The APPELO Project
Parallel Environment for Logic Programming

Claudio F. R. Geyer
Universidade Federal do Rio Grande do Sul
Instituto de Informática

Jorge L. V. Barbosa, Cristiano A. Costa and Adenauer C. Yamin
Universidade Católica de Pelotas
Escola de Informática

Inês C. Dutra
Universidade Federal do Rio de Janeiro
COPPE/Sistemas

Vítor M. M. Santos Costa and Fernando M. A. Silva
Universidade do Porto
Laboratório de Inteligência Artificial e Ciência de Computadores

Gopal Gupta and Enrico Pontelli
New Mexico State University
Laboratory for Logic, Database, and Advanced Programming

Abstract

This work focus on the APPELO project supported by CNPq (Protem-III/CC). Our APPELO project tackled three main aspects of parallel logic programming: the parallel runtime environment, compile-time analysis tools and post-run analysis tools. Regarding the parallel runtime environment we performed a thorough performance analysis of existing parallel logic programming systems by using a simulator of a distributed-shared memory (DSM) architecture. Our performance analysis confirms that our proposed runtime environment, DAOS – Distributing And/Or in Scalable machines, that combines DSM techniques with message passing will certainly obtain good performance on a distributed environment. We also proposed and analysed several scheduling strategies, Tamagoshi, Penelope and DSLP, necessary for the efficient distribution of work among the processors in our system. We also designed and developed a compile-time analysis tool, GRANLOG, that is responsible for annotating Prolog programs with useful information such as grain sizes of goals and clauses to guide scheduling decisions. In order to be able to identify bottlenecks and opportunities for optimising our parallel runtime environment, specially the schedulers, we designed and developed VisAll, a tool to visualise the parallel execution of Prolog programs. We also defined a benchmark set composed of Prolog programs taken from academia and from industries. We believe that our integrated system can run logic programs efficiently on parallel platforms offering to the user a very low cost software development environment.

Keywords: logic programming, parallelism, and-or parallelism, static analysis, scheduling, visualisation.

1 Introduction

Parallel processing is a very important alternative to obtain good performance from parallel programs. However in order to obtain good performance from parallel programs, programmers need to deal with a very complex programming model. Therefore it is very common to find that the development of parallel systems/programs take much longer than the development of their sequential versions. Our work focus on the **implicit** parallelisation of **logic programs**. Logic programming presents a declarative programming model that contrasts with the popular imperative programming model in that 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, i.e., programmers write their sequential programs and can run them efficiently in a parallel environment without any modification to the source code. This clearly reduces the cost of software development, as the user writes only one program that will run on one processor just as well as in several processors. Our work concentrates on developing techniques to efficiently implement parallel logic programming systems.

The rapidly growing interest in logic programming, and in parallel logic programming systems can be confirmed with the quality of papers presented in the most important conferences in the area: International Logic Programming Symposium (ILPS), International Conference on Logic Programming (ICLP), and Joint International Conference and Symposium on Logic Programming (JICSLP). Specific workshops on Parallelism and Implementation Techniques held with these main conferences have been very successful and produced published books with selected papers [47]. The Practical Applications of Prolog (PAP) conference and Practical Aspects of Declarative Languages Workshop also confirm the interest of industries on the use of Prolog as a language to software development. Many other important conferences (PLDI, PPOPP, Europar, PLILP and others) have published very good quality papers in the development of sequential or parallel logic programming systems. Lastly, the Journal of Logic Programming (JLP) is the specialised journal in the area.

Our main goal in having a parallel logic programming environment is to allow programmers to faster develop 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. The benchmark set we defined consists of such programs.

Logic programming exposes two main forms of parallelism that can be exploited implicitly:

- Or-parallelism: arises when we allow the parallel execution of several clauses that match a goal in the body of a clause.
- And-parallelism: that arises when we allow parallel execution of goals in the body of a clause. And-parallelism is further classified in dependent and-parallelism (DAP), when goals that share variables can proceed in parallel; and independent and-parallelism (IAP), when goals are only executed in parallel if they do not share variables.

In the literature we can find several systems that exploit only and-parallelism [61, 59], only or-parallelism [2, 77, 20, 108] or a combination of and- and or-parallelism. Systems that combine both and- and or-parallelism are still under research [55, 111, 112, 75, 76], 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 [79]. 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. Therefore, it is our objective to develop a parallel software that can run efficiently on scalable architectures such as the NCP2 [3] or the Alewife [1].

Parallel logic programming systems have been evolved in three main frontiers: (1) new algorithms and data structures for integrating both and-parallelism and or-parallelism, (2) efficient techniques to dynamically distribute work among the processors, and (3) compile-time analysis support for efficient parallelisation. Our work tackles several of these issues focusing on static analysis tools, post-run analysis tools, and implementation strategies for the runtime environment, including smart scheduling strategies.

Among algorithms and data structures to support or-parallelism, we can cite several research work in the literature, for shared memory architectures [71, 2], and for distributed memory [29, 73, 20, 77] architectures. As mentioned before, the most popular systems were implemented for shared-memory architectures because of the simpler programming model. These systems use either stack sharing (binding array model [108]) or stack copying [2] in order to maintain multiple bindings for different branches of the search tree.

Among the systems that exploit independent and-parallelism, we can mention &-Prolog [62], and &ACE [59], both implemented for shared-memory machines.

Systems that implement dependent and-parallelism are DASWAM [90], Parlog [52], AKL [60] and &ACE [59], also implemented for shared-memory machines.

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

- Andorra-I [87]: exploits or-parallelism a la Aurora [71] and determinate and-parallelism, i.e., goals that match at most one clause in the program are executed firstly and in parallel.
- Penny [74]: a parallel implementation of AKL [60][66] that also exploits or-parallelism.
- ParAKL [76]: another parallel implementation of AKL.
- ACE [55]: a system that integrates independent and-parallelism a la &-Prolog and or-parallelism a la Muse [58, 85].
- SBA [31]: another system that integrates independent and-parallelism a la &ACE with or-parallelism a la Aurora. 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 [54].

All these systems were designed and developed for shared-memory architectures. Several distributed memory PLPs have been proposed. Some support pure IAP [107], or pure ORP [19, 77]. Both ANDP and ORP are supported in Conery's And/Or model [30], in the dataflow based models [67], and in the Distributed Andorra Prolog [16].

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. Distributed implementations of parallel logic programming systems that combine both and-parallelism with or-parallelism require 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 [79]). In this work we propose a new parallel logic programming system that integrates the shared-memory programming model with the distributed-memory programming model, in order 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 by the authors 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 [86, 85, 84, 89, 25, 24, 23]. Our experiments also show that architectural parameters can affect significantly the performance of parallel logic programming systems [93, 92, 36]. These results confirm that our parallel runtime model designed to run in DSM platforms can obtain maximum performance from scalable architectures such as the NCP2 and Alewife already mentioned before.

Regarding work distribution, several research work have been done on the implementation of efficient techniques for distributing work among the processors during parallel execution of Prolog programs. The execution of Prolog programs tends to generate irregular computational patterns where work (both and-work and or-work) is created dynamically. Therefore, the scheduler is an important part of any parallel logic programming system that aims at efficiency. Several schedulers were written for distributing or-work in Aurora, for example: Argonne [21], Wavefront [18], Manchester [22], Bristol [15] and Dharma [95]. Each one of them deals with some kind of scheduling problem: dispatching on topmost or bottom-most work, parallel execution of side-effect predicates, execution of speculative work, application-dependent strategies etc.

In the context of our project, several scheduling strategies were implemented in order to maximise performance, minimise frequency and number of communication messages while keeping locality [105, 33, 28]. In order to develop and test these strategies simultaneously with the DAOS prototype, they were either simulated or implemented on our available parallel logic programming systems: Plosys [77] and Andorra-I [87].

Regarding compile-time analysis tools for parallelisation support, we concentrated on granularity and complexity analysis. Our work is based on the GRANLOG model [14] that extends and generalises previous models proposed in the literature [99, 38, 37]. 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 choicepoint, 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 [42, 43, 45, 46]. In the context of our work three research works were carried out utilising GRANLOG information: (1) the centralised scheduler used by the Plosys system was modified in order to support GRANLOG information [49, 50], (2) the Andorra-I reconfigurer [44] and the Andorra-I engine were modified to support GRANLOG information [65], (3) our DSLP scheme (Distributed Scheduling for Logic Programming) was designed and developed using information provided by GRANLOG [33].

Our preliminary 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).

Our work is organised as follows. Section 2 presents our DAOS (Distributing And/Or in Scalable machines) runtime environment describing the engine used, main data structures, memory management, algorithms to combine and-parallelism with or-parallelism and scheduling issues. Section 3 presents our compile-time analysis tools that generates useful information to efficiently exploit parallelism and guide scheduling decisions. Section 4 presents our post-run tool useful to visualise the parallel execution of Prolog programs and identify sources of overheads and opportunities for improving performance, mainly in the schedulers. Section 5 gives a brief description of the benchmark set we defined for Prolog. Last, section 6 presents our conclusions and future work.

2 Runtime Environment: DAOS

Parallel logic programming (PLP) systems should serve the broadest range of applications to become popular. To achieve this goal, PLP systems should support both ANDP and ORP for existing Prolog applications. In practice, exploiting just one form of implicit parallelism requires sophisticated system design and exploiting distinct forms of parallelism at the same time is even harder.

Arguably, shared memory PLP systems have been the most successful and widespread so far as mentioned before.

Several distributed memory PLPs have been proposed. In some models, computation nodes receive messages with goals to solve, and send answers to these goals. Unfortunately, sending goals and receiving solutions is expensive and may become a bottleneck to the system. Another alternative is to copy segments of the execution stacks between workers, as examplified by Opera [19, 77]. Because stacks can grow to be rather large, incremental copying of stacks is used to reduce message size. Although these systems perform well for ORP, they have difficulties with IAP, as IAP execution cannot be represented by a simple stack [91].

The differences between shared memory and message-based machines have become less significant in the last few years due to distributed shared memory systems (DSMs) that gives a shared memory abstraction. Work on PLPs for hardware DSMs has given interesting results. Silva's Dorpp or-parallel system [94] has shown that good performance is possible. More recently, Santos Costa et al [85] performed an extensive analysis of the Andorra-I system on a DASH-like simulated machine. Again, their numbers confirm that most read cache misses result from scheduling and from accessing the code area. Misses to the execution stacks varied from 8% on a ORP parallel application to 30% on an ANDP application. In fact, most of these misses were eviction or cold misses (the authors used 128K cache size). Only the ORP application has significant sharing misses for an execution stack, the choice-point stack, (60% of misses for this stack) and these misses are associated with the fact that this stack is also used for scheduling. We argue that the previous analysis suggests a new approach to distributed PLP systems:

- Most large data structures in these systems are built in the execution stacks. We should take the best advantage of caches to reduce network traffic.
- In contrast, the previous studies show high percentage of sharing misses in scheduler data-structures. This suggests using explicit messages for scheduling.

The *DAOS model: Distributing And/Or in Scalable machines* is the first parallel logic programming model for distributed systems that supports these innovations. The model's main contributions are:

- Using a binding data representation that is both simple to implement on a PLP system and that naturally adapts to DSM techniques. The DAOS model thus innovates over previous distributed logic programming systems while it can still use previously designed synchronisation and scheduling algorithms.
- Applying combined DSM and message passing techniques in the same framework. DAOS innovates over shared memory PLP systems by explicitly addressing the distribution problems inherent to scalable machines.

ORP and IAP are arguably two of the most interesting forms of parallelism available in Prolog programs. Several methods have been proposed to exploit

And/Or parallelism in PLP systems. We propose to use recomputation, according to the C-Tree concept [57]. To understand the motivation for recomputation, consider the following and-parallel query:

```
?- a(X) & b(Y).
```

where & represents a parallel conjunction. Figure 1 shows that both goals have several solutions. After running the goals `a(X)` and `b(Y)` in parallel, one needs to combine the solutions. One solution is to use a special node, that maintains pointers to each solution for each goal. Solutions to the conjunction are obtained by calculating the cross-product between values for `X` and for `Y`. This approach is known as *reuse* as presented in Figure 1(a) , because solutions to each goal are reused several times while calculating the cross-product.

Recomputation-based models are based on having an Or search tree, such that and-parallel goals correspond to building bits of the search tree in advance. Thus, when one starts `a(X)` and `b(X)` in parallel, the solution to `b(X)` is a continuation branch for `a(X)`, and is thus *associated with a specific solution for* a(X), as shown in Figure 1(b). These models are named recomputation-based because to exploit several alternatives for `a(X)`, the solution for `b(X)` has to be recomputed for each alternative. Recomputation has important advantages, namely, avoiding the overheads in implementing a cross-product node, and simplifying the support of Full Prolog, as semantics are closer. The price is that reuse saves some effort in recomputing goals to the right. There have been arguments that for Prolog applications reuse does not reduce much search space over recomputing solutions, as Prolog programmers avoid such situations [57].

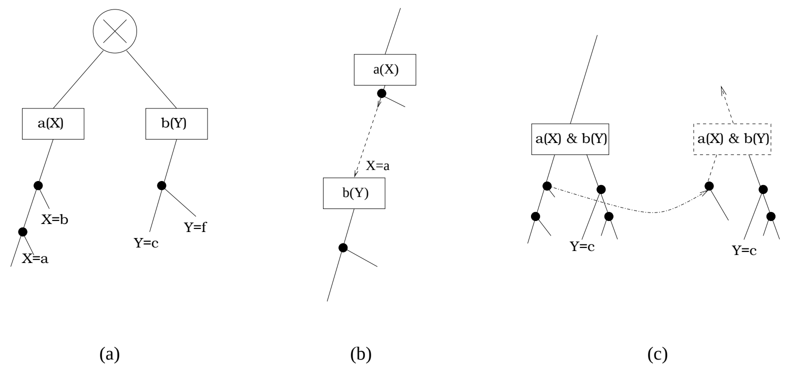

Fig. 1. (a) Reuse (b) Recomputation (c) C-Tree

The C-tree is shown in Figure 1(c). In the C-tree [57], whenever a worker picks an alternative from another process, it creates a copy of all parallel conjunctions found along the way and restarts work to the right. Traditionally, the C-tree has been associated with a notion of *team*, i.e, a group of workers. Normally, IAP work is explored inside a team and ORP work is executed between teams. DAOS gives freedom to system designer to decide if he or she wants to use or not teams. The use of teams has the advantage of simplifying scheduling allowing (re)use of IAP schedulers within teams and ORP scheduler between teams. This organization also simplifies the solution propagation in IAP through the possibility of using multicast inside a group.

DAOS aims at two goals: improved efficiency over traditional distributed systems, and good support for And/Or parallelism under Prolog semantics. It is a fundamental point in DAOS establish which data areas should be *private* to a worker, and which ones should be *virtually shared.* There are two opposite approachs: (a) all stacks must be private as in distributed PLP system, or (b) all stacks must be shared through a DSM layer. The later option seems interesting because we could use a previous implementation to shared memory systems. However, this may be inefficient due to scheduling data structures. DAOS presents a intermediate solution: the areas that implement the major data-structures will be logically shared and the areas that are used to implement work management should be private.

The key problem we address in DAOS is how to implement the virtually shared address space. This shared space contains the major data structures used in Prolog, such as all Prolog variables and compound terms. Because DAOS is designed to support or-parallelism, having a shared space suggests using a binding array (BA) based approach, such as the one used in Aurora [70] and Andorra-I [87]. The original BA was designed for ORP and keeps a private slot for every *variable* in the current search-tree. This slot stores all conditional bindings made by a worker, instead of writing on the shared space. Because all bindings to the shared stacks are indeed conditional, BA based models have the important property that workers can only write to their own address space. Accesses to other memory areas are read-only until this memory is reclaimed. This gives a nice single-writer, multiple-reader pattern for the shared memory.

Unfortunately, the original BA design does not extend to IAP, because one does not know beforehand how many cells each and-goal will need. Management of slots between workers running in and-parallel becomes highly complex [56]. In DAOS, we propose to use the Sparse Binding Array (SBA) [81] to manage bindings to shared variables. The SBA addresses the memory management problems in traditional BAs by *shadowing* the whole shared stacks, that is, *every cell* in a shared stack has a private "shadow" in a worker or team's SBA. Although the SBA was designed to address the problems in combining AND and OR parallelism, it has shown quite good performance even for purely ORP applications [81].

The initial SBA design depends on a shared memory environment: the design organises workers into teams. All workers in a team share the same choice-points, and thus can share the same SBA. Unfortunately, sharing the SBA eliminates the main advantage of the design, as one can expect the SBA to be write-intensive, and to have worse locality than the stacks. In DAOS, we propose a different solution:

– Each worker has a private SBA. SBAs are synchronised, through the trail, both when sharing ORP work and when sharing IAP work.

The *Heap* and *Environment Stack* support forward execution. The *Control Stack* is used both for backtracking and for parallelism. The *Trail* supports backtracking. The *Sparse Binding Array* (SBA) provides access to private bindings. Last, the *Goal Stack* gives goals to be exploited in IAP.

The two largest stacks in Prolog execution store terms and logical variables. Both have data structures that are read-only after sharing and should thus be virtually shared:

– The *Environment stack* stores environments, corresponding to the activation records of traditional imperative languages.
– The *Global stack* or *Heap* accommodates compound terms and the remaining logical variables.

False sharing may happen in two situations (see Silva [94] for the initial discussion of these problems in an ORP context). First, workers can make previous work *public*, and then continue execution on the same stack. In this case, their next updates to the stack might be sent to the sharing workers. Fortunately, such sharing can be easily attacked by standard techniques, such as using a lazy release consistency memory model [48]. Relaxed consistency ensures that new updates from the worker owning the stack segment will only be sent at synchronisation points.

A second source of false sharing is backtracking. In general, when all alternatives have been exploited, both global and environment stack space may be reclaimed, and next reutilised to store new work. Updates to this space may then be sent to the workers which had originally shared the work, unless one uses an invalidate-based protocol.

The two previous stacks take care of Prolog's forward execution. They do not address how to implement backtracking, ORP, and IAP. The *Control* stack stores choice-points, that is, open alternatives. Choice-points include pointers to the stacks and to the goal's arguments before creating the choice-point, plus pointers to the available alternatives. Some ORP systems also store scheduling information in choice-points, such as how many workers are exploiting alternatives from this choice-point. IAP system extend the control stack to also include *parcall frames*, that describe the conjunctions to be executed in parallel.

Originally, the control stack was only used to store alternatives. Then ORP systems had used this stack to manage the available work in the system. For instance, in the `chat-80` ORP-only benchmark running under Aurora, about a third of the sharing misses originated from the control stack [80] (the rest originated from the scheduler data structures). Although a similar study is not available for IAP systems, we expect parcall-frames to also be an important source of sharing misses.

The Trail Stack stores all *conditional* bindings. In a sequential execution this is required to undo bindings to variables when backtracking. In a BA-based system, the trail is a fundamental data-structure, as it is the one used to synchronise bindings between different branches. Since conditional bindings *may not* be placed directly on the stacks, the alternative is to store these bindings in the Trail. When workers fetch work, they read the bindings from the Trail and stored them in the SBA. Thus, the BA or SBA are optimised for access to a variable's binding, and the trail is optimised for installing and deinstalling bindings, as workers move around the search-tree.

In DAOS the first operation to be performed when sharing work is to install all conditional bindings in the SBA, which requires immediate access to corresponding trail section. This opens the question of whether to keep the trail under DSM, or whether to use explicit distribution, as for the Control Stack:

- The Trail is necessary to start work. In fact, all chunks of the Trail must be present in a worker *before* it can start work. This argues for sending the required trail segments immediately when we share choice-points or goals, and thus for a fully-distributed solution.
- The Trail tends to be a relatively small stack, as most bindings are unconditional [97]. After an initial delay, one may take advantage of the sharing facilities in a DSM system to actually have the Trail segments before they are asked. As the Trail is relatively small, this should not result in such heavy traffic. In fact results for Andorra-I showed that there is very little sharing in this area for ORP benchmarks (most misses were cold or eviction misses).
- IAP programs tend to require much larger stacks than ORP programs, as they perform much less backtracking, but they do tend to perform less conditional bindings. One will expect trail segments to grow larger.

A final decision will depend on several factors, such as the efficiency of the DSM system and of the message passing implementation. In the IDAOS implementation, our IAP part of DAOs, we have decided to initially follow a fully distributed implementation because trail copying can be naturally integrated with control-stack copying.

2.1 Scheduling

Work distribution (task scheduling) is one of the most important components of a distributed system. Its objective is to determine an assignment of tasks to processors and to determine the order in which tasks will be executed in order to improve performance by balancing the workload.

We can classify schedulers in parallel logic programming systems that combine and-parallelism with or-parallelism according to its functionality, i.e., to find and-work, to find or-work, or to choose between and-work and or-work available. Therefore we distinguish three kinds of schedulers:

- and-scheduler: chooses one or more goals from a goal list that can be executed in parallel.
- or-scheduler: chooses a choicepoint or alternative to be taken by a processor.

– and-or scheduler: decides what kind of work will be selected: and-work or or-work.

The schedulers are organised this way in order to minimise overheads of searching for work. Therefore, a processor first tries to find work that is closer to it in its subtree, either or-work or and-work. If there is no work available in its subtree, tries to select and-work or or-work from other subtrees.

We designed and developed several strategies to select or-work and to select between and-work and or-work. The DSLP (Distributed Scheduling for Logic Programming) model [33, 32] was designed for the DAOS system. Therefore deals with the problems of selecting between and-work and or-work. DSLP assumes that and-scheduling must be done in a centralised fashion, because of the low granularity of this kind of work and high overhead incurred by distributed algorithms on distributed platforms. On the other hand, or-parallelism is much coarser-grained and can utilise more sophisticated scheduling strategies adapted to the distributed environment.

We developed other strategies to distribute only or-work for the Plosys system and for the pclp(FD) model [103, 104]. **P**arallel **c**onstraint **l**ogic **p**rogramming over finite domains – pclp(FD) – is a model that exploits multi-sequential or-parallelism in a distributed environment. The model assumes that workers (processing elements) are interconnected via a network and executes constraint logic programs [40, 72]. The scheduling strategy is dynamic and totally distributed, based on incremental stack copying [2]. It is also based on Hwang's termination algorithm. In order to evaluate the strategy, the Tamagoshi prototype [105] was designed to simulate synthetic workloads in networks of processors. Tamagoshi simulates two kinds of strategies: one centralised as in the Plosys system and the one proposed for the pclp(FD) model.

The strategy developed for the Plosys system, Penelope [28], replaces the centralised algorithm, used originally, by a hierarchical scheduler that tries to minimise the bottleneck of the centralised scheduler while avoiding the overheads of totally distributed strategies.

The following sections describe Tamagoshi, Penelope and DSLP in more detail.

Tamagoshi The Tamagoshi (**Ta**sk Si**m**ul**a**tor Pro**g**ram and **O**R-Sc**h**eduler **I**nterface) system was designed with the objective of simulating a simplified execution of an or-parallel system. The scheduling protocols were developed independent on the actual abstract machine, what allowed to test the communication protocol between schedulers without interference of the actual execution. The simulator generates synthetic loads that are commonly found in the actual or-parallel Prolog execution. Our first results using a centralised strategy (as the one used in the Plosys system) and a totally distributed strategy shows that for our workloads the totally distributed strategy overperforms the centralised one.

Penelope This section presents Penelope [28], a hierarchical distributed scheduler for the Plosys system. Penelope aims at reducing number and size of messages exchanged between processors as well as to maintain data locality.

In the Plosys system, a Prolog program is executed by several workers. Each worker is designed with three threads: exporter, importer and a Prolog engine. The load of a worker consists of alternatives not yet explored from one or more choicepoints. Each worker sends its own load to the scheduler through the engine. A worker can assume three load status: (1) idle: it does not have any Prolog task to do, (2) quiet: the worker is active, but does not have enough work to share with another idle worker, and (3) overloaded: the worker has amount of work above a threshold, therefore it can share work with other idle workers.

The scheduling strategy applied by the Plosys system was originally centralised. Therefore the scheduler needed to maintain a global state about the system load using the information sent by workers every so often. The system uses full stack copying in order to transfer work from one worker to another. It also chooses the topmost choicepoint to share with other worker, trusting that topmost work is coarser-grained than bottom most work. In order to know what choicepoint is older, the system marks each choicepoint creation with a timestamp. This timestamp corresponds to its depth in the search tree. The strategy adopted by Penelope chooses choicepoints depending on its complexity given by the GRANLOG tool. It also tries to maintain locality by exporting/importing work from workers that already communicated sometime in the past.

Penelope also implements a symmetrically-initiated strategy, where idle workers ask for work, and overloaded workers inform idle workers that there is available work.

In the centralised scheduling model used by Plosys, only one worker is responsible for collecting information about workload in the system. This can produce a significant overhead, when we use a high number of processors. In order to avoid the bottleneck on this worker, we implemented a distributed hierarchical scheduler for the Plosy system.

In order to reduce the number of messages exchanged between processors, we established a partial order among the processors. This way only parent processes communicate with their children and vice-versa. In order to reduce the size of the messages exchanged in the system, we do not implement full stack copying as in the Plosys original design. We perform an incremental stack copying, where workers import/export only parts of the stacks that are not yet common to both workers involved in the communication. The stack copying algorithm used is similar to the one use by the Muse system [2].

The important point about incremental copying is that workers that share work have common parts of the Prolog stacks. In order to achieve this and maintain good locality we adopt a bottom up communication style, where work is transferred to other workers from the leaves to the root of the execution tree.

As all information flow from the leaves to the root of the execution tree, each scheduler keeps information about its descendents and about its parent.

DSLP This section presents a distributed scheduler for exploiting OR-Parallelism and Independent And-Parallelism in Logic Programming [33]. The scheduler is distributed and called DSLP – Distributed Scheduler for Logic Programming. DSLP works on parallel and distributed computers. The proposal uses two main information to aid scheduling: system information, detected before and during the execution and application information, generated statically by GRANLOG (Granularity Analyzer for Logic Programming). Some preliminary results are available.

The basic structure of DSLP is composed of or-nodes and and-nodes that are subordinated to their or-nodes forming a network ring. The or-nodes are composed of two processes: one is the worker and performs Prolog reductions, the second one is the scheduler responsible for choosing or-work for the or-nodes. The and-nodes are composed only of workers and are responsible for executing independent and-parallel work.

The communication protocol is bidirectional with maximum step. The network ring is bidirectional in order that each worker can communicate with its two neighbours. The maximum step is determined from the total number of schedulers in the ring. Whenever a worker sends a message from one worker to the others, the maximum step is packed with the message. Whenever the message arrives to a node it is unpacked and the maximum step is decremented by 1 unit.

The DSLP model has a prototype implemented in PVM (Parallel Virtual Machine) for SUN workstations with ethernet network. Our first results are encouraging and show that the model can exploit efficiently independent and-parallelism.

3 Static Analysis: GRANLOG

Static analysis consists of inferring useful information about a program at compile-time. There are several kinds of static analysis. Among them we can mention:

- Abstract Interpretation [34]: obtaining useful information of a program such as predicate modes, argument types, flow and variable dependencies through a simulation of the program execution using an abstract domain. This abstract domain preserves the general form of the program execution allowing for analysis of data flow and prediction of the dynamic program behaviour.
- Complexity Analysis: obtaining information about computational time and space complexity of programs.
- Granularity Analysis: obtaining grain size of goals and clauses for parallel execution.

In our work we concentrated on these three kinds of analysis and developed four main tools that were integrated to produce useful information to guide program execution and scheduling decisions.

- GRANLOG (GRanularity ANalyzer for LOGic programming): GRANLOG [14, 102, 9, 13] does automatic granularity analysis of Prolog programs generating an annotated program. It uses several advanced static analysis techniques such as Global Analysis [11] via Abstract Interpretation, Granularity Analysis [10] and Complexity Analysis [12];
- PARTY (Parallel Types Analyzer): PARTY [26, 27] is a compile-time tool that infers argument types in Prolog programs using abstract interpretation. It can utilise intervals to determine argument types.
- ORCA (OR Complexity Analyzer): ORCA [12] does complexity analysis of Prolog programs. It generates simplified information that can be utilised for determining or-complexity [12].
- MODA (Mode and Dependency Analyzer): MODA [4] does abstract interpretation of Prolog programs in order to obtain predicate modes and dependencies.

These modules were integrated on a unique module so-called GRANLOG. The GRANLOG system is composed of three modules:

- Global Analysis [11]: This module infers predicate modes, argument types, argument, goals and clauses sizes and dependencies in Prolog programs. PARTY and MODA are integrated in this module [5, 4]. The global analysis module accepts a Prolog program as input and generates an annotated program with predicate modes, argument types, argument, goals and clauses sizes, as well as goal dependencies.
- Granularity Analysis [10]: This module is responsible for inferring grain sizes of goals and clauses that are useful to exploit parallelism and distribute work among the processors. The sizes are inferred based on the dependency information generated by the global analysis module. The granularity analysis module accepts a Prolog program and information generated by the global analysis module as input and generates an annotated program with a description of the grain sizes existing in the program.
- Complexity Analysis [12, 100]: This module does the complexity analysis of Prolog programs. In this module we can find two integrated sub-modules: ORCA [12] and CASLOG [69]. ORCA is responsible for generating simplified information that can be used in the and-analysis or in the or-analysis [12]. CASLOG makes a very sophisticated static analysis of Prolog programs generating complexity information that can be as complex as a recurrent equation [12]. The complexity analysis module accepts an annotated program as input (output of the granularity analysis module) and generates an annotated program with information about complexity based on the granularity information [14].

In order to prove the usefulness of the information provided by the GRANLOG subsystem, we implemented scheduling strategies based on the granularity and complexity analysis generated by the GRANLOG subsystem [65, 49, 50].

3.1 Application of GRANLOG

GRANLOG in Andorra-I In previous works, we showed that dynamic scheduling strategies for distributing and-work and or-work in parallel logic programming systems yielded better performance than a fixed assignment of processors to exploit and-parallelism and or-parallelism. We then suggested that a combination of compile-time granularity information with dynamic scheduling strategies could improve the performance of the system even further [44, 43, 46].

In this work we show that a combination of compile-time granularity information with dynamic scheduling strategies can in fact improve the performance of parallel logic programming systems that use completely dynamic scheduling strategies.

Our target system is Andorra-I [111, 82, 112, 43], a parallel logic programming system that exploits both ORP and ANDP. ANDP in Andorra-I is exploited *determinately*, i.e., only goals that match at most one clause in the program are allowed to proceed in parallel.

In order to obtain compile-time granularity information, we used the ORCA system [8], that generates simplified granularity information based on Tick's algorithm [98] from goals and clauses of a Prolog-like program.

We modified the Andorra-I system to understand the ORCA outputs. Our results show that dynamic scheduling strategies can actually benefit from compile-time granularity information.

The ORCA system is responsible for generating an annotated Prolog program that is later compiled to an abstract code, the Andorra-I VRAM code [88], and provides granularity information about clauses and goals to the Andorra-I engine.

The ORCA system consists of three main parts. The first part reads the Prolog program and creates a table with names of all procedures and their respective clauses. It also generates a list of calls to clauses. The table is organised in a way that each entry contains complexity of the procedure and of the clauses belonging to that procedure. The second part of the model computes the complexity measures for the clauses and procedures. The third part annotates the source program with complexity information.

The algorithm used is based on Tick's algorithm[99] with some modifications to increase precision. Complexity is measured in terms of number of resolutions. In that case, the complexity of a fact in the database is considered as 1. The complexity of a clause is obtained by the sum of the complexities of each goal in the clause plus one call (considering the head call). The complexity of a procedure is given by the sum of the complexities of the clauses belonging to that procedure. The complexity of a recursive call is computed as the complexity of the corresponding procedure by considering each recursive call as having weight 1.

As an example, figure 3.1 shows the annotation for the clauses shown in figure 3.1.

The complexity of `d/2` is given by annotation `'__c_d/2'`. This annotation employs the Prolog syntax. The complexity of `samples` is given by `'__c_samples/2'`.

```
'__c_d/2'(1983,1,[1983],0,[]).
'__c_samples/2'(35,2,[2,33],18,[]).
```

Fig. 2. ORCA OUTPUT EXAMPLE

```
d(Data,Mode):-
scannerdata(Data,R,C,D1,D2),
scanner(Mode,R,C,D1,D2,Image).

samples([],Samples):- !, Samples = [].
samples([S|Spec],Samples):- !,
scannerdata(S,R,C,D1,D2),
Samples = [sample(R,C,D1,D2)|Smpls],
samples(Spec,Smpls).
```

Fig. 3. SOME CLAUSES OF THE `scanner` PROGRAM

The first argument of this annotated code corresponds to the complexity of the procedure, the second represents a list with the complexity of each clause in the procedure, the third argument corresponds to the recursive value of the procedure (if it has a recursive call), the last argument corresponds to a list of complexities for mutually recursive calls. As no one of the clauses shown in figure 3.1 have mutually recursive calls, this last argument is represented as an empty list.

Obviously, the numbers represented in each argument are computed based on the analysis of the whole program that we do not show here.

ORCA gives information on clauses and on goal invocations. This information finds a simple match in the process of Andorra-I compilation:

- The compiler already maintains clause information in order to support the or-scheduler. Currently, information is maintained through the `or_sched_info` instruction, which details how many cuts and commits exist in a clause, and whether the clause terminates in a `fail`. To insert the ORCA extension we just extended the information with the fields in the annotated code for each procedure.
- Each sub-goal invocation in Andorra-I corresponds to a `create` instruction. It is therefore natural to associate ORCA's annotation for each call in the clause to the `create` instruction.

The engine was adapted to access granularity information. This mainly consisted of adapting the loader and supporting the new instruction fields.

As this work concentrates on the reconfigurer, no modification was made in the and-scheduler and or-scheduler related to the work distribution algorithms. Modifications were done only to provide information to the reconfigurer.

The reconfigurer was implemented using two different strategies [44]. One, the efficiency-guided strategy, makes decisions based on past information and on *efficiency* of workers in a team [46]. The other, work-guided strategy, makes decisions based on instant information about the sizes of and-work and or-work available [45].

In this work we concentrated only on the work-guided strategy. The information used by the work-guided strategy assumes some estimates to work sizes. For example, in order to collect amount of and-work, the reconfigurer takes from the and-scheduler the width of the run queue of goals. This means that we do not keep any information about the actual predicted size of the work below that subtree, which may lead to a decision that allocates workers to a team that is working on a short branch of the execution tree. This may cause load imbalance, since other sources of work may need more workers.

Our experiments were done on a Sun SPARCstation-20 with 4 processors. We used only three processors and left one processor for OS duties. We ran 2 versions of Andorra-I. An original version that does not use any kind of compile-time information and a version using compile-time information generated by ORCA.

Our results show that the utilisation of granularity information can help in improving performance. Our improvement ranges from 1% (for the queens application with a 10x10 boardsize, `bqu10`) to 168% (for the cyphertext program, `cypher`). The program `fibonacci` does not have improvement in performance, because it is a deterministic application that contains only and-parallelism. As we do not use the granularity information in the and-scheduler, we observed only the effect of the overhead of supporting ORCA information.

GRANLOG in Plosys This section describes an integration of granularity information generated by GRANLOG in the PLoSys system. As mentioned before, granularity information can help scheduling decisions, in order that better tasks can be assigned to better workers.

The original Plosys system does not use any granularity information in order to guide scheduling decisions. This way, high overhead can be introduced to the system given that small tasks can be exported/imported very often causing a very high communication ratio.

From the information generated by the GRANLOG system, only grain_clause, that corresponds to granularity information about clauses, and grain_goal, that corresponds to the granularity information related to goals, were necessary to perform our work.

The example below shows in boldface, the parameters we used from grain_clause and grain_goal. Note that this information is static. The values are numbers inferred by the GRANLOG system expressed in resolution units.

:- grain_clause(procid, grain_name, mode_list, grain_size_list, **simplified_complexity**, compl_expression, I/O size,).

:-grain_goal(procid, grain_name, mode_list, grain_size_list,type_list, pending_resolvent, **simplified complexity**, **complexity_of_recursions**,**number_of_clauses**, compl_expression, I/O size).

The following parameters are used in our work: Simplified Complexity of a Clause (Cc); Simplified Complexity of a Goal (Cm); Complexity of recursions (Cr); Number of Clauses (Nc); Accumulator for the complexity of the pending resolvent (ACRPL); Complexity of choicepoints (CPE), and the Worker Load Register (RCT).

Cm provides the system with the complexity of a Prolog procedure. Cr corresponds to the simplified complexity of a recursion. If the procedure is not recursive, the value of this parameter is null. The Nc parameter corresponds to the number of clauses of the given procedure. The ACRPL parameter keeps the total sum of the complexities of pending goals for a given execution. The CPE parameter is calculated using Cm, Cc, Nc and ACRPL. The RCT parameter is a register that contains the partial sum of the complexities of choicepoints already created.

Modifications were done in the Plosys engine in order to support manipulation of the information provided by the GRANLOG system. We did experiments with some of our OR-benchmarks (described in section 5), and our results show that the utilisation of static information can significantly improve performance.

4 Post-run Tools: VisAll

4.1 Description

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. Some applications have very low degree of parallelism, either and-parallelism 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.

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 of the parallel systems even when the application has sufficient parallel tasks to supply the number of processors available. In most parallel logic programming systems these problems are directly related to workload distribution strategies since task switching is not a constant time operation [53]. In order to achieve good performance and scalability, parallel systems need to be optimised to the target parallel machine used.

Added to the difficulties posed above, parallel systems developers face the problem of debugging on a parallel environment where the response time is totally non-deterministic due to the interference of operating system operations.

One interesting way of dealing with the problems mentioned before is to provide the user with a visualisation tool that can mirror the parallel execution of the logic program. This work gives a major contribution in this direction with the design and implementation of a flexible visualisation tool for parallel logic programming systems, the **VisAll** system.

VisAll is a *flexible* and *universal* tool, whose main goal is to develop a parallel logic programming system that integrates exploitation of all kinds of parallelism in logic programming. The main goals of VisAll are (1) to allow developers to identify points of the parallel implementation that need to be optimised for the target parallel architecture used, and (2) to help in parallel debugging. Initially, the system was designed to help the development of SBA [31], a parallel logic programming system that exploits both or-parallelism and independent and-parallelism. Currently, VisAll is capable of visualising synthetic tracefiles of the target SBA system and other tracefiles taken from other parallel systems such as Muse, Aurora and &Prolog. The system is flexible because users can specify graphical objects and trace events to be visualised. By universal we mean that VisAll is capable of visualising any kind of major form of parallelism that is exploited in parallel logic programming systems.

Other visualisation tools have been developed to parallel logic programming systems, such as WAMtrace [41], VisAndOr [51], ViMust [96] and VACE [106]. Each one of these systems work for a different parallel logic programming system. VisAll is the first attempt at unifying the visualisation paradigms exhibited by the other visualisation tools adding extra more features important to system optimisation and debugging.

The algorithms used by VisAll do not pose any limit to the size of the tracefiles. The only memory restriction is the computer's.

Figure 4 shows a scheme of the system, with its components and their relationship.

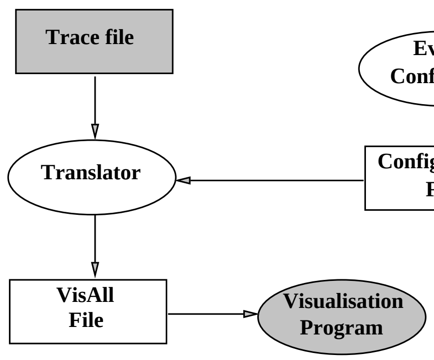

Fig. 4. The VisAll Tool

Figure 5 shows a visualization program example exhibiting all VisAll features.

The visualisation program is implemented on top of Tcl/TK. Currently, it displays the dependency graph contained in a VisAll file, shows some information associated to each event, a legend with information about each icon, and individual execution time for each processor.

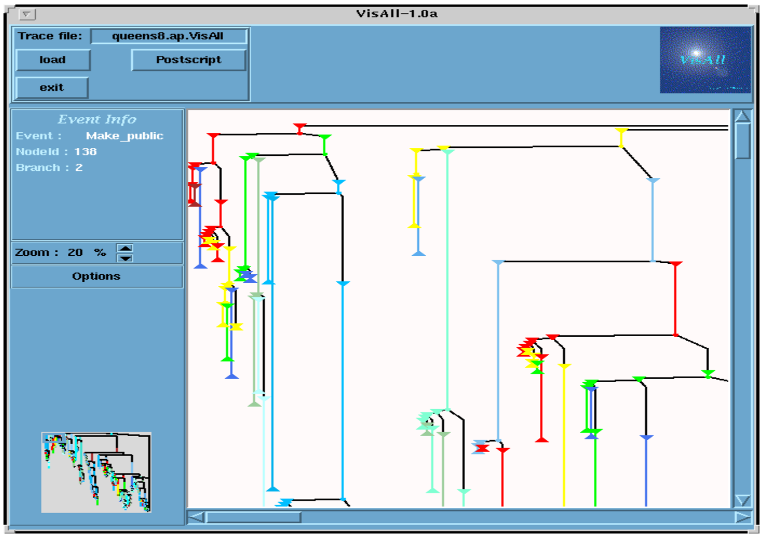

Fig. 5. VisAll - Visualization Interface

The whole graph is reduced and showed in the small picture on the left. The small picture also has a small box indicating the part that is being visualised in the larger image. The image on the right shows the graph at different sizes controlled via a zoom button. When the graph does not fit in the window, scrollbars are used to navigate through the graph. On the left hand side top corner of the graph, VisAll shows information about each event (node). It is enough to click on any node to obtain the information that corresponds to that event.

VisAll also allows the user to associate the processor execution time with the branch of the tree where the worker is positioned. It is enough to click on the worker bar in order to obtain the position where the worker is in the tree.

4.2 Application of VisAll in the Plosys system

In order to apply the VisAll tool to the Plosys system, two new files were created: trace.c e trace.h. These files implement the functions that store the VisAll events. Each worker generates a totally ordered tracefile. The set of tracefiles generated by all workers are merged in a single file also totally ordered. The following Plosys programs were instrumented to generate trace events: wam_engine.c e worker.c. The C application file also must emit an event corresponding to successful completion of a branch.

We faced a problem with the instrumentation given that the Plosys system runs in a totally distributed environment and therefore lacks a global clock. We had two alternatives to implement a global clock: to use synchronisation of the physical clock or synchronisation of a logical clock. The algorithm used to obtain a logical global clock is the one proposed by Lamport [68].

5 Benchmark Set for Prolog

In this section, we define a benchmark set for Prolog that contains programs normally used as benchmarks for sequential Prolog systems and other real applications developed in companies or in academia. The applications offer several opportunities for parallelisation, and presents a varied degree of parallelism. To the best of our knowledge, this is the first attempt to define a benchmark set for parallel Prolog systems.

In our benchmark set, we selected programs with predominantly and-parallelism, with predominantly or-parallelism, programs that present both kinds of parallelism in different phases of the computation and applications that have both kinds of parallelism appearing at the same computational phase. We classify them into four main groups depending on whether the parallelism is high or or low, and whether it is mainly and-parallelism or or-parallelism. For brevity, we call these groups: (1) *high or, high and*, (2) *high or, low and*, (3) *low or, low and*, (4) *low and, low or*, (5) *high and, low or*, (6) *high or*, and (7) *high and*. Group (1) has high degree of or- and and–parallelism, but the amount of or-parallelism is bigger than and-. Group (2) has high degree of or-parallelism with low degree of and-parallelism. Group (3) has low degree of or-parallelism and low degree of and-parallelism, but the amount of or-parallelism is bigger than the amount of and-parallelism. Group (4) has low degree of and- and or-parallelism, but the amount of and-parallelism is bigger than the amount of or-parallelism. Group (5) has high degree of and-parallelism, and low degree of or-parallelism. Group (6) has only or-parallelism. Group (7) includes programs that contain only and-parallelism.

Queens (`bqu`)
This is a queens program written using a Pandora technique [7]. Its or-parallelism arises when trying to solve the `cell` predicate. The problem is to place queens in a board (NxN) in order that no queen attacks each other in any column, row or diagonal. For any boardsize, this program has mainly or-parallelism with little amount of and-parallelism. Depending on the boardsize, this program can fall in groups 2 or 3.

Database Query (`chat_80`)
This is an example from a natural language question-answering system Chat-80. Chat-80 is a natural language question answering system written at the University of Edinburgh by Pereira and Warren [109]. This version of Chat-80 operates on the domain of world geography. The program makes a query to the Chat-80 database. It contains or-parallelism, and it has been used as one of the or-parallel benchmarks for both the Aurora and Muse systems. This program falls into group 6.

Crossword (`cross`)
This is a program that solves crossword problems according to the constraint technique used by van Hentenrick in [101]. A crossword generator was written with the purpose of automatically produce different kinds of crosswords. The crossword generator program accepts as an input the crossboard format,

horizontal words to be placed in the board and vertical words to be placed in the board. In the program generated by the crossword generator, the crossboard itself is expressed by a clause with a huge body defining the shape of the board in terms of horizontal and vertical blank spaces to be filled in the board. Each word is numbered and defined as a database of facts with as many facts for the same word as the number of letters in the word. Depending on the shape of the crossword, this program can fall into groups 3 or 4.

Theorem Prover (`mutest`)

This is a program that solves a problem described in [64]. The problem is to produce the string 'mu' from a given string, through production rules that can be applied in any order. One example of theorem to prove is that 'miiiii' produces 'mu'. If we replace the letters in the strings by numbers, this puzzle solves some interesting problems in number theory. This program contains only or-parallelism. This program falls into group 6.

Road Markings (`road_markings`)

This is a program to extract road markings developed by Hluchoweckyj [63] from the vision research group, in the department of Computer Science at the University of Bristol.

This program is part of a Ph.D. project to extract road markings from real world vision pictures that have been processed to identify a set of visible objects. The program fits straight lines or quadratic curves through these objects. The "best" set of curves is taken as being the road boundaries, where "best" is defined by certain heuristics such as the width between two road boundaries being constant and greater than a vehicle-width apart. This application contains and-parallelism corresponding to parallelising the operations on each pair of objects. We have an example that takes 16 objects as the input data. This program has mainly or-parallelism with little amount of and-parallelism. This program falls into group 5.

Flight Resource Allocation (`flypan`)

This is a version of a naval flying programme generator, initially developed by Software Sciences in the University of Leeds for the Royal Navy. It is an example of a real life resource allocation problem. The program allocates airborne resources (such as aircrafts) whilst taking into account a number of constraints. The problem is solved by using the technique of active constraints as first implemented for Pandora [6]. In this technique, the co-routining inherent to a system such as the Andorra model is used to activate constraints as soon as possible. The program has both or-parallelism, arising from the different possible choices, and and-parallelism, arising from the parallel evaluation of different constraints. The degree of and- and or-parallelism in this program varies according to the queries, but all the queries give rise to more and-parallelism than or-parallelism. Depending on the query, this program falls into groups 4 or 5.

Bit Mapping (`scanner`)

This is a scanner program to reveal the contents of a bitmap. The program is an AKL [66] benchmark developed at SICS. The program reveals the contents of a box (bitmap). The input is the number of dots on each row, column, left diagonal, and right diagonal. The problem was described in [39]. The program contains both and- and or-parallelism corresponding to parallel propagation of determinate bits and evaluation of constraints. This program gives rise to reasonable amounts of both and- and or-parallelism in interleaven phases. This program falls into group 1.

Decyphering Words (`cypher`)

This is a simple substituting decoding system developed by Rong Yang [110]. The system reads an encrypted text and decodes it (assuming the original message is in English). According to the statistics, if one knows about 100 of the most common words, one can respectively understand on average 60% of most text. Now, instead of having an entire English dictionary, we can solve a cipher using only a very small dictionary of common words (about 150 words). Them the program performs a lot of letter matching, first determinately in and-parallel, then when only non-determinate matching is left, the program makes a choice-point (giving rise to or-parallelism). An or-branch fails when the number of unmatchable words exceeds a certain limit. This program falls into group 2.

The length of the ciphertext we have is 56 letters, and the maximum number of unmatchable words is set to 2. This program has mainly or-parallelism with little amount of and-parallelism.

Broadcasting Network (`bcnet`)

This is another real world application developed by the British Telecom company [78]. It was developed on the scope of the BT contract with the Bristol University. The application involves finding paths through a network from a common source node to several destination nodes in such a way that the overall cost is minimised. The cost of using an individual link may reflect factors such as its length, its capacity, the average level of its demand, or the time of day at which it is demanded. Therefore the shortest route might not always be the cheapest. This program falls into group 4.

Floorplan Design (`floorplan`)

This is an application that has several applications, since hardware design till architectural or civil engineering design. The problem consists of placing several components on a surface in order to minimise the total area. Each component can have different shape. Our application deals only with rectangular shapes. The problem is implemented using a branch-and-bound approach with best-first search, and presents a great amount of or-parallelism, that stems from the different branches of the search tree. It belongs to group 6.

Traveling Salesperson 1 (`tsp_or`)

This program implements the classical traveling salesperson problem where we need to find a minimum route from a city A to another city B , visiting all cities on the way exactly once. It is also implemented using a branch-and-bound algorithm with best-first search, and presents a great amount of or-parallelism. This program falls into group 6.

Traveling Salesperson 2 (`tsp`)

This program also implements the classical traveling salesperson problem, but it was implemented in order to have only and-parallelism. This program was developed by the Reform Prolog group, and is used as benchmark in the Reform Prolog system [17]. This program falls into group 7.

Network Clustering (`bt_cluster`)

As an example and-parallel application, we used the clustering algorithm for network management from British Telecom [35]. The program receives a set of points in a three dimensional space and groups these points into *clusters*. Basically, three points belong to the same cluster if the distance between them is smaller than a certain limit. We have input data with 400, 500, and 1000 points. This program has very good and-parallelism, and, being completely determinate, no or-parallelism. This program falls into group 7.

6 Conclusions and Future Work

We performed a thorough performance analysis of existing shared-memory parallel logic programming systems, in order to identify main sources of overheads and opportunities for optimising these systems for distributed-shared memory (DSM) architectures. We have designed and partially developed a full parallel logic programming environment integrating our runtime module, DAOS, our static analysis module, GRANLOG, our post-run analysis module, VisAll, and implemented sophisticated scheduling strategies.

We have proposed a scheme for Distributed And/Or execution in Scalable systems, DAOS. The model innovates by taking advantage of recent work in distributed shared memory technology to obtain efficient sharing, and efficiently supporting both IAP and ORP in a distributed environment. We have found that the DSM mechanism is quite effective in simplifying our design. In fact, we found that using DSM techniques allowed us to focus on the issues that we believe have a major impact in performance.

Work in the IDAOS prototype is progressing at Universidade Federal do Rio Grande do Sul (UFRGS), Federal do Rio de Janeiro (UFRJ) and Porto (UP). Our target is a network of workstations connected by a fast network, such as Myrinet or Fast Ethernet. Both the UFRGS and the UP groups have access to such networks. Changes required to support IAP and ORP have already been contributed to the base system, the `wamcc`, and work will next move on to experimenting with the distributed platform. We expect that after implementing the message mechanism on top of MPI most of the work will move on to scheduler design, as it is traditional in parallel logic programming systems.

One motivation in using the `wamcc` is that the `wamcc` supports the `clp(fd)` constraint system. A parallel implementation of `clp(fd)` requires only minor changes to IDAOS, meaning that IDAOS will support the parallelisation of both logic programming and constraint applications. DAOS was also the motivation from one of the author's research into copy-on-write models, and specifically into the δ-COWL [83]. Copy-based models have lower overheads than BA based models, but in a distributed setting will require more sophisticated DSM mechanisms than what we are currently using.

We have also designed and developed a static analysis tool, GRANLOG, that is capable of inferring predicate modes, argument types and sizes, goal and clause sizes, that are useful for efficiently exploiting parallelism and guiding scheduling decisions.

We presented a way of using compile time information to make better scheduling decisions in the Andorra-I and Plosys systems. The information is provided by the Granlog subsystem, ORCA, incorporated to the Andorra-I and Plosys compilers, and consulted by the schedulers. We described the Andorra-I reconfiguring algorithms and explained which parameters and estimates of sizes of and-work and or-work are used. We claim that we can obtain a much better scheduler by using the information provided by Granlog, since the reconfigurer introduces very little overhead and the utilization of the compile time information is implemented in a very simple way.

We designed and developed **VisAll**, a flexible and universal tool to visualise the parallel execution of logic programs.

VisAll is capable of visualising several kinds of parallelism, and-parallelism and or-parallelism, and displaying individual processor execution times. It can visualise traces of different parallel logic programming systems allowing the user to zoom in and out at parts of the execution tree to see dynamic execution and to obtain information about the logic program source. It integrates two paradigms of visualisation allowing the user to see execution through an execution tree or through individual process execution. VisAll is capable of visualising more sources of parallelism than other visualisers.

We published several papers in international and brazilian conferences, and supervised several undergraduate and graduate students.

Acknowledgments

We would like to acknowledge, Ricardo Bianchini, Denise Bandeira, Patricia Kayser, Luis Fernando Pias de Castro, and Kish Shen for their major contribution and influence to this work. We would like also to thank all students that collaborated to the success of this project.

This work has been supported by the CNPq project Appelo/ProTem-CC. The work presented here has been partially supported by funds granted to LIACC through the Programa de Financiamento Plurianual, Fundação para a Ciência e Tecnologia and Programa PRAXIS.

References

1. A. AGARWAL, R. BIANCHINI, D. CHAIKEN, K. L. JOHNSON, D. KRANZ, J. KUBIATOWICZ, B. LIM, K. MACKENZIE, and D. YEUNG. The MIT Alewife Machine: Architecture and Performance. In *22nd Annual International Symposium on Computer Architecture (ISCA'95)*, June 1995.
2. ALI, K. A. M. and KARLSSON, R. 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.

3. C. L. AMORIM, R. BIANCHINI, G. SILVA, R. PINTO, M. HOR-MEYLL, M. de MARIA, L. WHATELY, and J. BARROS Jr. A Segunda Geração de Computadores de Alto Desempenho da COPPE/UFRJ. In *VII Simpósio Brasileiro de Arquitetura de Computadores, SBAC-PAD*, July 1996.
4. AZEVEDO, S. A. and BARBOSA, J. L. V. (Orientador). Moda: Um analisador estático de modos e dependências para programação em lógica. Projeto de diplomação, Universidade Católica de Pelotas, Pelotas: UCPel, 1998.
5. AZEVEDO, S. C., BARBOSA, J. L . V., GEYER, C. F. R., and CASTRO, L. F. P. Integrao party-granlog: Interpretao abstrata aplicada a paralelizao de programas em lgica. *III Congresso Argentino de Cincia de la Computacion (CACIC 98)*, 1998. Neuquen, Argentina.
6. REEM BAHGAT. Solving Resource Allocation Problems in Pandora. Technical report, Imperial College, Department of Computing, 1990.
7. REEM BAHGAT. *Non-Deterministic Concurrent Logic Programming in Pandora*, volume 37. World Scientific, Singapore, 1993. Series in Computer Science.
8. J. L. V. BARBOSA and C. F. R. GEYER. Análise de Complexidade na Programação em Lógica: Taxonomia, Modelo Granlog e Análise OU. In *XXIII Conferência Latino Americana de Informática, V Encontro Chileno de Computação da Sociedade Chilena de Computação (CLEI – PANEL'97)*, November 1997.
9. BARBOSA, J. L. V. and GEYER, C. F. R. Um modelo para análise automática de granulosidade na programação em lógica. *X SIMPÓSIO BRASILEIRO DE ARQUITETURA DE COMPUTADORES - PROCESSAMENTO DE ALTO DESEMPENHO*, page 639, July 1995. Canela, RS.
10. BARBOSA, J. L. V. and GEYER, C. F. R. Análise de grãos na programação em lógica. *Seminário Integrado de Software e Hardware*, 1996. Recife, PE.
11. BARBOSA, J. L. V. and GEYER, C. F. R. Análise global na programação em lógica. *Simp'osio Brasileiro de Linguagens de Programação*, 1996. Belo Horizonte, MG.
12. BARBOSA, J. L. V. and GEYER, C. F. R. Análise de complexidade na programação em lógica: Taxonomia, modelo granlog e análise ou. *XXIII Conferência Latino Americana de Informática/V Encontro Chileno de Computação da Sociedade Chilena de Computação (CLEI - PANEL'97)*, 1997. ValParaíso, Chile.
13. BARBOSA, J. L. V., WERNER, O, and GEYER, C. F. R. Automatic granularity analysis in logic programming. In *TENTH LOGIC PROGRAMMING WORKSHOP - WLP94*, ZURICH: INSTITUT FOR INFORMATIK DER UNIVERSITE ZURICH, September 1994.
14. BARBOSA, J.L.V. Granlog: Um modelo para análise automática de granulosidade na programação em lógica. Dissertação de mestrado, Universidade Federal do Rio Grande do Sul, 1996.
15. BEAUMONT, A., RAMAN, S. M., SZEREDI, P., and WARREN, D. H. D. Flexible Scheduling of OR-Parallelism in Aurora: The Bristol Scheduler. In *PARLE91: Conference on Parallel Architectures and Languages Europe*, volume 2, pages 403–420. Springer Verlag, June 1991.
16. 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.
17. JOHAN BEVEMYR, THOMAS LINDGREN, and HÄKAN MILLROTH. Reform Prolog: The Language and its Implementation. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 283–298. MIT Press, June 1993.

18. BRAND, P. Wavefront Scheduling. Internal Report, Gigalips Project, 1988.
19. J. BRIAT, M. FAVRE, C. GEYER, and J. CHASSIN. Scheduling of Or-parallel Prolog on a Scalable, Reconfigurable, Distributed-Memory Multiprocessor. In *Proceedings of Parallel Architecture and Languages Europe*. Springer Verlag, 1991.
20. GEYER, C. e CHASSIN, J. BRIAT, J., FABRE, M. Opera: a parallel prolog system and its implementation on supernode. In *International Workshop on Compilers for Parallel Computers*, 1990.
21. BUTLER, R., LUSK, E., McCUNE, W., and OVERBEEK, R. Parallel logic programming for numeric applications. In Ehud Shapiro, editor, *Third International Conference on Logic Programming, London*, pages 375–388. Springer-Verlag, 1986.
22. CALDERWOOD, A. and SZEREDI, P. Scheduling or-parallelism in Aurora – the Manchester scheduler. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 419–435. MIT Press, June 1989.
23. VANUSA MENDITI CALEGARIO. Análise de Desempenho de Sistemas de Programação Lógica. Master's thesis, COPPE/Sistemas, UFRJ, MSc. thesis, Mar 1999.
24. VANUSA MENDITI CALEGARIO and INÊS DE CASTRO DUTRA. Parallel Conventional Systems versus Parallel Logic Programming Systems on Distributed Shared Memory Architectures. Technical Report ES-489/98, COPPE/Systems Engineering and Computer Science, Dezembro 1998.
25. CALEGARIO, V. M. and DUTRA, I. C. Performance Comparison between Conventional and Logic Programming Systems. Technical Report ES-478/98, COPPE/Systems Engineering and Computer Science, Setembro 1998.
26. CASTRO, L. F. P. Um modelo analisador estático baseado na interpretação abstrata direcionado à paralelização de programas em lógica. Dissertação de mestrado, Universidade Federal do Rio Grandedo Sul, 1997.
27. CASTRO, L. F. P. and GEYER, C. F. R. ParTy – A Parallel-Types Analyser. together with the ILPS'97, Oct 1997.
28. CENTENO, A. P. B. and GEYER, C. F. R. Penelope: Um modelo de escalonador hierrquico para o sistema plosys. In *X Simpsio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1998. Búzios, RJ.
29. CLOCKSIN, W. F. Principles of the DelPhi parallel inference machine. *Computer Journal*, 30(5):386–392, 1987.
30. J.S CONERY. *The AND/OR Process Model for Parallel Interpretation of Logic Programs*. PhD thesis, University of California at Irvine, 1983.
31. CORREIA, M. E., SILVA, F. M. A., and SANTOS COSTA, V. The SBA: Exploiting orthogonality in OR-AND Parallel Systems. In *Proceedings of the 1997 International Logic Programming Symposium*, Outubro 1997. Also published as Technical Report DCC-97-3, DCC - FC & LIACC, Universidade do Porto, April, 1997.
32. COSTA, C. A. Uma proposta de escalonamento distribudo para a explorao do paralelismo na programao em lgica. Dissertação de mestrado, CPGCC - UFRGS, July 1998.
33. COSTA, C. A. and GEYER, C. F. R. 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.
34. P. COUSOT and R. COUSOT. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In *ACM SYMP. ON PRINCIPLES OF PROGRAMMING LANGUAGES, 4*. ACM Press, 1977.

35. BARRY CRABTREE. A clustering system to network control, British Telecom, March 1991.
36. MÁRCIO GON¸CALVES da SILVA. Influência de Parâmetros Arquiteturais em Sistemas Paralelos Programação Lógica. Master's thesis, COPPE/Sistemas, UFRJ, MSc. thesis, Mar 1999.
37. DEBRAY, S. K., GARCA, P. L., HERMENEGILDO, M., and LIN, N. Estimating the computacional cost of logic programs. In *International Static Analysis Symposium*, volume 864 of *Lecture Notes in Computer Science*, Namur, Belgium, September 1994. Springer-Verlag.
38. DEBRAY, S. K. and LIN, N. Cost analysis of logic programs. *ACM Transactions on Programming Languages and Systems*, 15(5):826–875, November 1993.
39. A. K. DEWDNEY. Mathematical Recreations – A Compendium of Math Abuse from around the World. *Scientific American*, 263(5):142–145, September 1990.
40. DANIEL DIAZ and PHILIPPE CODOGNET. A Minimum Extension of the WAM for clp(FD). In *10th International Conference on Logic Programming*, pages 774–792. The MIT Press, 1993.
41. TERRENCE DISZ and EWING 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.
42. 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.
43. 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. Also available as technical report CSTR-94-09, from the Department of Computer Science, University of Bristol, England.
44. 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. available at http://www.cos.ufrj.br/∼ines.
45. I. C. DUTRA. Performance Analysis of a Strategy to Distribute And-work and Or-work in Parallel Logic Programming Systems. In *Proceedings of the VIII Brazilian Symposium on Computer Architecture and High Performance Processing – SBAC-PAD*, pages 449–463, 1995.
46. 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.
47. INÊS C. DUTRA, V. S. COSTA, M. CARRO, E. PONTELLI, G. GUPTA, and F. SILVA, editors. *Parallelism and Implementation of Logic and Constraint Logic Programming*. Nova Science, Inc., 1999.
48. C. AMZA et. al. TreadMarks: Shared memory computing on networks of workstations. *IEEE Computer*, 19(2):18–28, February 1996.
49. FERRARI, D. N. O uso e implementação de informações de granulosidade no plosys. Trabalho individual, Universidade Federal Do Rio Grande Do Sul, 1998.
50. FERRARI, D. N., VARGAS, P. K., GEYER, C. F. R., and BARBOSA, J. L. V. Modelo de integração plosys-granlog: Aplicação de 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'98)*, 1999. Submitted.

51. 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.
52. GREGORY, S. *Parallel Logic Programming in PARLOG*. Addison–Wesley, 1987.
53. GOPAL GUPTA and BHARAT 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.
54. GOPAL GUPTA and VíTOR 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.
55. GUPTA, G., HERMENEGILDO, M., PONTELLI, E., and SANTOS COSTA, V. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proc. ICLP'94*, pages 93–109. MIT Press, 1994.
56. GUPTA, G., HERMENEGILDO, M., and SANTOS COSTA, V. And-Or Parallel Prolog: A Recomputation based Approach. *New Generation Computing*, 11(3,4):770–782, 1993.
57. GUPTA, G. and HERMENEGILDO, M. V. Recomputation based Implementations of And-Or Parallel Prolog. In *International Conference on Fifth Generation Computer Systems 1992*, pages 770–792. ICOT, Tokyo, Japan, June 1992.
58. GUPTA, G., HERMENEGILDO, M. V., PONTELLI, E., and SANTOS COSTA, V. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, Italy, June 1994.
59. GUPTA, G., PONTELLI, E., and HERMENEGILDO, M. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.
60. HARIDI, S. and JANSSON, S. Kernel Andorra Prolog and its Computational Model. In D.H.D. Warren and P. Szeredi, editors, *Proceedings of the Seventh International Conference on Logic Programming*, pages 31–46. MIT Press, 1990.
61. MANUEL HERMENEGILDO. An Abstract Machine for Restricted And-Parallel Execution of Logic Programs. In Ehud Shapiro, editor, *Proceedings of the Third International Conference on Logic Programming*, pages 25–39. Springer-Verlag, 1986.
62. HERMENEGILDO, M. V. and GREENE, K. &-Prolog and its Performance: Exploiting Independent And-Parallelism. In *Proceedings of the Seventh International Conference on Logic Programming*, pages 253–268. MIT Press, June 1990.
63. LYDIA T. HLUCHOWECKYJ. *Recognising Road Marking Structures for Autonomous Vehicle Navigation using Computer Vision*. PhD thesis, University of Bristol, Bristol, England, 1993.
64. DOUGLAS R. HOFSTADTER. *Gödel, Escher, Bach: an eternal golden braid*. Harmondsworth: Penguin, 1980.
65. I. C. DUTRA and V. SANTOS COSTA and J. L. V. BARBOSA and C. F. R. GEYER. Using Compile-Time Granularity Information to Support Dynamic Work Distribution in Parallel Logic Programming Systems. available from http://www.cos.ufrj.br/ ines/projs/ai-orca.ps, June 1998.
66. JANSSON, S. and HARIDI, S. Programming Paradigms of the Andorra Kernel Language. In *Proceedings of the 1991 International Logic Programming Symposium*, pages 167–186. MIT Press, October 1991.
67. Peter Kacsuk and Michael J. Wise, editors. *Implementations of Distributed Prolog*. Wiley, Series in Parallel Computing, 1992.

68. LESLIE LAMPORT. Time, Clocks and the ordering of Events in a Distributed System. *Communications of the ACM*, 21(7), Jul 1993.
69. LIN, N. Automatic Complexity Analysis of Logic Programs. PhD Thesis, Department of Computer Science, University of Arizona, Tucson: University of Arizona, 1993.
70. LUSK, E. 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.
71. LUSK, E. et al. The Aurora or-parallel Prolog system. *New Generation Computing*, 7(2,3):243–271, 1990.
72. KIM MARRIOT and PETER J. STUCKEY. *Programming with Constraints.* MIT Press, 1998.
73. HIDEO MASUKAWA, KOUICHI KUMON, AKIHIRO ITASHIKI, KEN SATOH, and YUKIO SOHMA. "Kabu-Wake" Parallel Inference Mechanism and Its Evaluation. In *1986 Proceedings Fall Joint Computer Conference*, pages 955–962. IEEE Computer Society Press, November 1986.
74. JOHAN MONTELIUS. *Penny, A Parallel Implementation of AKL.* PhD thesis, Swedish Institute for Computer Science, SICS, Sweden, May 1997.
75. MONTELIUS, J. Penny, A Parallel Implementation of AKL. In *ILPS'94 Post-Conference Workshop in Design and Implementation of Parallel Logic Programming Systems, Ithaca, NY, USA*, November 1994.
76. MOOLENAAR, R. and DEMOEN, B. A parallel implementation for AKL. In *Proceedings of the Programming Language Implementation and Logic Programming: PLILP '93, Tallin, Estonia*, pages 246–261, 1993.
77. E. MOREL, J. BRIAT, J. CHASSIN de 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.
78. DEWI MUNAF and BRIAN TESTER. And/Or Parallel Programming in Practice. Technical Report, Project 1251:WP 12:1203, British Telecom, November 1993.
79. PROTIC, TOMASEVIC, and MILUTINOVIC. Distributed Shared Memory: Concepts and Systems. 4(2), 1996.
80. V. SANTOS COSTA and R. BIANCHINI. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of Europar 1998, Southampton, UK*, 1998.
81. V. SANTOS COSTA, MANUEL EDUARDO CORREIA, and FERNANDO SILVA. Performance of Sparse Binding Arrays for Or-Parallelism. In *Proceedings of the VIII Brazilian Symposium on Computer Architecture and High Performance Processing – SBAC-PAD*, August 1996.
82. V. SANTOS COSTA, D. H. D. WARREN, and R. YANG. The Andorra-I Preprocessor: Supporting full Prolog on the Basic Andorra model. In *Proceedings of the Eighth International Conference on Logic Programming*, pages 443–456. MIT Press, June 1991.
83. VíTOR SANTOS COSTA. Cowl: Copy-on-write for logic programs. In IEEE Press, editor, *Proceedings of the IPPS99*, May 1999.
84. SANTOS COSTA, V. and BIANCHINI, R. Optimising Parallel Logic Programming Systems for Scalable Machines. In *Proceedings of the EUROPAR'98*, pages 831–841, Sep 1998.

85. SANTOS COSTA, V., BIANCHINI, R., and DUTRA, I. C. 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.
86. SANTOS COSTA, V., BIANCHINI, R., and DUTRA, I. C. 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.
87. SANTOS COSTA, V., WARREN, D. H. D., and YANG, R. 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 PPOPP*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.
88. SANTOS COSTA, V. and YANG, R. Andorra-I user's guide and reference manual. Technical report, University of Bristol, Computer Science Department, Sept 1990. Internal Report, Gigalips Project.
89. SANTOS COSTA, V. S. and BIANCHINI, R. and DUTRA, I. C. Parallel Logic Programming Systems on Scalable Architectures. *Journal of Parallel and Distributed Computing*, 1999. Submitted.
90. KISH SHEN. An Overview of DASWAM — An Implementation of DDAS. Technical Report CSTR-92-08, Computer Science Department, University of Bristol, 1992.
91. KISH SHEN and MANUEL HERMENEGILDO. On the virtues of spaghetti. In *ICLP'93 Postconference Workshop on Practical Implementations and Systems Experience*, 1993.
92. MÁRCIO G. SILVA, INÊS C. DUTRA, RICARDO BIANCHINI, and VíTOR SANTOS COSTA. The Influence of Parallel Computer Architectural Parameters on Parallel Logic Programming Systems. Technical Report ES-477/98, COPPE/Systems Engineering and Computer Science, Setembro 1998.
93. MÁRCIO G. SILVA, INÊS C. DUTRA, RICARDO BIANCHINI, and VíTOR 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), Lecture Notes in Computer Science 1551*, pages 122–136, January 1999. Also available as Technical Report ES/477-98, COPPE Systems Engineering, Sep/98.
94. SILVA, F. M. A. *An Implementation of Or-Parallel Prolog on a Distributed Shared Memory Architecture*. PhD thesis, Dept. of Computer Science, Univ. of Manchester, September 1993.
95. SINDAHA, R. Y. *Branch-Level Scheduling in Aurora: The Dharma Scheduler*. PhD thesis, University of Bristol, Department of Computer Science, In preparation, 1993.
96. J. SUNBERG and C. SVENSSON. Muse trace: A graphic tracer for or-parallel prolog. Technical report, SICS, 1990.
97. EVAN TICK. Prolog Memory-Referencing Behavior. Technical Report CSL-TR-85-281, Computer Systems Laboratory, Stanford University, Stanford, CA 94305, September 1985.
98. EVAN TICK. Compile Time Granularity Analysis for Parallel Logic Programming Systems. *New Generation Computing*, 7(2,3):325–337, 1990.
99. TICK, E. Compile Time Granularity Analysis for Parallel Logic Programming Systems. *New Generation Computing*, 7(2,3):325–337, 1990.

100. TREVISAN, T. S. and BARBOSA, J. L. V. (Orientador). Orca 2.0 (or complexity analyzer). Projeto de diplomação, Universidade Católica de Pelotas, Pelotas: UCPel, 1998.
101. PASCAL van HENTENRYCK. *Constraint Satisfaction in Logic programming.* MIT Press, 1989.
102. VARGAS, P. K. Implementação de um analisador de granulosidade para prolog. Projeto de Diplomação, July 1995. CPGCC - UFRGS.
103. VARGAS, P. K. and GEYER, C. F. R. 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.
104. VARGAS, P. K. and GEYER, C. F. R. 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.
105. VARGAS, P. K. V., GEYER, C. F. R., and DUTRA, I. C. Tamagoshi - plataforma para avaliao de escalonamento de tarefas em programao em lgica paralela. In *X Simpsio Brasileiro de Arquitetura de Computadores e Processamento de Alto Desempenho*, 1998. Búzios, RJ.
106. RICK VAUPEL, ENRICO PONTELLI, and GOPAL GUPTA. Visualization of And/Or-Parallel Execution of Logic Programs. In *Proceedings of the Fourteenth International Conference on Logic Programming*, pages 271–285, 1997.
107. ANDREW R. VERDEN and HUGE GLASER. Independent And-Parallel Prolog for Distributed Memory Architectures. Technical report, Department of Electronics and Computer Science, University of Southampton, April 1990.
108. 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.
109. DAVID H. D. WARREN and FERNANDO C. N. PEREIRA. An Efficient Easily Adaptable System for Interpreting Natural Language Queries. Technical Note, Dept of AI, University of Edinburgh, 1981.
110. RONG YANG. Solving simple substitution ciphers in Andorra-I. In *Proceedings of the Sixth International Conference on Logic Programming*, pages 113–128. MIT Press, June 1989.
111. RONG YANG, VíTOR SANTOS COSTA, and DAVID 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.
112. YANG, R., BEAUMONT, T., DUTRA, I. C., SANTOS COSTA, V., and WARREN, D. H. D. 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.