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

Geoff Sutcliffe

James Cook University*

Abstract. This paper presents an embedding of the Linda parallel programming paradigm into Prolog, resulting in a coarsely grained parallel Prolog. The embedding provides a distributed tuple space, using unification for matching and Prolog style deduction in tuple space queries. Access to the tuple space is based on a general service mechanism that provides flexible and unrestricted manipulation of tuples. A useful mechanism, called an 'abandon request', has been introduced in the implementation, to facilitate time dependent tuple space access. A controller process is used to provide remote I/O facilities for all background processes. Two applications have been developed using Prolog-D-Linda.

1 Introduction

Prolog-D-Linda[2]. is an embedding of the Linda paradigm into SICStus Prolog (Carlsson, et al., 1991). The original motivation for embedding Linda into Prolog was the naturalness and ease with which it could be done. This naturalness has also been noted by De Bosschere and Jacquet (1992). There have been five implementations since the original (Sutcliffe & Pinakis, 1990) in muProlog (Naish, 1985), moving from a single processor running all the application processes and the centralized tuple space, through to the current version which supports applications running over an internet of processors with a distributed tuple space (hence Prolog-D(istributed)-Linda). Many aspects of the development path have been influenced by experience gained in the development of two non-trivial applications. The viability of Prolog-D-Linda as a practical parallel logic programming language has been illustrated by the success of these applications.

Section 2 of this paper describes the original Linda paradigm, and Section 3 describes how the paradigm has been adapted for Prolog-D-Linda. The implementation of the embedding is described in Section 4. Two particularly interesting aspects of Prolog-D-Linda are the deductive nature of the tuple space, and the 'abandon request' mechanism used in the implementation. These are discussed in Sections 5 and 6 respectively. Two applications developed in Prolog-D-Linda, mentioned above, are described in Section 7. Section 8 concludes the paper, highlighting the main features of Prolog-D-Linda and contrasting Prolog-D-Linda with related systems.

* The assistance of Stuart Kemp in programming Prolog-D-Linda v2 is acknowledged.

[2] Specifically, this paper describes Prolog-D-Linda v2.

2 The Linda Paradigm

Linda (Gelernter, 1985) is a coordination language (Gelernter & Carriero, 1992) that is combined with existing computation languages, to produce new parallel programming languages. Examples of computation languages that have been combined with Linda are Modula-II (Borrmann, Herdieckerhoff & Klein, 1988), C (Berndt, 1989), LISP (Yuen & Wong, 1990), Joyce (Pinakis & McDonald, 1991), and Russell (Butcher & Zedan, 1992). 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 and the Tuple Space

Tuples are collections of fields. Every field has a data type drawn from the computation 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, while an actual field has both a type and a value.

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. All processes that have access to the tuple space have access to all tuples in it. Thus the tuple space is a shared data object.

2.2 The Linda Operators

The `out` operator inserts a tuple into the tuple space. The tuple is supplied as the argument to `out`.

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 the 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. If there is more than one matching tuple then one is chosen indeterminately. If no matching tuple can be found in the tuple space, `in` blocks and waits for a matching tuple to be inserted into the tuple space. A side-effect of the blocking is to 1 the communicating processes.

The `rd` operator (pronounced read) is similar to `in`, but leaves the matched tuple in the tuple space.

The `inp` and `rdp` operators perform tasks equivalent to `in` and `rd` but are non-blocking. Instead they return a Boolean value which indicates the success of the operation.

The final operator is `eval`. The `eval` operator is syntactically similar to `out`. When `eval` is invoked, new processes are created to evaluate each of the fields in the tuple argument. When the evaluation of all fields has terminated, the tuple is placed in the tuple space.

3 Prolog-D-Linda

Prolog-D-Linda represents tuples by Prolog clauses. Both Prolog rules and facts can exist in the tuple space. Facts correspond almost directly to standard Linda tuples. The necessary existence 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 is enforced by some Linda implementations (Leichter, 1989). This requirement does not reduce the generality of the system. Formals in Prolog-D-Linda tuples are implemented by unbound variables. As data in Prolog is untyped (everything is a term) the data in Prolog-D-Linda's tuples is untyped. The effect of rules in the tuple space is discussed in Section 5.

The Prolog-D-Linda tuple space is represented by a collection of Prolog databases. Each database stores a partition of the tuple space, as determined by a user supplied Prolog program. The program defines a many-to-one relation between tuples and tuple space partitions (details are given in Section 4), and a tuple resides in the partition specified by this relation.

The `out`, `in`, `rd`, `inp`, and `rdp` operators are provided as unary Prolog predicates. The argument to each of these must be a Prolog clause, either a fact or rule. Tuples are added to the appropriate tuple space partition (the `out` operation) using Prolog's `assertz`, and removed (the `in` operation) using `retract`. Tuples in the tuple space are examined (the `rd` operation) using Prolog's query mechanism. The tuple matching method is thus generalized to Prolog's unification. As a consequence of this, formals can match and be extracted from the tuple space.

Prolog-D-Linda's `eval` operation is different from that in the original Linda paradigm. The `eval` operator 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. The `eval` operator is provided as a ternary Prolog predicate. Its arguments are: the name of the processor on which to execute the client; a Prolog query; and a list of Prolog source file names. This form of `eval` is more general than the original, and can implement the original. Restricting `eval` to starting a single process is supported by Nadkarni (1992).

4 The Implementation

Prolog-D-Linda has been implemented in SICStus Prolog v2.1 (Carlsson, et al., 1991). The development has been done on a network of DEC stations, running Ultrix and connected via an Ethernet. This environment provides access to a shared file system via Sun's Network File System. Prolog-D-Linda has also been tested on a network of SUN Sparc stations running Sun OS. The Prolog-D-Linda architecture is shown in Figure 1.

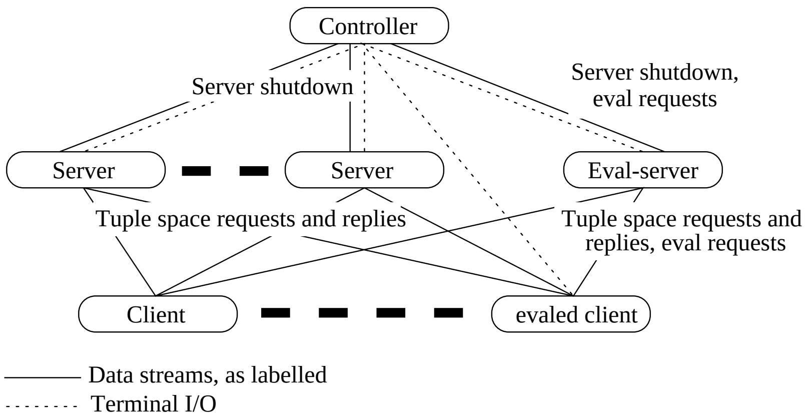

Fig. 1. The Prolog-D-Linda Architecture

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 a partition of the tuple space. Linda operations in *client* processes are translated into *requests* which are passed to an appropriate server. One of the servers is designated to be the *eval-server*. In addition to being responsible for a partition of the tuple space, the eval-server is responsible for servicing all `eval` requests. A *wait-queue*, which can store requests, is maintained within each server. The standard use for the wait-queues is to hold `in` and `rd` requests that could not be immediately satisfied, and which are waiting for an appropriate tuple to be `out`ed. Each wait-queue is implemented as a list, stored as the argument of a fact in the server's database.

Prolog-D-Linda is controlled by a single *controller* process, which must be associated with a terminal device. The controller is responsible for: starting and stopping the server processes; for reading and displaying the terminal input and output of servers; for starting clients in `eval` operations; and for reading and displaying the terminal input and output of `eval`ed clients.

Communication between the controller and servers, between the controller and `eval`ed clients, and between servers and clients, is via internet-domain stream sockets. When a server or `eval`ed client is started, its terminal I/O stream is connected to a file descriptor in the controller. In addition to these I/O streams, each server establishes a connection to the controller and every client establishes a connection to each server. The server-controller connections are used by the controller to tell the servers when to shutdown, and by the eval-server to pass information regarding `eval` requests to the controller. The client-server connections are used for sending Linda operation requests to the servers, and for receiving corresponding replies. The controller and servers monitor their connections for incoming data, and process the data as described in Section 4.2.

A configuration file must be supplied to Prolog-D-Linda. The configuration file is in the form of a Prolog program which defines three predicates :

– A single `servers__/1` fact that has a list of processor names as its argument. The processors listed each execute a server process. No processor name may appear more than once in the list.
– A single `eval_server__/1` fact that has the name of the eval-server as its argument. The eval-server name must also appear in the `servers__/1` list.
– One or more `select_server__/2` clauses that define how the tuple space is to be partitioned amongst the servers. The second argument of `select_server__/2` must return the name of the server (appearing in the `servers__/1` list) that is responsible for the tuple that is supplied as the first argument.

A sample configuration file is listed in the Figure 2. The configuration file specifies two servers, `coral` and `curlew`, of which `coral` is nominated as the eval-server. The tuple space is partitioned so that `coral` stores tuples of arity 0 and 1, and `curlew` stores all other tuples.

```
%------------------------------------------------------------------------
%---- The tuple space is partitioned between two processors.
servers__(['coral.cs.jcu.edu.au','curlew.cs.jcu.edu.au']).

%---- coral does the eval requests
eval_server__('coral.cs.jcu.edu.au').

%---- coral stores tuples with 1 or 0 arguments
select_server__('coral.cs.jcu.edu.au',Tuple):-
    functor(Tuple,_,Arity),
    Arity =< 1.

%---- curlew stores all other tuples
select_server__('curlew.cs.jcu.edu.au',Tuple):-
    functor(Tuple,_,Arity),
    Arity >= 2.
%------------------------------------------------------------------------
```

Fig. 2. A Sample Configuration File

4.2 Operation

Prolog-D-Linda is started by executing the controller. The controller reads the configuration file to get the names of the server processors. When the `servers__/1` fact has been found, the controller uses the `rexec()` system call to start each server. Each server establishes a connection to the controller, then waits for: connections from new clients; requests from existing clients; or a shutdown instruction from the controller. Client processes may be started independently at a terminal, or via the `eval` operator (see Section 4.3). New clients consult the configuration file to get the names of the server processors, to determine how the tuple space is partitioned amongst the servers, and to identify the eval-server. They then establish a connection to each server. To determine where to send a Linda operation request, a client simply evaluates an appropriate `select_server_/2` or `eval_server_/1` query.

There are two types of requests that are sent from clients to servers; *action* requests and *abandon* requests. Action requests consist of a query, a success action, and a failure action. They are serviced by evaluating the query and, depending on the success of the query, performing either the success or failure action. The available success and failure actions are: to do nothing; to send some data back to the client that sent the request; to place the request on the server's wait-queue; to re-service all requests on the server's wait-queue; or to report an error. The <query,success,failure> triplet provides a general mechanism for servicing action requests - *any* query can be passed to a server for evaluation. Abandon requests consist only of a success action and a failure action, but they are serviced in the same manner as action requests. The implicit query of every abandon request is to remove an action request, that came from the same client, from the server's wait-queue. The success or failure action of the abandon request is executed depending on the outcome of this implicit query.

The controller and servers are shutdown by entering the keyword `halt.` at the controller's terminal. When the controller reads this, it closes its ends of the connections from the servers. This causes an end-of-file condition to be sent to the servers. Upon receipt of these, the servers terminate. This in turn sends end-of-file conditions to the controller on the servers' terminal I/O streams, indicating to the controller that the servers have terminated. Although Linda operation requests can no longer be serviced, the controller continues to run until all `eval`ed clients have also terminated. This is necessary as the controller is still responsible for such clients' terminal I/O (see Section 4.4).

4.3 The Operators

Each of the Linda operators is implemented in the clients by sending an action request to the appropriate server.

The query of an `in` request is to `retract` the tuple template. The success action is to send the template, now with variables possibly instantiated, back to the client that made the request. The failure action is to place the request on the server's wait-queue. Such waiting requests are re-serviced after an `out` operation. After sending the request the client attempts to read a reply from the server, thus causing the client to block until a reply is received.

The `rd` operation is implemented in a manner similar to `in`, with the difference that the query is to evaluate the tuple template as a Prolog query.

The query of an `out` request is to `assertz` the tuple into the server's database. The success action is to re-service all requests on the server's wait-queue. This may allow some waiting requests to succeed. The failure action of an `out` request is to report an error.

The `inp` and `rdp` operations are implemented by sending the same action requests as for `in` and `rd`, but then sending an abandon request before attempting to read a reply from the server. The success action of the abandon request is to send the atom `fail` back to the client, and the failure action is to do nothing. If the action request's query succeeds, either immediately or at any time before the abandon request is received, then the requested tuple will be returned to the client and the abandon request will fail with no further action. If the action request's query fails and the action request remains on the server's wait-queue, then the abandon request's query will succeed and the `fail` reply is sent to the client. This reply is used in the client to cause the operation to fail.

Prolog-D-Linda's `eval` operator takes three arguments : the name of the processor on which to execute the client, a Prolog query, and a list of Prolog source file names. The query of `eval`'s action request calls an eval-server procedure, with these three arguments, to implement the `eval`. The success action of the request is to do nothing, and the failure action is to report an error. The eval-server procedure forwards the name of the new client processor to the controller. The controller starts a new client by `rexec`ing a Prolog saved state on the specified processor. (The Network 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 its connections to the servers. In the interim, the eval-server procedure places a tuple of the form `<client processor>(<the query>,<list of source files>)` into its tuple space. The new client `in`s this tuple, `consult`s the source files, and evaluates the query. On completion of the query the client closes the server connection and terminates.

4.4 Terminal I/O

The terminal output of servers and `eval`ed clients 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 or client from which the output originated. Input to be sent to a server or client is entered at the controller's terminal, prefixed by the descriptor number which identifies the server (the descriptor number is obtained from previous output). The controller strips the descriptor number from the input and forwards the remainder on that descriptor. This I/O system permits all servers and clients to be interactive, even though they may not be associated with a terminal device.

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 generalization of relational database to deductive database. Lloyd (1987) gives an introduction to this work. It is a logical step to extend a 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` queries may be satisfied by facts or using rules. Rules are evaluated using normal Prolog deduction. If a deduction is to succeed in the tuple space, all the required tuples (rules and facts) have 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 time required to evaluate deductive `rd`s. The second problem, which is an extreme case of the first, is the danger of a server entering an infinite deduction. Client requests to that server will not be serviced. In particular, requests that could terminate the infinite deduction are not serviced. A solution to this second problem is to restrict the nature of the deductive tuple space 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` multiple tuples of 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. To `rd` any one of `Tuple1`, `Tuple2`, ... , `TupleN` :

  ```
  rd((Tuple1;Tuple2; ... ;TupleN)).
  ```

  The disjunction between the alternate tuples is provided by Prolog's interpretation of `;`.
- A deductive tuple space has the potential for extreme space saving. Some groups of tuples can only be finitely stored in a deductive manner. For example :

  ```
  out_even:-
      out((even(Negative):-Negative < 0,!,fail)),
      out(even(0)),
      out((even(Number):-N2 is Number-2,even(N2))).
  ```

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

6 The 'abandon request'

Historically, the `inp` and `rdp` operators have been criticized because of the temporal difficulties in implementing them correctly, especially in a distributed tuple space (Leichter, 1989; Nadkarni, 1992). The criticisms are largely valid (although less convincing in the coarse grained environment of Prolog-D-Linda), but this does not remove the requirement of time dependence in 'real world' applications. The abandon request was designed to allow time dependence in client processes, but such that the server processes are not aware of any temporal issues.

The operational semantics of an abandon request have been described in Section 4.2. It is evident that it is the responsibility of the client to determine when an abandon request is sent. The recipient server simply tries to satisfy the implicit goal whenever an abandon request arrives. The server makes no guarantee that the tuple in question is not in the process of being `out`ed. However, if it is known a priori on which server a given tuple will reside (as is the case in Prolog-D-Linda) then the abandon operation can be targeted at the appropriate server. In this scenario it is guaranteed that the tuple is not in the tuple space when the waiting action request is abandoned.

Currently the abandon request is an internal mechanism within the Prolog-D-Linda implementation, and it is used only in the implementation of `inp` and `rdp`. However, other uses for abandon requests are envisaged :

- Generalizations of `inp` and `rdp` can be provided. They would take two arguments; a tuple template as is usual for `inp` and `rdp`, and a time limit on the operation. The implementation sends the normal `inp` or `rdp` action request to the server, sets an alarm interrupt to go off when the time limit is reached, then attempts to read a reply from the server. If a reply is received (the reply would be a matching tuple), then the alarm is turned off. If the read is interrupted by the alarm, then an abandon request (the same as in `inp` and `rdp`) is sent to the server, and the client again waits for a reply. Execution proceeds as for `inp` and `rdp`.
- Preemptive forms of `in` and `rd` could be provided. These forms would make their requests but not immediately wait for a reply. Rather, another interim query is evaluated. When the evaluation of the interim query completes, an abandon request is sent and the client waits for a reply from the server, as above. In this manner a client can 'lay claim' on a tuple that may appear in the tuple space at some point, but the client need not wait idle for the tuple to be `out`ed. This approach to processing while waiting for data is more light-weight than `eval`ing a separate process to wait for the data, and also avoids the problems associated with reliably transferring the data back to the original process.

7 Applications

Two non-trivial applications have been developed using Prolog-D-Linda.

7.1 Automated Deduction

Prolog-D-Linda has been used to implement distributed automated deduction systems (Sutcliffe, 1991, 1992). The deduction systems have multiple deduction components, each of which executes as a separate client process. Each component runs a different format of deduction system. Lemmas created in each of the deduction components are passed to some of the other components, via the tuple space. An extended version of one of the systems, in which the lemmas created are distributed via a separate 'lemma control' component, has also been developed. The speed-ups obtained by these distributed deduction systems are largely due to cross-fertilization between the deduction components. Prolog-D-Linda makes the implementation of the systems highly modular, and new components can easily be added.

7.2 Genetic Algorithms

Prolog-D-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 `rd`s two parent solution tuples from the tuple space; performs a crossover to produce two child solutions; for each child, `in`s a 'sucker' solution chosen at random; and then `out`s the child solution if

$$e^{\frac{(SuckerObjective - ChildObjective)}{T}} > random([0, 1))$$

otherwise `out`s the sucker (i.e., if the child has a better objective value than the sucker, then the child is always `out`ed; 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 parallelized. Similar work has been done by Ackley (1987) and Robertson (1987).

8 Conclusion

Prolog-D-Linda is a truly distributed logic programming environment. The 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[3]. In the context of parallel Prolog architectures, it has been argued that "exploiting as much fine grain parallelism as

[3] Rough timings indicate that it takes about 0.05 sec. to communicate a simple tuple between two processes. 1000 tuples of the form `tuple(blah,foo(N))`, with N running from 1 to 1000, were put into the tuple space using the `out` operator. The tuples were then immediately all removed using the `in` operator. The time of 0.05 sec. is the runtime (as provided by the SICStus Prolog `statistics/2` predicate) taken, divided by 1000. The experiment was performed with the server and clients running on separate DEC stations, connected by a lightly loaded Ethernet.

possible may be a flawed strategy; any gains through increased parallelism are wasted due to communication overheads" (Wise, 1991, p. 2). The use of a blackboard style architecture for parallel logic programming is growing in popularity in the logic programming community.

The distribution of the tuple space in Prolog-D-Linda is a significant feature. 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 general request servicing mechanism, used in the implementation of the Prolog-D-Linda servers, provides flexible and unrestricted access to the tuple space. The abandon request has approached the temporal difficulties of the `inp` and `rdp` operators from a new, cleaner, angle. The abandon request also has the potential to extend the capabilities of the Linda paradigm.

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($\uparrow$) implementation described by Shapiro, et al. (1989).

The closest relations to Prolog-D-Linda appear to be Shared Prolog (Brogi & Ciancarini, 1991), its successor, PoLiS Prolog (Ciancarini, 1992), Multi-Prolog (De Bosschere & Jacquet, 1992), and the blackboard system in BinProlog (Tarau, 1992).

– PoLiS Prolog extends upon the basic Linda model by providing multiple tuple spaces. In both Shared Prolog and PoLiS Prolog the manner in which the tuple space operators can be used is restricted to preactivation and postactivation parts, and the number of client processes cannot be changed at run-time. These constraints contrast with the unrestricted nature of Prolog-D-Linda.
– Multi-Prolog should be the fastest of this type of parallel Prolog, due to the passive nature of its blackboard, which is implemented in the shared memory of a multiprocessor. Multi-Prolog has a very rich set of operators corresponding to the Linda `in` and `rd` operators, but restricts the terms on the blackboard to be ground. Each background process spawned in Multi-Prolog evaluates its query using the same clauses as its parent, and the query must be ground. This contrasts with Prolog-D-Linda where new clients consult whatever source code is required, and any query can be evaluated.
– The BinProlog blackboard is accessed by a suite of low level primitives, which distinguish between naming and copy functions. Access to a given object on the blackboard is via a 2-key hashing table, rather than the associative lookup provided by the Linda paradigm. Any type of Prolog term can be stored on the blackboard. There is no immediate facility in this system for

starting parallel processes.

In none of these systems is the tuple space distributed or deductive.

Prolog-D-Linda is available by anonymous ftp from `coral.cs.jcu.edu.au` in `pub/prolog-linda`.

References

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

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

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

Brogi A., and Ciancarini P. (1991), The Concurrent Language, Shared Prolog, In ACM Transactions on Programming Languages and Systems 13(1), ACM Press, New York, NY, 99-123.

Butcher P., and Zedan H. (1992), Lucinda - A Polymorphic Linda, In Banatre J.B., Le Metayer D. (Ed.), Proceedings of the Workshop on Research Directions in High-Level Parallel Programming Languages (Mont Saint-Michel, France, 1991), (Lecture Notes in Computer Science 574), Springer-Verlag, New York, NY, 110-125.

Carlsson M., Widen J., Andersson J., Andersson S., Boortz K., Nilsson H., and Sjoland T. (1991), SICStus Prolog User's Manual, T91:11B, Swedish Institute of Computer Science, Kista, Sweden.

Ciancarini P. (1992), Parallel logic programming using the Linda model of computation, In Banatre J.B., Le Metayer D. (Ed.), Proceedings of the Workshop on Research Directions in High-Level Parallel Programming Languages (Mont Saint-Michel, France, 1991), (Lecture Notes in Computer Science 574), Springer-Verlag, New York, NY, 110-125.

De Bosschere K., and Jacquet J-M. (1992), Multi-Prolog: A Blackboard-based Parallel Logic Programming Language, In DeGroot D., Kacsuk P. , Succi G., Talia D. (Ed.), Proceedings of the Joint Workshop on Distributed and Parallel Implementation of Logic Programming Systems (Washington, DC, 1992).

Gelernter D.H. (1985), Generative Communication in Linda, In ACM Transactions on Programming Languages 7(1), ACM Press, New York, NY, 80-112.

Gelernter D., and Carriero N. (1992), Coordination Languages and their Significance, In Communications of the ACM 35(2), ACM Press, New York, NY, 97-107.

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

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

Nadkarni P.M. (1992), Parallel Programming with Linda: An Advanced Introduction.

Naish L. (1985), muProlog 3.2 Reference Manual, Technical Report 85/ 11, Department of Computer Science, University of Melbourne, Melbourne, Australia.

Pinakis J., and McDonald C. (1991), 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, 45.1-45.11.

Robertson G. (1987), 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, 129-140.

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

Sutcliffe G. (1991), A Parallel Linear and UR-Derivation System, In Kanal L.N., Suttner C. B. (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, 211-215.

Sutcliffe G. (1992), A Heterogeneous Parallel Deduction System, In Hasegawa R., Stickel M. E. (Ed.), Proceedings of the Workshop on Automated Deduction: Logic Programming and Parallel Computing Approaches, FGCS'92 (Tokyo, Japan, 1992), Institute for New Generation Computer Technology, Tokyo, Japan,

Sutcliffe G., and Pinakis J. (1990), 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, 331-340.

Tarau P. (1992), Blackboard based Language Extensions in BinProlog, TR92-04, Dep't d'Informatique, Universite de Moncton, Moncton, Canada.

Wise M.J. (1991), MB-Prolog: A Distributed Prolog with Communication via Message-Brokers, Distributed at the International Workshop on Parallel Processing for Artificial Intelligence.

Yuen C.K., and Wong W.F. (1990), 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.