VisAll: A new Tool to Visualise Parallel Execution of Logic Programs

Nuno Fonseca and Vítor Santos Costa
LIACC, Universidade do Porto
{nf,vsc}@oat.ncc.up.pt

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

Abstract

Logic programming allows to explore the full potential of parallelism in the programs in a transparent manner. Several systems were developed to explore the implicit parallelism of logic programs. The development of these systems brings some hard problems such as parallel debugging and dynamic distribution of varied sized work among processors. This paper describes **VisAll**, a new tool to visualise the parallel execution of logic programs, whose main objective is to help users to develop parallel logic programming systems. VisAll can visualise several kinds of parallelism including or-parallelism, independent and-parallelism, and combinations of both. The system is modular and reasonably flexible in order to be able to visualise a great spectrum of parallel logic programming systems VisAll allows users or system developers to modify or add new trace formats, define the visualisation of new forms of parallelism, and define the icons associated to each trace event.

Keywords: logic programming, parallelism, visualisation tool, trace files.

1. Introduction

One of the most important advantages of logic programming is that it allows the transparent exploitation of parallelism. Several forms of paralellism can be exploited in logic programming systems. The major forms are or-parallelism, independent and-parallelism and dependent and-parallelism. Independent and-parallelism was exploited in systems such as &-Prolog [12] and &-Ace [11, 9], or-parallelism in systems such as Aurora [14] and Muse [1], and dependent and-parallelism in systems such as KLIC [22], Parlog [3] and DASWAM [21]. More recently, parallel logic programming systems have been developed to exploit a combination of these forms of parallelism. These include Andorra-I [19], ACE [10], SBA [2], Penny [17], ParAKL [18] and the EAM [24].

Experience from these systems has shown that the development of *efficient* parallel logic programming systems is not an easy task. System implementors need to tackle hard problems such as parallel debugging and dynamic distribution of varied sized work among processors. Some applications use a sequential programming style and have a very low degree of parallelism, be it and-parallelism or or-parallelism. Moreoveor, this parallelism can be rather fine-grained. Acceptable performance for these applications often depends on smart scheduling decisions.

Logic programs that present reasonable amounts of paralellism can have good, even excellent, performance on parallel systems. However, in practice, there are several issues that may result in bad performance of the parallel system even when the application has sufficient parallel tasks to supply the available processors. In most parallel logic programming systems these problems are directly related to workload distribution strategies since task switching is not a constant time operation. 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 a complex system on a parallel environment where the response time is totally non-deterministic due to the interference of operating system operations.

To deal with these problems parallel system designers and users need tools that can allow a high-level view of a parallel execution. Visualisation tools are particularly important towards this goal, as good visualisation tools can both allow a general overview of an execution and focussing on the important execution details of the 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 designed to support the visualation of all kinds of parallelism in logic programming. The main goals of VisAll are (i) to allow developers to identify points of the parallel implementation that need to be optimised for the target parallel architecture used; and (ii) to help in parallel debugging. Initially, the system was designed to help the development of SBA [2], a parallel logic programming system that exploits both or-parallelism and independent and-parallelism. Currently, VisAll is capable of visualising synthetic tracefiles for the target SBA system and also other tracefiles taken from parallel systems such as Muse, Aurora and &-Prolog. The system is flexible because users can specify graphical objects and trace events to be visualised. It is universal in that VisAll is capable of visualising all major forms of parallelism that are exploited in parallel logic programming systems.

Our systems follows other visualisation tools have been developed to parallel logic programming systems, such as WAMtrace [5], VisAndOr [7], ViMust, and VACE [23]. Each of these systems supports a different parallel logic programming system. VisAll is the first attempt at unifying the visualisation paradigms exhibited by the other visualisation tools whilst adding the extra features important to system optimisation and debugging.

The paper is organised as follows. Section 2. gives some basic notions on visualisation of parallel execution of logic programs. Section 3. describes VisAll and its implementation. In section 4. we compare VisAll with two other well-known visualisation tools. Finally, section 5. gives some conclusions and future work.

2. VisAll In Operation

In this section we present some terminology used throughout this paper and demonstrate some VisAll applications. We borrow terminology from VisAndOr [15] whenever possible.

VisAll is a trace-based event-driven visualisation tool. An *event* describes a relevant point in the execution, together with the information needed to distinguish it from other event in the same execution. VisAll reads and interprets a set of these events from a *trace* file and transforms them in a graphical visualisation.

Typically an event contains its type, a timestamp, a process agent, references to related events and additional information. The *timestamp* corresponds to the time that the event occurred. The *process agent* corresponds to the process that generated the given event. A *trace* is thus a collection of events that corresponds to a particular execution of a program.

VisAll transforms the trace-file in a dependency graph, where nodes correspond to events and arcs to dependencies (or relations) between two events. For example, a colour line between two nodes represents a process at work. In VisAll each process will receive a different colour. Black lines indicate that no worker has taken this work. Each event is represented by an icon. The vertical axis represents either the time-line or event order, that increases from the top to the bottom of the image. In this paper we use event order for the VisAll pictures.

The main objective of VisAll is to help the user to understand his/her parallel system execution. In order to do so VisAll gives to the user three views of the parallel execution: (1) a static view of the whole execution tree and processor allocation; (2) a dynamic view of the execution for a more detailed study of the applications' behaviour; and (3) a view of process utilisation along time. Note that the dynamic view of the execution is obtained from the static execution tree.

One important characteristic of VisAll is that it is built in such a way that it can visualise traces of different parallel systems. This includes visualising traces for well known logic programming systems, and also visualising synthetic trace files that can be used to predict the performance of parallel systems not yet implemented. To do so, VisAll is distributed with several filters that can translate from the formats used in other tools to the more general internal VisAll format.

In the following sections we show how VisAll is applied to some well known parallel logic programming systems.

2.1. Visualising Or-parallel Execution

For or-parallel systems VisAll can display, but is not limited to, parallel execution of the Muse [1] and Aurora [14] systems. Or-parallelism arises in applications when one executable goal matches more than one clause in the database. Or-parallel execution can be described through or-trees.

Implementors of or-parallel systems must deal with problems such as side-effects, that sequentialise execution, and work distribution. Moreover, and depending on the system, or-trees for the same problem from different systems may look quite different.

Figure 1 shows how VisAll visualises the parallel execution of the queens problem, as run in Muse with four processors for a 8x8 board size. The small picture to the bottom left shows the whole picture of the parallel execution, and gives the structure of the search tree. The main picture shows the zoomed part of the execution tree. The triangles represent events associated to branches. A normal filled triangle indicates that a branch has been completed successfully, while an inverted triangle represents the start of a branch. The inverted stripped triangle represents branch suspension while a normal triangle after a suspension indicates resumption of the suspended branch.

One of the utilities of our tool can be immediately identified with this picture. By observing figure 1, one can see that for this execution too many suspensions were made in very short intervals right after starting goals. This may indicate that Muse is being too eager to suspend branches and seek for better (less speculative) work in the tree.

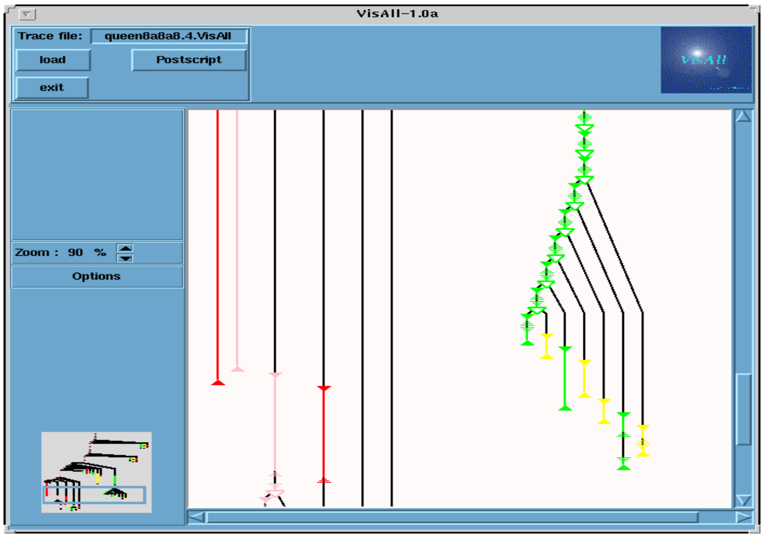

Figure 1: Or-parallelism: Visualisation of the eight queens problem in Muse

VisAndOr shows the same kind of tree, but its precision does not allow one to spot the "too many suspensions" problem. Figure 2 shows the same trace file as visualised by VisAndOr. We do not compare VisAll with ViMust in this work, since the dynamic part of VisAll is not yet implemented.

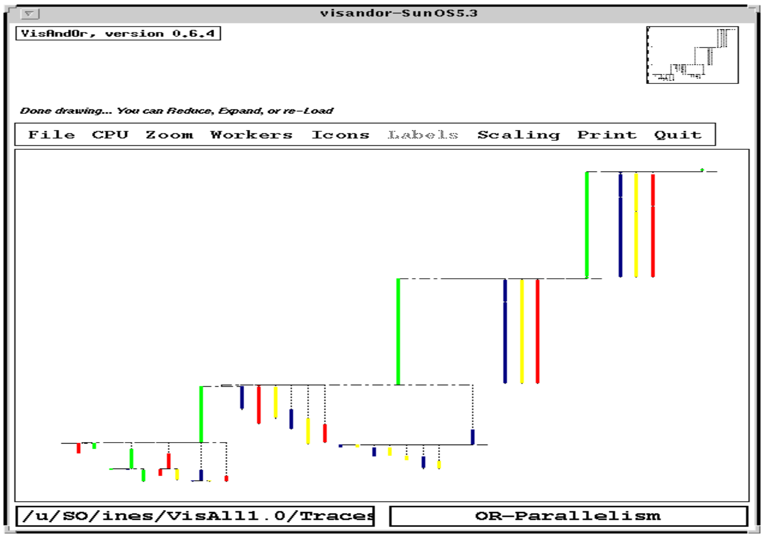

Figure 2: Or-parallelism: Visualisation of the eight queens problem in Muse

Aurora [14], as Muse, exploits or-parallelism in logic programming applications. Aurora and Muse have major differences in their design. They use different data structures to manage conditional bindings and use different or-scheduling techniques. Using different data structures to manage or-parallel bindings does not necessarily make the execution trees look different, but different scheduling strategies definitely will make the trees look different. Figure 3 shows the parallel execution of the eight queens problem with four processors, in Aurora.

VisAll clearly shows how the execution trees of Muse and Aurora are quite different. While Aurora concentrates workers on the left hand side of the search tree, Muse spreads workers evenly across branches, suspending work in a given branch from time to time in order to seek for better sources of work.

2.2. Visualising And-parallelism

And-parallelism arises in applications where several sub-goals can execute in parallel. VisAll can visualise parallel execution of systems that exploit dependent and independent and-parallelism. We next concentrate on independent and-parallelism, as in &-Prolog [13]. In these systems goals that do not share variables are allowed to execute in parallel. Independent and-parallelism arises in divide and conquer problems, for example.

Figure 4 shows an &-Prolog execution. The trace-files generated by &-Prolog have the same format as the ones for VisAndOr and can be converted to the VisAll format. We present the visualisation of a parallel execution of the quicksort program run in &-Prolog with four workers. Notice that the work always *forks* from a conjunction, is run in parallel by the individual workers, and at the *joins* to provide a partial solution. Although the structure of the execution is quite regular, tasks at the same level have quite different execution times, showing the need for good load-balancing.

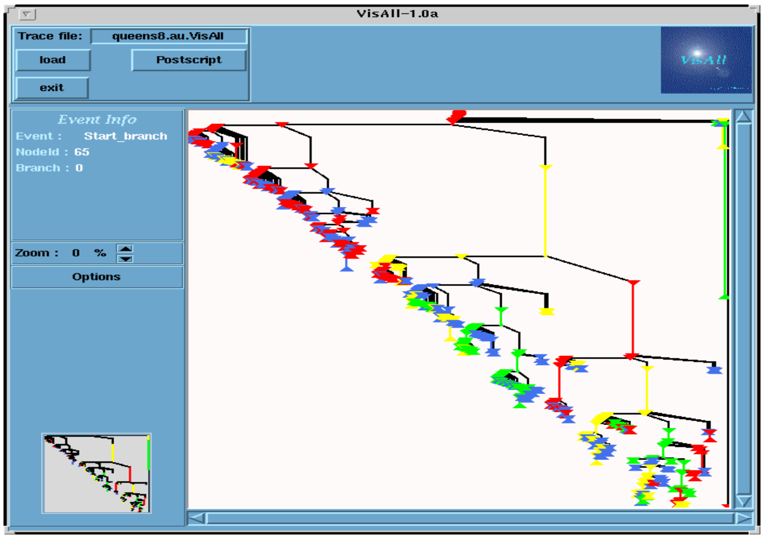

Figure 3: Or-parallelism: Visualisation of the eight queens problem in Aurora

The figure also shows for the first time the VisAll's *activity map* for each processor. At any time if a worker is idle its bar is painted black, otherwise it is painted in the worker's color. One can move from the activity map to the main visualisation and back, allowing for easy investigation of the points at which workers became idle for longer, and vice-versa. In the example, the map indicates some load imbalance, especially at the beginning and end of the execution.

2.3. Visualising both And-parallelism and Or-parallelism

Andorra-I [26, 20, 25, 6] is a system that exploits *or-parallelism* and *determinate and-parallelism*. By determinate and-parallelism we mean that the system only allows parallel execution of goals when they match at most one clause in the database. This makes the parallel execution tree looks like an or-tree with some "bushes" that indicate the parallel execution of determinate and-goals in a branch.

Figure 5 shows the last phase of the parallel execution of a flight allocation program run in Andorra-I. The program firsts sets constraints on the flights by using the Pandora method and then evaluates the system of constraints by first propagating the constraints then trying alternative solutions to the open constraints. Execution has two sources of parallelism: and- and or-. And parallelism is dominant in a first phase, where initial constraints are set and some resolved deterministically. In a second phase the system searches through a space of possible solutions and both and- and or-parallelism are available. The picture only shows this second phase.

Note that and-parallelism is depicted by parallel coloured lines that join in a same main branch of the execution tree. Both figures seem to indicate lack of sufficient work to sustain eight workers. The search tree is pretty small, and workers are quite often idle or jumping to find work in other branches. The activity map confirms this insight, with long periods of idleness for most processors.

A different approach for combining different kinds of parallelism in logic programs is to support both or-parallelism and independent and-parallelism. The combination of these two forms of parallelism in a recomputation based approach offers the promise to obtain good performance and low overheads in relation to sequential Prolog systems. The SBA model was designed to exploit combined parallelism

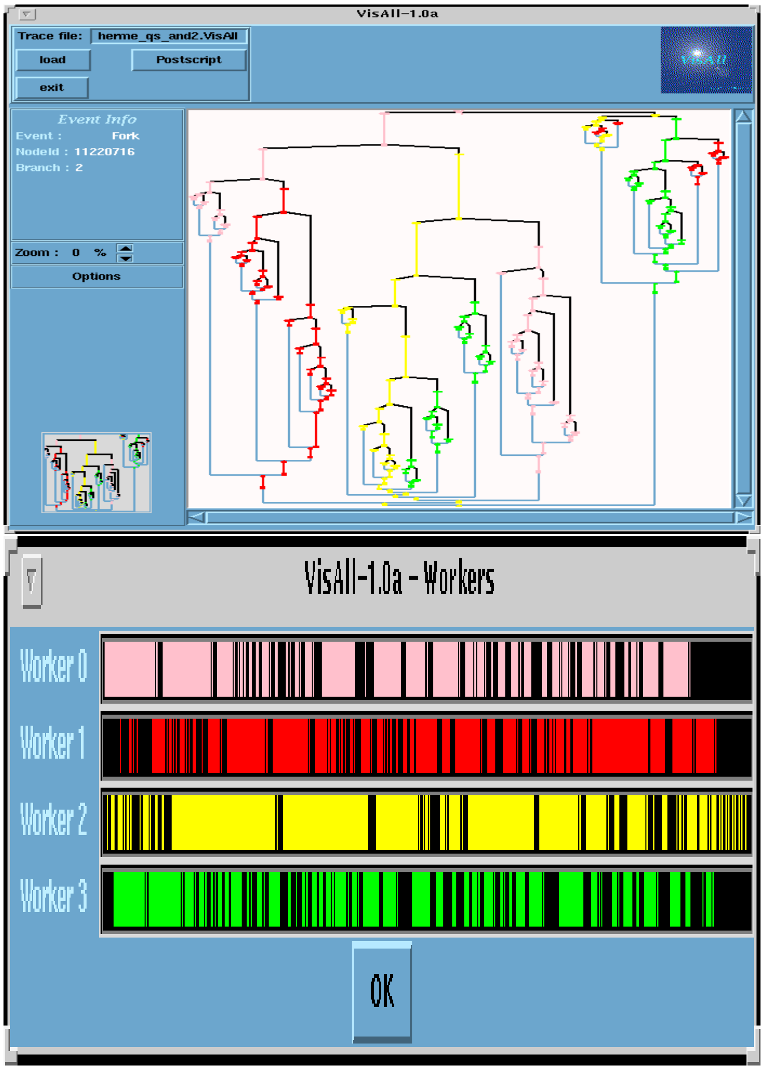

Figure 4: &-Prolog according to VisAll

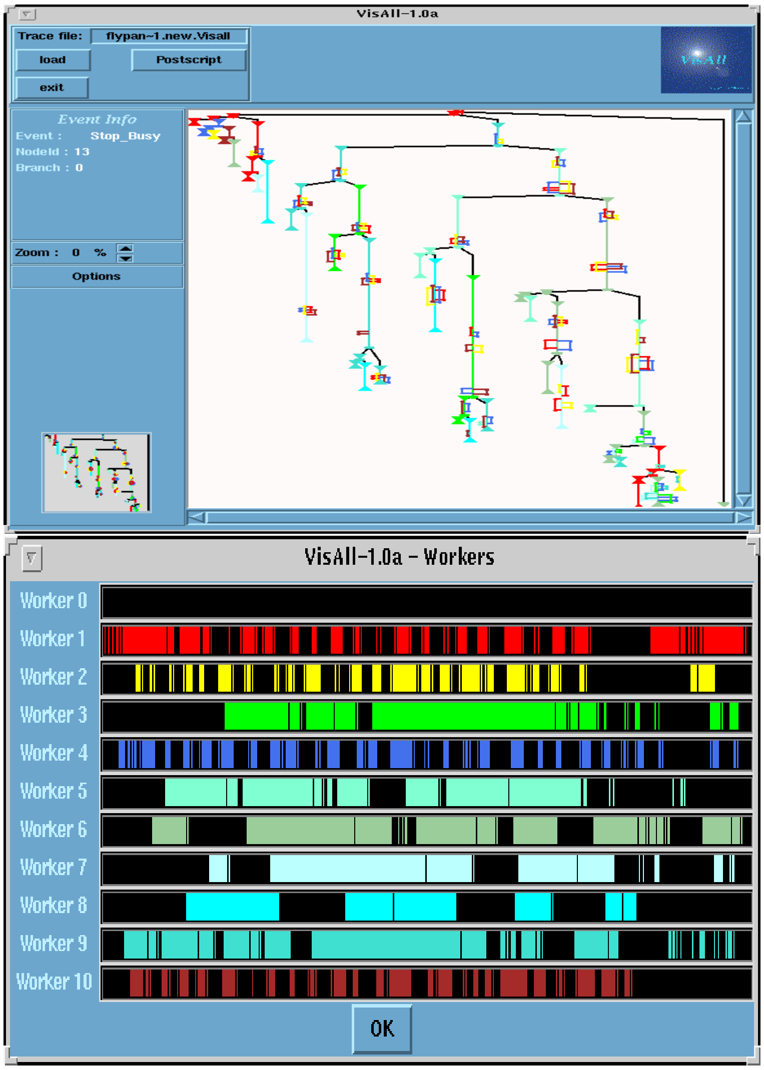

Figure 5: VisAll Visualisation of an Andorra-I Benchmark

by introducing a novel approach to the problem of variable binding representation, the sparse binding array. In this representation, each position in the shared stacks is shadowed by each active or-parallel branch binding stack, so that the complex problems of memory management for independent and-parallel systems need only to be resolved for the shared area.

The SBA system [2] is being developed at LIACC. VisAll is being developed in the context of the same project and therefore should be able to visualise parallel execution of the model subsumed by the SBA system. Ultimately, the main objective of this project is to build a parallel logic programming system that integrates all sorts of parallelism, dependent, independent and-parallelism, and or-parallelism. Currently, VisAll is capable of visualising **synthetic** trace-files for the SBA system, since this system is not fully implemented yet.

Figure 6 shows the visualisation of a synthetic trace-file to the SBA model, as visualised by VisAll. The file was generated by Dutra's recomputation *or* plus *independent &-parallelism* simulator.

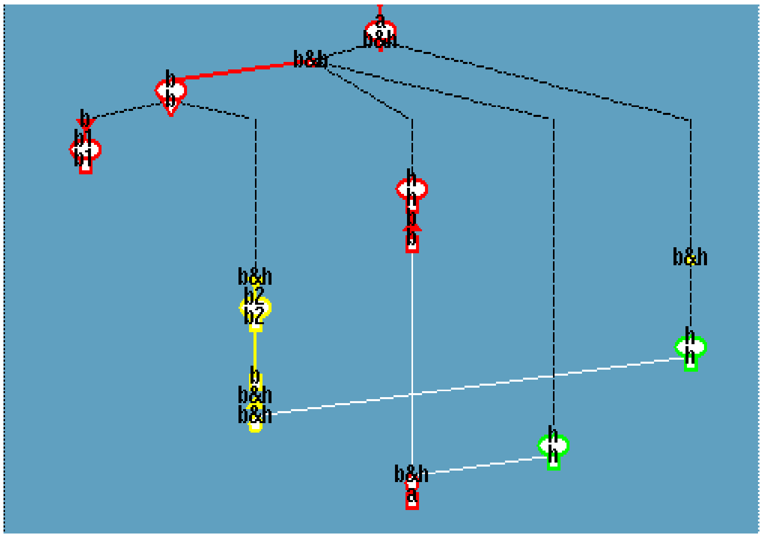

Figure 6: VisAll representation of independent-and and or-parallelism

The execution of `?-a.` is shown in figure 6, and the corresponding Prolog program is shown in the next lines.

```
a :- b & h.
b :- b1 ; b2.
b1. b2. h.
```

The simulator generates VisAll traces, so information on the name of goals is also available. The goal `a` unfolds in `a & b`. A C-node is created for this goal just at the top of the figure. The C-node has two alternatives. The first is obtained by taking `b1` and running `h` in parallel. The second by taking `b2` and running `h` in parallel with it.

Notice that differently from VisAndOr and VACE, VisAll represents join nodes. These joins connect and-goals that can be executed in parallel inside a CGE (Conditional Graph Expression) [4]. They are particularly useful when the and-parallel computation is not the last goal in a clause.

3. VisAll Implementation

One of the objectives of VisAll was to allow developers of parallel logic programming systems to debug and optimize their implementations. To do so, the system should be extremely flexible. Moreover, we wanted for VisAll to be an universal system, that could be useful in the new systems being developed. Note that implementors are still required to capture in their programs the events understood by VisAndor in order to generate trace events whenever appropriate in the source code. VisAll's contribution is that it allows implementors to add new forms of visualisation or to modify existing forms. In order to do so, VisAll was designed as three components: (1) an *event configurer*, (2) a *visualisation program* and (3) *translators*.

The *event configurer* is a program based in TCL/TK that allows users to configure symbols associated to events. For example, the user may wish to represent a *cut* by $\boxed{!}$ or by other icon.

The *translators* have the job of converting a given trace file and to create a VisAll trace format file containing a dependency graph representing the trace.

The *visualisation program* displays the VisAll files created by the translators.

The algorithms used by VisAll do not pose any limit to the size of the trace-files. Trace-file size is only restricted by the computer's virtual memory size.

Figure 7 shows a scheme of the system, with its components and their relationship.

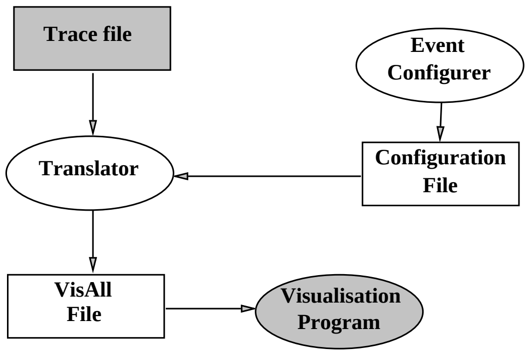

Figure 7: The VisAll Tool

3.1. The Trace Files

The trace files are the base for the visualisation. The use of trace files makes unnecessary the direct access to the parallel system during visualisation, thus reducing overheads. It also allows to visualise the parallel execution in an environment different from the system where the trace was generated.

In order to create trace-files, the developer of the parallel system must instrument the parallel system to generate events. These events must be written on a disk trace-file. Each trace file line must refer to only one event at a time, and has to be ordered chronologically (by the timestamp value). Visandor accepts several events on the same timestamp.

Currently, VisAll accepts two trace formats. One is VisAndOr's original format. The internal VisAll format generalises the VisAndOr format in that it includes (1) new fields to visualise icons related to new forms of parallelism; and (2) a string field used to store source program information. The main events for VisAll closely follow the Visandor's events and are described in table 3.1.. The *Start* and *End*

Event	Description
Start Time	Start the execution of a program
End Time	End the execution of a program
Create C-Node	C-Node was created
Create CGE	New alternative to C-Node (CGE)
Start Goal	Parallel goal started execution
Finish Goal	Parallel goal completed
Join	Parallel conjunction completed
Make Public	Or-node available to other teams
Start Branch	Branch in or-node taken
Succ Branch	Branch finished successfully
Suspend Branch	Team left non-completed branch
Resume Branch	Team returns to completed branch

Table 1: Main VisAll Events

Times events indicate the beginning and the end of the execution. The *Create C-Node* events corresponds to a a set of and-parallel goals which can have both and and or-parallelism according to the Composition tree model [8]. The *Create CGE* event corresponds to starting a traditional and-parallel conjunction for independent and-parallelism. *Start* and *Finish Goal* correspond to and parallel goals. Last, *Join* corresponds to the end of parallel conjunction execution. The next events correspond to or-parallelism. *Make Public* corresponds to making a goal available for parallel execution, and *Start Branch* to some processor taking a branch from that node. A branch may succeed or may be suspended and then later resumed, as shown in the next events.

3.2. The Translators

The task of a translator is to take a trace in a format specific to that translator and to create a VisAll file. The file includes a list of events, in a format similar to the Trace-files described in the previous section, and an associated dependency graph. The graph is also represented as ASCII text, as a sequence of lines of the form *Command: Fields*, where the number of fields depends on the commands. The main commands are *addInfo* and *addNode*, that together describe a node, and *addEdge*, that describes an edge of the graph. Note that the translator could be included in the visualiser. Implementing it separately avoids the overheads of always building and setting the coordinates of the graph whenever we visualise a trace.

Translators are currently written in C. Each translator includes a common set of routines and two separate modules: (1) the *trace module* that defines the trace format, and (2) the *graph module*, that defines the dependencies of each event (defines a graph). To define a new trace format it is only necessary to write a new trace module that understands the new trace format and converts to the internal representation. The graph module that defines the graph is essentially implemented as a series of rules, one for each kind of event, that introduce the edges to the graph. In this way one can generate different graphs supporting different visualisation paradigms by rewriting just this module.

The common code provides control and is responsible for the general look of the visualisation graph. The translator inputs the trace file line by line. For each line, or in other words, each event, the translator calls the trace module function for the event. This function will in its turn call the `addNode` glue function, which calls the graph module for the visualisation dependent rules defined for the event. After all events are loaded, the X and Y coordinates of the nodes (events) and edges of the graph must be calculated. The Y values can be obtained either from the timestamp field or from the event order, and the X values are obtained from an algorithm designed to balance the graph width. If necessary, the algorithm adjusts the Y coordinates in order to prevent that icons are overlapped. By default, each unit of time corresponds to one pixel on the screen, which makes the visualisation accurate in terms of time.

Each edge has associated a type and a color. The types are either dotted or full line. The color indicates the worker that generated that event. The dotted lines are used to show relations between events, while full lines show working agents.

3.3. The Visualisation Program

The visualisation program is implemented on top of the Tcl/TK script language and toolkits. It currently 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 processors.

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 sufficient 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 sufficient to click on the worker bar in order to obtain the position where the worker is in the tree.

4. Related Work

To the best of our knowledge, VisAndOr [16] was the first system that attempted to visualise parallel logic programming execution. It was originally designed for &-Prolog, and then extended to support Aurora, Muse, and Andorra-I. VisAll differs from VisAndOr in three characteristics: (1) it allows for visualisation of more parallel systems; (2) it represents information for individual execution of each process; (3) it implements a more sophisticated scheme of zoom, where the user can examine information attached to each node of the execution tree.

Figure 8 shows two visualisations of the same program. The left one is the one shown by VisAll. The right one shows the same program visualised by VisAndOr. Note the extra info provided by VisAll for this program.

Another well-known tool for visualisation is VACE. VACE was designed to visualise parallel execution of the ACE system. VACE uses an approach different from VisAll to visualise parallel logic programming systems. VACE builds a graph that is then designed by using the DaVinci system.

VisAll is different from VACE in two main aspects: (1) VisAll visualises more parallel logic programming systems, and (2) VisAll visualises parallel execution according to event time and not according to execution flow and shape of the tree. Figure 9 gives a flavour of VACE and VisAll displays.

5. Conclusions and Future Work

This work presented **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 capable of visualising more sources of parallelism than other visualisers.

VisAll is a tool still under construction. Our next steps are (1) to add new forms of parallelism, namely non-determinate dependent and-parallelism, (2) to implement an animated simulation of the execution, and (3) to extend the tool to give more detailed information about the system, such as why a processor (or worker) suspended, why it positioned somewhere in the tree, and where in the source program is is executing code. We are particularly interested in finding what kinds of information are of interest to parallel logic programming system designers.

Needless to say, we are continuing experimentation of VisAll with the recent parallel systems in order to demonstrate its utility.

Acknowledgments

This work has been supported by the PRAXIS Proloppe project, PRAXIS/3/3.1/TIT/24/94. Vítor Santos Costa would like to acknowledge support from the JNICT Melodia project. Inês Dutra would like to acknowledge support from the PROTEM Appelo project.

References

[1] Khayri A. M. Ali and Roland Karlsson. The Muse or-parallel Prolog Model and its Performance. In *Proceedings of the 1990 North American Conference on Logic Programming*, pages 757–776. MIT Press, October 1990.

[2] Manuel Eduardo Correia, Vtor Santos Costa, and Fernando Silva. The SBA: Exploiting Orthogonality in AND–OR Parallel Systems. In *ILPS'97, Port Jefferson, NY, USA,*, 1997.

[3] J. A. Crammond. The Abstract Machine and Implementation of Parallel Parlog. Technical report, Dept. of Computing, Imperial College, London, June 1990.

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

[5] 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.

[6] 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.

[7] Luis Gómez, Manuel Carro, and Manuel Hermenegildo. Some Paradigms for Visualising Parallel Execution of Logic Programs. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 184–202, 1993.

[8] Gopal Gupta, M. Hermenegildo, and V. Santos Costa. And-Or Parallel Prolog: A Recomputation based Approach. *New Generation Computing*, 11(3,4):770–782, 1993.

[9] Gopal Gupta and M. V. Hermenegildo. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *LNCS 569, ICLP'91 Pre-Conference Workshop on Parallel Execution of Logic Programs*, pages 146–158. Springer-Verlag, June 1991.

[10] Gopal Gupta, M. V. Hermenegildo, E. Pontelli, and V. Santos Costa. ACE: And/Or-parallel Copying-based Execution of Logic Programs. In *Proceedings of the Eleventh International Conference on Logic Programming*, Italy, June 1994.

[11] Gopal Gupta, Enrico Pontelli, and Manuel Hermenegildo. &ACE: A High Performance Parallel Prolog System. In *Proceedings of the First International Symposium on Parallel Symbolic Computation, PASCO'94*, 1994.

[12] M. V. Hermenegildo and K. Greene. &-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.

[13] 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.

[14] Ewing Lusk, David H. D. Warren, Seif Haridi, et al. The Aurora Or-parallel Prolog System. *New Generation Computing*, 7(2,3):243–271, 1990.

[15] M.Hermenegildo M. Carro, L. Gómez. Implementation of an event driven scheme for visualizing parallel execution of logic programs.

[16] M.Hermenegildo M. Carro, L. Gómez. Some paradigms for visualizing parallel execution of logic programs. 1993.

[17] Johan Montelius. *Penny, A Parallel Implementation of AKL.* PhD thesis, Swedish Institute for Computer Science, SICS, Sweden, May 1997.

[18] Remco Moolenaar and Bart Demoen. Hybrid tree search in the Andorra Model. In Pascal Van Hentenryck, editor, *Proceedings of the Eleventh International Conference on Logic Programming*, pages 110–123. MIT Press, June 1994.

[19] V. Santos Costa, D. H. D. Warren, and R. Yang. Andorra-I: A Parallel Prolog System that Transparently Exploits both And- and Or-Parallelism. In *Third ACM SIGPLAN Symposium on Principles & Practice of Parallel Programming*, pages 83–93. ACM press, April 1991. SIGPLAN Notices vol 26(7), July 1991.

[20] Vítor Santos Costa, David H. D. Warren, and Rong 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, 1991.

[21] Kish Shen. Memory Management and Goal Scheduling for Parallel Execution Revisited. Internal Report, Bristol University, 1993.

[22] T. Chikayama, T. and Fujise, and H. Yashiro. A Portable and Reasonably Efficient Implementation of KL1. In *Proceedings of the Eleventh International Conference on Logic Programming*, June 1993.

[23] Richard S. Vaupel. Vace - a visualization system for ace. Internal Report, New Mexico State University, 1995.

[24] David H. D. Warren. The Extended Andorra Model with Implicit Control. Presented at ICLP'90 Workshop on Parallel Logic Programming, Eilat, Israel, June 1990.

[25] Rong Yang, Tony Beaumont, Inês Dutra, Vítor Santos Costa, and David H. D. Warren. Performance of the Compiler-Based Andorra-I System. In *Proceedings of the Tenth International Conference on Logic Programming*, pages 150–166. MIT Press, June 1993.

[26] 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.

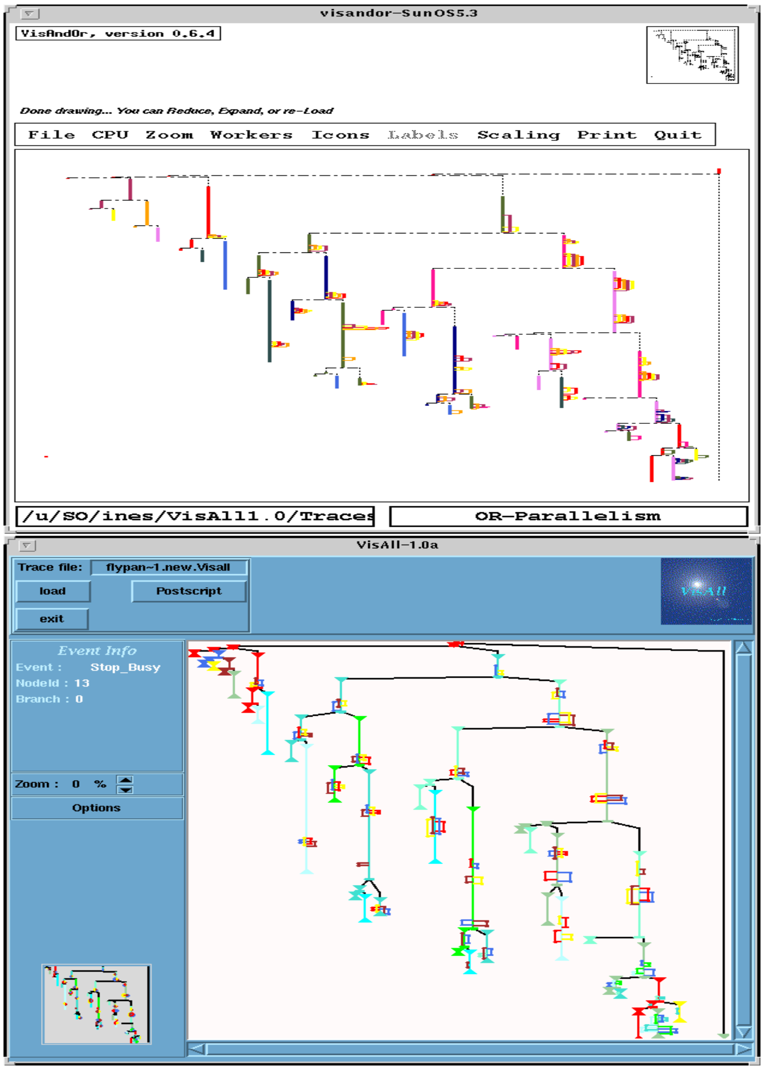

Figure 8: Flight allocation application executed in Andorra-I according to VisAndOr and VisAll, respectively

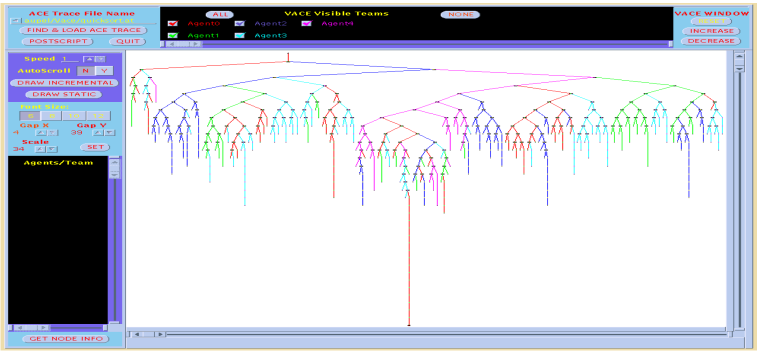

Figure 9: VACE visualisation