Assembling a resolution multiprocessor from interface, programming and distributed processing components

Assembling a Resolution Multiprocessor from Interface, Programming and Distributed Processing Components

Hamish Taylor
Heriot-Watt University, Edinburgh, Scotland

Abstract - An effective resolution multiprocessor can be built from distributed processing, logic programming, and interface elements. Widely used, portable, components can be modularly composed into a portable parallel system that displays good resistance to premature obsolescence by software evolution. A virtual multiprocessor offering common message passing and configuration services integrates a distributed mesh of sequential resolution engines. Users configure and control the resolution engines and virtual multiprocessor through a GUI using an embedded command language to drive its facilities. Prolog programs either explicitly control parallel execution through message passing or would have to rely on program transformation techniques to extract parallelism implicitly.

Prolog PVM X Window TCL/TK Expect

1 Introduction

Modular assembly of parallel programming systems from widely used, off the shelf components more quickly produces systems that are less likely to become obsolete than designing and building them from scratch. Timeliness is particularly important with the development of parallel systems where the lead times tend to be much longer. If sequential systems are developed in too short a time compared with parallel systems then a parallel system based on today's sequential implementation technology may actually run slower than the latest sequential implementation when the parallel system is eventually built. Even where this doesn't quite happen, the latest sequential implementation will often yield much better performance relative to the cost of the processing platform deployed than the parallel system that was so much longer in development.

Process based programming systems offer part of a strategy for evading this difficulty. Since they use sequential processes as their building blocks, they can use the latest in portable sequential processing technology. 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 as their components evolve.

Interest in process based logic programming systems featuring explicit communication and distributed execution has recently revived [1], [2], although research has been continuing in this area over the years [3], [4]. This approach to developing parallel logic programming systems is not as promising for exploiting parallelism as more specialised approaches such as [5]. However, it substantially reduces the effort required to build such systems, and avoids making the processing elements of the systems architecture so special that evolution in sequential processing technology will soon render them out of date.

A process based approach is not enough to avoid early obsolescence. Parallel logic programming systems also face changes created by continual improvements to processors, better platform designs, and enhanced operating system functionality. They mean that parallel implementations, which have to be targeted at some parallel architecture, become dated with the version of the parallel platform they are customised to. While substantial effort can be invested to make them maintain pace with the platform's evolution, their user base tends to be small and does not make this investment worthwhile. Furthermore, diversity in operating system characteristics and in parallel platform configurations, and lack of standardisation in their architectures make parallel implementations awkward to port among platforms, and likely only to run on a small minority of available systems. Thus a process based logic programming system would have to be able to abstract from the differences among parallel platforms and view them in a common way, if it was to avoid being made obsolete by platform evolution and was to be readily available on many parallel architectures.

Virtual multiprocessors allow idiosyncratic features of the underlying platform to be hidden. They supply a software abstraction that lets parallel programs running on the virtual multiprocessor maintain a uniform view of underlying processing resources. 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. Their substantial user base also justifies investing effort to make the virtual multiprocessor available on many parallel platforms.

Graphical User Interfaces or GUIs are another source of demands to alter and adapt the software of a parallel programming system with time. GUIs make it easier for users to develop programs interactively and to control the runtime behaviour of parallel programs. However, unless such interfaces are readily reconfigurable to meet changing user needs and developing ideas about how to

realise current needs, they are unlikely to continue to be readily usable. To ensure that a GUI's reconfigurability doesn't impose high maintenance costs and jeopardise the reliability of the parallel programming system, the interface between them needs to be clean and minimal. This could be achieved, if the parallel system is assembled out of sequential processes that already support interactive user interfaces, by building the GUI to the whole system on top of existing user interfaces. Recent developments in scripting languages for embedded control of applications and their GUIs [6] [7] allow a process based parallel programming system to be given a GUI frontend with the required modularity.

2 Design Choices of PAN

PAN is a parallel logic programming system that incorporates these aims 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 and a high level embedded command language that drives GUI facilities and composes interface elements. PAN is designed to have a sustainable systems architecture that should adapt gracefully 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 build it. PAN combines resolution engines, a model of multiprocessing, a virtual multiprocessor and a graphical user interface to get a systems architecture with evolutionary resilience and high portability.

2.1 Model of Multiprocessing

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. SICStus Prolog fitted the requirements well, so it was chosen as PAN's resolution engine component.

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 [8], independent and-parallelism by &-Prolog [9] and stream and-parallelism by Parallel NU Prolog [10]. Combinations of forms of parallelism are also being exploited such as or-parallelism and deterministic and-parallelism by Andorra I [11]. However, such capabilities require major alterations to conventional sequential Prolog engine technology particularly to manage memory. Thus while impressive speedups can be obtained, such sys-

tems are not resilient to evolution in their enabling technologies and tend to be specialised to current versions of certain parallel platforms. Thus to track technology and platform evolution in order to remain state of the art, such systems require substantial maintenance effort which is not warranted by the small number of persons using them.

Explicitly parallel Prologs exploit parallelism in a control driven manner. By reflecting control up to the program level, they allow much greater flexibility in their exploitation. Thus data or demand driven processing can be emulated on a 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 or message based depending upon whether it is done through a shared store or by explicit message passing.

Blackboard based systems usually follow the LINDA model [12]. 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 [1] 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 [4] and PMS-Prolog [2]. A common design choice of such Prologs has been to support dynamic Prolog process creation and only to support synchronous message reception.

Dynamic process creation provides flexibility and expressiveness but has high overheads. It is normally realised by forking and overlaying UNIX processes. Dynamic creation would make better sense if lighter weight processes were employed as IC-Prolog-II [14] does. However, the process and memory interleaving required to do this, rules it out for PAN as too big a departure from the norm. 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.

Existing message passing Prologs only support synchronous message reception [4] [2]. Asynchronous message reception has higher implementation overheads, but is more flexible in being demand driven, letting multiple messages be simultaneously in flight, and avoids the idle time 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.

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 and communicates by synchronous and asynchronous message passing. Extra primitives can control message passing and achieve synchronisation. Prolog engines can be farmed out to the width of process parallelism required 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 to be resilient to platform evolution.

2.2 Virtual Multiprocessor

The Parallel Virtual Machine or PVM [15] is a widely used virtual multiprocessor that runs on many different 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. The PVM 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 workstations running suitable operating systems to be configured as multiprocessors, it greatly expands the availability of platforms for parallel processing purposes.

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.

2.3 User Interface

Users of multiprocessors and networked workstations suitable for running PVM typically exercise their applications under UNIX within a graphical user interface environment such as X Window. However, it is not straightforward to realise such an X interface if it is also required that the interface be both readily reconfigured and have a simple and minimal connection to the parallel programming system.

It would be possible to avoid building a new interface to the parallel programming system from scratch, if the compositionality inherent in a process based approach was exploited to build a collective interface out of existing individual interfaces to each process. Powerful new techniques of using an embedded command language to drive GUI facilities and compose interface elements in a high level way can glue these process interfaces together [6].

These techniques are centred round a string processing imperative programming language TCL that interfaces with the operating system in the manner of UNIX shell scripts. TCL has an extension called TK that lets TCL programs create a rich range of X widgets and service X Window callbacks. The combination allows a TCL/TK script to be embedded in an application and to be interpreted during a session so as to drive an X interface and mediate its interactions with the application in the manner of a shell script.

TCL/TK also offers important new functionality that allows separate TCL interpreters to exchange TCL programs. This enables TCL/TK interfaces to adopt a hypertools approach and compose an interface from separately scripted elements that are coordinated through message exchange. Since PAN will be composed of Prolog processes and a PVM console process that already support interactive interfaces to terminals or terminal emulators, this suggests creating a composite interface to the whole parallel programming system by supporting these terminal emulators in TCL/TK and using TCL program exchange among them to weld the elements into a composite.

Unfortunately, TCL/TK lacks the capabilities to support a terminal emulator under UNIX. Conventional X Window applications use the xterm executable to do this. TCL/TK requires Expect to achieve the same effect. Expect [7] "talks" to other interactive programs according to a script which specifies what can be expected from a program and what the correct response should be. Expect uses TCL to provide branching and high-level control structures to direct the dialogue. Its extension Expectk supports TK as well. Expect allows users to take control and interact directly when desired, afterward returning control to the script.

Use of Expect enables PAN to support terminal emulators that can both serve

as interface widgets to Prolog processes and support exchange of TCL programs amongst themselves. It also allows an Expect script to control the PVM through its console before handing control back to the user. These contributions allow PAN to use the latest versions of SICStus Prolog and PVM as they are supplied off the shelf. Avoiding specialising the code of either, preserves a high degree of modularity among these components which greatly eases their ability to adapt gracefully and easily to technology evolution.

Modern Prologs such as SICStus present a simple command interface to users under the control of a terminal. Such interfaces can easily be realised under X Window using windows that emulate terminals. Adding a button panel above it allows pull down menus and button activated dialog boxes to provide short cuts to formulating query commands within the virtual terminal. The terminal emulator and button panel can readily be realised by an Expect/TK script.

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 an extra window is added 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.

FIGURE 1

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.

A virtue of this approach is that it preserves unaltered the user's interface to each Prolog engine. Queries and interactive debugging can be performed as with an ordinary sequential Prolog which eases users migrating to using PAN. Another virtue is that each engine is allowed to remain ignorant of the X 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. Further session control provided by PAN console buttons is realised by using PVM to send operating system signals to remote Prologs.

TCL's ability to pass TCL programs between TCL interpreters provides the mechanism to broadcast a Prolog goal to each engine. A global query is elicited by a dialog box popped up from the console widget and passed by this means to the TCL interpreters (with TK and Expect extensions) that establish the terminal emulator interfaces to each engine. The same mechanism also enables the console widget to provide collective layout control facilities that affect the size and layout of all PAN widgets. This is important where many engines are part of PAN and crowd the screen with their widgets.

3 Implementation of PAN

PAN has been implemented under 4 versions of UNIX, SUNOS4.1+, Ultrix, HPUX and Solaris5+ and on 4 hardware platforms, Sun-3s, Sun-4s, HP 9000s and DEC Workstations. It currently runs on a local area network of over 90 workstations having a common distributed filing system supported by an automount facility.

FIGURE 2

Currently hosts in PAN are of three different types

Host Type	**Main Job**
Interface	display all X widgets
Communications	manage Parallel Virtual Machine
Worker	run Prolog Engine

The Interface host is unique and only displays widgets for processes that run on the Communications and Worker hosts. The Communications host is also unique and runs the master PVM daemon, an Expect process to drive the PVM console and an Expectk process that emulates a terminal under which the PVM console runs. Worker hosts support an engine each of the PAN system. Figure 2 shows the various processes running on 6 hosts to support a PAN configuration of 4 engines. The Parallel Virtual Machine consists of a master daemon running on the Communications host, PE4, and 4 local daemons running on Worker hosts. The PVM console running on the Communications host enables monitoring and interactive control over PVM. Alongside the PVM console and Prolog engines run Expectk processes interpreting Expect scripts to sustain terminal emulators to which the PVM console and Prologs are attached through pseudo-tty device interfaces. All widgets display on PE5.

A PAN session is launched in four main stages:

Stage	**Host Type**	**Job**
start up	interface PE	start console widget with PAN launcher
start PVM	comms PE	farm out daemons to hosts probed as alive
farm out Prologs	comms PE	farm out widgets with attached Prologs
initialise PAN	worker PE	Prologs register with PVM

A PAN session is started from the Interface host. An Expectk process emulating a terminal with button and menu panels is farmed out to the Communications host. It displays its X widget on the Interface host. The PAN launcher process is attached to this terminal and interprets a TCL script. It starts by probing hosts on the PAN configuration file to establish which hosts may be used in the session and then establishes the PVM on whichever of these hosts are running. The PVM is set going by starting a master daemon and a console on the local machine. The rest of the PVM is then established by getting an Expect process to issue appropriate commands to the PVM console. The course of this interaction is displayed on the master X widget's virtual terminal. Driving the PVM from an Expect script keeps the nature of that interaction, visible and readily reconfigurable.

The launcher process then farms out Expectk processes with attached Prolog processes to the Worker hosts. When the Prologs 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 pass control over the PVM console to the user.

In order to ensure maximum evolutionary resilience, standard Prolog engines are used. They are only specialised for PAN at boot time by using SICStus's configuration file to load extra code in to define a few primitives and to induce initialisation with PVM. A standard version of PVM is also used. Thus upgrading PAN, as SICStus Prolog and PVM evolve, is readily and easily performed.

4 PAN's Programming Model

PAN adds a few primitives to Prolog to allow programs to pass messages among each other and to obtain session information. Prolog engines are identified by a number from zero up to the number of engines running in a session. The four communication primitives are:

rx(Term, Id)	blocks until caller synchronises with transmission from Prolog Id and then Term is unified with message sent
tx(Term, Id)	blocks until caller synchronises with receiving Prolog Id and then Term is sent
rxnb(Term, Id)	Term is unified with message (to be) sent from Prolog Id
txnb(Term, Id)	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 PVM 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 variable are awoken and run immediately. Thus goals such as

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

will set up *handle(Term, Id)* to deal with the next receivable message sent asynchronously to the caller. As asynchronous message reception can happen unpredictably 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 record them in the clause database for later handling. PAN supports the passing of arbitrarily large terms among engines which permits much flexibility in developing applications. Unbound variables in communicated terms get freshly renamed on reception to ensure all variables have purely local scope. PAN follows PMS-Prolog's reasons [2] in not supporting any form of backtracking on communication.

The two informational primitives

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

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

5 Performance

Communication among PAN engines involves indirection through PVM daemons making message passing slow. A short message sent between two Prologs on Sun 4s on a common Ethernet typically takes 10 msecs. Long lists take significantly longer to pass. Thus effective exploitation of PAN's parallelism requires that the significant overheads of sending a goal back and forth to a remote processor are repaid by sufficient saving in effort of having it remotely processed. The computation of perfect numbers illustrates some performance issues. A perfect number is an integer such as 6 which is equal to the sum of all its divisors less itself.

FIGURE 3

The benchmark in figure 4 is exercised by a query such as *perfects(1,500,L,[])* which computes all the perfect numbers in the range 1 to 500. 254,195 reductions are required to compute L as being [6,28,496]. On SICStus Prolog 2.1#9 on a SUN 4 IPX, it takes 43.2 seconds. The procedure *perfects/4* has two recursive calls which can be executed in and-parallel. This benchmark was adapted for PAN in figure 5 by adding code to process goals remotely and to farm out subgoals. The top level query *go(N)* starts off execution on all engines where N is bound to the number of engines to be used. Engine 0 initiates the benchmark while the other engines are set to process goals for other Prologs.

FIGURE 4

Timings are obtained using the PAN primitive *clock/1*. A task farmer *solve/2* solves its 1st argument on the given 2nd argument engines splitting the processing resources among the two recursive body goals for *perfects/4*. The first is dispatched for remote processing while the other is set aside for local processing. Goals are transmitted without blocking while receives of answers are made to block but only after all transmissions and local processing are done.

On an Ethernet LAN of Sun 4 hosts running SUNOS4.1.3U1 (ELCs, SLCs, IPCs, IPXs etc), some of which were on separate subnets and accessed across gateways, the following performance results were obtained for a 16 host PAN session with one SICStus Prolog engine 2.1#9 running per workstation where all timings were the average of the three best runs.

Prologs	1	2	3	4	5	6	7	8
Time (secs)	43.5	32.2	22.4	18.8	16.0	16.3	16.8	12.4

Prologs	9	10	11	12	13	14	15	16
Time (secs)	12.4	14.3	17.2	16.7	9.4	12.0	9.9	9.8

A best case performance limit for the benchmark can be calculated by analysing how the program divides up the work into subtasks and computing the time the largest subtask would take from the number of reductions it requires and an assumed rate of processing of the benchmark of 5,884 reductions per second (i.e. 254,195 reductions in 43.2 seconds). This would estimate the execution time, if all remote processing had no overhead at all. The following table gives these figures.

Prologs	1	2	3	4	5	6	7	8
Time (secs)	43.5	32.3	18.8	18.8	13.5	13.5	10.0	10.0

Prologs	9	10	11	12	13	14	15	16
Time (secs)	8.8	8.8	8.8	8.8	6.1	6.1	5.2	5.2

There are several significant discrepancies between the actual and the estimated best performance. Examining a few of them illuminates some performance issues in relation to the architecture. The workstations in the study were a mixture of older and newer Sun 4s (IPCs, IPXs, SLCs, ELCs etc), many of them discless nodes, and not all on the same subnet. While this complicates performance analysis, it is typical of expected conditions of use of an architecture such as PAN.

The first significant discrepancy occurred with three processors. The estimated performance was based on a division of the overall 254,195 reductions into three tasks of 64,424, 79,236 and 110,533 reductions which are executed by Engines 1, 2 and 0 respectively. The bottleneck task on Prolog 0 was computed as taking 18.8 seconds to complete. In fact Prolog 2 running on an old IPC ran 73% slower than Prolog 0 on a newer IPX. Thus the actual bottleneck task was Prolog 2's smaller task. The best estimate of 23.3 seconds for it to compute, is very close to the measured performance of 22.4 seconds.

Further significant discrepancies occurred where 9 to 14 processors were involved, when the engine with the bottleneck task was running on an old, discless Sun 4 on a different subnet from Prologs 0 to 3. When the Prolog allocated

the bottleneck task returned to being the central Prolog 0, where 15 and 16 processors were involved, the overhead in seconds between the theoretical best and the actual best narrowed significantly. The best case estimate for a speed up for 16 processors is 8.3 while the measured speedup was only 4.4. The shortfall is significant, but 4.4 is still an effective speedup under conditions where communications on LANs take time and processors with different performance can undermine static load balancing.

The task farming strategy adopted by PAN's harness for the perfect numbers benchmark is suboptimal. The processing of the higher half of a range of numbers takes significantly longer. A better split of processing resources would favour the computation of the higher range. The result of changing the distribution of Prolog engines to allocate one third rather than a half of them to compute the lower range is shown below. It was effected by changing the second body goal of *do/7* to M is N // 3.

Prologs	1	2	3	4	5	6	7	8
Time (secs)	43.5	31.8	22.4	22.9	16.4	16.3	9.6	17.1
Prologs	9	10	11	12	13	14	15	16
Time (secs)	17.1	8.9	11.9	11.9	11.9	9.5	8.7	8.8

The biased allocation of processing resources tends somewhat unevenly to improve the speedup obtained by concentrating processing resources better. However, it suffers from an even worse case of a retrogressive effect on performance in task allocation in the 8 and 9 processor cases than with a more even handed approach to allocating processing resources and for the same sorts of reasons as before.

6 Drawbacks

While PAN has good evolutionary resilience and high portability, it buys them at a significant price. It refuses to specialise the functionality of SICStus Prolog beyond defining a few new predicates via the foreign language interface. This sacrifices many opportunities to exploit parallelism via alternatives to SLD resolution such as committed choice resolution or the extended Andorra principle [16] or via adaptations of Prolog implementation techniques such as multiple binding environments [17] and stack sharing and copying schemes [8].

PAN employs a virtual multiprocessor's services to set up communications among the Prolog engines rather than setting up socket connections among them itself. This separates the layers of functionality better which eases PAN's management and its ability to evolve but incurs significant overheads on each

communication. Furthermore PAN eschews specialising its functionality to the particular platforms used, which loses significant opportunities for optimising its performance. PAN trades in these performance optimisations in exchange for ready adaptation to technology evolution and the ability to run on a wide range of parallel platforms.

7 Conclusion

A flexible and sustainable architecture for logic programming in parallel can be assembled in a modular way from widely used 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 be familiar to its potential users. PAN aspires to be such a system by combining SICStus Prolog, PVM and TCL/TK/Expect technology to create a message passing parallel Prolog system running on a virtual multiprocessor under a script controlled X Window interface. The PAN architecture has been developed as part of a wider endeavour to create an effective, available vehicle for executing ordinary Prolog programs in parallel. The eventual aim is to use program and granularity analysis techniques in a preprocessor and runtime scheduling system under development to exploit parallelism transparently on PAN.

References

[1] 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.

[2] 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.

[3] 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.

[4] 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.

[5] P. Kacsuk. *Execution Models of Prolog for Parallel Computers*. Pitman, London, 1990.

[6] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, 1993.

[7] D. Libes. Expect: Curing those uncontrollable fits of interactivity. In *Proceedings of the Summer 1990 USENIX Conference*. Anaheim, California, June 11-15, 1990.

[8] K.A.M. Ali and R. Karlsson. The Muse approach to or-parallel Prolog. *International Journal of Parallel Programming*, 19(2):129–162, April 1990.

[9] 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.

[10] 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, August 1988.

[11] 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.

[12] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–459, 1989.

[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] Chu D. and Clark K. *I.C. Prolog II: a Multi-threaded Prolog System*. Department of Computing, Imperial College, London, 31 May 1993.

[15] A. Geist and Sunderam V. Network-based concurrent computing on the PVM system. *Concurrency Pract. Exper.*, 4(4):293–311, 1992.

[16] D.H.D. Warren. Extended Andorra principle. Invited address, 10th International Conference on Logic Programming, Budapest, June 1993.

[17] D.H.D. Warren. The SRI model for or-parallel execution of Prolog — abstract design and implementation issues. In *Proceedings of the 1987 Symposium on Logic Programming*, pages 92–102. IEEE Computer Society Press, San Francisco, August 1987.

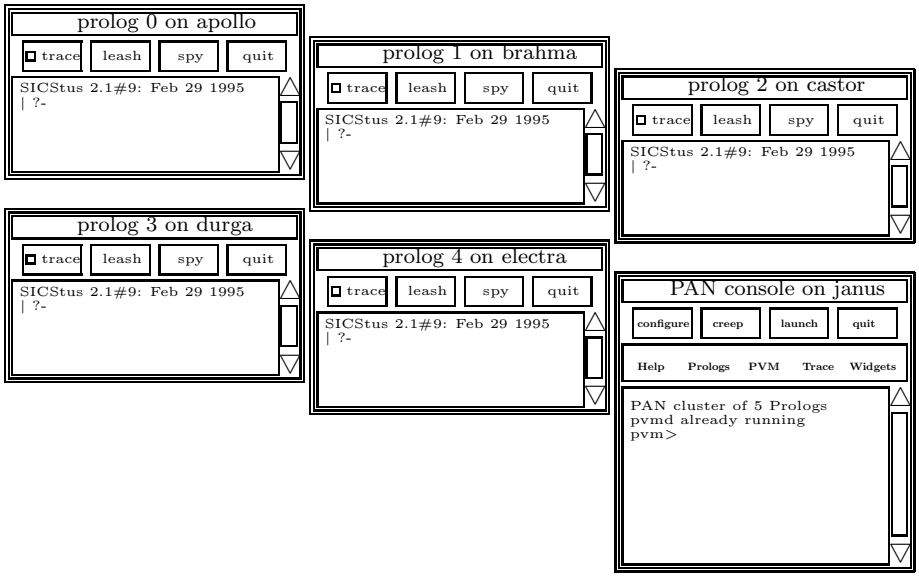

Fig. 1. Interface to 5 Prologs

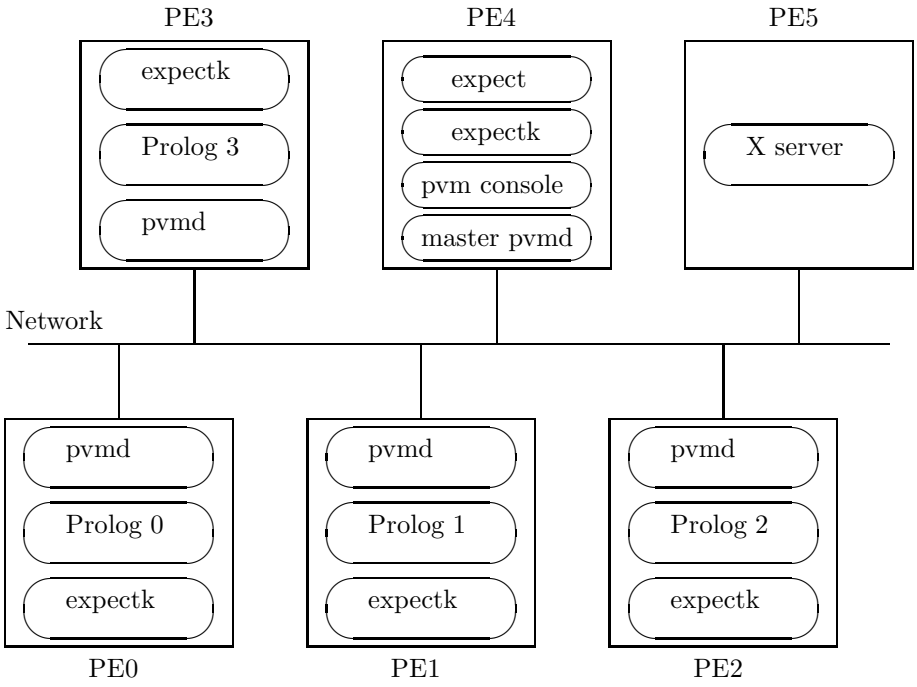

Fig. 2. PAN Configuration of 4 Prologs

```
perfects(N, Range, L, L2)   :-  Range > N, !,
                                Range1 is N + ( Range - N ) // 2,
                                Range2 is Range1 + 1,
                                perfects(N, Range1, L, L1),
                                perfects(Range2, Range, L1, L2).
perfects(N, N, [N|L], L)    :-  perfect(N), !.
perfects(N, N, L, L).

perfect(N)   :-  gen(1, N, Range), divisors(N, Range, L), sum(L, N).

gen(Max, Max, [])    :-  !.
gen(N, Max, [N|L])   :-  N < Max, N1 is N + 1, gen(N1, Max, L), !.

divisors(_, [], []).
divisors(N, [H|T], [H|L])   :-  N mod H =:= 0, !, divisors(N, T, L).
divisors(N, [_|T], L)       :-  divisors(N, T, L).

sum([], 0).
sum([H|T], Sum)   :-  sum(T, S), Sum is H + S.
```

Fig. 3. Perfect Numbers Benchmark

```
go(N)    :-  prolog_id(Id), go(Id, N).

go(0, N)    :-  clock(T1),
                other_prologs(1, N, L),
                solve(perfects(1, 500, L, []), L),
                clock(T2),
                write('X = '), write(X), nl,
                T is T2 - T1, write(T), write(' secs'), nl.
go(Id, N)   :-  Id < N, rx(G, M), call(G), txnb(G, M).
go(_, _).

do(G, [], [], R, R, L, [G|L]).
do(G, [E|V], X, R, [rx(solve(G, W), E)|R], L, L)   :-  length(V, N),
                                                       M is N // 2,
                                                       split(M, V, W, X),
                                                       txnb(solve(G, W), E).

solve(perfects(N, R, L, L2), [E|Es])   :-  R > N, !,
                                           R1 is N + ( R - N ) // 2,
                                           R2 is R1 + 1,
                                           do(perfects(N, R1, L, L1),
                                              [E|Es], Es1, [], Rs, [], Ls),
                                           solve(perfects(R2, R, L1, L2), Es1),
                                           calls(Ls),
                                           calls(Rs).
solve(G, _)                            :-  call(G).

calls([]).
calls([G|Gs])   :-  call(G), calls(Gs), !.

split(_, [], [], [])              :-  !.
split(0, Es, [], Es)              :-  !.
split(N, [E|Es], [E|Et], Eu)      :-  M is N - 1, split(M, Es, Et, Eu), !.

other_prologs(N, N, []).
other_prologs(N, Max, [N|T])   :-  M is N + 1, other_prologs(M, Max, T).
```

Fig. 4. PAN harness for Perfect Numbers Benchmark

Contact Details
Author: Hamish Taylor
E-mail: hamish@cee.hw.ac.uk
FAX: (44) 131 451 3431
Summary of Paper

An effective and usable resolution multiprocessor can be built in a modular manner from separate distributed processing, logic programming, and interface elements. By carefully composing widely used, state of the art components, that are well maintained and highly portable, a powerful parallel logic programming system can be developed that runs on a broad range of platforms and incorporates good resistance to premature obsolescence by software evolution.

This paper describes a parallel logic programming system called PAN that has been built on a virtual multiprocessor under the control of a highly configurable composition of interacting processing and interface components. The computational platform is a meshed set of daemons on a network of processors offering common message passing, configuration and job control services. Sequential resolution engines run on these nodes and interact through these services.

Users configure and control the PAN architecture through a graphical user interface using an embedded command language that drives GUI facilities and composes interface elements. Distinct widgets support interfaces to each resolution engine, to the virtual multiprocessor's console, and to a widget management console, and window based methods are used for simultaneously invoking all resolution engines together. Users can run programs that either explicitly control parallelism through passing general terms synchronously or asynchronously among PAN's resolution engines or would have to rely on program transformation to extract parallelism implicitly from Prolog programs.

About the Author - Hamish Taylor received MA and MLitt degrees in philosophy from Cambridge University and an MSc and a PhD degree in Computer Science from Heriot-Watt University, Scotland. He has been a lecturer in Computer Science at Heriot-Watt University since 1990. Before that he was a research assistant at QMW in the University of London working on formal models for automated reasoning and a research associate at Heriot-Watt University on an Alvey project "PARLOG on Parallel Architectures". His current research interests are in parallel logic programming, performance modelling of parallel databases, and intelligent access to information on distributed hypermedia. He is a member of his department's Database Research Group and holds government grants in performance modelling of parallel relational databases and intelligent discovery of Internet resources.