) probabil-
ity that a proof can be found in a certain part of the search space. There is certainly
a large potential of useful information to be exchanged, but also a tradeoff between
the amount of data to be transferred as well as the overhead to determine which
information is worth while transferring, and the gain that arises from the availability
of these data.

**Precision** Whereas in the competition approach different inference mechanisms are
employed, precision relies on the use of basically the same inference mechanism, but
with various degrees of precision through changes in the unification procedure. The
range of precision is from the propositional “skeleton” of the formula (no unifica-
tion) over weak unification (substitutions are only computed locally) and unification
without occur check (cyclic substitutions can occur) to predicate logic with full unifi-
cation. Inference mechanisms with different precision operate fully in parallel or in a
pipelined fashion, where the ones with low precision proceed faster, thus eliminating
dead branches earlier and reducing the remaining search space for the more precise
mechanisms.

**Modularity** A widely accepted methodology for the development of large software
systems is to structure the overall system into largely independent subsystems with
clearly defined interfaces. This approach obviously is also relevant for inference sys-
tems and logic programs, and emerges in three variants. First, it is possible not to
impose any additional syntactical structuring on large programs, but to rely on the
inherent structuring as result of the particular problem together with the style of the
particular programmer. The resulting program is analysed, and independent parts
are identified. The second approach is to explicitly characterize modules in the source
code, which eliminates the potentially costly analysis of the program. A further step
is to integrate object-oriented concepts into the logic programming paradigm.

The common theme for these approaches is to group the program into parts which
have few interactions among each other, and thus can be evaluated more or less
independently. Restrictions in the parallel evaluation, however, can still be imposed through time or data dependencies.

**Reductions** An analysis of the program to be evaluated is also a central point for reductions. The goal here, however, is to eliminate redundant or unnecessary code, thus restricting the search space to be traversed. The application of reductions must maintain important properties of the program, and they are usually grouped into equivalence-preserving and satisfiability-preserving transformations. Some reductions operate locally on parts of the program (single clauses, pairs or sets of clauses), whereas others require information about the program as a whole. While reductions may change the text of the program, they can be applied in parallel since the transformations do not change the important properties of the program.

**Recursion** The use of recursion on one hand can be used to write simpler programs, but on the other hand is difficult for parallelization because the unrolling of recursive clauses typically is done dynamically at execution time. This is due to the fact that the number of iterations can only be determined from the actual goal together with the program. Once that goal is available, however, simple calculations often can be made to determine or estimate the number of iterations. Then in some cases it is possible to apply conventional parallelization techniques to transform the sequence of operations in the loop into operations executed in parallel [Millroth, 1991]. The potential gain from this technique is especially large if it can be used to predetermine the structure of the AND/OR-tree, and thus reduce the complexity of managing of the binding environments.

**Spanning Sets** One important advantage of the spanning set concept as well is to make the run-time management of bindings simpler. This is based on a statiscal analysis of the program with the goal of identifying parts of the program (spanning sets of connections) which represent alternative solutions [Bibel, 1987, Wang, 1989, Ibañez, 1988, Ibañez, 1989, Kurfeß, 1990] (see also the contribution of Wang, Marsh and Lavington in this volume). These solutions then can be computed completely independent of each other, without the necessity of maintaining complex run-time environments for variable bindings. The limitation of this concept lies in the treatment of recursive parts, because these cannot in general be expanded statically. A combination of spanning sets with the techniques for unrolling recursion looks quite promising at least for certain simple kinds of recursion.

**OR-Parallelism** Spanning sets and OR-parallelism exploit the same feature, namely the computation of alternative solutions, but with different techniques. Whereas spanning sets use statiscal analysis, OR-parallelism usually invokes new threads of computation dynamically whenever there are multiple clauses for resolving a subgoal. This technique is a straightforward extension of the resolution mechanism and
can be incorporated into existing PROLOG implementations without major problems. The disadvantage, however, is the management of the variable bindings for different threads of computation, which must be open to backtracking in the case that a candidate for a solution turns out to be invalid.

Routes The routes concept again relies on a statical analysis of the program, identifying sequences of connections (or resolution steps) which will have to be followed through in the evaluation, and which contain OR-parallel connections only at the end points [Ibañez, 1988, Ibañez, 1989, Kurfeß, 1990]. The crucial point with respect to parallelism is that these routes do not necessarily have to be treated in a sequential way; they may as well be combined pairwise with logarithmic execution time instead of linear. The problem here is the same as with the spanning sets: the appearance of recursive parts requires dynamical treatment, but again a combination with the unrolling of recursion can improve the situation.

AND-Parallelism This category is widely used in parallel logic programming, and relies on a concurrent evaluation of the literals in the clause body (subgoals). Problems occur if variables are shared between literals, since they must assume identical values [DeGroot, 1984, DeGroot, 1988]. On the other hand, shared variables are a convenient way to express synchronization between different threads of computation, which makes languages based on AND-parallelism also well suited for low-level tasks such as systems programming. Especially when OR- and AND-parallelism are combined in one execution model, the management of variable bindings can get very complicated, so that sometimes the parallel evaluation is restricted to independent subgoals, i.e. subgoals which do not share variables.

Pipelining-Parallelism Instead of evaluating the subgoals in a clause fully in parallel, they can be processed in a pipelined fashion [Beer and Giloi, 1987, Beer, 1989]. The advantage is the full compatibility with the standard PROLOG evaluation mechanism, which proceeds from left to right in the body of a clause. The performance gain achieved through this scheme (which relies on dedicated hardware), however, is very limited, and is easily outperformed by the performance progress in general-purpose architectures.

Term Parallelism The task of evaluating a logic program can conceptually be divided into two parts: one is to navigate through the search space as indicated by the relations between the predicates in the program, the other to guarantee that the variable bindings made during the traversal are consistent. The second is based on unification as underlying operation for determining the variable bindings in order to make two terms equal. In analogy to the simultaneous application of inference operations to distinct logic programs, unification can be performed simultaneously on different term pairs (or, more generally, sets of terms). Parallelism on the term level
underlies two important restrictions: first, the task of unification can be inherently sequential in the worst case [Yasuura, 1984, Dwork et al., 1988]; second, the grain size for unification tasks tends to be rather small with a PROLOG-based evaluation model. The first restriction is of a fundamental nature, but depends to a large degree on the formulation and representation of the problem to be solved. The second restriction can be overcome by choosing a different evaluation mechanism [Bibel, 1987, Wang, 1989, Ibañez, 1988, Ibañez, 1989, Kurfeß, 1990].

**Unification Competition** Conceptually similar to competition in the inference mechanism, competition can be used for unification by applying different unification mechanisms to one and the same term pair. It is questionable, however, if this kind of parallelism is very useful since the variety of unification mechanisms is not too large, and performance differences largely can be attributed to the term size, and not so much to the internal structure of the terms [Corbin and Bidoit, 1983].

**Unification Cooperation** The use of cooperation during unification has a close affinity to unification competition described above: it seems feasible, but with current techniques probably does not result in substantial advantages.

**Occur Check** The task of the occur check is to identify situations where a term is substituted by a subterm of itself, resulting in cyclic substitutions [Plaisted, 1984]. Whereas it is possible to define a semantics which allows the occurrence of infinite substitutions [Colmerauer, 1982], or in many situations to just neglect potential problems and omit an occur check (as in many PROLOG implementations), there are cases where cyclic substitutions should not be accepted. The occur check is basically a search of a graph for cycles, an operation which can be implemented independent of unification per se. It is not possible to perform these two operations completely in parallel since the structure of the terms involved is changed during unification, and cyclic substitutions may be introduced through unification. At best unification and occur check can be performed in a pipelined fashion, where the occur check is performed incrementally on the section of the terms already unified [Hager and Moser, 1989].

**Separability** This category corresponds to modularity on the program level by identifying sections in the terms to be unified which are more or less independent. It is probably only useful to exploit this kind of parallelism if there is already a partitioning induced by the structure of the program or its representation, since the cost of performing such an analysis is not much smaller than actually performing unification.

**Function Call** A considerable amount of work during the unification process can consist of evaluating functions; this work may be done independent of the rest of unification, possibly even at compile time. The basic idea is the to try to evaluate
all or some of the function calls in parallel to the rest of unification, replacing the calls by the computed results. To some degree this parallelism is speculative since work may be done which actually is not needed; the advantage, however, is that it can be used to occupy idling processing elements, for example in the initial phase of unification when only a few processing elements are working close to the roots of the terms.

**Forwarding** The process of unifying two terms can be viewed as assignment or comparison for atomic components (variable or constant symbols), and propagating the task of unification to the corresponding subterms. Forwarding denotes the propagation of unification to subterms in parallel. A problem arises for multiple occurrences of one and the same symbol (or whole subterm), which must have the same value. It can be solved by a suitable representation, for example as dags (directed acyclic graph), where multiple occurrences of substructures are represented only once, but with several incoming pointers.

**Bottom Up** In this case, activation starts from the constants and instantiated variables in order to reduce the complexity of the terms to be unified by replacing function calls with their results and instantiating variables where possible. This is again a case of speculative parallelism since the correlation between substructures from the left and right term can only be determined in a top-down way.

**Data Parallelism** Data structures in logic programs are usually characterized through terms, which may have a highly irregular internal structure. In many practical cases, however, regular data structures are used, the prototype being an array of elements of some type. Such regular data structures open the door to another category of parallelism, usually referred to as data parallelism [Hillis and Steele, 1986]. Here, one and the same operation is applied to all or a subset of the single elements in a SIMD (Single Instruction Multiple Data) way. Whereas typical logic programs may not contain many data structures where data parallelism can be applied, there certainly are many applications with highly regular data structures which can profit from this combination, such as deductive data bases, image processing, or scientific computations [Fagin, 1991, Succi and Marino, 1991].

**Streams** The concept of streams is based on an incremental evaluation of large data structures, e.g. lists which change dynamically [Ito et al., 1987, Takeuchi et al., 1987]. Usually operations involving the whole data structure, like comparing two lists, are only applicable if and when the whole data structures are available. In some cases, operations can be applied while these data structures still evolve, like appending elements to the end of the list. This category may be on the borderline of parallelism, but eases some synchronization constraints in the parallel evaluation of programs.
Subsymbolic Representations  All the categories described so far implicitly rely on a symbolic representation, where an atomic part of the program has a direct correspondence to the internal representation in the execution vehicle, typically one or more memory cells. The most important operations here are assignment, where the value of a symbol is set to the result of an operation, and pointer manipulation, where references between symbols are established or changed. An important aspect of this representation scheme is that applied to symbols, or at least to substantial parts of symbols. It is conceivable – and considerable interest has been devoted to this recently – to construct inference mechanisms based on subsymbolic representations, where there is no direct correlation between symbols and machine-internal representations [Touretzky and Hinton, 1985, Touretzky and Hinton, 1988, Ballard, 1986, Smolensky, 1987, Lange and Dyer, 1989, Shastri, 1988, Shastri and Ajjanagadde, 1989, Pinkas, 1990]. One approach is to represent symbols and programs as patterns of activation and interconnection distributed a network of simple units, which are connected through weighted links [Hölldobler, 1990a, Hölldobler and Kurfeß, 1991]. The computation then is performed by a spreading activation scheme which settles into a stable state when a solution is found. This category opens up a whole new dimension of parallel evaluation due to its different representation and computation paradigm.

3 Exploitation of Parallelism

The identification of sources for parallelism in inference systems is an interesting topic for itself, but in order to be practically applicable must be accompanied by an investigation of which kinds of parallelism are worth while to be exploited. This becomes rather complicated, especially when one tries to combine different categories of parallelism in one evaluation mechanism.

3.1 Analysis of Parallelism

An important method for the exploitation of parallelism in logic is the analysis of programs, for example by determining the potential parallel factor [Harland and Jaffar, 1987] or more complex measures. This can be done in a general investigation, analysing many logic programs with the aim of identifying common, useful ways to exploit parallelism for (classes of) logic programs [Onai and et al., 1984, Delcher and Kasif, 1989, Debray, 1989, Debray et al., 1990]. The same can be done for individual programs through a static analysis at compile time, or through symbolic evaluation to capture also some dynamic aspects of the program execution. To go even further, one can perform sample executions of a program with a set of typical goals, thus gaining information for the dynamic behavior of the program in real use.

\[^1\]see also this volume
3.2 Control of Parallelism

The analysis of programs before the actual execution can provide important information about the expected dynamic behavior, but will in many cases not be sufficient to completely determine the actual execution pattern. The control of parallelism at run-time can be achieved on three different levels: through specifications provided by the user in the program itself, through strategies determined by the compiler, and through measures of the operating system. A problem here is to find a balance between two conflicting goals: high performance, and easy program development. Highest performance can be achieved by giving the user direct access to operating system features, e.g. for communication, synchronization, load balancing, I/O, etc. The price to be paid is that in this case the user needs an intimate knowledge of the execution mechanism, the operating system, and the underlying hardware architecture. The other extrem is to relieve the user form all these low-level tasks, only requiring a formal problem specification based on logic. This, however, may result in disappointingly poor performance, at least as long as the development in terms of automated analysis and control mechanisms is not developed very far.

A partial solution to this dilemma might lie in the use of control abstraction, which follows a strategy similar to data abstraction [Crowl and LeBlanc, 1991]. The basic idea is to provide a safe way to introduce explicit, user-definable control constructs for parallelism. The introduction of these constructs is separated into two parts: one describes the desired properties in a formal definition, the other provides the actual implementation, based both on the formal definition as well as on features of the machine used for execution. Whereas this method still requires some effort from the user’s side in order to achieve high performance, it allows the definition of problem-specific control constructs based on efficiently implemented control primitives provided by the operating system. Another advantage is better portability, since the architecture-dependent features are concentrated in one place, namely the particular implementation of the construct.

This method fits nicely into the logic framework e.g. through modules (or objects) and meta-evaluation, but requires some higher-order logic features.

3.3 Restrictions

While the abstract potential of parallelism in logic is quite high, its practical exploitation is subject to a number of restrictions. Some of these are under a limited control from the user, like the problem structure, specification and representation, as well as the actual encoding in the form of a program. Others are implied through the underlying evaluation mechanism, the runtime environment (shared binding environments), operating system, and hardware architecture.
4 Conclusions

The goal of this paper has been to illuminate the potentiality of parallelism in logic programming and inference systems. For this purpose, we investigated different categories of parallelism derived from the evaluation of predicate logic program, without having a particular calculus, abstract machine, or underlying hardware in mind. The outcome is a variety of different categories, far beyond the AND-/OR-parallelism usually found in attempts to parallelize PROLOG. Admittedly some of these categories are not very relevant in combination with present technology. Others, however, in particular the ones based on static program analysis in combination with programming methodologies aiming at easier program development while maintaining the possibility to finetuning for high performance, have a very good potential for the exploitation of parallelism in inference systems.

References


22


