VisAll: A Universal Tool to Visualise the Parallel Execution of Logic Programs

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

Inês de Castro Dutra
Department of Systems Engineering and Computer Science
COPPE/UFRJ, Federal University of Rio de Janeiro, Brazil
ines@cos.ufrj.br

Abstract

One of the most important advantages of logic programming systems is that they allow the transparent exploitation of parallelism. The different forms of parallelism available and the complex nature of logic programming applications present interesting problems to both the users and the developers of these systems. Graphical visualisation tools can give a particularly important contribution, as they are easier to understand than text based tools, and allow both for a general overview of an execution and for focusing on its important details.

Towards these goals, we propose *VisAll*, a new tool to visualise the parallel execution of logic programs. VisAll benefits from a modular design centered in a graph that represents a parallel execution. A main graphical shell commands the different modules and presents VisAll as an unified system. Several input components, or translators, support the well-known VisAndor and Vace trace formats, plus a new format designed for independent and-parallel plus or-parallel execution in the SBA. Several output components, or visualisers, allow for different visualisations of the same execution. These visualisaters can operate simultaneously and can collaborate by using the execution graph.
Keywords: logic programming, parallelism, visualisation tool, trace files.

1 Introduction

One of the most important advantages of logic programming systems is that they allow the transparent exploitation of parallelism. Several forms of paralellism can be exploited in logic programming systems. The major forms are or-parallelism (ORP), independent and-parallelism (IAP) and dependent and-parallelism (DAP). IAP was exploited in systems such as &-Prolog [12] and &-Ace [11, 10], ORP in systems such as Aurora [13] and Muse [1], and

DAP in systems such as KLIC [22], Parlog [3], and DASWAM [20]. More recently, parallel logic programming systems have been developed to exploit a combination of these forms of parallelism. These include Andorra-I [17], ACE [10], SBA [2], Penny [14], ParAKL [15] and the EAM [24]. These systems have demonstrated the usefulness of parallel logic programming for non-trivial applications.

The different forms of parallelism available and the complex nature of logic programming applications present interesting problems to both the users and the developers of these systems. From the developer's point of view, designing and implementing a parallel logic programming system is a complex task. In order to obtain a correct and efficient implementation one must address problems such as synchronisation and dynamic distribution of varied sized work among processors. Moreover, debugging parallel programs is not simple. These are complex systems operating on a parallel environment where the response time is non-deterministic due to the interference of operating system operations. And even when the system is found to be working correctly, it may not perform well.

The evolution of parallel logic programming research and the more sophisticated computer architecture models further amplify these problems.

In order to alleviate the developers job, parallel systems must provide a set of tools that can utilise data extracted from execution (actual or simulated) and present it in a form that allows for qualitative reasoning. Graphical visualisation tools can give a particularly important contribution, as they are easier to understand than text based tools, and allow both for a general overview of an execution and for focusing on its important details.

It is therefore unsurprising that since early on most parallel logic programming systems have included a related visualisation tool. Examples include Aurora and WAMtrace [5], Muse and Must [21], &-Prolog and Visandor [8], or ACE and Vace [23]. These systems have been very important in the development of parallel logic programming.

The aim of this work is to design a general visualisation tool that addresses the problems mentioned thus far. We would like (i) to support the visualisation of all kinds of parallelism in logic programming, and (ii) to support different visualisation paradigms in the same framework. The resulting system should not be overly dependent on functionality from a single technology, such as a graphical toolkit, and should not depend on a specific parallel system.

We propose *VisAll*, as a *flexible* and *universal* visualisation tool that achieves these goals. To do so, VisAll relies on a simple idea, the *representation of parallel execution as a graph*. This graph is the cornerstone of the system.

VisAll is the subject of this paper. We first present the general structure of the system. Next we concentrate on the main implementation issues, concerning the structure and characteristics of the execution graph, how to design the filters that translate from external formats, and what visualisation paradigms are currently supported. We next present some examples of VisAll visualisation, comparing them throughout with other well-known visualisation tools. Finally, we give the conclusions and discuss future work.

2 System Design

VisAll was initially designed as a tool to support the development and the utilisation of the SBA [2], a parallel logic programming system that exploits both IAP and ORP. The SBA is a system in continuous development, and soon it became clear that the visualiser would require extensibility as new components of the SBA are being developed and introduced to the system. Moreover, our experience with other systems motivated us in designing VisAll to be an universal system, one that should be used with both previous and novel parallel logic programming systems.

In order to obtain a general and flexible system, VisAll is based on the key idea of an *event graph*. VisAll assumes that the input to the visualisation process is a *trace file*, consisting of a sequence of *events*. Each event includes a *timestamp*. One can say that the events form a graph, where edges represent direct dependencies between events. Usually, this graph of events is a directed acyclic graph.

VisAll makes no assumptions regarding how events were generated. Events may have been generated by actual execution on a parallel system, or by execution on some simulator, and we have in fact used trace files originating from both forms of execution. As most other visualisers, the current VisAll implementation works off-line, meaning that it does assume execution has completed and the trace file is closed when the visualisation process starts. With current technology, off-line execution is necessary for detailed and accurate visualisation.

VisAll does not impose any specific trace format for visualisation. On the contrary, from early on it became clear that VisAll would only benefit if it would understand other common trace formats, such as VisAndor's or Vace's format. To do so, we separate between an external trace file, with the events generated by the system, and the internal representation. Internal nodes and events have semantics that are understood by the *visualisers*. *Translators*, or filters, translate from an external trace format to this internal graph representation.

One should remark that in general, and for completeness sake, we would prefer for the internal representation to be at least as general as any external representation accepted by the system. By doing so we never lose information. Alternatively, one could accept more general external trace formats and only process a projection that can be understood by VisAll.

A main graphical program commands the visualisers. The system has been designed to present a simple interaction to external users, and presents itself as a single, integrated, system. The *event configurer* currently allows users to associate symbols with events. For example, the user may wish to represent a *cut* by $\boxed{!}$.

We next describe the individual components in detail.

2.1 The Structure of the Graph

The major VisAll component is the graph representing execution. This graph representation is generated from a trace file format. As a starting point we chose the VisAndOr format [8], that has been the most popular for parallel logic programming. An example of a node format used to visualise the SBA execution is shown in Table 1.

TS	Ev	NId	Br#	C-Node	#Goals	CGE#	goal#	PID	Info

Table 1: An example of a SBA node format

Each node contains information on **(i)** the timestamp, ie., the point in time where the node was generated; **(ii)**, an event code, that determines the type of the node; **(iii)**, a unique identifier; **(iv)** an or-branch number corresponding to an edge connecting the node to the node that started the current or-branch, for support of ORP; **(v)**, a C-Node number for the support of the recomputation based models for ORP/IAP [9]; **(vi)**, the number of and-parallel goals from the current CGE (Conditional Graph Expression representing a conjunction that can be executed in parallel [4]); **(vii)**, a CGE-number connecting the node to the parent CGE, for the support of IAP; **(viii)**, a goal number connecting the node to the CGE branch, also for support of IAP; **(ix)**, a worker id, corresponding to the worker (or agent, or processor) that generated the event; and **(x)**, a fragment of text with information on the node. This fragment of text can contain information such as the name of a goal, the number of goals in a CGE, or the reason for a worker to have chosen a certain alternative.

Note that this format generalises the VisAndOr format in that it includes (1) new fields to visualise icons related to new forms of parallelism (**v** to support ORP/IAP); and (2) a string field used to store source program information or the behaviour of the parallel system (field **x**).

The event code field is particularly important in that it describes the type of event to visualise. The main events for VisAll closely follow the Visandor's events and are described in table 2. More details about the events can be found in [7].

Arguably, the simplest form of representing the graph would be to represent it as an ordered sequence of events in ASCII text. Instead, and for efficiency reasons VisAll uses a rather different strategy. The original execution graph is manipulated within the translator, and the translator is responsible for generating coordinates that can be used by a graphical tool.

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 (to support IAP)
Create CGE	New alternative to C-Node (to support IAP)
Start Goal	Parallel goal started execution
Finish Goal	Parallel goal completed
Join	Parallel conjunction completed (to support IAP)
Start_Busy	Worker joins a new team (to support dyn. sch.)
Stop_Busy	Worker leaves its team (to support dyn. sch.)
Make Public	Or-node available to other teams (to support ORP)
Start Branch	Branch in or-node taken (to support ORP)
Succ Branch	Branch finished successfully (to support ORP)
Suspend Branch	Team left non-completed branch (to support ORP)
Resume Branch	Team returns to completed branch (to support ORP)

Table 2: Main VisAll Events

The resulting representation is organised as two components: (1) a sequence of events, represented as a line with an event number and skeletal information on the event, and (2) a sequence of graph building commands. Graph building commands use a simple language of the form *Command: Fields*, where the number of fields depends on the commands.

One major contribution of VisAll is representing the parallel execution graph as a list of commands. These commands do not just describe how the graph is composed, they actually *do indicate how to build the graph*, that is, how to physically place the nodes and edges of the graph. This greatly facilitates the implementation of the graphical components.

Three major commands are available. The *addInfo* command gives detailed information on a node. The *addNode* command describes how to place a node in the graph. The *addEdge* command describes how to place an edge of the graph, and gives information on its type.

The current implementation makes the system more efficient by combining in the translator two different components, actual translation and graph representation. In fact, the components do pursue separate goals and have been developed separately, as explained in the next section.

2.2 The Translators

Translators take a trace from an external format and create a VisAll style internal file. Translators include a format dependent component, the *trace module*, that converts from an external representation to VisAll's internal representation, and a format independent component that actually builds the graph, the *graph module*. The two modules are managed by common code that provides control and are responsible for the general look of the visualisation graph.

Input trace files consist of lists of events. Most external formats are text based, such that each line in the trace file line must refer to a single event, and has to be ordered chronologically (by a timestamp). Differently from other systems, VisAll accepts several events with the same timestamp.

Current translators are written in `C`. Translators have been written for the SBA visualisation format, for the VisAndor format, and for the Vace format. The trace module is the component concerned with the external trace format and with the conversion to the internal representation. The translator inputs the trace file line by line. For each line, or in other words, each event, the translator calls an *event specific function* that depends on its type.

The graph module that defines the graph is essentially implemented as a series of rules, one for each kind of event. These rules then introduce the nodes and edges to the graph. This way one can generate different graphs supporting different visualisation paradigms just by rewriting this module. The rules are implemented through the `addNode` interface function. This function is called from the top level of the translator after a new event is read and translated from the input graph. The function receives a full description of the event and starts visualisation dependent rules defined for this type of event. In a second stage, 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 icons from overlapping. By default, each unit of time will correspond to one pixel on the screen, which makes the visualisation accurate in terms of time.

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

2.3 Visualisation

We have decided to implement the visualisation program with the `Tcl/Tk` language and toolkit. These graphical systems provide great flexibility, are portable, web-aware, and allow for easily building nice-looking graphical systems. They do have worse performance than `C` or `C++`, but this is not a problem in our case as most heavy processing is performed by the translator.

Figure 1 shows the standard main VisAll interaction window. The top pane gives the standard control options. As usual this includes loading the visualisation graph from a file, saving it to a postscript file, or exiting. The interface accepts both external and translator processed trace files. The load status bar is useful during loading of a new trace file, as it informs how much of the file has been loaded so far. The icon to the right is the VisAll icon.

The bottom pane presents the standard static based form of visualisation. The main subwindow, to the right, displays the dependency graph contained in a VisAll file and shows the execution graph with information on each node (event). This information includes a user specified icon for each type of event and a tag with a string of text, if available from the original trace. The tool also displays execution time for each processor. As in other tools, it is possible to zoom in and out of the graph. In the example, the graph is zoomed and cannot fit the window, so scrollbars are used to navigate through the graph. VisAll uses the standard `Tcl/Tk` facilities to implement this functionality.

Control over the visualisation process is provided by a secondary pane to the left of the main visualisation window. The whole graph is reduced and shown in the small picture bottom-left. This picture also includes a small box indicating the part that is being visualised in the larger image. A zoom button controls the level of detail one wants to see in the main window. Last, on the left hand side top corner of the graph, VisAll shows detailed information about each event (node). It is sufficient to click on any node to obtain the information that corresponds to that event.

VisAll was designed from the beginning to allow several graphical interaction tools. VisAll supports a processor activity map, as shown in Figure 4.This tool works from the same graph, but only displays the points in time where each worker was active. VisAll provides as standard functionality the ability to jump between several sub-tools. In this case, one can just click on a worker bar in order to obtain the position where the worker is in the tree, and vice-versa.

The third tool currently supported in VisAll is the animated execution tool. The tool is activated from the main VisAll window using the “Animation Controls” bar shown in Figure 1. The bar includes a start button, a step-by-step button, a pause button and a stop button. Last it allows controlling execution speeding and auto-scrolling. Figure 7 shows an example of the animated execution tool in action. Note that the modular structure of VisAll allows having different animation tools for the same trace file. The one we currently have implemented uses a Vace-like approach to represent the dynamics of building execution trees.

3 VisAll In Operation

VisAll was designed and built so that it could visualise traces of different parallel systems. This includes both visualising traces from well known logic programming systems, and visualising synthetic trace files that can be used to predict the performance of parallel systems not yet implemented. To achieve this goal, VisAll is distributed with several filters that can translate from the formats used in other tools to VisAll's internal graph representation.

In the following sections we show how the current imlementation of VisAll visualises the major forms of parallelism and how it compares with other parallel logic programming visualisers.

3.1 Visualising ORP

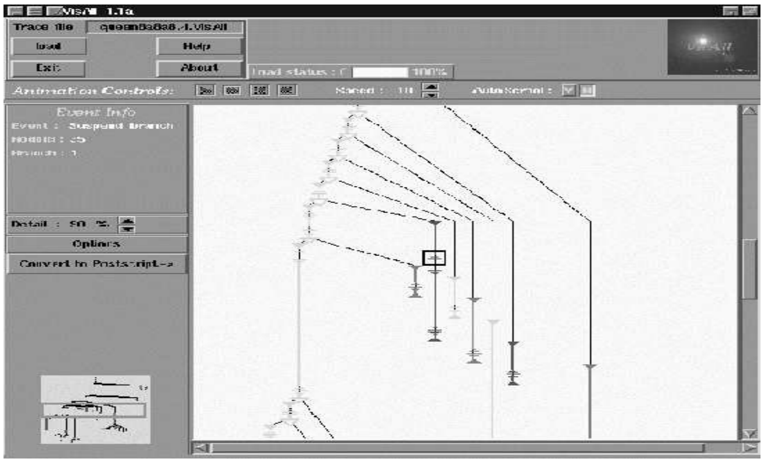

Figure 1: ORP: the 8-queens problem in Muse by VisAll

Or-parallel execution can be described through or-trees. VisAll can display, but is not limited to, parallel execution of the Muse [1] and Aurora [13] or-parallel systems.

Implementors of or-parallel systems have experimented with very different approaches to problems such as work distribution or side-effects in Prolog programs, that require sequentialised execution of parts of the program. Thus, execution trees for the same problem from different systems may look quite different.

Figure 1 shows how VisAll visualises the parallel execution of the all-solutions 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. An upwards pointing triangle (▲) indicates that a branch has been completed successfully, while a downwards pointing triangle (▼) represents the start of a branch. The downwards pointed stripped triangle represents branch suspension while an upwards pointing triangle after a suspension indicates resumption of the suspended branch.

One of the applications for our tool can be immediately identified in 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 alternatives. This may indicate that Muse is being too eager to suspend

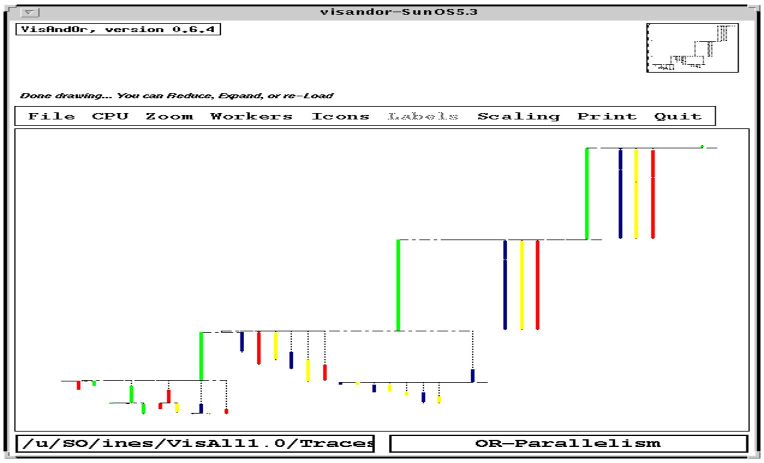

Figure 2: ORP: the 8-queens problem in Muse by VisAndOr

branches and seek for better (less speculative) work in the tree, resulting in fine-grained work.

Figure 2 shows the same trace file as visualised by VisAndOr. VisAndOr shows the same kind of tree, but does not show details. It is therefore harder to spot the "too many suspensions" problem.

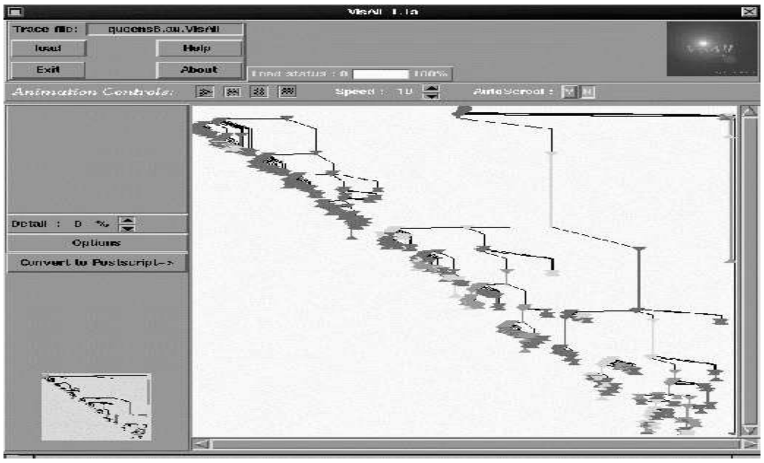

Figure 3: ORP: the 8-queens problem in Aurora by VisAll

Aurora [13], as Muse, exploits ORP in logic programming applications. Although both systems are based on SICStus Prolog, Aurora and Muse have major differences in their design. In particular, 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. Figure 3 shows the all-solution 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. Aurora tends to concentrate workers on the left hand side of the search tree. In contrast, 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.

3.2 Visualising And-parallelism

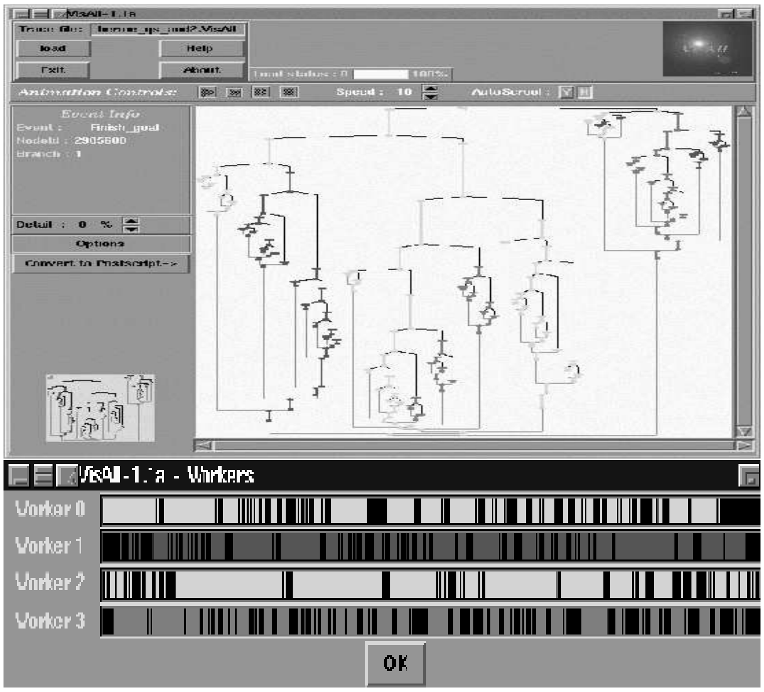

Figure 4: &-Prolog according to VisAll

VisAll can visualise parallel execution of systems that exploit DAP and IAP. Figure 4 shows an execution for the &-Prolog IAP system. The trace files generated by &-Prolog use the VisAndOr format and can be converted to the VisAll format. In Figure 4 we present the visualisation of a parallel execution of the quicksort program run in &-Prolog with four workers. Notice that work always *forks* from a conjunction, is run in parallel by the individual workers, and at the end *joins* to provide a partial solution. This is the *fork-join* structure typical of IAP. 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 4 also shows the VisAll's *activity map* for each processor allowing for easy investigation of the points at which workers became idle for longer time, and vice-versa.

3.3 Visualising DAP plus ORP

Andorra-I [26, 18, 25, 6] is a system that exploits ORP 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.

And-parallelism is depicted by parallel coloured lines that join in a same main branch of the execution tree. The top figure seems 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.

3.4 Visualising IAP plus ORP

A different approach for combining different kinds of parallelism in logic programs is to support both ORP and IAP. Examples include ACE [10] and the SBA [2]. One of the major goals of VisAll is to visualise execution of SBA. Since the system is not fully implemented, VisAll can only visualise *synthetic* trace files for the SBA style of execution.

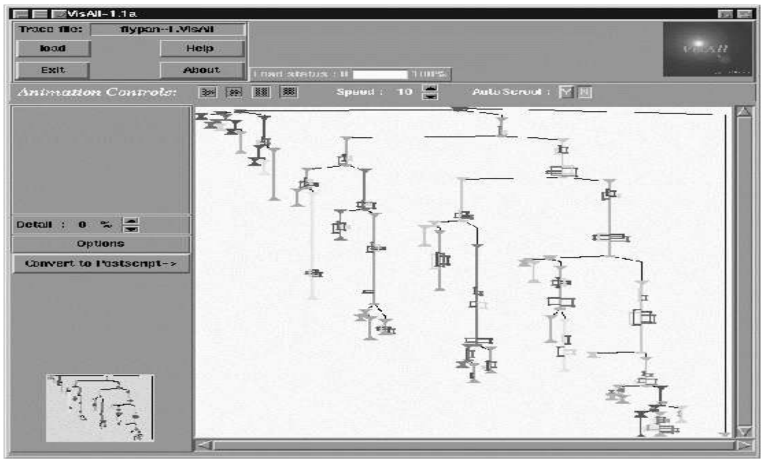

Figure 5: VisAll Visualisation of an Andorra-I Benchmark

Figure 6 shows the visualisation of a synthetic trace file of the SBA model, as visualised by VisAll.

The synthetic traces include information on the name of goals. A goal `a` unfolds in `b & h`. 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.

Alternatively, VisAll allows the visualisation of Vace trace files. The Vace trace format is based on the VisAndOr trace format, to which it adds

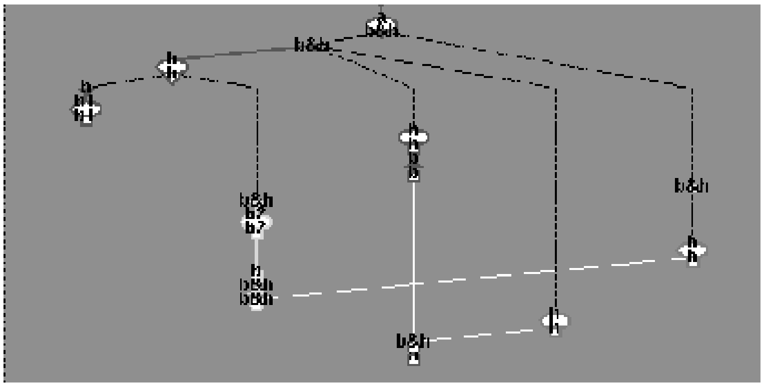

Figure 6: VisAll representation of IAP and ORP

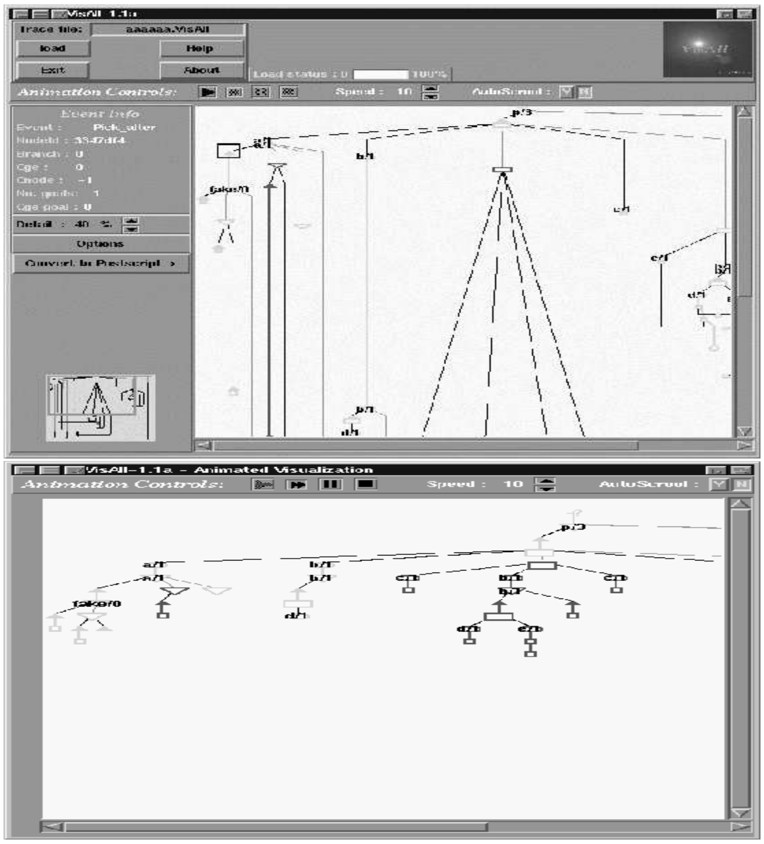

Figure 7: VisAll representation of an ACE program

extensions deemed necessary to support IAP and ORP through copying.

Figure 7 shows the execution of a small ACE program. Note that the Vace format also includes a field for procedures names. This field is understood by the translator, allowing the display of procedure names.

The top figure shows the static execution. ACE workers are grouped into *teams*. These teams execute in and-parallel, until they run out of and-goals to execute. When this happens, they ask other teams for work.

VisAll represents all the elements of a team with the same colour. Copying is represented as a light blue line. A copying operation consists of a set of light blue lines all terminating at approximately the same point in time. Colours are very important in VisAll to express several kinds of visualisations such as number of copies made by teams of workers, amount of and-parallel execution exploited by a team, or exchange of signals between workers. As the text is black and white, those details are only visible through actual demonstrations of the software.

Execution under ORP plus IAP can arguably be rather more complex than just ORP or IAP. VisAll's animated execution window is particularly useful to understand these cases. The lower window in Figure 7 shows a snapshot of this tool. The window shows how teams move in the search tree and is particularly useful in recognising how teams work together in exploring the search tree.

4 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, such as IAP, DAP, and ORP. It can display traces of execution statically and dynamically, while also displaying individual processor activity maps. All these components work together, in an integrated fashion. VisAll allows users to zoom in and out at parts of the execution tree, to see dynamic execution, and to obtain information about the logic program source. Moreover, VisAll was designed so that it can visualise traces of different parallel logic programming systems, even if those traces were obtained for other visualisation tools, and currently supports the VisAndor and Vace input trace formats.

Development of VisAll continues active. In the future, we plan to add new forms of parallelism, namely non-determinate dependent and-parallelism, in collaboration with Shen [19], and Pontelli and Gupta [16]. Moreover, we would like to extend the tool to give more detailed information about the system, such as why a processor (or worker) suspended, why it is positioned somewhere in the tree, and where in the source program it is executing code. We are particularly interested in finding what kinds of information are of interest to parallel logic programming system designers. We are also starting research into using 3D visualisation for visualising the complex execution of

AND-OR systems such as SBA or ACE.

So far, the development of VisAll has been directed towards the needs of parallel logic programming system developers. As parallel logic programming systems mature, systems like VisAll will be more and more important for developing and testing applications in parallel systems. We believe the modular structure of our system is ideal towards directing our tool towards the requirements of application developers.

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. Nuno Fonseca would like to acknowledge support from the Subprograma Ciência e Tecnologia do 2o Quadro Comunitário de Apoio.

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, Vítor 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. 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.

[7] N. A. Fonseca. VisAll Documentation. Technical report, LIACC, Universidade do Porto, 1998.

[8] 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.

[9] 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.

[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] 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.

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

[15] 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.

[16] E. Pontelli and G. Gupta. Non dependent and-parallelism revisited. Technical Report 1, Laboratory for Logic, Databases and Advanced Programming, New Mexico State University, Las Cruces, New Mexico, USA, February 1996.

[17] 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.

[18] 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.

[19] Kish Shen. Exploiting And-parallelism in Prolog: the Dynamic Dependent And-parallel Scheme (DDAS). In *Logic Programming: Proceedings of the Joint International Conference and Symposium on Logic Programming*, pages 717–731, 1992.

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

[21] J. Sunberg and C. Svensson. Muse trace: A graphic tracer for or-parallel prolog. Technical report, SICS, 1990.

[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] 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.

[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.