The University Of Western Australia
Department Of Computer Science

Technical Report 91/7

Prolog-D-Linda : An Embedding of Linda in SICStus Prolog

Geoff Sutcliffe and James Pinakis
Dep't of Computer Science, The University of Western Australia,
Nedlands, 6009, Western Australia
Email: geoff@cs.uwa.edu.au

ABSTRACT
This paper presents an embedding of the Linda parallel programming paradigm in Prolog, resulting in a coarsely grained parallel Prolog. The work extends that reported in [Sutcliffe and Pinakis, 1990][1]. This embedding supports a distributed tuple space and a control hierarchy that provides remote I/O facilities for client processes. As before, the embedding uses unification and Prolog style deduction in the tuple space. Two applications of Prolog-Linda are described.

[1] To make this paper self contained it has been necessary to duplicate some of the information given in [Sutcliffe and Pinakis, 1990].

1 Introduction

A categorisation of parallel Prolog systems may be made according to how the problem is divided into parallel tasks. This *task granularity* can be divided into three categories : unification parallelism, goal parallelism, and process parallelism:

- Unification parallelism introduces parallelism by performing parts of unification operations in parallel. This form of parallelism has been used by Ito [1985].
- Goal parallelism encompasses AND-parallelism, OR-parallelism and AND-OR-parallelism. Here subgoals of a single Prolog program are evaluated in parallel. Examples in this category are GHC [Ueda, 1985], EPILOG [Wise, 1986], Parlog [Gregory, 1987], the family of Concurrent Logic Programming languages [Shapiro, 1987, 1989b], Strand [Foster and Taylor, 1989], and Andora [Haridi and Janson, 1990].
- Process parallelism is that form of parallelism in which multiple sequential Prolog programs execute in parallel, with some mechanism being provided for inter-process communication. Examples are Delta-Prolog [Cunha, Ferreira, and Pereira, 1989], CS-Prolog [Futo and Kacsuk, 1989], the Quintus Prolog multi-processing package [Quintus, 1989], Prolog-1-Linda and Prolog-N-Linda [Sutcliffe and Pinakis, 1990], Amoeba Prolog [Shizgal, 1990], PMS-Prolog [Wise, 1991a, Forthcoming], MB-Prolog [Wise, 1991b], the SICStus Prolog multi-processing package [Forthcoming], and Prolog-D-Linda.

Orthogonal to the task granularity, parallel Prolog systems may be categorised according to how the parallel tasks are executed on the host computer. This *execution granularity* is often influenced by the task granularity. The finest grain executes the tasks as light weight processes within a single operating system level process. At the next level of granularity the tasks are executed as distinct operating system level processes, but are restricted to a single processor. In the coarsest granularity the tasks also execute as distinct operating system processes, but the processes may be distributed over a network of processors. Combinations across categories are possible. The network category is the only one which can harness more computing power; the others simply provide conceptual parallelism.

Prolog-D-Linda (Prolog-Distributed-Linda) is an embedding of the Linda paradigm into an existing Prolog system. The embedding supports a distributed tuple space, unification and Prolog style deduction in the tuple space, and a control hierarchy that provides remote I/O facilities for client processes.

The reader is presumed to be familiar with Prolog.

2 The Linda paradigm

Linda is a programming framework of language-independent operators. These operators may be injected into the syntax of existing programming languages, such as Modula-II [Borrman, Herdieckerhoff, and Klein, 1988], C [Berndt, 1989], LISP [Yuen and Wong, 1990], Joyce [Pinakis and McDonald, 1991], and Russell [Butcher and Zedan, 1991], resulting in new parallel programming languages. Linda permits cooperation between parallel processes by controlling access to a shared data structure called the *tuple space*. The tuple space contains ordered collections of data called *tuples*. Manipulation of the tuple space is possible only by using the set of Linda operators.

2.1 Tuples

Tuples are collections of *fields*, of any arity. Every field has a data type drawn from the host language. The type of a tuple is the cross product of the types of its fields. A field can be a *formal* field or an *actual* field. A formal field has a type but no value, and can be thought of as a variable that has not been assigned a value. The type of a formal field is the type of the variable. A formal field is specified by the variable name preceded by a `?`. An actual field has both a type and a value. The type of an actual field is the type of its value. Example : if `s1` is a variable of type `string` containing the value `"hello"`, and `f1` is a variable of type `float`, then `(s1,9,?f1)` is a tuple of arity 3. The first two fields are actual fields, the first being of type `string` with value `"hello"`, the second being of type `integer` with value `9`. The third field is a formal field of type `float`. The type of the tuple is `string` × `integer` × `float`.

The tuple space contains any number of tuples, and identical tuples may exist in the tuple space. Processes communicate by inserting, removing and examining tuples in the tuple space. Thus the tuple space is a shared data object. All processes having access to a tuple space have access to all tuples in it.

2.2 Operations on tuples

The `out` operator inserts a tuple into the tuple space. Following the example above, `out(s1,9,?f1)` inserts the tuple `("hello",9,?f1)` into the tuple space.

The `in` operator removes a tuple from the tuple space. Its argument is a template against which tuples are matched. A template matches a tuple if all corresponding fields match. Two actual fields match if they have the same type and value. A formal field and an actual field match if they have the same type. Two formal fields cannot match. If a match for a template is found, the matched tuple is removed from the tuple space and formal fields in the template are given the values of the corresponding actual fields in the tuple. For example, if `i1` is an integer variable, the operation `in("hello",?i1,27.0)` could remove the previously inserted tuple from the tuple space. In addition to removing the tuple, the value `9` would be assigned to the variable `i1`. If more than one tuple matches a template, only one is chosen. If no matching tuple can be found in tuple space, `in` will block and wait for a matching tuple to be inserted by an `out` operation.

The `rd` operation (pronounced read) is similar to `in`, but leaves the matched tuple in the tuple space. `rd` is used for its binding and synchronization side-effects.

Two related operators are `inp` and `rdp`. These perform tasks equivalent to `in` and `rd` but are non-blocking. Instead they return a boolean value which indicates the success of the operation. Recent research [Leichter, 1989] argues against the use of these operators.

2.3 Process creation

The final operation provided by Linda is the `eval` operation. The `eval` operation is syntactically similar to `out` except that a new process is created to evaluate each of the fields in the tuple. When the evaluation of all fields has terminated, the tuple becomes an ordinary tuple in the tuple space. For example, let `sqrt` be the square root function. The operation `eval("hello",sqrt(81),?f1)` will create a new process to evaluate each of the fields. The first and last fields evaluate trivially, but the second process will continue to execute in parallel with others. When the process finally terminates, the tuple `("hello",9,?f1)` will appear in the tuple space and can be manipulated in the usual ways. While the `sqrt` process is executing the tuple is unavailable.

3 Prolog-Linda

Prolog-Linda implements the Linda tuple space as a collection of Prolog clauses in the Prolog database. Both Prolog rules and facts can exist in the tuple space. The effect of rules in the tuple space is discussed in section 5. Facts correspond almost directly to standard Linda tuples. The necessity of a predicate symbol in a fact is analogous to requiring that the first field of a tuple be an actual field with a string literal value, as enforced by some Linda implementations [Leichter, 1989]. (This requirement does not reduce the generality of the system.) Formals in tuples are implemented by unbound variables. As data in Prolog is untyped (everything is a term) the data in Prolog-Linda's tuples is untyped.

The `out` operation adds tuples to the tuple space using Prolog's `assertz` database operation, and the `in` operation removes tuples using `retract`. The `rd` operation interrogates the tuple space simply by using Prolog's query mechanism. The tuple matching method is thus generalised to Prolog's unification. As a consequence of this formals can match and be extracted from the tuple space. Prolog-Linda's `eval` operation differs from that of the original Linda paradigm. An `eval` operation is used to start a new Prolog environment containing specified clauses and evaluating a specified Prolog query. The evaluation of the query may of course cause a tuple to be inserted in the tuple space. This form of `eval` is more general than the original, and can implement the original.

4 Implementation

Prolog-D-Linda has been implemented in SICStus Prolog [Carlsson, 1990]. SICStus Prolog provides an interface for accessing functions coded in C, and this is used for process creation and inter-process communication. Prolog-D-Linda runs on a network of Sun SPARC station-1s running SunOS 4.1.1, and connected via an Ethernet. This environment provides access to a shared file system via Sun's Network File System [Sandberg, 1985].

4.1 Overview

Prolog-D-Linda's tuple space and associated operations are implemented in *server* processes. Multiple servers can be used, each being responsible for part of the tuple space. Linda operations in *client* processes are translated into requests which are passed to an appropriate server. Prolog-D-Linda is controlled by a single *controller* process, which must be associated with a terminal device. The controller is responsible for : (i) starting and stopping the server processes, (ii) for reading and displaying the terminal input and output of servers, and (iii) for reading and displaying the terminal input and output of clients that are started via an `eval` request.

A configuration file must be supplied to Prolog-D-Linda. The configuration file is in the form of a Prolog program which specifies (i) the names of processors that will execute server processes, (ii) the name of the server which will deal with `eval` requests, and (iii) how the tuple space is to be partitioned amongst the servers. A sample configuration file is listed in the appendix. Prolog-D-Linda is started by executing the controller, which reads the configuration file to determine the names of the server processors. The names are stored in a `servers__/1` clause, whose argument is the list of processor names. When this clause has been found, the controller uses the `rexec()` system call to start each of the servers.

Client processes may be started independently at a terminal, or via the `eval` primitive. Clients `consult` the configuration file, and use the consulted clauses to determine how the tuple space is partitioned amongst the servers. The tuple space partition information is in the form of `select_server__/2` clauses. The first argument returns the name of the server processor that is responsible for the tuple that is supplied in the second argument. To determine where to send a tuple space operation request, a client simply evaluates an appropriate `select_server__/2` goal.

One of the servers is designated to be the *eval-server*. In addition to processing tuple space operation requests, the eval-server is responsible for processing all `eval` requests. The eval-server starts new clients using an `rexec()` system call. The name of the eval-server is stored in the configuration file as the argument of an `eval_server__/1` clause.

4.2 Communication

Communication between the controller and servers, and between servers and clients, is via internet domain stream sockets, as illustrated in Figure 1.

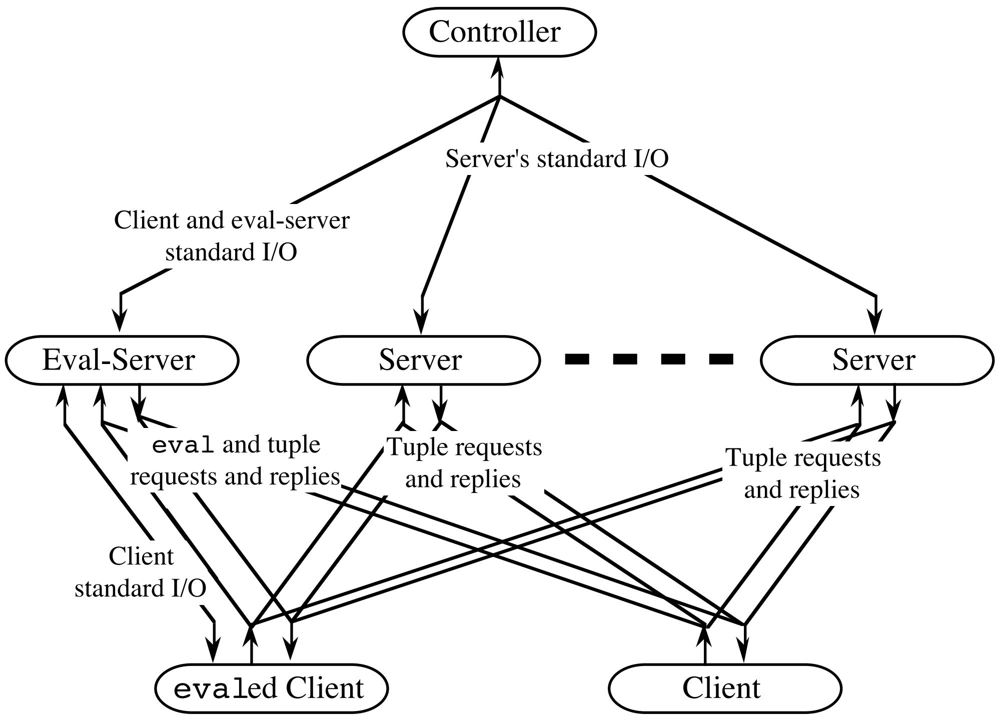
Controller
Server's standard I/O
Client and eval-server standard I/O
Eval-Server
Server
Server
eval and tuple requests and replies
Tuple requests and replies
Tuple requests and replies
Client standard I/O
evaled Client
Client

Figure 1 - The Prolog-D-Linda System

When a server is started its terminal input and output streams are connected to a file descriptor in the controller. The descriptor is obtained from the `rexec()` system calls used to start the server. Similarly, when a client is started by the eval-server, the client's terminal input and output streams are connected to a file descriptor in the eval-server. As well as the terminal I/O connection, every client establishes two further connections to each server. One connection is used for sending Linda operation requests to the server, and the other is used for receiving replies. The controller and servers monitor their input descriptors for incoming data, and process incoming data as described below.

4.3 Linda primitives

The servers read Linda operation requests from the request connections opened by the clients. The requests are serviced by evaluating them as Prolog queries. The requests are thus simply queries on Prolog procedures which implement the required operations. The use of Prolog evaluation to service requests is a general mechanism, and allows any query to be passed to a server for evaluation.

Prolog-D-Linda's `eval` operation takes three arguments : the name of a processor on which to execute the client, a Prolog goal, and a list of Prolog source file names. When the eval-server receives an `eval` request, it starts the new client by `rexec`ing a SICStus Prolog saved state, on the specified processor. (The shared file system provides transparent access to files on remote processors.) The saved state has the client running so that it immediately `consults` the configuration file and opens the request and reply connections to the servers. In the interim, the eval-server places a tuple of the form `<client processor>(<the goal>,<the source files>)` into its tuple space. The client `ins` this tuple, `consults` the source files, and evaluates the goal. On completion of the goal the client closes the connections and terminates.

Tuple space `rd` requests received by servers are implemented by evaluating the requested tuple template as a Prolog goal, and `in` requests attempt to `retract` the requested tuple template. When an `in` or `rd` request is satisfied, the tuple template, with variables instantiated, is sent back to the requesting client on the reply connection. If a server is unable to satisfy an `in` or `rd` request, the request is placed on a global wait queue in the server, to wait for an appropriate tuple to be `out`ed. Tuple space `out` requests are implemented by `assertz`ing the supplied tuple into the server's database. After the `assertz,` all requests on the wait queue are re-evaluated. The `inp` and `rdp` operations return the atom `fail` to the requesting client if the request cannot be immediately satisfied. This is used in the client to cause the operation to fail.

4.4 Terminal I/O

The terminal output of servers is read by the controller, off the descriptors obtained from its `rexec()` calls. The output is displayed on the controller's terminal, prefixed by the descriptor number from which it was read. This number uniquely identifies the server from which the output originated. Input to be sent to a server is entered at the controller's terminal, prefixed by the descriptor number which identifies the server (the descriptor number is obtained from previous output from that server). The controller strips the descriptor number from the input and forwards the remainder of the input to the server, on that descriptor.

The terminal output of clients which have been started via an `eval` request, is read by the eval-server, off the descriptors obtained from its `rexec()` calls. The output is forwarded to the controller by writing the output to the eval-servers terminal output. The forwarded output is prefixed by the descriptor number from which it was read. The output is received by the controller and displayed on its terminal in the manner described above. Output from clients is therefore displayed with two prefixed descriptor numbers : firstly the descriptor number upon which it arrived at the controller (the descriptor number that identifies the eval-server), and secondly the descriptor number upon which it arrived at the eval-server. These numbers uniquely identify the client from which the output originated. Input to be sent to a client is entered at the controller's terminal, prefixed by the two descriptor numbers which identify the client (the descriptor numbers are obtained from previous output from that client). The controller strips the first descriptor number from the input and forwards the remainder of the input to the eval-server, on that descriptor. The eval-server strips the second descriptor number from its input and forwards the remainder to the client, on that descriptor.

This I/O hierarchy permits all clients to be interactive, even though they may not be associated with a terminal device. If there is to be a lot of such client I/O, then the system can be configured so that the eval-server is not responsible for any part of the tuple space, i.e. the eval-server only deals with `eval` requests and client I/O. The I/O hierarchy is also used to halt the system. If the keyword `halt.` is entered at the controller's terminal, `halt.` is written to all servers' terminal inputs. When a server reads the `halt.` message, it immediately closes all open descriptors and terminates. After writing to all the servers, the controller closes all its open descriptors and terminates.

5 Deductive tuple spaces

The Linda tuple space and associated operations are very similar to a standard concurrent access relational database system. The `in` and `out` operations effect database updates, and the `rd` operation effects database queries. The difference is that the Linda paradigm is viewed as providing communication between, and synchronization of, parallel processes, whereas a relational database is viewed only as storing data. Much research has been done on the generalisation of relational database to deductive database, in Prolog. Lloyd [1987] gives a good summary of this work. It is a logical step to extend the Prolog-Linda tuple space to a deductive tuple space. By allowing rules as well as facts to be added to and removed from the tuple space, the tuple space becomes deductive. Tuple space `rd` and `rdp` requests may be satisfied by facts, or using rules. Rules are evaluated using normal Prolog deduction, including backtracking. If a deductive tuple space is used it is necessary for all the required tuples (rules and facts) to be stored in the same partition of the tuple space, i.e. in one server.

A deductive tuple space greatly increases the capabilities of the tuple space, but not without some penalty. The first problem is the increased possibility of a bottleneck on the execution of the client, as the server must spend time evaluating deductive `rd`s. The second problem, which is an extreme case of the first, is the danger of the server entering an infinite deduction. Client requests will not be evaluated, in particular requests that may terminate the infinite deduction. Clients that make `in` or `rd` requests will be blocked indefinitely. A solution to this second problem is to restrict the nature of the deductive database to be hierarchical [Lloyd, 1987]. Despite the problems associated with a deductive tuple space, such a model provides facilities that are not available from a standard tuple space. Two examples are described here.

- In Linda it is awkward to simultaneously `rd` tuples of two different signatures. A method suggested in [Leichter, 1989] requires the `out`ing process to know that the tuples will be requested in this way. A deductive tuple space provides a direct solution :

```
make_switch(Tuple1,Tuple2):-
    out((switch(Tuple1):-Tuple1)),
    out((switch(Tuple2):-Tuple2)),
%----Wait for Tuple1 or Tuple2 to be outed
    rd(switch(Which)).
%----Which contains the outed tuple
```

- A deductive tuple space has the potential for extreme space saving. There are indeed some groups of tuples that can only be finitely stored in a deductive manner. For example :

```
recognise_even:-
    out((even(Negative):-Negative < 0,!,fail)),
    out(even(0)),
    out((even(Number):-Number_less_2 is Number-2,
even(Number_less_2))).
```

  would effectively place all tuples `even(X)` into the tuple space, where `X` is an even natural number.

6 Applications

6.1 Automated deduction

Prolog-Linda has been used to implement a distributed automated deduction system [Sutcliffe, 1991]. The deduction system, called GLD||UR, has two deduction components, which execute as separate client processes. One component runs a chain format linear deduction system and the other a UR-deduction system. Lemmas created in each of the deduction components are passed to the other component, via the tuple space. An extended version of GLD||UR, in which the lemmas created are distributed via a separate 'lemma control' component has also been developed. The speed-ups obtained in GLD||UR are largely due to cross-fertilisation between the deduction components. The implementation of GLD||UR is highly modular, and new deduction or other components can easily be added to the system.

6.2 Genetic Algorithms

Prolog-Linda has been used to implement a genetic algorithm, in which multiple clients access and update the solution pool in parallel. Each candidate solution is stored as a tuple containing the solution and its objective value. Each client process repeatedly (i) `rd`s two parent solution tuples from the tuple space, (ii) performs a crossover to produce two child solutions, (iii) for each child, `ins` a 'sucker' solution chosen at random, (iv) `outs` the child solution if $e^{(\mathrm{SuckerObjective} - \mathrm{ChildObjective})/T} >$ random([0,1)), otherwise `outs` the sucker. (I.e. if the child has a better objective value than the sucker, then the child is always `outed`; if the child has a worse objective value, then the child may still be saved by virtue of the Boltzman distribution, with temperature T.) Some variants of this algorithm have also been implemented. It is the iterative nature of this genetic algorithm that permits it to be parallelised. Similar work has been done by Ackley [1987] and Robertson [1987].

7 Conclusion

Prolog-D-Linda is a truly distributed logic programming environment. This distribution allows applications to take advantage of the added computing power available, as well as to be structured in a parallel fashion. The parallelism obtained is acknowledged to be coarse. In the context of parallel Prolog architectures, it has been argued that "exploiting as much fine grain parallelism as possible may be a flawed strategy; any gains through increased parallelism are wasted due to communication overheads" [Wise, 1991b, p 2].

The distribution of the tuple space in Prolog-D-Linda makes it superior to its predecessors. As the partitioning is user controlled, it is possible to tune the use of the tuple space so that bottlenecks are avoided. The introduction of a deductive tuple space is a significant enhancement to the capabilities of the Linda paradigm. A deductive tuple space provides direct solutions to problems that were previously difficult or impossible. The only system that provides features similar to Prolog-D-Linda, is the SICStus Prolog multi-processing package. As this product has only just been released, little is known about it. Email exchange with its author indicates that it is based on the Linda paradigm, and distributes clients over a network of processors. Nothing is known about its tuple space organisation.

The Prolog-D-Linda embedding of Linda in Prolog is very natural : the pattern matching and database features of Prolog have been used directly in the embedding; garbage collection and hashing in the tuple space are provided free by the Prolog implementation; the implementation of formals in tuples is direct; the specification of how the tuple space is to be partitioned is done as a Prolog program. This naturalness contrasts with the FCP(↑) implementation described by Shapiro [1989a].

8 References

Ackley D.H., *A Connectionist Machine for Genetic Hillclimbing*, Kluwer Academic Publishers, Dordrecht, The Netherlands, (1987).

Berndt D.J., C-Linda Reference Manual, Version 2.0, Scientific Computing Associates Inc., New Haven, (1989).

Borrmann L., Herdieckerhoff M., and Klein A., Tuple Space Integrated into Modula-2, Implementation of the Linda Concept on a Hierarchical Multiprocessor, In Jesshope, Reinartz, Ed., *Proceedings of CONPAR '88* (1988), Cambridge University Press, Cambridge, England, (1988), 1-8.

Butcher P., and Zedan H., Lucinda - An Overview, In *ACM SIGPLAN Notices* 28(8), ACM Press, New York, NY, (1991), 90-100.

Carlsson M., and Widen J., SICStus Prolog 0.7 User's Manual, R88007C, Swedish Institute of Computer Science, Kista, Sweden, (1990).

Cunha J.C., Ferreira M.C., and Pereira L.M., Programming in Delta Prolog, In Levi G., Martelli M., Ed., *Proceedings of the 6th International Conference on Logic Programming* (Lisbon, Portugal, 1989), MIT Press, Cambridge, MA, (1989), 487-502.

Foster I., and Taylor S., Strand: A Practical Parallel Programming Tool, In Lusk E.L., Overbeek R. A., Ed., *Proceedings of the 1989 North American Conference on Logic Programming* (Cleveland, OH, 1989), MIT Press, Cambridge, MA, (1989), 497-512.

Futo I., and Kacsuk P., CS-Prolog on Multitransputer Systems, In *Microprocessors and Microsystems* 13(2), IPC Science and Technology Press, Guildford, England, (1989), 103-112.

Gregory S., *Parallel Logic Programming in PARLOG*, Addison-Wesley, Reading, MA, (1987).

Haridi S., and Janson S., Kernel Andorra Prolog and its Computational Model, In Warren D.H.D., Ed., *Proceedings of the 7th International Conference on Logic Programming* (Jerusalem, Israel, 1990), MIT Press, Cambridge, MA, (1990), 31-46.

Ito N., Data-flow based Execution Mechanism of Parallel and Concurrent Prolog, In *New Generation Computing* 3, Springer-Verlag, New York, NY, (1985), 15-41.

Leichter J.S., Shared Tuple Memories, Shared Memories, Buses and LAN's - Linda Implementations Across the Spectrum of Connectivity, Ph.D. Thesis, Yale University, Yale, CT, (1989).

Lloyd J.W., *Foundations of Logic Programming, 2nd Edition*, Springer-Verlag, New York, NY, (1987).

Pinakis J., and McDonald C., The Inclusion of the Linda Tuple Space Operations in a Pascal-based Concurrent Language, In Gupta G., Lions J., Ed., *Proceedings of the 14th Australian Computer Science Conference* (Kensington, Australia, 1991), Department of Computer Science, University of New South Wales, Kensington, Australia, (1991), 45.1-45.11.

Quintus Computer Systems Inc., Quintus Prolog Multiprocessing Package, Quintus Prolog version 1.0 Manual, Quintus Computer Systems Inc., (1989).

Robertson G., Parallel Implementation of Genetic Algorithms in a Classifier System, In Davis L., Ed., *Genetic Algorithms and Simulated Annealing*, *Research Notes in Artificial Intelligence* Pitman Publishing, London, England, (1987), 129-140.

Sandberg R., The Design and Implementation of the Sun Network File System, In *Proceedings of the USENIX Association Conference* (Portland, OR, 1985), USENIX Association, El Cerrito, CA, (1985), 119-130.

Shapiro E., Concurrent Prolog: A Progress Report, In Bibel W., Jorrand P., Ed., *Fundamentals of Artificial Intelligence*, Springer-Verlag, New York, NY, (1987), 277-313.

Shapiro E., Technical Correspondence, In *Communications of the ACM* 32(10), ACM Press, New York, NY, (1989a), 1244-1258.

Shapiro E., The Family of Concurrent Logic Programming Languages, In *Computing Surveys* 21(3), ACM Press, New York, NY, (1989b), 412-510.

Shizgal I., The Amoeba-Prolog System, In *The Computer Journal* 33(6), Cambridge University Press, England, (1990), 508-517.

Sutcliffe G., and Pinakis J., Prolog-Linda - An Embedding of Linda in muProlog, In Tsang C.P., Ed., *Proceedings of AI'90 - the 4th Australian Conference on Artificial Intelligence* (Perth, Australia, 1990), World Scientific, Singapore, (1990), 331-340.

Sutcliffe G., A Parallel Linear and UR-Derivation System, In Kanal L., Ed., *Informal Proceedings of PPAI-91, International Workshop on Parallel Processing for Artificial Intelligence* (Sydney, Australia, 1991), International Joint Conferences on Artificial Intelligence, Inc., Sydney, Australia, (1991), 211-215.

Ueda K., Guarded Horn Clauses, In Wada E., Ed., *Proceedings of Logic Programming '85 - The 4th Japanese Conference on Logic Programming (Tokyo, Japan, 1985), (Goos G., Hartmanis J., Ed., Lecture Notes in Computer Science 221*), Springer-Verlag, New York, NY, (1986), 168-179.

Wise M.J., *Prolog Multiprocessors*, Prentice-Hall, Englewood Cliffs, NJ, (1986).

Wise M.J., Introduction to PMS-Prolog: A Distributed Coarse-Grain-Parallel Prolog with Processes, Modules and Streams, In Kanal L., Ed., *Informal Proceedings of PPAI-91, International Workshop on Parallel Processing for Artificial Intelligence* (Sydney, Australia, 1991), International Joint Conferences on Artificial Intelligence, Inc., Sydney, Australia, (1991a), 222-228.

Wise M.J., MB-Prolog: A Distributed Prolog with Communication via Message-Brokers, Wise M.J., Distributed at Parallel Processing for AI Workshop, IJCAI'91, (1991b).

Wise M.J., Jones D.G., and Hintz T., PMS-Prolog: A Distributed Coarse-Grain Parallel Prolog with Processes, Modules and Streams, In Kacsuk P., Wise M. J., Ed., *Distributed Prolog*, John Wiley, (Forthcoming), Yuen C.K., and Wong W.F., BaLinda Lisp: A Parallel Lisp Dialect for Biddle with the Concurrent Facilities of Linda, Technical Report TRA1/ 90, Department of Information Systems and Computer Science, National University of Singapore, Kent Ridge, Singapore, (1990).

Appendix

Listed below is a sample Prolog-D-linda configuration file. The configuration specifies two servers - `bison` and `budgie`, of which `bison` is nominated as the eval-server. The tuple space is partitioned so that `bison` maintains tuples of arity 0 and 1, and `budgie` maintains all other tuples.

```
/*----The tuple space is partitioned between two processors */
servers__([bison,budgie]).

/*----bison does the eval requests                          */
eval_server(bison).

/*----bison maintains tuple with 1 or 0 arguments           */
select_server__(bison,Tuple):-
    functor(Tuple,_,Arity),
    Arity =< 1.

/*----budgie maintains all other tuples                     */
select_server__(budgie,Tuple):-
    functor(Tuple,_,Arity),
    Arity >= 2.
```