Parallelism in Logic Programming

Cláudio F.R. Geyer
Patrícia Kayser Vargas
II-UFRGS
{geyer,kayser}@inf.ufrgs.br

Inês de Castro Dutra
COPPE-UFRJ
ines@cos.ufrj.br

Abstract

This text summarises the main research work being carried out on the implementation of parallel logic programming systems. It concentrates on describing techniques for exploiting and-parallelism and or-parallelism while showing important aspects of some systems on shared-memory, distributed-memory and distributed-shared memory architectures. It also presents some important pointers to journals, conferences and sites with related information.

1 Introduction

Parallel processing is a very important alternative to obtain good performance from programs, be them applications or basic software. Parallelism can be exploited implicitly or explicitly. Most well-known parallel systems are based on the *imperative* programming model and parallelism is exploited explicitly, making the programming task hard. Contrasting with imperative programming, *declarative* programming presents a higher-level programming model where the programmer needs to worry on **what** to solve and not on **how** to solve a problem. Because of this declarative characteristic, logic programming offers some opportunities to exploit implicit parallelism. This clearly reduces the cost of software development, as the user writes only one application that will run on one processor just as well as in several processors.

While logic programming provides a higher-level programming paradigm that releases the programmer from "programming in parallel", systems that exploit parallelism from logic programs are usually written "in parallel" using an imperative paradigm in order to obtain maximum performance from existing parallel architectures.

Our text addresses the main issues of parallel and distributed logic programming concentrating on techniques to **efficiently** implement parallel logic programming systems on several parallel architectures. This text has the following structure: section 2 briefly introduces logic programming concepts. Section 3 presents main aspects of parallel logic programming concerning mainly the implicit control parallelism. Section 4 presents some scheduling issues. Last section draws our conclusions and presents open problems for future work.

2 Logic Programming

Logic programming [60] has its basis in studies of formal logic mathematics, in particular, Predicate Calculus. Therefore all its basic constructs are inherited from logic. A logic program is a set of axioms, or rules, defining relations between objects. A computation of a logic program is a deduction of consequences, which is its meaning.

Kowalski [61] defines algorithm through the equation *Algorithm = Logic + Control*. In this context, many authors consider that a programmer can concentrate in Logic when using logic programming since the control is implicit in the execution environment. Because the programmers can concentrate more on what to solve and not how to solve a problem, logic programming languages are called *declarative languages*.

Logic programming has been largely used in many research centers as an alternative to the imperative programming paradigm. Logic programming is suitable to the development of many applications such as natural language processing, theorem proving, databases, expert systems and symbolic problems in general. The Journal of Logic Programming has one entire volume dedicated to applications developed using logic programming languages such as intelligent information management and electronic CAD [1]. Besides there are conferences dedicated to practical applications such as PACLP (The Practical Application of Constraint Technologies and Logic Programming).

There are several logic programming languages but Prolog [21, 89, 7] was the first practical logic programming language implemented and it is still the most widely known and used. Prolog was created in the early 1970s by Alain Colmerauer, Robert Kowalski, Phillipe Roussel and colleagues. Modern implementation began with a virtual machine called the *Warren Abstract Machine* [101, 3] implemented by David H. D. Warren at Edinburgh, which defines a stack-based abstract architecture. Since initial Prolog interpreters were not optimised, a myth of "logic programming = slow implementations" developed and still survives [73]. Only when Prolog compilers appeared, this myth started declining. Nowadays there are extremely fast implementations available [95] [6]. Almost all compiled Prolog implementations use the (*WAM*) with some extensions and/or optimisations.

The Prolog language is now defined by an ISO (1995) Standard [31]. However, much current theoretical value comes from important extensions, modifications and experimental aspects [82]. These include: Constraint Logic Programming (CLP), Parallelism and concurrency, Linear Logic Programming(LLP), Inductive logic Programming (ILP), Committed Choice Languages, Higher-Order Logic (HOL), and Object-oriented Logic Programming (OOLP).

In next paragraphs, basic terminology employed in this tutorial will be presented. For a more detailed discussion on the Prolog syntax and its origin, the reader is reported to the following references [23, 21, 89, 7].

The data structures manipulated in Prolog are first order *terms*. A term can be either a *variable*, started with an upper case letter, a *constant*, started with a lower case letter, or a *structure* in the format f1(t_1, ..., t_n) where f is a *functor* and t_i is a term.

Variables in Prolog have some important features, i.e., variables: (1) are logical variables which can be instantiated only once; (2) are untyped until instantiation; and (3) are instantiated via *unification*, a pattern matching operation finding the most general common instance of two data objects.

Logic Programming considers that each *clause* expresses a number of (maybe zero) conditions that imply a number of (maybe zero) conclusions. The set of clauses with at most one conclusion are called *Horn clauses*. Horn clauses are the basis of many logic languages including Prolog. In Prolog, the conditions of a clause are called the *body* of the clause. The conclusion is called the *head* of the clause.

Prolog programs are sets of Horn clauses (assertives). A clause can be either a fact or a rule. *Facts* are simple and unconditional, i.e., they express a tautology. Syntactically, a fact can be considered a clause without a body. *Rules* express conditional relations. All rules present head and body separated by the symbol ":-". The head has one single term representing a possible conclusion and the body is composed by a set of terms also called *literals* or *goals*. Each goal must be successfully proved in order that all clause succeed. *Queries* are special rules without head that are always executed.

Figure 1 presents a small Prolog code composed by facts, rules and queries. In this example, the predicate *grandfather* has two rules to indicate *what* means to be a grandfather "Someone is grandfather of X **if** he is father of one of the parents of X". Both rules explain also *how* to find out if someone is grandfather of another one: " X is grandfather of Y **if** X is father of Z **and** Z is father of Y" (first clause) **or** " X is grandfather of Y **if** X is father of Z **and** Z is mother of Y" (second clause).

```
father(paulo, joao).
father(paulo, jose).
father(paulo, ana).           facts
father(jose, maria).
father(ana, lucas).

grandfather(X,Y) :-
      father(X,Z),
      father(Z,Y).            rules
grandfather(X,Y) :-
      father(X,Z),
      mother(Z,Y).

:- grandfather(X, maria).     query
```

Figure 1: Example of Prolog code

Thus a logic program can have a *declarative* meaning ("*what*") and a procedural meaning ("*how*"). The declarative interpretation reads the clauses statically, textually and understands the definitions corresponding to the rules and facts. The procedural interpretation tries to understand how the logic program produces a solution.

The procedural interpretation of Horn clauses forms the basis of the operational semantics of logic programming. Indeed, the set of all clauses with the same head predicate symbol **p** and with the same number **n** of arguments (*arity*) can be considered as the definition of the *procedure* **p** frequently called procedure **p/n**. Each goal in the body of a clause can thus be considered as a procedure call. At a given step of the execution of a logic program, the set of goals that remain to be executed, i.e. the continuation of the computation, is called the current *resolvent*. The execution of a logic program starts by taking the query as initial resolvent and proceeds by transforming the current resolvent into a new one as follows [57]. First select any goal G of the resolvent, and apply a *resolution step*, which consists in finding a clause of the program whose head unifies with G1, then replacing G by the body of the unifying clause to produce a new resolvent. The *bindings* of variables resulting from the unification apply to the whole resolvent. This process iterates until either the resolvent is empty, in which case the computation ends with success, or no unifying clause can be found during a resolution step, in which case a *failure* occurs and backtracking takes place. *Backtracking* consists in restoring the previous resolvent and selecting an alternative unifying clause in order to perform an alternative resolution step, i.e. at backtracking Prolog investigates the different alternatives in the order in which they are listed in the program. If all possible alternatives have been tried unsuccessfully and it is not possible to backtrack further, the computation terminates with failure. Figure 2 illustrates the execution of the previous example through the *search tree* until the first solution is found. Notes that a special structure called *choice-point* or *or-node* is created for every call to a goal that has more than one candidate clause. This structure stores and controls still available alternatives.

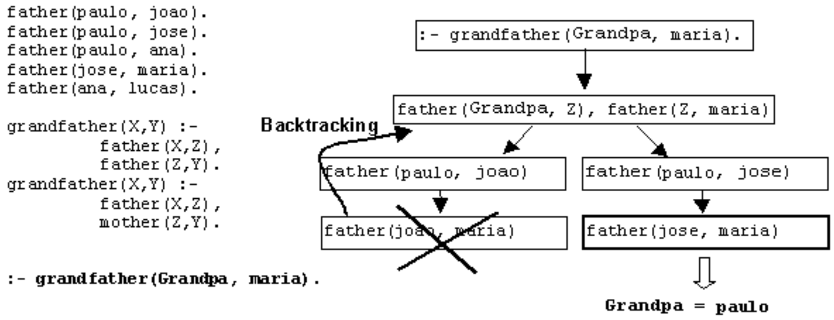

Figure 2: Example of Prolog execution

The Prolog sequential computation strategy consists in a depth-first left-to-right search of the program tree. This execution mechanism is responsibility of the execution environment and the programmer does not need to worry about this. One important concept used in this tutorial is *Prolog semantics* [73] whose meaning is closely related to the execution strategy. Prolog extends pure Horn clause logic with some additional features whose the most relevant are:

- *Meta-logical predicates*: are used to manipulate terms and to query the state of the proof (e.g. instantiation state of the variables).

- *Cut*: represents the only non-logical operator in Prolog which affects the control of the execution. The cut allows to prune parts of the search tree (also called search space).

- *Extra-logical predicates*: are used to perform I/O (e.g. read and write) and to dynamically modify the structure of the program (e.g. adding new clauses to the program using assert).

In this text, *Prolog semantics* refers to the ordering of the exploration of the search tree. Extra-logical predicates whose behaviour depends on the order of execution are said *order-sensitive predicates.*

3 Parallelism in Logic Programming

Parallel logic programming [57, 17, 46, 73, 100] is an important research field whose history is closely related to logic programming languages evolution. The main goal in having parallel logic programming environments is to allow programmers to develop faster applications that are suitable to be developed in a declarative language such as optimisation problems, relational databases, inductive logic programming and resource allocation problems among many others. Independently from the specific paradigm considered, in order to execute a program which exploits parallelism, the programming language must supply the means to [73]: (1) *identify parallelism* by recognizing the components of the program execution that will be (potentially) performed by different processors; (2) *start* and *stop* parallel execution; (3) *coordinate* the parallel execution, e.g. specifying and implementing interactions between concurrent components.

These parallel environments could use explicit structures to allow programmers to define parallelism or could explore parallelism automatically without programmers interference. These two types of approaches are called respectively explicit and implicit parallelism.

Explicit parallelism is characterized by the presence of explicit constructs in the programming language, aimed at describing to a certain degree of detail the way in which the parallel computation will take place. The main advantage of this approach is its flexibility which allows to code a wide variety of patterns of execution, giving a considerable freedom in the choice of what should or not run in parallel. However, the management of parallelism is a complex task involving activities as detecting the components of the parallel execution and controlling processes synchronization.

Implicit parallelism allows programmers to write their applications without any concern about the exploitation of parallelism. Parallelism is transparent to the programmer because it is automatically performed by compiler and/or runtime system. The main advantage of implicit parallelism is keeping the complexity of parallel software development to special systems, and consequently releasing programmers from additional programming effort.

However, the parallelizing system may try to parallelize code which is inherently sequential or to exploit very fine-grained parallelism. These situations normally lead to an inefficient execution.

For Logic Programming, there are both implicit and explicit parallel execution models. Gupta et al [46] point out that there are many proposals for extending a logic programming language with explicit constructs for concurrency. They are largely put into three categories:

1. those that add explicit message passing primitives to Prolog e.g. Delta Prolog [72] and CS-Prolog [41]. Multiple processes are run in parallel that communicate with each other via messages.

2. those that add blackboard primitives to Prolog, e.g. De Bosschere et al [26] and Shared Prolog [18]. These primitives are used by multiple Prolog process running in parallel to communicate with each other via the blackboard.

3. those based on guards, committed choice, and data-flow synchronization, e.g. Parlog [45], GHC [93, 91], and Concurrent Prolog [83].

However, this tutorial will focus only on implicit parallel approaches. In Logic Programming there are some implicit sources of parallelism that facilitate the automatic execution. The two main sources are the following:

- **OR-parallelism**: is the parallel execution of clauses that compose a predicate. The simultaneous development of several resolvents that would be computed successively by backtracking in a sequential execution is called OR-Parallelism since we can consider that the clauses are connected by a logical *or* operator ($\vee$).

- **AND-parallelism**: is the parallel execution of goals that compose a clause. It is called AND-parallelism since goals are connected by a logical *and* ($\wedge$). The main inherent difficulty of exploiting AND-parallelism is to obtain coherent bindings for variables shared by several goals executed in parallel. This has led to the design of execution models where goals which may possibly bind shared variables to conflicting values are either serialised or synchronised. This gives place to two distinct classifications:

  - **independent AND-parallelism**: only independent goals which do not share any variable are executed in parallel;
  - **dependent AND-parallelism**: concurrent execution of goals sharing variables is possible.

Both kinds are called *control-parallelism* and they will be presented in detail in the next sections. There are other implicit sources of parallelism called *data-parallelism* which exploits parallelism that originates from the data. These are *unification parallelism*, *path parallelism* and *recursion parallelism*. They are very fine-grained, and therefore difficult to achieve good performance in parallel systems and may require specialised architecture in order to get speedups. Nevertheless, many researchers pursue the objective of exploiting data-parallelism [44, 88].

In the literature we can find several systems that exploit only and-parallelism [53, 54] [50] [86], only or-parallelism [4] [70] [12, 13] [103] or a combination of and- and or-parallelism. Systems that combine both and- and or-parallelism are still under development and research [47] [106, 105] [67, 68] [69], and several problems remain to be solved. For instance, what kind of data structure is more adequate to combine and-parallelism with or-parallelism, or what kind of and-parallelism is more common in application programs, or yet what kind of implementation strategy to use when developing parallel logic programming systems for hybrid parallel architectures such as distributed shared memory systems [74]. Most well-known parallel logic programming systems currently available were implemented for shared-memory architectures, because of its simpler programming model. However some applications need more scalable architectures. Recent work has been done on studying the performance of such systems on modern architectures [78, 77, 76, 87, 14, 81] in order to devise a suitable model that would run efficiently on these modern architectures [15].

3.1 OR-Parallelism

Or-parallelism is present in many application areas such as parsing, optimisation problems and databases. Or-parallelism is one of the simplest forms of implicit parallelism, since it can be executed without including any annotation in the code. At the same time, the execution model is close to the sequential one. This helps efficient sequential techniques reuse.

All efficient implementations use the multi-sequential model. The *multi-sequential* approach is characterised by a set of processes (*workers*) that have a full Prolog Engine, i.e. each worker can solve a complete branch of the search tree.

The fundamental problem to solve in an or-parallel logic programming system is the maintenance of multiple bindings for the same variable in different branches of the search tree [35]. In theory, a new resolvent is generated at each unification of a goal against the head of a clause, by renaming all the variables of the current resolvent. In practice, resolvents are not copied and the same memory locations are used to compute alternative resolvents [57]. As a goal can match to several different clauses and produce different bindings for a variable, a system that allows several clauses to proceed in parallel need a mechanism to avoid conflicting bindings. In sequential Prolog this problem does not arise because each branch is executed at once and the conditional bindings are annotated in a special stack called *Trail* to be unbound later on backtracking. In parallel, each processor needs to have its own copy of the conditional variable, as processing of each branch can be done separately and simultaneously with other processors. Several solutions have been proposed to cope with this problem. One of the frequently used classifications consider three groups: copying, sharing and recomputation of stacks [57].

1. **copying of stacks**: In this scheme, each worker maintains a complete copy of the stacks in its workspace. An importer worker copies the stacks of the computation state down to the node providing work. The importer will need to restore the stack bindings to the previous state, i.e. when the imported choice-point was created. This way, they keep their own independent environments sets to produce multiple bindings to a variable. Some systems that use this approach are **Muse** [4, 5, 56], OPERA [12, 13], pclp(FD) [96] and YapOr [75]. The overhead of copying can be reduced through the *incremental copying of stacks* technique. Incremental copying is based in fact that one idle worker can share a part of the search tree with the exporter. So, both share parts of the stacks. In this technique, the importer backtracks until the last common choice-point before copying the segments of the stacks that are *younger* than the last common choice-point.

2. **sharing of stacks**: workers share the parts of the stacks that they have in common. Since several logical variables may have the same identification, which is a location in a shared stack, special data structures are used to store logical variables. **Aurora** [65, 66] and PEPSys [9, 58]/ECLiPSe [2] are some or-systems that uses this technique.

3. **recomputation of stacks**: in this technique each worker computes a pre-determined path of the search tree described by an "oracle" allocated by a specialized process called "controller". To simplify the path description through a bit-string, programs are rewritten to obtain an arity 2. Delphi [22] is the main example of this approach.

Copying and sharing of stacks are the two approaches more successfully employed. Sharing stacks can be done using different techniques as for example hash-windows, variable importation, directory trees, and binding arrays.

The idea of **Hash-windows** [11, 58] is based on the observation that unification of a clause head with its goal arguments, generally, only binds a small number of variables in the ancestor environments. Thus, these variables' bindings can be stored in a small "window" which is only accessible to the process that created the window and its descendant or-processes. Windows are linked for each process and are visible between choice-points (see figure 3). The disadvantage of this method is that the access time for each variable is non constant, because when a descendant process does not find the variable on its own window, it needs to scan its ancestor's windows until finding the variable.

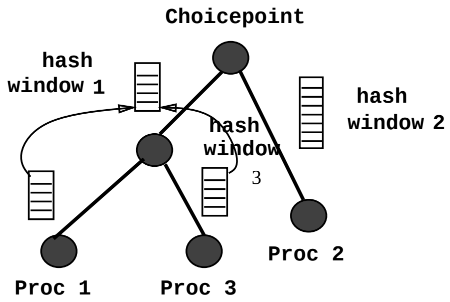

Figure 3: Hash-windows technique

Variable importation [64] (see figure 4) is based on the idea that the only variables in the ancestor environment which can be affected by unification are those which are unbound. If these variables are "imported" into the local frame for a particular process, the unification can only update variables in the local environment, independently of other or-parallel processes. The difference between variable importation and hash windows is that in the variable importation method, variables are searched for in environments local to the processor, rather than in special variable tables.

Directory trees [19] (figure 5) are based on the idea that only those environments that contain unbound variables need to be copied to create a separate environment. Ancestor environments which contain no unbound variables can be shared between processes, while copies are made of other environments, with the relevant entries in the directory being changed to point to the new copy.

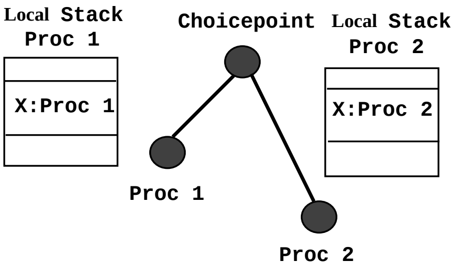

Figure 4: Variable importation technique

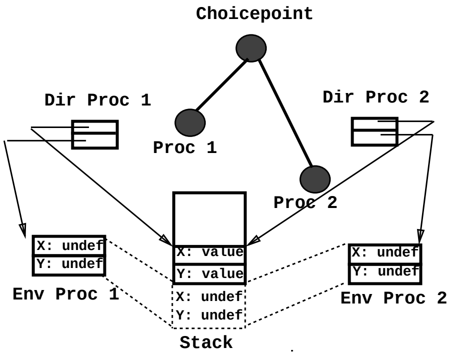

Figure 5: Directory trees technique

A **Binding-array**[103] (figure 6) is an array private to each worker which contains conditional logical variables. Conditional variables are all variables that were unbound after the creation of a choice-point. Whenever a process needs to trail a variable, it caches its value into the binding array. Every process has the same entry to the same variable. In the Binding Arrays, the operation of taking work is considerably faster since the only activity required is updating the content of the binding arrays. On the other hand, the model has an additional overhead due to the fact that every conditional variable is now stored in binding arrays, and every access to it requires one step of indirection. Additionally, the use of binding arrays requires that, during trailing, both the address of the variable and its current value are saved in the trail stack (which doubles the cost of trailing/untrailing).

Each technique has an associated non-constant-time cost that is incurred at task creation time, at variable access time, or at task switching time. All three operations cannot be performed in constant-time [48]. Therefore in order to reduce the overhead costs, systems need to reduce these three costs

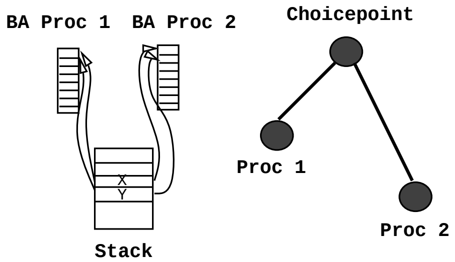

Figure 6: Binding Arrays technique

to as low as possible. Unless the system employs a static scheduling strategy to execute the programs, task switching is always non-constant time, and in this case, it becomes one of the critical parts of the system to be optimised. From the techniques we study here, binding arrays is the only one that keeps both variable access and task creation at a constant time cost.

Another way of maintaining multiple environments for variables is by doing stack-copying. More recently, Gupta et al [49] have proposed the *Stack-splitting* scheme, a modification of the stack-copying technique for environment representation.

The advantage of using stack-sharing is that there is no duplication of amounts of memory used. Copying has also the advantage of having exactly a standard sequential execution in each worker, at least as long as this computation is limited to the "private" part of the tree.

If a parallel Prolog system wants to maintain the *Prolog semantics*, the execution of *side-effects* requires special attention: workers should guarantee that the execution of order-sensitive predicates across the branches of the or-tree is performed in the correct order. This typically involves delaying the execution as long as the current branch is not leftmost in the or-tree.

3.2 Independent AND-Parallelism (IAP)

The main problem in the independent AND-parallelism is the detection of independence between goals. A possible approach consists in perform half of the detection work at compile time to limit the runtime overhead.

Independent and-parallelism is exploited when the parallel system guarantees that two or more goals in the body of the clause do not share any variables. If the arguments of the goals are found to be independent at runtime then the goals are allowed to proceed independently. A system that implements and-parallelism generally has the following steps:

- an ordering phase that is needed to detect dependencies among goals;
- a forward execution phase, whose objective is to select goals for and-parallel execution; and
- a backward execution phase that is activated whenever a goal fails.

The ordering phase can be done in several ways. It can use user defined *modes* to indicate dependencies among goals. Arguments of clauses are annotated as being "input" or "output". Another method is to do static data dependency analysis of goals in the clauses. This is done at compile-time and annotates the variables as being independent. This method may lose much parallelism, with the advantage of having minimum runtime overhead. Another method is to build, at runtime, dependency graphs. This method incurs a very high runtime overhead, but can exploit all parallelism available. The final method is to generate code for dependency checking at runtime. This combines static data dependency analysis with runtime dependency checking. It is a compromise solution. Compile-time tools have been built to implement it [71].

The execution model of a system that implements IAP can be done by allowing all independent goals to produce all solutions and then *combine the solutions*, or *recomputing* all goals to the right for every solution produced by goals to the left.

One example of a system that exploits independent and-parallelism is Hermenegildo's &-Prolog system [54]. In &-Prolog, processing elements called WAM agents cooperate with each other to execute independent goals in parallel. Programs are compiled into graphs called *Extended Conditional Graph Expressions* (ECGE, based on DeGroot's CGEs [29, 30]). The ECGE graphs are independence tests annotated to the Prolog source code at compile-time and checked at runtime as illustrated in table 1. The first line of the table, for example, shows a pair of conditions generated at compile-time that will be tested at runtime, i.e., in order to run goals `p/2` and `q/2` in parallel, the runtime system needs to guarantee that the variable `Y` is ground (completely instantiated) and that variables `X` and `Z` are independent (not shared by goals `p/2` and `q/2`).

The generation of the ECGEs can be a problem, because many CGEs are possible for the same clause. The number of variables to be checked for independence can also be a problem. In practice even simple runtime checks, such as testing for groundness of a complex nested Prolog term, can be very expensive. Groundness checking can introduce a significant overhead to the

Prolog goals	**&-Prolog CGEs**
p(X,Y), q(Y,Z)	ground(Y), indep(X,Z)
p(Y), q(Y)	ground(Y)
p(Y,Z), q(W)	indep([(Y,W), (Z,W)])
p(Y,Z), q(W)	indep([(W, [Y,Z])])

Table 1: Examples of Prolog mapping to &-Prolog

system. &-Prolog uses global analysis to infer as much information as possible from the programs [71], and produce reasonable annotated code. The system implements goal recomputation instead of combination of solutions. If cross-product is used, there is no need for recomputation. This increases the quantity of parallelism exploited, but the cross-product algorithm can be a bottleneck. Cross-product can also be the source of an incorrect Prolog execution order, since solutions are produced in any order. If recomputation is used only the first solution is found in parallel. All the other solutions will be found by backtracking. Parallelism in this case is limited by the number of solutions each goal has.

Another system that also exploits IAP is APEX (And-Parallel EXecution of logic programs) [62]. This model exploits and-parallelism on shared-memory architectures. The major features of the implementation are (1) dependency analysis between literals of a clause is done dynamically without incurring excessive runtime overhead; and (2) backtracking is done intelligently at the clause level without incurring any extra cost for the determination of the backtrack literal; Performance results on a Sequent Balance 21000 show that APEX can achieve linear speedups on dozens of processors.

3.3 Dependent AND-Parallelism

Dependent and-parallelism allows goals that share variables to go ahead simultaneously, but under some conditions [35]. Usually, it is exploited in concurrent logic languages. The Parlog system [45] exploits dependent and-parallelism in a producer-consumer fashion. If two goals share a variable, one is considered to be the producer for the variable and the other one will be the consumer. The goal is selected to be a consumer or a producer through *mode declaration*, i.e. the programmer or a preprocessor annotates the program to indicate the mode (input or output) of the arguments for each predicate. As long as an output argument is assigned one value, the other goal (the consumer) can start execution. If the variable is instantiated to a list or structure, the consumer goal can start to consume the partially instantiated argument. A subset of languages that support dependent and-parallelism do not support backtracking, such as Parlog. These are called committed-choice languages. Only one clause is allowed to be chosen through the evaluation of *guards*, usually implemented as calls to built-in predicates that make clauses mutually exclusive. Once a guard succeeds, the system commits to the clause containing the guard, and all other clauses are ignored. Other examples of these systems include GHC [93, 91], FGHC [94], Concurrent Prolog [83] and KL1 [59].

Concurrent Prolog is a logic programming language in that a program is a collection of universally quantified Horn-clause axioms, and a computation is an attempt to prove a goal – an existentially quantified conjunctive statement – from the axioms of the program. The goal statement describes an input/output relation for which the input is known; a successful (constructive) proof provides a corresponding output. The difference between Concurrent Prolog and other logic programming languages (e.g., pure Prolog) is in the mechanism they provide for controlling the construction of a proof. Prolog uses the order of clauses in the program and the order of goals in a clause to guide a sequential search for a proof, and uses the cut operator to prune undesired portions of the search space. Concurrent Prolog searches a proof in parallel. To control the search, CP embodies two familiar concepts: guarded-command indeterminacy, and dataflow synchronisation. They are implemented using two constructs: the commit operator and the read-only annotation.

KL1 is a system based on the KL1 language, that by its turn is based on GHC (Guarded Horn Clauses). In these languages the control is also added by the user through *guards* that guarantees only one clause to be committed. The KL1 language was designed in the scope of the Fifth Generation Computer Systems Japanese project.

Yang [104] designed a language, P-Prolog, based on mutual exclusion relations where a set of clauses can be "backtrackable". Clauses in a predicate in this language are grouped. Clauses in the same group behave in the same way as GHC, but if a fail occurs for a group of clauses, another group can be tried. The P-Prolog language introduced a new concept of programming in committed-choice languages, by allowing a degree of don't know non-determinism to the programs.

Another system that exploits dependent and-parallelism, but in a different way, is DDAS [84, 86]. In DDAS the variable dependency is checked at compile-time and the program is annotated with extended CGEs similar to the ones used in &-Prolog. Goals to the right of a conjunction are allowed to proceed in parallel as long as they do not bind the dependent variable.

Only the leftmost goal is allowed to bind the dependent variable in order to keep the operational semantics of Prolog. Hence rightmost goals that try to bind a dependent variable are suspended until become leftmost in the CGE.

3.4 AND-OR Parallelism

Parallel Logic Programming systems should serve the broadest range of applications to become popular. To achieve this goal, these systems should support both or- and and-parallelism since most programs exhibit mostly one kind of parallelism. Requirements for a usable parallel WAM engine are [46, 100]: (1) parallelism should be transparent to the programmer; (2) both and- and or-parallelism should be supported as a minimum; (3) the performance of the WAM engine on a single processor should be similar to that of a sequential WAM; (4) the parallel WAM should incorporate as many of the known sequential optimisations as possible; and (5) the runtime overhead should be low.

In practice, exploiting one single form of parallelism requires sophisticated system design and exploiting distinct forms of parallelism at the same time is even harder. And/Or-parallel systems clearly suffer of overhead coming from both and- and or-parallelism. Unfortunately, this overhead may get worse due the combination of the two forms of parallelism.

Scheduling becomes, in general, considerably more complex. The presence of both and- and or-parallelism requires an additional scheduling step, in order to decide which form of parallelism to exploit next. All additional scheduling requirements can be achieved only through the use of additional data structures which collect the information required by the scheduler.

Since integration of and-parallelism with or-parallelism is a very complex task, few systems were successful in doing it:

- Andorra-I [79]: exploits or-parallelism a la Aurora [66] and determinate and-parallelism, i.e., goals that match at most one clause in the program are executed firstly and in parallel.

- Penny [67, 68]: a parallel implementation of AKL [52][55] that also exploits or-parallelism.

- ParAKL [69]: another parallel implementation of AKL.

- ACE [47]: a system that integrates independent and-parallelism a la &-Prolog and or-parallelism a la Muse [4, 77].

- SBA [24]: system that integrates independent and-parallelism a la &ACE with or-parallelism a la Aurora [24]. To the best of our knowledge, SBA is the first system to tackle the problem of efficiently representing variable bindings when combining independent and-parallelism with or-parallelism. It uses the Paged Binding Array model [51].

All these systems were designed and developed for shared-memory architectures. As an example of system that integrates and-parallelism with or-parallelism designed and developed for distributed machines we can mention the Distributed Andorra Prolog [10].

Systems implemented for centralised memory architectures present several limitations: (1) the hardware is not scalable, (2) algorithms and data structures designed for shared-memory architectures sometimes are not scalable when running on scalable hardwares. Distributed implementations of parallel logic programming systems that combine both and-parallelism with or-parallelism requires quite complex parallel programming skills.

New software/hardware platforms have been developed that combine the shared memory programming model with a scalable hardware (distributed shared memory systems – DSMs [74]). A new parallel logic programming system that integrates the shared-memory programming model with the distributed-memory programming model, so-called DAOS [15], aims to obtain an efficient system capable of running on a hybrid architecture (for example, a cluster of workstations connected via a fast network that allows a shared-memory programming model). Research carried out about performance of parallel logic programming systems originally written for shared-memory machines, shows that it is possible to achieve significant performance improvements when running these systems on DSM architectures by only making modifications to data layout and making few modifications to the algorithms [78, 76]. These experiments also show that architectural parameters can affect significantly the performance of parallel logic programming systems [87]. These results confirm that parallel runtime model designed to run in DSM platforms can obtain maximum performance from scalable architectures.

These studies indicate that data structures used in scheduling normally lead to false sharing. DAOS innovates by defining a hybrid solution: areas (stacks) which implement the major data-structures are logically shared and areas that are used to implement work management are private. To solve binding conflicts it uses an adaptation of SBA [24].

3.5 Tools for performance evaluation

The development of efficient parallel logic programming systems is not an easy task. These systems need to tackle some hard problems such as parallel debugging and dynamic distribution of varied sized work among processors. Debugging in parallel environments is difficult due to non-deterministic response time of tasks.

Besides, some applications have very low degree of parallelism, either and- or or-parallelism or have very small input data size. Usually programs that present very low degree of parallelism use sequential programming style, what makes them difficult to parallelise transparently [42]. Programs that present a reasonable amount of parallelism can lead to good performance on parallel systems. However, in practice, there are several issues that contribute to bad performance even when the application has sufficient parallel tasks to supply available processors due normally to workload distribution strategies.

One way of dealing with these problems is to provide to parallel system developers and users a visualisation tool for parallel logic programming systems. Several visualisation tools have been developed such as WAMtrace [32], VisAndOr [43], ViMust [90], and VACE [99]. Each one of these works for a different parallel logic programming system. VisAll [39] is a more recent tool which attempts at unifying the visualisation paradigms exhibited by the other visualization tools adding extra features to system optimisation and debugging.

4 Scheduling

Parallel or distributed execution means that different pieces of the program will be divided in several tasks and these tasks will be assigned to available processors. The task scheduler is the entity responsible for deciding what, when and to whom export tasks. All scheduling policies must distribute tasks over the available computational resources improving one or more performance measures. The scheduler is responsible for the efficient utilisation of processors. By efficient utilisation we mean to use only the resources required by the application. A reasonable scheduling strategy will also complete a parallel application in a time close to the earliest possible time. So, scheduling is a fundamental aspect in any parallel system since it is closely related to system performance and thus is an important research field in parallel logic programming.

In parallel logic programming systems, the execution of logic programs generally produces irregular computations that can be mapped into trees. Some applications produces very "bushy" trees, while others produce "thinner" trees. The trees have nodes that are distributed throughout the computing resources to be executed.

In or-parallel systems the nodes of the execution tree are or-nodes (also called choice-points) that contain alternatives left to be executed. A collection of or-nodes connected in some hierarchical order form an *or-tree*. Finding or-work, i.e., finding new alternatives to execute, is the task performed by an or-scheduler. The *or-scheduler* deals with problems such as which alternative to select next to continue working or which branches need to be pruned.

And-parallel systems generate the same kind of trees as or-parallel systems, but with and-nodes. A collection of and-nodes connected in some hierarchical order form an *and-tree*. In and-parallel systems, the *and-scheduler* is responsible to select the next goal to be executed.

In systems that exploit both and-parallelism and or-parallelism there is a new issue to be tackled which is *what kind* of work to be chosen next: and- or or-? There is not much research on this scheduling issue, because the combination of and- and or-parallelism is a relatively new issue and there are not many implemented systems that exploit both and- and or-parallelism.

Systems that exploit both and- and or-parallelism usually generate *and-or trees*, although several variations of exploitation of and-parallelism may lead to different conceptual trees. A *task* to be selected by a scheduler to allocate to a processor is any work done by the system engine between two invocations of the scheduler. In that case, depending on the implementation system, the task sizes may vary. A *task switch* is performed when the scheduler is invoked to find work, and the worker actually switches to another task.

Next we discuss some or-only and and-only scheduling techniques used for some well known parallel logic programming systems that exploit either and- or or-parallelism, and and/or scheduling techniques proposed for systems that exploit both and- and or-parallelism (sections 4.1, 4.2 and 4.3).

4.1 Scheduling Or-Parallel Work

In a parallel logic programming system an or-scheduler has the function of choosing the best choicepoint from which to take an alternative to be explored, in order to utilise resources in a better way. It should not incur too high costs in task switching, and it should cope with a limited number of processors in the system. Another very important issue is to reduce the amount of interaction between the workers. Some schedulers also deal with the problem of "speculativeness". This means that whereas some branches of the search tree are to be executed compulsorily, others (speculative or-branches) might be pruned because of a cut or commit operator. A branch of the or-tree can also be speculative if we can find ways of knowing that it does not lead to a valid solution.

Several schedulers have been proposed to distribute or-work in or-parallel logic programming systems. In shared memory systems implementations such as Aurora [66] and Muse [4], or-schedulers try to minimize frequency of task switching, and amount of interaction between the workers. In the implementation of other models [102], methods are used that keep the cost of task switching constant by maintaining the cost of task creation or variable access as non constant. Task switching costs are directly related to scheduling. It is known from [20] that it is more efficient to keep the cost of task switching non constant while keeping other costs – variable access and task creation – constant.

Contrary to distributed systems, schedulers for shared memory systems do not need to tackle the processor communication problem. Instead, the greatest problem to be tackled by an or-parallel scheduler is to control the frequency of task switching in systems that do not have constant time for this operation. Indirectly, two problems of equal importance arises when we do not control the frequency of task switching: cache corruption and scheduling overheads. These problems can also occur in distributed memory systems where the penalty for not maintaining locality of references is more serious, incurring in memory accesses that can be 10,000 times slower than a shared memory access.

There are several different schedulers for or-parallel systems, but the main basic strategy to follow, in general, is: *to give preference to mandatory work. If there is no mandatory work select the least speculative work and repeatedly seek for better work.* This is the algorithm that seems to fit all or-parallel systems nowadays and proved to be the best so far. Around this main algorithm, several different strategies implement some or all of this.

There are in the literature several strategies describing different schedulers for different models, and even for the same model. For example, the Aurora system has at least 5 scheduler strategies that compete with each other for parallel efficiency. The strategies vary mainly in the way that the work is chosen in the execution search tree. The schedulers also use different data structures that affect in different ways the performance of the or-parallel system. Some of them, the most recent ones, deal with speculative work. All these strategies are supposed to be dynamic, i.e., decisions are made based on information collected at runtime.

4.2 Scheduling And-Parallel Work

In systems that implement **independent and-parallelism** the main problem in and-scheduling is to correctly implement backtracking inside a CGE. If any goal is selected to execute in and-parallel, and the implementation is WAM-based with contiguous stacks, on backtracking, a worker may not be able to backtrack and continue a previous goal, because there may be another goal being executed after the goal that needs to start again. This is shown in picture 7. From goals `a & b & c`, worker 1 (w1) selects goals `a` and `c`, while worker 2 (w2) selects goal `b`. Assume that worker 1 finishes goal `a` before worker 2 finishes goal `b`. In that case, worker 1 starts goal `c`. Assume that worker 2 finishes execution of goal `b`. It needs to backtrack and find another solution to `a`, but `a` cannot continue, because goal `c` did not finish yet. This is illustrated as the *trapped goal problem.*

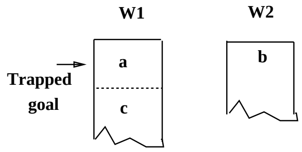

Figure 7: THE TRAPPED GOAL PROBLEM

The strategy currently used by the &-Prolog system is random selection of parallel goals from any agent's stack. As shown by the explanation before, this strategy is dangerous if we allow goals inside a ECGE to backtrack. In this case it can cause the problem of "trapped" goals where the topmost goal in an agent's stack backtracks on top of an older goal that did not finish execution yet. In order to solve this problem, there are three approaches. The first one [54] was designed for the &-Prolog system itself, and consists of only selecting a goal for parallel execution if it is the oldest pushed on the goal list. Therefore, WAM agents only execute goals in a certain order that is the sequential Prolog's order of goals. The second approach is to use a more flexible scheduling strategy that instead of restricting selection of goals by age, can select any goal. This is achieved by maintaining a *separator marker* between goals that can be executed in parallel. This scheduling strategy is described by Shen [85] and used in the DDAS system. A third approach that will probably be used by the PBA implementation is to employ a memory management that allocates memory segments dynamically instead of using contiguous chunks of memory.

4.3 Scheduling And/Or-Parallel Work

With the most recent systems that aim at exploiting both and- and or-parallelism, the new issue is *what* kind of work to distribute. Some solutions were found that use some kind of global analysis of the program to predict the amount of work to be done in each branch [27, 8]. Other solution is to consider that all Prolog applications when executed in parallel generate a certain amount of work that can be mapped onto a statistical distribution [63]. Assuming that all tasks are to be executed with the same probability distribution, an algorithm is used that obtains the probability to the branches in order to distribute tasks evenly among the processors. This technique although applied to a model that aims to exploit and- and or-parallelism, only solves the or-parallel work distribution.

Another solution is used by Gao et al [107] and involves generating feedback information to the system by profiling previous interactive system sessions. The solution proposed tries to solve almost all problems by using a combination of static, dynamic, and hybrid approaches to get granularity sizes of Prolog programs, and feed these results into the system together with most-frequent used modes for calls to predicates. The scheduling strategy for or-parallel tasks is to choose the nodes nearest to the root. The strategy to choose and-goals is based on goal granularity (given at compile-time). Nothing is reported about how to decide between and- and or-work, and no results are reported at all.

Andorra-I [80, 79] is the first and/or parallel logic programming system that has reported results running real applications. Andorra-I is the first reasonably mature implementation of a system that exploits both and- and or-parallelism. Andorra-I exploits or-parallelism as in Aurora by using the binding arrays technique of the SRI model to solve the problem of different conditional bindings per branch of the search tree. It exploits a simple form of and-parallelism by allowing only *determinate* goals to be executed in parallel. If goals try to bind a common variable with different values the whole branch fails.

Workers in Andorra-I work much the same as in the Aurora system, but instead of having a single worker to explore a branch, several workers cooperate with each other in a *team* in order to explore and-parallelism if this exists in the branch. The first version of Andorra-I used a fixed configuration of workers into teams, and once the user started the execution with a given configuration, this would never change. If the application had a tree that did not suit the chosen configuration, Andorra-I would lose parallelism. Andorra-I had received a reconfigurer [35] whose objective is to rearrange workers into teams according to the parallelism existing in the application. This strategy allows workers to be redeployed from one team to another to look for work. It also decides **what** kind of work to choose – and- or or- – in order to achieve reasonable load balancing and overall better performance. The other new issue in parallel logic programming systems, either and-parallel systems or or-parallel systems, that is tackled by the reconfigurer is the efficient utilisation of resources by the parallel system.

4.4 Compile-Time Analysis

Regarding compile-time analysis tools for parallelisation support, we will concentrate on granularity and complexity analysis. Consider as a discussion case the GRANLOG model [8] that extends and generalises previous models proposed in the literature [92, 28, 27]. The GRANLOG tool composed of several modules accepts a Prolog program as input and generates an annotated Prolog program containing information about complexity of goals and clauses. The information produced by GRANLOG is very useful to guide scheduling decisions. For example, based on the number and grain size of alternatives in a choice-point, the scheduler can select better pieces of work to assign to processors. Global information about amounts of and-parallelism and or-parallelism can be inferred from the grain sizes provided by the GRANLOG tool, in order that we can choose between sources of and-parallel work or sources of or-parallel work [33, 34, 36]. In the context of APPELO project [42] three research work were carried out utilising GRANLOG information: (1) the centralised scheduler used by the Plosys system was modified in order to support GRANLOG information [38], (2) the Andorra-I reconfigurer [35] and the Andorra-I engine were modified to support GRANLOG information [37], (3) the DSLP scheme (Distributed Scheduling for Logic Programming) was designed and developed using information provided by GRANLOG [25].

Results are very encouraging and show that logic programming can be an alternative to efficient software development producing very good performance at a very low programming cost. Static analysis can produce useful information to guide scheduling decisions, and our runtime model can be more efficient than systems developed for specific parallel platforms (shared-memory only or distributed-memory only).

4.5 Discussion

Systems that exploit or-parallelism are quite mature and their scheduling techniques are improving with each new algorithm and implementation. Researchers have vastly studied or-scheduler systems and it is becoming clear what is required of an or-parallel scheduler to make an or-parallel system efficient. General conclusions are that low communication between workers is important, in order to be able to perform well with both speculative applications and non-speculative applications. We also need to have an efficient way of keeping the left-to-right order of the branches in the tree and have an efficient algorithm for searching for deepest leftmost work. It is also clear that an or-parallel system can benefit from schedulers that are capable of disposing some processors if the application does not need them. The Bristol-scheduler, the Dharma-scheduler and the Muse-scheduler have been successful in doing their jobs.

Other or-scheduling methods such as the method proposed by Lin [63] are valid, although some assumptions are made that are not easy to accept in logic programming. E.g., structural imbalance is an intrinsic characteristic of logic programs, therefore to remove structural imbalance is not an easy task. To do that at runtime can introduce too many overheads. This strategy is hence more suitable for a static scheduling model than for a dynamic scheduling model. It is arguable the efficiency of such systems, because the scheduler needs to be centralised in order to keep track of number of idle processors and number of tasks not yet taken at a given time of the execution, and allocate all idle processors to the work at once.

The and-schedulers are not so studied as the or-schedulers, but some conclusions can be drawn. It is not a very good idea to maintain a global pool of goals to schedule and-goals. Moreover, in systems that exploit IAP it is also important to prevent the trapped goal problem.

The method used by Tick [92], that introduced a compile-time technique to help decisions at runtime, although very simple, proved to be not very efficient for scheduling and-parallelism. It might be useful for scheduling or-parallelism though, as Wai-Keong [40] tried to show. To collect granularity size of programs can be a good idea if one can spot points in the program that presents a regular behaviour. This would help the schedulers, both and- and or-schedulers, in some cases. In the cases that we cannot conclude anything about the shape of the execution tree, some strategy that does not rely on compile-time analysis data, needs to be used anyway. To assume that the amount of work in the applications follows a given probabilistic distribution seems unreal, given that the shape of Prolog application search trees varies widely. The idea of profiling is interesting and can be a source for future research, although we can profile and draw conclusions for some classes of applications, but not generalise for all applications.

With respect to scheduling both and- and or-parallelism Andorra-I [105] is an and/or parallel logic programming system that tackles this problem. So far the results have been quite encouraging. Andorra-I is capable of running a wide range of applications that contain both and- and or-parallelism or either kind alone and achieve good speedups without the need for manually adjusting the number of workers to exploit and- and or-parallelism.

The other presented And/Or-system is DAOS [15]. Several scheduling strategies are available in order to maximise performance, minimise frequency and number of communication messages while keeping locality for distributed environments [98, 25, 16, 97]. In order to develop and test these strategies simultaneously with the DAOS prototype, they were either simulated or implemented on other available parallel logic programming systems (Plosys [70] and Andorra-I [79]).

5 Conclusion

Some of the main issues in adding parallelism to logic programming are:

- *What kind of Parallelism to Exploit?* Regarding the programming paradigm, parallelism can be exploited implicitly or explicitly.

- *What kind of Architecture to Choose?* The system can run in any architecture as long as it runs *efficiently* which means to suit the system model to the architecture chosen.

Implicit parallel logic programming has the advantage of allowing program reuse and transparency at user level. However it is not a trivial task and several techniques and approaches have been developed in the last years as discussed in this document. Nowadays, some important research fields in parallel logic programming are: (1) efficient ways to combine and- and or-parallelism; (2) efficient scheduling and execution in scalable systems; and (3) compile-time (static) analysis to help run-time decisions; (4) extensions of logic programming to prune the search space (such as tabling and constraints). Systems should support both or- and and-parallelism to serve the broadest range of applications since most programs exhibit mostly one kind of parallelism. In practice, exploiting one single form requires sophisticated system design and exploiting distinct forms of parallelism is even harder.

In order to exploit parallelism implicitly, it is necessary to detect sources of parallelism in the applications. Since tasks in logic programming are created dynamically, we need a good runtime support to identify sources of parallelism. Compile-time granularity analysis can help to improve performance in some cases.

Results of few parallel logic programming systems show that it is possible to achieve good performance on parallel architectures with a minimum effort from the end user.

For interested readers, we can indicate sereral pointers to the Logic Programming World. One very interesting and complete site is located at `http://www.comlab.ox.ac.uk/archive/logic-prog.html`. For a comprehensive bibliography site goto `http://liinwww.ira.uka.de/bibliography/LogicProgramming/index.html`. The ALP (Association for Logic Programming) site can be found at `http://www.cwi.nl/projects/alp/index.html`. There you have access to the ALP Newsletter. There is also the Compulog-Americas initiative that contains information about logic programming groups and research in the Americas: `http://www.cs.nmsu.edu/~compulog`. The Journal of Logic Programming is the most important publication in the area. Several important conferences are: ICLP (International Conference on Logic Programming), PPDP (International Conference on Principles and Practice of Declarative Programming, old PLILP – Programming Languages and Implementation of Logic Programming), PAP (Practical Applications of Prolog), PACLP (Practical Applications of Constraint Logic Programming), and PADL (Practical Aspects of Declarative Languages). Pontelli keeps in his home page an interesting chapter on "Adventures in Parallel Logic Programming" (`http://www.cs.nmsu.edu/~epontell/advent.html`). Enjoy your trip!

References

[1] *The Journal of Logic Programming*, 26(2), February 1996. Special Issue: Applications of Logic Programming.

[2] A. AGGOUN et al. *ECLiPSe 3.5 User Manual.* ECRC, December 1995.

[3] H. AÏT-KACI. *Warren's Abstract Machine — A Tutorial Reconstruction.* MIT Press, 1991.

[4] K. A. M. ALI and R. KARLSSON. The Muse Or-parallel Prolog Model and its Performance. In *Proceedings of the North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[5] K. A. M. ALI and R. KARLSSON. Performance of Muse on the BBN Butterfly TC2000. In *LNCS 605, PARLE'92 Parallel Architectures and Languages Europe*, pages 603–616. Springer-Verlag, June 1992.

[6] J. ANDERSSON, S. ANDERSSON, K. BOORTZ, M. CARLSSON, H. NILSSON, T. SJOLAND, and J. WIDÉN. SICStus Prolog User's Manual. Technical report, Swedish Institute of Computer Science, November 1997. SICS Technical Report.

[7] K. APT. *From Logic Programming to Prolog.* Prentice Hall, 1997.

[8] J. L. V. BARBOSA. Granlog: Um modelo para análise automática de granulosidade na programação em lógica. Dissertação de mestrado, Universidade Federal do Rio Grande do Sul, 1996.

[9] U. BARON, J. C. CHASSIN de KERGOMMEAUX, M. HAILPERIN, M. RATCLIFFE, P. ROBERT, J.-C. SYRE, and H. WESTPHAL. The Parallel ECRC Prolog System PEPSys: an Overview and Evaluation Results. In *International Conference on Fifth Generation Computer Systems 1988*, pages 841–850. ICOT, Tokyo, Japan, November 1988.

[10] V. BENJUMEA and J. M. TROYA. A Static Implementation of the Basic Andorra Model for Distributed Memory Systems. In *Post-JICSLP96 Workshop on Parallelism and Implementation of Logic Programming Systems*, 1996. Bonn, Germany.

[11] P. BORGWARDT. Parallel prolog using stack segments on shared memory multiprocessors. In *International Symposium on Logic Programming*, pages 2–12, Silver Spring, MD, February 1984. Atlantic City, IEEE Computer Society.

[12] J. BRIAT, M. FABRE, C. F. R. GEYER, and J. CHASSIN. Opera: a parallel prolog system and its implementation on supernode. In *International Workshop on Compilers for Parallel Computers*, 1990.

[13] J. BRIAT, M. FAVRE, C. F. R. GEYER, and J. CHASSIN. Scheduling of or-parallel prolog on a scalable, reconfigurable, distributed-memory multiprocessor. In *PARLE91*, 1991.

[14] CALEGARIO, V. M. and DUTRA, I. C. Performance Evaluation of Or-Parallel Logic Programming Systems on Distributed Shared Memory Architectures. In *Proceedings of the EUROPAR'99*, pages 1484–1491, Aug-Sep 1999.

[15] L. F. P. CASTRO, V. SANTOS COSTA, C. F. R. GEYER, F. SILVA, P. K. VARGAS, and M. E. CORREIA. DAOS – Scalable And-Or Parallelism. In *EUROPAR '99*, pages 899–908, 1999.

[16] A. P. B. CENTENO and C. F. R. GEYER. PENELOPE: Um modelo de Escalonador Hierárquico para o Sistema PLoSys. In *X Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1998. Búzios, RJ.

[17] P. CIANCARINI. Parallel Programming with Logic Languages: A Survey. *Computer Languages*, 17(4), October 1992.

[18] P. CIANCARINI. Blackboard programming in shared prolog. In *Languages and Compiler for Parallel Computing*. MIT Press, June 1993.

[19] A. CIEPIELEWSKI and S. HARIDI. A Formal Model for Or-Parallel Execution of Logic Programs. In *IFIP 83 Conference*, pages 299–305. North Holland, 1983.

[20] A. CIEPIELEWSKY, K. A. M. ALI, and P. BISWAS. Execution Models and Parallel Implementations of Prolog. unpublished, June 1993.

[21] W. CLOCKSIN and C. MELLISH. *Programming in Prolog*. Springer-Verlag, 1986.

[22] W. F. CLOCKSIN. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.

[23] J. COHEN. A View of the Origins and Development of Prolog. *Communications of the ACM*, 31(1):26–36, 1988.

[24] CORREIA, M. E., SILVA, F. M. A., and SANTOS COSTA, V. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *ILPS'97*, Outubro 1997. Also published as Technical Report DCC-97-3, DCC - FC & LIACC, Universidade do Porto, April, 1997.

[25] C. A. COSTA and C. F. R. GEYER. Uma Proposta de Escalonamento Distribuıdo para Exploração de Paralelismo na Programação em Lógica. In *X Simpsio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1998. Búzios, RJ.

[26] K. DE BOSSCHERE and P. TARAU. Blackboard-based Extensions in Prolog. *Software — Practice and Experience*, 26(1):49–69, Jan. 1996.

[27] S. K. DEBRAY, P. L. GARCA, M. V. HERMENEGILDO, and N. LIN. Estimating the computacional cost of logic programs. In *International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, Namur, Belgium, Sept. 1994. Springer-Verlag.

[28] S. K. DEBRAY and N. LIN. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, Nov. 1993.

[29] D. DeGROOT. Restricted And-Parallelism. In H. Aiso, editor, *Proceedings of the 1984 International Conference on Fifth Generation Computer Systems*, pages 471–478. Institute for New Generation Computing, Tokyo, 1984.

[30] D. DeGROOT. A Technique for Compiling Execution Graph Expressions for Restricted And-Parallelism in Logic Programs. In C. Polychronopoulos, editor, *Lecture Notes in Computer Science 297, Supercomputing*, pages 1074–1093, 1987.

[31] P. DERANSART, A. ED-DBALI, L. CERVONI, and A. A. ED-BALL. *Prolog, The Standard: Reference Manual.* Springer Verlag, 1996.

[32] T. DISZ and E. LUSK. A Graphical Tool for Observing the Behavior of Parallel Logic Programs. In *Proceedings of the 1987 International Logic Programming Symposium*, pages 46–53, 1987.

[33] I. C. DUTRA. A Flexible Scheduler for Andorra-I. In A. Beaumont and G. Gupta, editors, *Lecture Notes in Computer Science 569, Parallel Execution of Logic Programs*, pages 70–82. Springer-Verlag, June 1991.

[34] I. C. DUTRA. Strategies for Scheduling And- and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 1994 International Logic Programming Symposium*, pages 289–304. MIT Press, 1994.

[35] I. C. DUTRA. *Distributing And- and Or-Work in the Andorra-I Parallel Logic Programming System.* PhD thesis, University of Bristol, Department of Computer Science, February 1995. PhD thesis.

[36] I. C. DUTRA. Distributing And-Work and Or-Work in Parallel Logic Programming Systems. In *Proceedings of the 29th Hawaii International Conference on System Sciences*, pages 646–655. IEEE, 1996.

[37] I. C. DUTRA, V. SANTOS COSTA, J. L. V. BARBOSA, and C. F. R. GEYER. Using Compile-Time Granularity Information to Support Dynamic Work Distribution in Parallel Logic Programming Systems. In *X Simpósio Brasileiro de Arquitetura de Computadores, SBAC-PAD*, October 1999.

[38] D. N. FERRARI, P. K. VARGAS, J. L. V. BARBOSA, and C. F. R. GEYER. Modelo de Integração Plosys-Granlog: Aplicação da Análise de Granulosidade na Exploração do Paralelismo OU. In *XXV Conferência Latino Americana de Informática/V Encontro Chileno de Computação da Sociedade Chilena de Computação (CLEI - PANEL'99)*, September 1999. Assuncion, Paraguay.

[39] N. FONSECA, I. C. DUTRA, and V. SANTOS COSTA. VisAll: A Universal Tool to Visualise Parallel Execution of Logic Programs. In *Joint International Conference and Symposium on Logic Programming*, pages 100–114, 1998. Manchester, Inglaterra.

[40] W.-K. FOONG. Or-Parallel Prolog with Heuristic Task Distribution. In *Lecture Notes in Artificial Intelligence 592, Logic Programming Russian Conference*, pages 193–200, 1991.

[41] I. FUTO. Prolog with Communicating Processes: From T-Prolog to CSR-Prolog. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 3–17, June 1993.

[42] C. F. R. GEYER, J. L. V. BARBOSA, I. C. DUTRA, V. SANTOS COSTA, and G. GUPTA. The appelo project – parallel environment for logic programming. In *Protem-cc III*, pages 30–55. CNPq, 1998. Available at http://www.inf.ufrgs.br/procpar/opera/APPELO/.

[43] L. GÓMEZ, M. CARRO, and M. HERMENEGILDO. Some Paradigms for Visualising Parallel Execution of Logic Programs. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 184–202, 1993.

[44] A. GONZÁLEZ and J. TUBELLA. The Multipath Parallel Execution Model for Prolog. In *Proceedings of PASCO'94 (Parallel Symbolic Computation Conference)*, pages 426–432. World Scientific Publishing, 1994.

[45] S. GREGORY. *Parallel Logic Programming in PARLOG*. Addison–Wesley, 1987.

[46] G. GUPTA, K. A. M. ALI, M. CARLSSON, and M. V. HERMENEGILDO. Parallel execution of prolog programs: A survey. Preliminary Version, 1995.

[47] G. GUPTA, M. V. HERMENEGILDO, E. PONTELLI, and V. SANTOS COSTA. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, Italy, June 1994.

[48] G. GUPTA and B. JAYARAMAN. On Criteria for Or-parallel Execution of Logic Programs. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 737–756. MIT Press, October 1990.

[49] G. GUPTA and E. PONTELLI. Stack-Splitting: a Simple Technique for Implementing Or-Parallelism and And-parallelism on Distributed Machines. In *Proceedings of the Sixteenth International Conference on Logic Programming*. MIT Press, 1999. To appear.

[50] G. GUPTA, E. PONTELLI, and M. V. HERMENEGILDO. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.

[51] G. GUPTA and V. SANTOS COSTA. And-Or Parallelism in Full Prolog with Paged Binding Arrays. In *LNCS 605, PARLE'92 Parallel Architectures and Languages Europe*, pages 617–632. Springer-Verlag, June 1992.

[52] S. HARIDI and S. JANSSON. Kernel Andorra Prolog and its Computational Model. In D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, 1990.

[53] M. V. HERMENEGILDO. An Abstract Machine for Restricted And-Parallel Execution of Logic Programs. In E. Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 25–39. Springer-Verlag, 1986.

[54] M. V. HERMENEGILDO and K. J. GREENE. The &-prolog system: Exploiting independent and-parallelism. *New Generation Computing*, 9(3,4):233–256, 1991.

[55] S. JANSSON and S. HARIDI. Programming Paradigms of the Andorra Kernel Language. In *Proceedings of the 1991 International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.

[56] R. KARLSSON. *A High Performance OR-parallel Prolog System*. SICS Dissertation Series 07, The Royal Institute of Technology, 1992.

[57] J. C. KERGOMMEAUX and P. CODOGNET. Parallel logic programming systems. *Computing Surveys*, 26(3):295–336, September 1994.

[58] J. C. KERGOMMEAUX and P. ROBERT. An Abstract Machine to Implement Or-And Parallel Prolog Efficiently. *The Journal of Logic Programming*, 8(3), May 1990.

[59] Y. KIMURA and T. CHIKAYAMA. An Abstract KL1 Machine and its Instruction Set. In *International Symposium on Logic Programming*, pages 468–477. San Francisco, IEEE Computer Society, August 1987.

[60] R. A. KOWALSKI. Predicate Logic as a Programming Language. In *Proceedings IFIPS*, pages 569–574, 1974.

[61] R. A. KOWALSKI. Algorithm = logic + control. *Communications ACM*, 22(7), July 1979.

[62] Y.-J. LIN and V. KUMAR. An Execution Model for Exploiting And-Parallelism in Logic Programs. *New Generation Computing*, 5(4):393–425, 1988.

[63] Z. LIN. Self-Organising Task Scheduling for Parallel Execution of Logic Programs. In *Proceedings of the 1992 International Conference on Fifth Generation Computer Systems*, pages 859–868. ICOT, 1992.

[64] G. LINDSTROM. Or-Parallelism on Applicative Architectures. In *Proceedings of the 1985 International Logic Programming Symposium*, pages 159–170, July 1984.

[65] E. LUSK et al. The Aurora or-parallel Prolog system. In *International Conference on Fifth Generation Computer Systems 1988*, pages 819–830, epelewski, A. and Hausman, B., Nov 1988. ICOT, Tokyo, Japan.

[66] E. LUSK et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.

[67] J. MONTELIUS. *Penny, A Parallel Implementation of AKL*. PhD thesis, Swedish Institute for Computer Science, SICS, Sweden, May 1997.

[68] J. MONTELIUS and P. MAGNUSSON. Using SIMICS to evaluate the Penny system. In J. Małuszyński, editor, *Proceedings of the International Symposium on Logic Programming (ILPS-97)*, pages 133–148, Cambridge, Oct. 13–16 1997. MIT Press.

[69] R. MOOLENAAR and B. DEMOEN. A parallel implementation for AKL. In *Proceedings of the Programming Language Implementation and Logic Programming: PLILP '93, Tallin, Estonia*, pages 246–261, 1993.

[70] E. MOREL, J. BRIAT, J. C. d. KERGOMMEAUX, and C. GEYER. Side-Effects in PloSys OR-parallel Prolog on Distributed Memory Machines. In *JICSLP'96 Post-Conference Workshop on Parallelism and Implementation Technology for (Constraint) Logic Programming Languages, Bonn, Germany*, September 1996.

[71] K. MUTHUKUMAR and M. V. HERMENEGILDO. The CDG, UDG, and MEL Methods for Automatic Compile-time Parallelization of Logic Programs for Independent And-parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 221–237. MIT Press, June 1990.

[72] L. M. PEREIRA, L. MONTEIRO, J. CUNHA, and J. APARíCIO. Delta Prolog: a distributed backtracking extension with events. In E. Shapiro, editor, *Third International Conference on Logic Programming, LNCS 225*, pages 69–83. Springer-Verlag, 1986.

[73] E. PONTELLI. Adventures in parallel logic programming, 1996. Available at http://www.cs.nmsu.edu/~epontell/adventure/.

[74] J. PROTIC, M. TOMASEVIC, and V. MILUTINOVIC. Distributed Shared Memory: Concepts and Systems. *IEEE Parallel and Distributed Technolog*, 4(2), 1996.

[75] R. ROCHA, F. SILVA, and V. SANTOS COSTA. YapOr: an Or-Parallel Prolog System based on Environment Copying. Technical report, DCC-97-14, DCC - FC & LIACC, Universidade do Porto, December 1997. (submitted for publication).

[76] V. SANTOS COSTA and R. BIANCHINI. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of the EUROPAR'98*, pages 831–841, Sep 1998.

[77] V. SANTOS COSTA, R. BIANCHINI, and I. C. DUTRA. Evaluanting Parallel Logic Programming Systems on Scalable Multiprocessors. In *Proceedings of the 2nd International Symposium on Parallel Symbolic Computation (PASCO'97)*, pages 58–67, July 1997.

[78] V. SANTOS COSTA, R. BIANCHINI, and I. C. DUTRA. Evaluating the impact of coherence protocols on parallel logic programming systems. In *Proceedings of the 5th EUROMICRO Workshop on Parallel and Distributed Processing*, pages 376–381, 1997. Also available as technical report ES-389/96, COPPE/Systems Engineering, May, 1996.

[79] V. SANTOS COSTA, D. H. D. WARREN, and R. YANG. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice*

of Parallel Programming, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[80] V. SANTOS COSTA, D. H. D. WARREN, and R. YANG. The andorra-i preprocessor: Suporting full prolog on the basic andorra model. In *1991 International Conference on Logic Programming*. MIT Press, June 1991.

[81] SANTOS COSTA, V., BIANCHINI, R., and DUTRA, I. C. Parallel logic programming systems on scalable architectures. *Journal of Parallel and Distributed Computing*, 1999. to appear.

[82] M. SCHNEIDER. Links on objects and components/prolog, 1999. Available at http://www.cetus-links.org/oo_prolog.html.

[83] E. SHAPIRO, editor. *Concurrent Prolog–Collected Papers*. MIT Press, 1987.

[84] K. SHEN. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Proceedings of the 1992 Joint International Conference and Symposium on Logic Programming*, 1992.

[85] K. SHEN. Memory Management and Goal Scheduling for Parallel Execution Revisited. Internal Report, Bristol University, 1993.

[86] K. SHEN. Overview of DASWAM: Exploitation of Dependent And-parallelism. *The Journal of Logic Programming*, 29(1–3), 1996.

[87] M. G. SILVA, I. C. DUTRA, R. BIANCHINI, and V. SANTOS COSTA. The Influence of Computer Architectural Parameters on Parallel Logic Programming Systems. In Gopal Gupta, editor, *Workshop on Practical Aspects of Declarative Languages (PADL99), LNCS 1551*, pages 122–136, January 1999.

[88] D. A. SMITH. MultiLog: Data Or-Parallel Logic Programming. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 314–331. MIT Press, 1993.

[89] L. STERLING and E. SHAPIRO. *The Art of Prolog*. MIT Press, 1994.

[90] J. SUNBERG and C. SVENSSON. MUSE TRACE: A Graphic Tracer for Or-Parallel Prolog. Technical report, SICS, 1990.

[91] J. TANAKA, K. UEDA, T. MIYAZAKI, A. TAKEUCHI, Y. MATSUMOTO, and K. FURUKAWA. Guarded Horn Clauses and Experiences with Parallel Programming. In *1986 Proceedings Fall Joint Computer Conference*, pages 948–954. IEEE Computer Society Press, November 1986.

[92] E. TICK. Compile Time Granularity Analysis for Parallel Logic Programming Systems. *New Generation Computing*, 7(2,3):325–337, 1990.

[93] K. UEDA. *Guarded Horn Clauses*. PhD thesis, University of Tokyo, 1986. Ph.D. thesis.

[94] K. UEDA and M. MORITA. A New Implementation Technique for Flat GHC. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 3–17. MIT Press, June 1990.

[95] P. VAN ROY. *Can Logic Programming Execute as Fast as Imperative Programming?* PhD thesis, University of California at Berkeley, November 1990.

[96] P. K. VARGAS and C. F. R. GEYER. Introduzindo o paralelismo ou na programação em lógica com restrições. *IX Simpósio Brasileiro de Arquitetura de Computadores e processamento de Alto Desempenho*, 1997. Campos do Jordão, SP.

[97] P. K. VARGAS and C. F. R. GEYER. Uma poltica de escalonamento distribuda no modelo pclp(fd) de explorao de paralelismo ou. *XXIV Conferência Latino Americana de Informática/V Encontro Chileno de Computação da Sociedade Chilena de Computação (CLEI - PANEL'98)*, 1998. Quito, Ecuador.

[98] P. K. V. VARGAS, C. F. R. GEYER, and I. C. DUTRA. TAMAGOSHI - Plataforma para Avaliação de Escalonamento de Tarefas em Programação em Lógica Paralela. In *X Simpósio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1998. Búzios, RJ.

[99] R. VAUPEL, E. PONTELLI, and G. GUPTA. Visualization of And/Or-Parallel Execution of Logic Programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 271–285, 1997.

[100] von LAHUIZEN, M. P. Parallel Logic Programming Techniques. Technical report, Delft University of Technology, PDS-1998-003, February 1998. Available at http://pds.twi.tudelft.nl/reports/.

[101] D. H. D. Warren. An Abstract Prolog Instruction Set. Technical Note 309, SRI International, 1983.

[102] D. H. D. WARREN. Or-Parallel Execution Models of Prolog. In *TAPSOFT'87, The 1987 International Joint Conference on Theory and Practice of Software Development, Pisa, Italy,* pages 243–259. Springer-Verlag, March 1987.

[103] D. H. D. WARREN. The SRI model for or-parallel execution of Prolog—abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102, 1987.

[104] R. YANG. *P-Prolog a Parallel Logic Programming Language*, volume 9. World Scientific, Singapore, 1987. Series in Computer Science.

[105] R. YANG, T. BEAUMONT, I. DUTRA, V. SANTOS COSTA, and D. H. D. WARREN. Performance of the Compiler-Based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.

[106] R. YANG, V. SANTOS COSTA, and D. H. D. WARREN. The Andorra-I Engine: A parallel implementation of the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 825–839. MIT Press, 1991.

[107] G. YAOQING et al. Intelligent Scheduling And- and Or- Parallelism in the Parallel Logic Programming System RAP/LOP-PIM. In *Proceedings of the 1991 International Conference on Parallel Processing*, August 1991.