

# Design of a Resolution Multiprocessor for the Parallel Virtual Machine

Hamish Taylor

Department of Computing and Electrical Engineering  
Heriot-Watt University, Edinburgh EH14 4AS, Britain

e-mail: hamish@cee.hw.ac.uk

## Abstract

PAN is a process based Prolog system running in a distributed manner on the nodes of a virtual multiprocessor. It was designed to be a general purpose parallel logic programming system that would be resilient to software and hardware evolution. Programs can explicitly control parallelism through passing general terms synchronously or asynchronously among PAN's Prolog engines. Alternatively several forms of parallelism can be implicitly extracted by compilation techniques and runtime scheduling on top of this architecture. The computational platform or Parallel Virtual Machine is a meshed set of daemons on a network of heterogeneous workstations offering common message passing, configuration and job control services. PAN is presented to users under X Window as a multi-headed extension of single processor Prolog systems. An interface to each Prolog engine is displayed in a separate X widget together with a console window to the PVM, and window based methods are supported for simultaneously invoking all engines together.

Keywords: Prolog, PVM, X Window, evolutionary, message passing, process based

## 1 Introduction

Interest in process based logic programming systems featuring explicit communication and distributed execution has recently revived [2], [15], although research has been continuing in this area over the years [6], [8]. This type of approach to developing parallel logic programming systems is not as promising for exploiting parallelism as more specialised approaches such as [11]. However, it substantially reduces the effort required to build such systems, and avoids making the systems architecture so special that evolution in enabling technologies will soon render it out of date. Design and performance improvements to hardware, machine architectures, operating systems, and to sequential processing technology happen continually, making niche programming systems specialised for particular platforms obsolete. While much effort can be invested to make such systems maintain pace with developments, their small user base, and the long lead times to develop them alongside continual improvements in sequential processing technology tend not to make this investment worthwhile.

Process based logic programming systems offer part of an appealing strategy for evading this difficulty. Since they use sequential processes as their building blocks, they can use the latest in portable sequential processing technology to empower these processes. With careful modular design to allow a clean separation of these processes from extensions to support interprocess communication, synchronisation, and system configuration, such systems should be able to evolve gracefully and easily with evolution in the processing technology of their components.

However, processing platforms also continually evolve. Improvements to processors, better machine designs, and enhanced operating system functionality mean that parallel implementations, which

have to be targeted at some form of parallel architecture, become dated with the parallel machine version they are customised to. Furthermore, diversity in operating system characteristics and in parallel platform configurations, and lack of standardisation in their architectures make parallel implementations awkward to port between platforms, and likely only to run on a small minority of available systems. If a process based logic programming system could abstract from the differences among parallel platforms and view many of them in a common way, it would be less vulnerable to being made obsolete by platform evolution and would be more portable to diverse parallel machines.

Virtual multiprocessors offer the capability of hiding the idiosyncratic features of the underlying processing platform. They supply a software abstraction that lets parallel programs running on the virtual platform maintain a uniform view of underlying processing resources. Using them incurs a performance penalty, but one that can often be quite acceptable where straightforward mappings of virtual multiprocessor services can be made onto the underlying platform. Since virtual multiprocessors offer general purpose functionality of broad applicability, their much wider user base can justify the tracking of technology evolution in hardware and operating systems that specialised parallel logic programming systems cannot do.

## 2 Design Choices of PAN

PAN is a parallel logic programming system that has incorporated these objectives of easy adaptation to technology evolution and wide availability into its design. It combines a process based approach to logic programming with use of a virtual multiprocessor to attain a sustainable systems architecture that should gracefully adapt to change and be available on a wide range of platforms. PAN uses actively maintained, off the shelf components to achieve this, which has greatly eased the effort involved to construct it.

PAN combines

- resolution engines
- model of multiprocessing
- virtual multiprocessor
- user interface

to realise a systems architecture with evolutionary resilience.

### 2.1 Resolution Engines

Prolog was the obvious choice for a resolution engine if PAN was to be as general purpose as possible yet use off the shelf conventional technology where available. The only other mature style of resolution engine executes the committed choice languages. However, Prolog is more expressive, generally familiar, and suitable for the most relevant class of applications, knowledge based systems, than languages like KL1, FCP or PARLOG. Mixing the two different styles of resolution engines would also have been an attractive possibility to enhance functionality [14] [4], but would have created significant extra complexities in mediating interactions among the different styles of resolution.

Among the many available Prolog implementations, a version was sought that was actively maintained, state of the art in performance, highly portable, provided access to source code, was widely used, conventional in character, cheap to acquire, had a C interface, and supported delayed goals. SICStus Prolog fitted these requirements well, so it was chosen as PAN's resolution engine component.

## 2.2 Model of Multiprocessing

Conventional Prolog engines get deployed for multiprocessing where the parallelism is either exploited implicitly or explicitly. Implicitly parallel Prologs extract various opportunities for parallel execution transparently at the resolution processing level in a data or demand driven manner. For example or-parallelism is exploited implicitly by Muse [1], independent and-parallelism by &-Prolog [10] and stream and-parallelism by Parallel NU Prolog [12]. Combinations of forms of parallelism are also being exploited such as or-parallelism and deterministic and-parallelism by Andorra I [5]. However, such capabilities usually require major alterations to conventional sequential Prolog engine technology particularly to manage memory. Thus while impressive speedup results can be obtained, such systems are not robust to evolution in their enabling technologies and require substantial maintenance effort to remain state of the art.

Explicitly parallel Prologs exploit parallelism in a control driven manner. By reflecting control up to the programmer's level, they allow much greater flexibility in their exploitation. In particular they allow emulation of data or demand driven processing on top of their control driven approach. Process based Prologs offer a good accommodation between a control driven approach and a conservative use of mainstream Prolog technology. They sacrifice the ability to exploit parallelism in a fine-grained way in return for using sequential processing technology directly. Multiprocessing is achieved by spawning multiple Prolog processes and enabling communication among them in ways that allow synchronisation. Communication among process based Prologs tends either to be

- blackboard
- message

based depending upon whether it is done through a shared store or by explicit message passing.

Blackboard based logic programming systems usually follow the LINDA model [3]. A shared tuple space serves as the message exchange zone and Linda operations on the tuple space coordinate and organise the multiple threads of execution. Representative examples of blackboard based logic programming systems include Multi-Prolog [2] and Prolog-D-Linda [13]. Such systems provide appealingly expressive ways for explicitly controlling multiple threads of resolution. However, the reliance such systems place in communicating via a shared store builds in a contention bottleneck that is a threat to their scalability.

Message based systems have the advantage over blackboard based systems of being inherently more scalable. They also avoid the inefficiency of making all interprocess communication indirect. They are not quite as expressive, because they force users to address all messages or consign them to named channels, but in practice this drawback can be finessed. Representative examples include CS-Prolog [8] and PMS-Prolog [15]. A common design choice of such Prologs has been to support dynamic process creation and to adopt a purely synchronous style of message reception.

Dynamic process creation provides flexibility and expressiveness but has rather high overheads, since it is normally realised by forking and overlaying processes. Dynamic creation would make better sense if lighter weight processes were employed. IC-Prolog-II shows how cheaply such a means can create new threads of resolution. However, the process and memory interleaving required to underpin this functionality, rules it out for PAN as too radical a departure from the well maintained norm of sequential processing technology to ensure evolutionary resilience. The more obvious design choice is to avoid dynamic process creation overheads by determining the width of process parallelism at start up time. Since the grain of parallelism is not readily flexible at a reasonable cost, efficient multiprocessing dictates exploiting parallelism in only a coarse grained manner. Granularity analysis of logic programs at compile time would be an important aid to doing this most effectively [7].

The choice of existing message passing Prologs to support only a synchronous style of message reception is also curious. Asynchronous message reception has higher implementation overheads,

but can be more flexible in being demand driven, letting multiple messages be simultaneously in flight, and avoiding the idle time involved in blocking the receiver until the message is available. There is also a natural way of handling asynchronous communication within Prolog using delayed goals. That demand sensitive communication is needed by applications is clear from the support offered in existing message passing Prologs for such unresolution like activities such as polling. Such ad hoc and inelegant measures could be avoided by supporting both synchronous and asynchronous forms of message reception. Doing that would also close the gap of expressiveness of such message passing Prologs with blackboard based Prologs.

Combining these considerations suggested the adoption of a model of multiprocessing that

- is control driven
- exploits parallelism in a coarse grained way
- creates processes statically
- communicates by synchronous and asynchronous message passing

Extra primitives added to the Prolog language can be used to control message passing and achieve synchronisation. Prolog engines can be farmed out to the width of process parallelism required on the parallel platform at start up time. The ratio of work done to communication can be made low by chunking it to exploit parallelism coarsely. Expressiveness and functionality can be maintained by supporting asynchronous communication as well. However, PAN needs further support if it is also to be resilient to platform evolution.

## 2.3 Virtual Multiprocessor

The Parallel Virtual Machine or PVM [9] is a widely used virtual multiprocessor that runs on many different kinds of platforms. It allows a heterogeneous network of parallel and serial computers to appear as a single concurrent computational resource. It consists of daemon processes to run on participating nodes and a shared library of routines for initiating remote processes, for communicating among processes, and for changing the configuration of the machine. It offers a suitable software abstraction from diverse parallel architectures and supports portability across multiple different platforms. It also offers the important new capability of integrating networks of suitable serial computers into a single parallel machine. By allowing local area networks of single processor workstations running suitable operating systems to be configured as multiprocessors, it greatly expands the availability of platforms for parallel processing purposes.

Two features of PVM aid its usefulness as a virtual multiprocessor for PAN. PVM supports a console interface that allows interactive control and monitoring of its processing resources and tasks running on them. This is important for PAN as it is intended that it should be able to run on LANs of UNIX workstations whose use is shared with others. Coping with availability problems of such nodes for a PVM session is greatly aided by the ability to configure and monitor the PVM separately from the parallel Prolog system that runs on top of it. PVM also supports asynchronous communication as its basic message passing mechanism, reflecting the indirection provided by using daemons to mediate in communication. PVM tasks only directly pass messages to and from their local daemon, which in turn pass messages among themselves at their own pace to complete the communication circuit. Since asynchronous communication can be used to implement synchronous communication but not vice versa, PVM's predisposition for asynchronous communication is readily suited to support both styles of communication.

## 2.4 User Interface

Modern computer users expect WIMP interfaces offering direction manipulation facilities. In the world of concurrent operating systems with full networking capabilities, that means X Window preferably with the Motif widget set. In that environment Prolog is usually presented to the user via a simple interactive command interface within a virtual terminal window. An obvious augmentation of that interface is to add a button panel in a pane above the virtual terminal that can be directly manipulated to pop up menus and panels in ways that provide a short cut to formulating query commands within the virtual terminal.

Such an interface naturally extends to becoming an interface to a cluster of Prologs running on different processors if it is replaced by a cluster of virtual terminal windows with button panels each interfaced to a different Prolog engine. If we add an extra window to provide access to the PVM console and to offer button panel control over the whole system, an interface configuration such as shown in figure 1 is obtained.

A parallel session on such a system might be initiated by going to each interface in turn and making each Prolog consult its own program and then execute a distinctive query over it. Clearly this would be too cumbersome where many Prolog engines are involved. A simpler idea is to provide functionality in the console window to broadcast the same goal to each Prolog interface. Each Prolog can be made to consult a common Prolog program and then to execute a common query. If the common Prolog program is written to allow each Prolog engine upon executing the common query to test which Prolog it is and then to solve its part of the common program, each Prolog would be executing its own code.

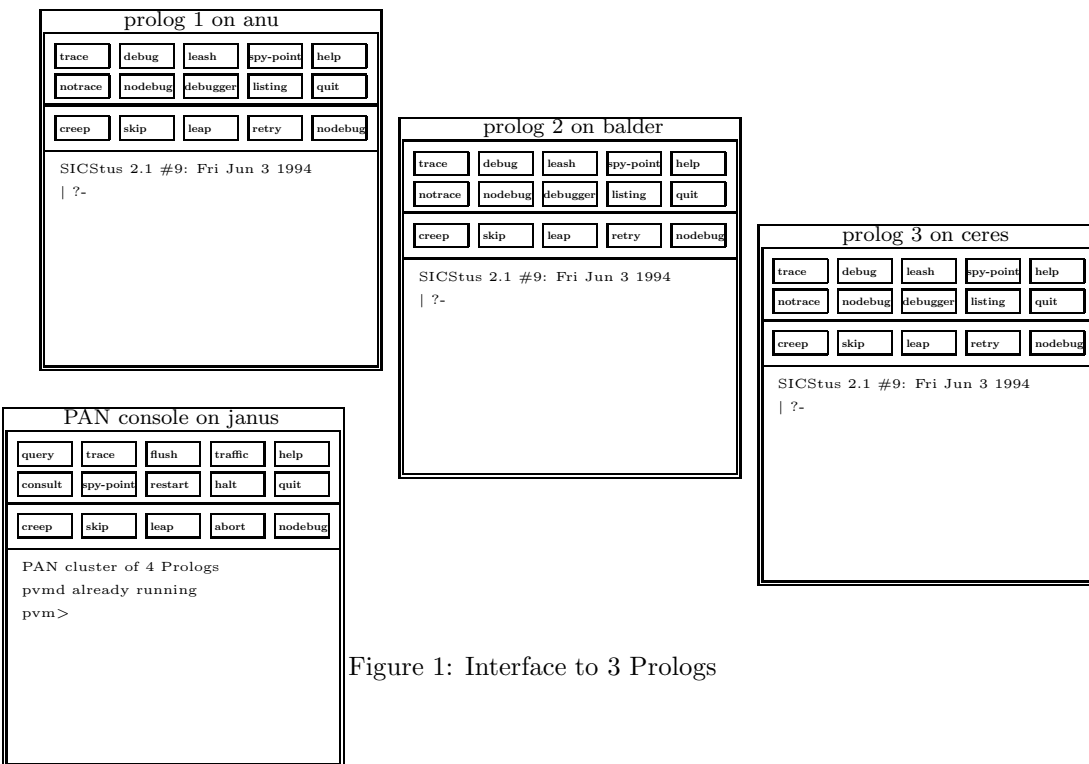


Figure 1: Interface to 3 Prologs

This was the solution adopted for the PAN interface. One major virtue is that it preserves unaltered the user's familiar interface to each Prolog engine. Queries and interactive debugging can be performed in exactly the same way as with an ordinary sequential Prolog process. Thus users

should find it relatively easy to migrate to using PAN from using SICStus Prolog. A second major virtue is that it allows each Prolog engine to remain completely ignorant of the X Window interface. All commands to a Prolog engine even those activated by pressing buttons are pasted into its virtual terminal pane and seen by it on its input stream. Thus the extra interface functionality is provided purely at an X Window level, which keeps the fat X libraries out of the Prolog executable. Further session control provided by PAN console buttons is realised by using PVM to send operating system signals to remote Prologs.

### 3 Implementation of PAN

PAN has been implemented under 3 versions of UNIX, SUNOS4.1+, Ultrix and Solaris5+ and on 3 hardware platforms, Sun-3s, Sun-4s and DEC Workstations. It currently runs on a local area network of over 50 workstations having a common distributed filing system supported by an automount facility. PAN has booted on up to 50 separate hosts using combinations of 4 different operating system and hardware configurations. Management of the different executables for the different architectures on so many different hosts was largely made possible by the uniform view provided by the distributed file system of the executables' locations.

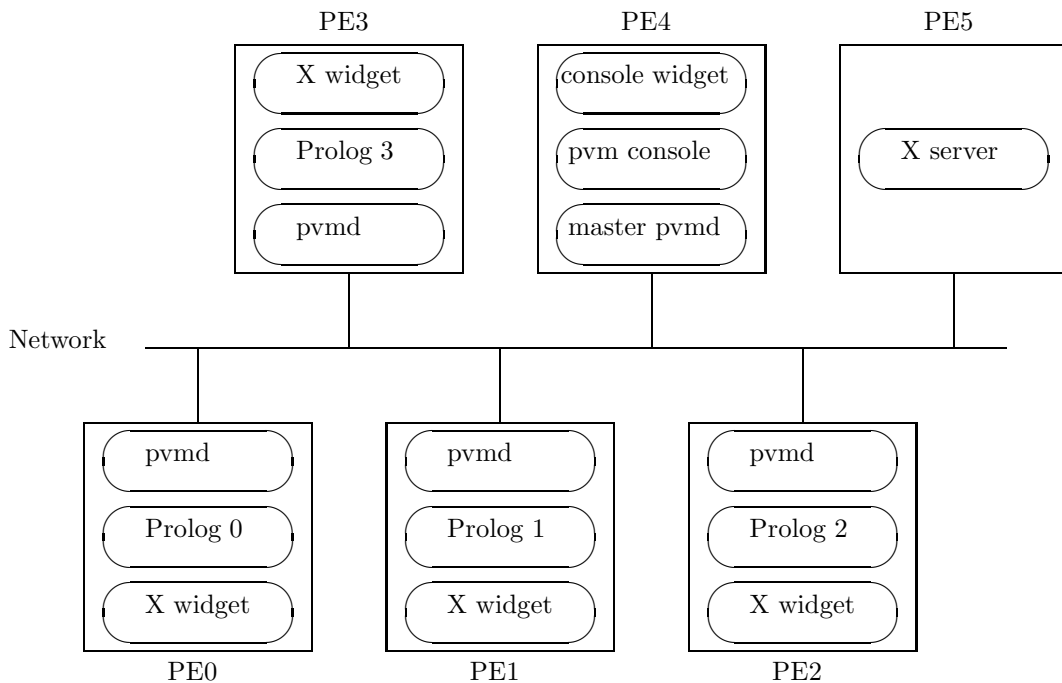


Figure 2: PAN Configuration of 4 Prologs

Currently hosts in PAN are of three different types

<b>Host Type</b>	<b>Main Job</b>
<i>Interface</i>	display all X widgets
<i>Communications</i>	manage Parallel Virtual Machine
<i>Worker</i>	run Prolog Engine

The Interface host is unique and only displays the windows of X widget processes that run on the Communications host and Worker hosts. The Communications host is also unique and runs the master PVM daemon and a PVM console process. Worker hosts support a Prolog engine each of the PAN system. Figure 2 shows the various processes running on 6 hosts to support a PAN configuration of 4 Prolog engines. The Parallel Virtual Machine consists of a master PVM daemon running on the Communications host, PE4, and 4 local PVM daemons running on Worker hosts. A PVM console process is also run on the Communications host to allow monitoring and interactive control over PVM. Alongside the PVM console process and the Prolog engines run X widget processes to which the PVM console and Prologs are attached through pseudo-tty device interfaces. The X widgets are ordinary xterm processes with their functionality augmented by extra code to support their button panels. All widgets display on PE5.

A PAN session is launched in four main stages:

Stage	Host Type	Job
start up	interface PE	farm out master X widget with PAN launcher
establish PVM	communications PE	farm out daemons to hosts probed as alive
farm out Prologs	communications PE	farm out X widgets with attached Prologs
initialise PAN	worker PE	Prologs register with PVM and learn of others

A PAN session is started from the Interface host which farms out the PAN launcher attached to the master X widget to the Communications host. The PAN launcher starts by probing hosts on the PAN configuration file to establish which hosts may be used in the PAN session and then ensures the PVM is established on these hosts with the master daemon running locally. Prolog engines with attached X widgets are then farmed out to the Worker hosts. When the Prolog engines boot, they register with the PVM and learn of each other so that they are fully ready, when they come up, to participate in a PAN session. The last act of the PAN launcher process is to **exec** over itself the PVM console process which displays a simple command interface in the virtual terminal pane of the PAN console X widget.

In order to ensure maximum evolutionary resilience, completely standard SICStus Prolog engines are used. They are only specialised for PAN at boot time by using SICStus Prolog's *.sicstusrc* configuration file to ensure that extra code is loaded in to define a few PAN related primitives and to induce an initialisation sequence with PVM. A standard version of PVM is also used. Thus upgrading PAN, as SICStus Prolog and PVM evolve, should be completely straight-forward where its components maintain relevant user level backward compatibility.

## 4 PAN's Programming Model

PAN adds a few extra primitives to SICStus Prolog to allow programs to pass messages among each other and to obtain PAN session information. The number of primitives is kept small to minimise the learning effort to use PAN. Primitives use the convention of identifying Prolog engines by a number from zero up to the number of Prolog engines running in a session. The four communication primitives are:

<code>rx(Term, Id)</code>	blocks until caller synchronises with transmission from Prolog Id and then Term is unified with message sent
<code>tx(Term, Id)</code>	blocks until caller synchronises with receiving Prolog Id and then Term is sent
<code>rxnb(Term, Id)</code>	sets up Term to be unified with message sent from Prolog Id
<code>txnb(Term, Id)</code>	sends Term asynchronously to Prolog Id

The synchronous communication primitives *tx/2* and *rx/2* block until a complementary pair synchronise whereupon a message is passed. The asynchronous communication primitives *rxnb/2* and *txnb/2* don't block. Asynchronous transmission sends off a message which either is received immediately if an *rxnb/2* has already set up the means to do so or sits within the PVM delivery system until an appropriate *rxnb/2* or *rx/2* predicate is executed. Asynchronous reception is set up by *rxnb/2* which appoints a variable to be bound to the next receivable message. This message may already be awaiting delivery or not yet be delivered. Delivery takes place at the earliest opportunity to do so. Backtracking across the point at which an *rxnb/2* was executed cancels the association of its message variable with being bound to the next deliverable message. Leaving the Prolog identifier unbound in either an *rx/2* or an *rxnb/2* predicate signifies that any sender will do, and the argument is bound to the identifier of whichever Prolog happens to supply a receivable message.

When a message is delivered asynchronously, the message variable is bound to it, and any goals delayed on the message variable will be awoken and run immediately. Thus a conjunction of goals such as

```
freeze(Term, handle(Term, Id)), rxnb(Term, Id)
```

will set up the goal *handle(Term, Id)* to deal with the next receivable message sent asynchronously to the caller. As asynchronous message reception can happen unpredictably at any point and then be undone by local backtracking across it, handler goals have to treat their messages as transient data and either deal with them immediately or else record them in the database for later handling by other parts of the program.

PAN supports the passing of arbitrarily large terms between Prolog engines in one go which permits much flexibility in developing applications on top of PAN. Unbound variables in communicated terms are allowed but get freshly renamed on reception to ensure all variables have purely local scope. PAN follows PMS-Prolog's reasons [15] in not supporting any form of backtracking on communication.

The two informational primitives

<code>prolog(Id)</code>	unifies Id with identifier of calling Prolog
<code>no_of_prologs(N)</code>	unifies N with number of Prologs in PAN session

allow all the Prolog engines in a PAN session to execute a common Prolog program and then to conditionalise their behaviour upon who they are and how many other Prolog engines are also running. This works well in conjunction with PAN's broadcast method for posing the same query to all the Prolog engines in a session.

## 5 Conclusion

A flexible and sustainable architecture for logic programming in parallel needs to be assembled from widely used, modern logic programming components based on well established, portable and evolving state of the art technology. The system can then grow with its components, be usable by many, and have a chance of being familiar to its potential users. PAN aspires to be such a system by combining SICStus Prolog, PVM and X Window technology to create a message passing distributed Prolog system running on a virtual multiprocessor. The PAN system has been successfully implemented and has currently been tested on up to 50 heterogeneous workstations on a local area network that shares a common distributed filing system. Future work on PAN aims to develop compilation and runtime scheduling techniques to enable the automatic extraction of and- and or-parallelism from ordinary sequential Prolog programs, to develop the PAN architecture further particularly in the area of interface support for parallel debugging, to benchmark performance, and to develop real applications that will demonstrate PAN's parallel programming potential.

## 6 Acknowledgements

The author would like to acknowledge the contributions of several postgraduate students in developing versions of PAN and in particular the efforts of Joseph Totten, Paul Connolly, and David Wagner.

## References

- [1] K.A.M. Ali and R. Karlsson. Full prolog and scheduling or-parallelism in muse. *International Journal of Parallel Programming*, 19(6):445–475, December 1990.
- [2] K. De Bosschere and J-M. Jacquet. Multi-prolog: Definition, operational semantics and implementation. In D.S. Warren, editor, *Proceedings of 10th International Conference on Logic Programming*, pages 299–313. MIT Press, Budapest, 1993.
- [3] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–459, 1989.
- [4] D. Chu and K. Clark. *I.C. Prolog II: a Multi-threaded Prolog System*. Department of Computing, Imperial College, London, 31 May 1993.
- [5] V. Santos Costa, D.H.D. Warren, and R. Yang. The andorra-i engine: A parallel implementation of the basic andorra model. In *Proceedings of 8th International Conference on Logic Programming*, pages 825–839. MIT Press, 1991.
- [6] J.C. Cunha, P.D. Medeiros, M.B. Carvalhosa, and L.M. Pereira. Delta prolog: A distributed logic programming language and its implementation on distributed memory multiprocessors. In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 335–356. John Wiley, Chichester, 1992.
- [7] S.K. Debray, N-W. Lin, and M. Hermenegildo. Task granularity analysis in logic programs. In *Proceedings of the 1990 ACM Conference on Programming Language Design and Implementation*. ACM Press, June 1990.
- [8] S. Ferenczi and I. Futo. CS-prolog: A communicating sequential prolog. In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 357–378. John Wiley, 1992.
- [9] A. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and Vaidy Sunderam. *PVM 3.0 User's Guide and Reference Manual*. Oak Ridge Nat. Lab., Tennessee, February 1993.

- [10] M.V. Hermenegildo and K.J. Greene. &-prolog and its performance: Exploiting independent and-parallelism. In *Proceedings of the 7th International Conference on Logic Programming*, pages 253–268. MIT Press, 1990.
- [11] P. Kacsuk. *Execution Models of Prolog for Parallel Computers*. Pitman, London, 1990.
- [12] L. Naish. Parallelizing nu-prolog. In R.A. Kowalski and K.A. Bowen, editors, *Proceedings of the 5th International Conference/Symposium on Logic Programming*, pages 1546–1564. MIT Press, Seattle, Washington, January 1988.
- [13] G. Sutcliffe. Prolog-d-linda v2 : A new embedding of linda in SICStus prolog. In K. De Bosschere, J-M. Jacquet, and P. Tarau, editors, *Proceedings of ICLP'93 Conference Workshop on Blackboard Based Logic Programming*, pages 105–117. ICLP'93, Budapest, June 1993.
- [14] H. Taylor. Coupling committed and trial binding resolution engines. In *Data and Knowledge Engineering*, volume 6, pages 159–178. North-Holland, 1991.
- [15] M.J. Wise, D.G. Jones, and T. Hintz. PMS-prolog: A distributed, coarse-grain-parallel prolog with processes, modules and streams. In P. Kacsuk and M.J. Wise, editors, *Implementations of Distributed Prolog*, pages 379–404. John Wiley, Chichester, 1992.