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

Nuno Fonseca and Vítor Santos Costa and Inês de Castro Dutra

Technical Report Series: DCC-97-17

Departamento de Ciência de Computadores – Faculdade de Ciências

&

Laboratório de Inteligência Artificial e Ciência de Computadores

Universidade do Porto

Rua do Campo Alegre, 823 4150 Porto, Portugal

Tel: +351+2+6001672 – Fax: +351+2+6003654

http://www.ncc.up.pt/fcup/DCC/Pubs/treports.html

VisAll: A new Tool to Visualise the 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

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). Independent and-parallelism was exploited in systems such as &-Prolog [11] and &-Ace [10, 8], or-parallelism in systems such as Aurora [12] and Muse [1], and dependent and-parallelism 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 [9], 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 further amplifies these problems. Novel research is tackling more sophisticated models of execution, such as efficiently combining different forms of problems, or how to obtain efficient operation on distributed programming machines or on virtual shared memory platforms. At the same time as parallel execution becomes more complex, parallel systems also become more sensitive to architectural parameters.

From the user's point of view, implicit parallelism simplifies the user's task, but is not a magic wand. Often, the kind and amount of parallelism available in the application are not clear. The

programmer may believe its application has great parallalelism, whereas actual execution only obtains fair speedups. In fact, small changes to the original program might result in huge performance improvements. Or it might be case the program exposes gaps in the parallel logic programming system, and the user should try other systems for ideal performance. Ideally, one would like for the user to be able to quickly understand the key factors in the parallel execution, without having to go through the gory details. Users should be able to understand system execution in their own language, so that they can more easily reason about system bottlenecks.

To fulfill these goals, 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 [4], Muse and Must [21], &-Prolog and Visandor [6], or ACE and Vace [23]. These systems have been very important in the development of parallel logic programming.

The design of visualisation systems can be seen from two major perspectives. The first is the interface between the visualiser and the parallel system. Traditionally, most systems have been designed towards visualising a particular system and thus a specific model of execution. Usually, there is some independence between the system and the visualiser, as they communicate through a well-defined interface, in most cases a trace-file consisting of a set of well-defined *events*, but the visualiser itself is bound to these events. This traditional design simplifies development, but makes it hard to adapt the system to the evolution of parallel logic programming.

A good case in point is Visandor. Although the system was originally designed for IAP, it was later extended to ORP (eg, Muse and Aurora), and then to DAP plus ORP (eg, Andorra-I). This development showed the virtues of the event based approach, as different trace formats from different systems could be processed in much the same way. On the other hand, it showed some of the problems. Mainly, good communication between both developers of VisAndor and of the parallel systems was needed to include support for new forms of parallelism. Moreover, whereas ORP has a structure in many ways similar to IAP, DAP does not. Hence it was simpler to obtain a good representation for ORP than to DAP in this framework.

The second major perspective results from seeing the visualiser as a graphical application. The visualiser presents a high level (be it static or dynamic) view of an execution graph. This view is in turn supported by a lower-level graphical toolkit. Two problems arise here: first, what form of presentation to use, and second, how best to exploit the underlying graphical environment.

It is our argument that visualisation tools should not be designed for a specific form of presentation. This goes against the direction taken by most of the current systems, but there are strong arguments favouring our point of view. First, a practical visualisation system should be able to present a different view of the system to developers, who are mostly interested in its inner workings, and to users, who are concerned with the general perspective and tracking down bottlenecks. Second, a system should be seen from different perspectives such as static execution tree, dynamic execution, or resource utilisation through time. All these paradigms should both be independent and consistent. This argues for starting from a paradigm free design, one that can provide functionality that can then support specific presentations. Last, it is now clear that the best form of presenting, say, DAP might not work as well for, say, IAP. To be usable for different parallel systems, the visualiser should have the ability to shift between different paradigms.

A more technical, but still fundamental, issue concerns the interaction with the graphical environment. To obtain maximum portability, one should use as little as possible from underlying toolkits. The price is code complexity. In fact, even portability is not guaranteed as even simple toolkits either evolve through time or will disappear. An alternative is to use a high-level toolkit, hence simplifying implementation. An example is DaVinci, a graph drawing toolkit that has been excellently used by Vace. Again, there is a price. One is largely bound by the design path of the toolkit. In the worst case, the toolkit may bind us to a particular presentation paradigm.

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:

- Visualisation of different parallel systems can be obtained by translating from their native trace formats to the general graph representation.
- The same graph can be used as a basis for different visualisations, depending on the intended audience and on the graph structure.
- In fact, several visualisations may be active simultaneously. In this case, the graph also operates as a channel between visualisers, ensuring consistency and enabling sharing.
- Last, the graph is the key to system organisation. As such it simplifies modularisation, and allows the use of different tools. For example, `C` is a natural language for non-interactive translation to the standard graph, whereas `Tcl/Tk` offers a good environment for graphical interaction, as we discuss later.

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 characterists 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.

For parallel logic programming implementors to actually use VisAndor, their systems must generate a sequence of events and dump it to a trace file. 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.

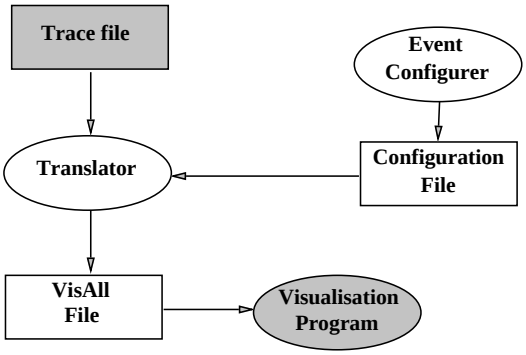

Figure 1: The VisAll Tool

Figure 1 shows the structure of VisAll. One can see the three major components: (1) the event configurer, (2) the visualisation programs, and (3) the translators:

The *translators* have the job of converting a given trace file to a VisAll internal format file containing a dependency graph representing the trace. One should note that the system was designed so that 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 available virtual memory.

The *visualisation programs* displays the VisAll files created by the translators. Two forms of visualisation are currently available, one through trees and the other through execution maps. Both communicate via the internal representation.

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 !.

We next describe the individual components in detail.

2.1 The Structure of the Graph

The major VisAll component is the graph representing execution. We had two options in designing the graph. We could have designed a neutral graph format, with no semantic information, or we could use a general format, where each node would include sufficient fields to contain all information of interest. Efficiency considerations made us choose the latter approach for the current version of VisAll. As a starting point we chose the VisAll format [13], the format that has been the most popular for parallel logic programming. The current node format is shown in Table 1.

Timestamp	Event	NodeId	Branch#	C-Node	#Goals	CGE#	goal#	PID	Info

Table 1: Fields in a VisAll Node

Each node contains information on **(i)** the timestamp, that is the point in time where the node was generated; **(ii)**, an event code, that determines the type of the node; **(iii)**, an unique identifier;

(iv) an or-branch number corresponding to a 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 OR/IAP [7]; **(vi)**, the number of and-parallel goals from the current CGE; **(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; and (2) a string field used to store source program information.

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 2: Main VisAll Events

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. The *Start* and *End 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 [7]. The *Create CGE* event corresponds to starting a traditional and-parallel conjunction for IAP. The *Start* and *Finish Goal* correspond to and parallel goals. Lastly, *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.

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 resposible for generating coordinates that can be used by a graphical tool. The resulting representation is organised as two components:

- A sequence of events, represented as a line with an event number and skeletal information on the event.
- 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 ara managed by common code that provides control and is 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 convertion 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 a *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 2 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 quitting. 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 5, page 11. 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 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 2. 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 8, page 14, 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 Or-parallel Execution

Or-parallelism arises in applications where executable goals match more than one clause in the database. 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 [12] 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 2 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 2, 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 branches and seek for better (less speculative) work in the tree, resulting in fine-grained work.

Figure 3 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.

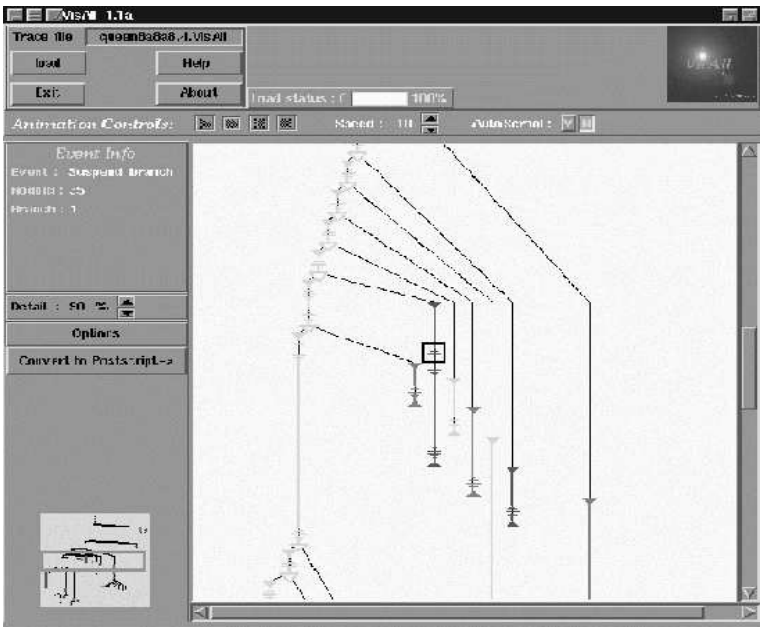

Figure 2: Or-parallelism: Visualisation of the eight queens problem in Muse by VisAll

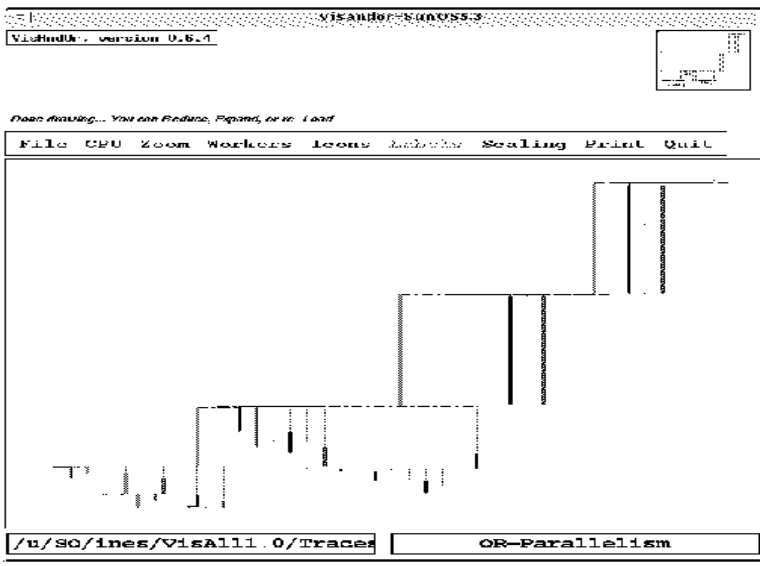

Figure 3: Or-parallelism: Visualisation of the eight queens problem in Muse by VisAndOr

Aurora [12], 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 4 shows the all-solution parallel execution of the eight queens problem with four processors, in Aurora.

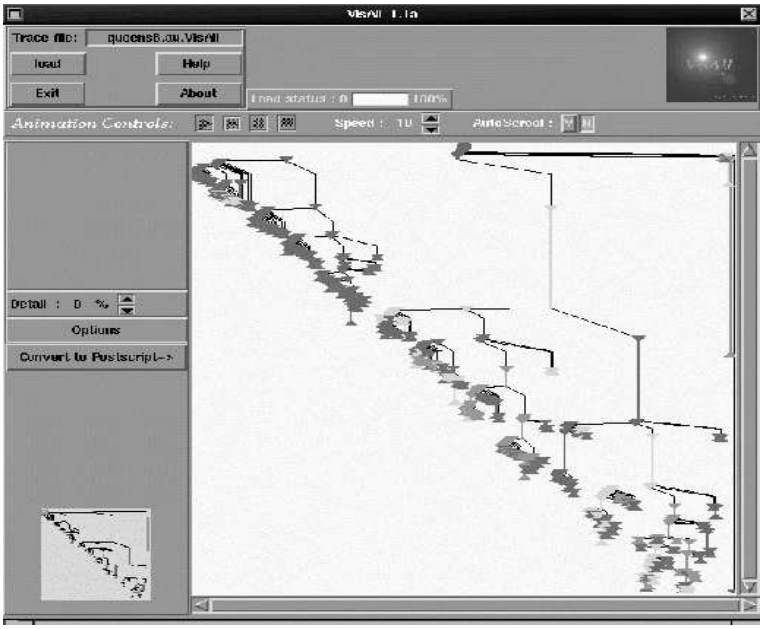

Figure 4: Or-parallelism: Visualisation of the eight queens problem in Aurora by VisAll

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

VisAll can visualise parallel execution of systems that exploit dependent and independent and-parallelism. Figure 5 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 5 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 5 also shows 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 time, and vice-versa. In the example, the map indicates some load imbalance, especially at the beginning and end of the execution.

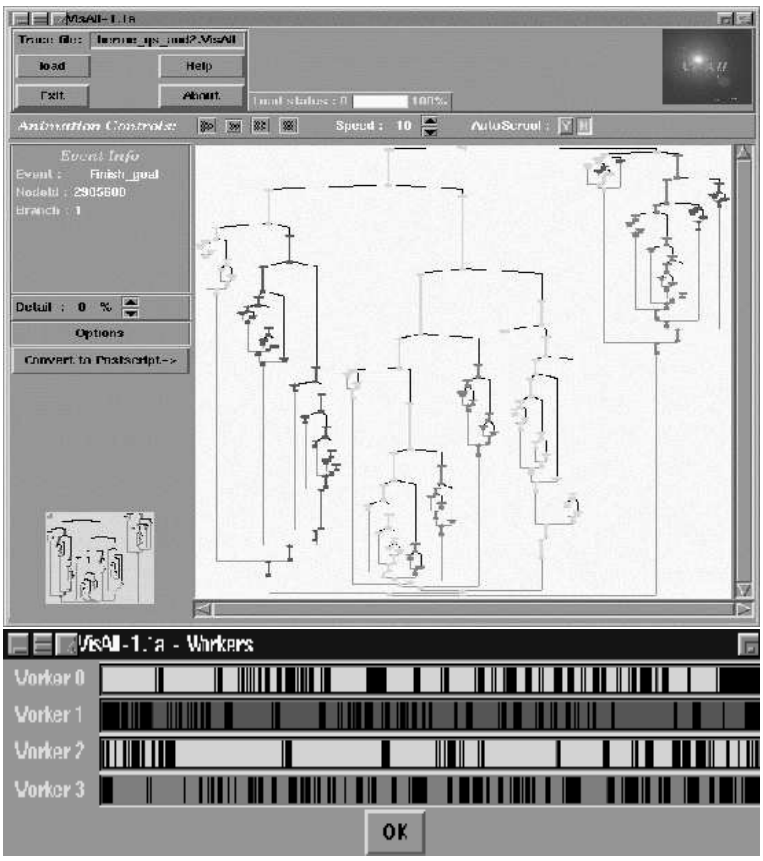

Figure 5: &-Prolog according to VisAll

3.3 Visualising Dependent And-parallelism plus Or-parallelism

Andorra-I [26, 18, 25, 5] 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 6 shows the last phase of the parallel execution of a flight allocation program run in Andorra-I. The program first sets constraints on the flights by using the Pandora method and then evaluates the system of constraints by first propagating the constraints and then trying the available alternatives to the unsolved 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.

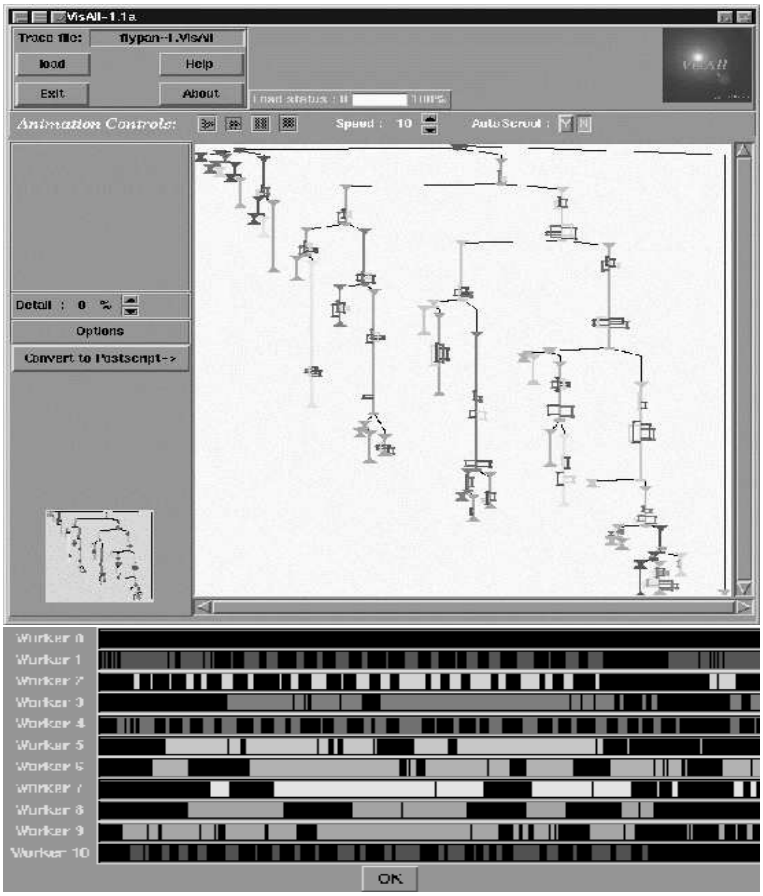

Figure 6: VisAll Visualisation of an Andorra-I Benchmark

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. The activity map confirms this insight, with long periods of idleness for most processors.

3.4 Visualising Independent And-parallelism plus Or-parallelism

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. Examples include ACE [9] 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 7 shows the visualisation of a synthetic trace-file of the SBA model, as visualised by VisAll. The file was generated by Dutra's recomputation Or plus Independent &-parallelism simulator.

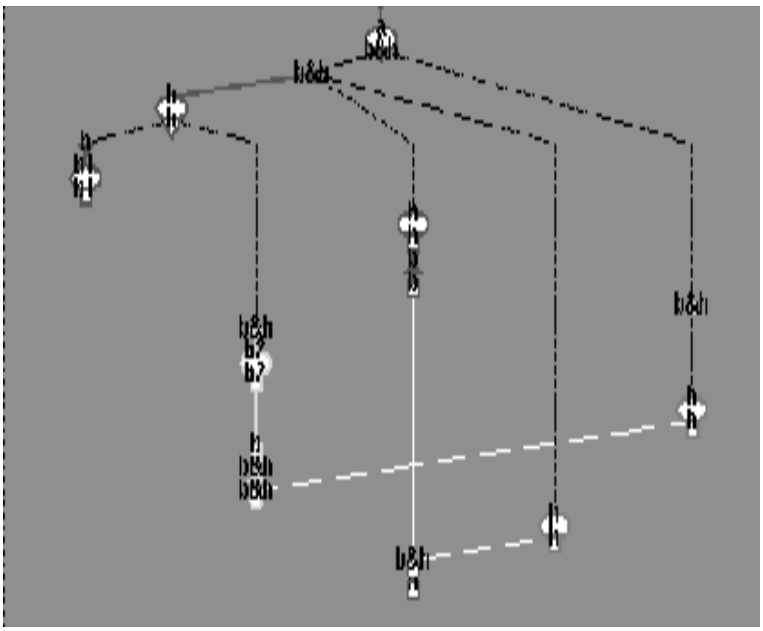

Figure 7: VisAll representation of independent-and and or-parallelism

Execution corresponds to a simple Prolog program shown in the next lines.

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

?- a.
```

The simulator generates VisAll-style traces that also include information on the name of goals. The 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 extensions deemed necessary to support independent and-parallelism and or-parallelism through copying.

Figure 8 shows the execution of a small VACE 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.

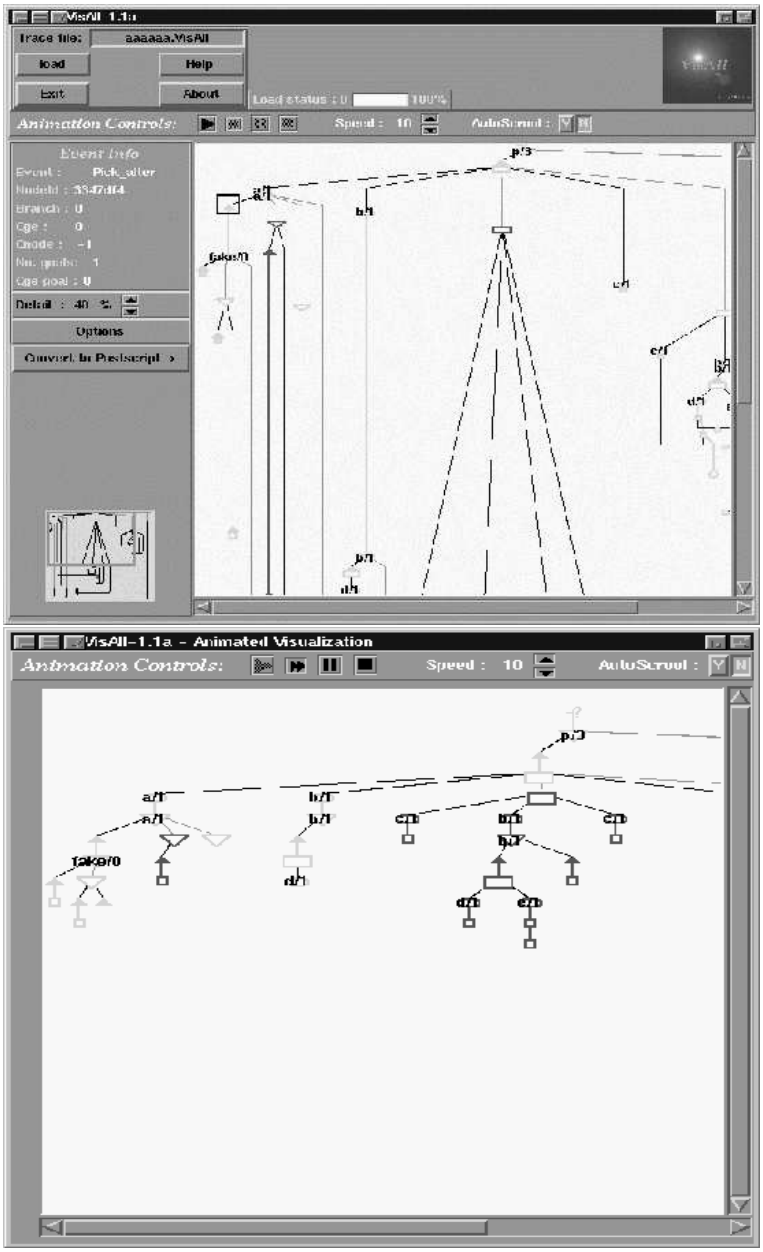

Figure 8: VisAll representation of an ACE program

The top figure shows the static execution. To follow the picture one must first understand that in 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. If such work is available, say team T_0 finds out that team T_1 has work, team T_0 will copy a number of alternatives and CGEs (Conditional Graph Expressions containing IAP) from T_1 into its workspace.

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 times all terminating at approximately the same point in time. One can notice that in this example only three copying operations were performed, and that in all cases the copies were taken from the pink team. The first team to copy was the red team. One can notice the red team copied two choice-points and one CGE. The second copying operation was performed by the yellow team, who copied four choice-points and the same CGE.

The Figure clearly shows and-parallel within the pink team. Note that the first alternative to `a/1` and the deterministic goal `b/1` were executed in parallel within the same team. Execution of the yellow team, to the right of the picture, also shows some issues in IAP execution. When the yellow team starts its copy of the CGE, it launches an inner CGE for `b/1`, which itself launches goals `d/1` and `e/1` in parallel. These two goals succeed and perform a join but the ensuing computation results in failure. This failure is propagated to other workers under the CGE through a *kill* signal (✪) that eventually forces the computation under the second CGE to be relaunched.

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 8shows 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 independent and-parallelism dependent and-parallelism, and or-parallelism. 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 VACE.

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 by the Subprograma Ciência e Técnologia do 2o Quadro Comunitàrio 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] 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.

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] M.Hermenegildo M. Carro, L. Gómez. Implementation of an event driven scheme for visualizing parallel execution of logic programs. In *JICSLP'92 Workshop on the Parallel Execution of Logic Programs*, 1992.

[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] 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.