PROLOG-LINDA :
AN EMBEDDING OF LINDA IN muPROLOG

GEOFF SUTCLIFFE
Department of Computer Science,The University of Western Australia,
Nedlands, 6009, Western Australia

and

JAMES PINAKIS
Department of Computer Science,The University of Western Australia,
Nedlands, 6009, Western Australia

ABSTRACT
This paper presents an embedding of the Linda parallel programming paradigm in Prolog. A mono-processor and a multi-processor implementation are described. Both implementations provide coarse grain parallelism to Prolog. The embedding of Linda in Prolog extends Linda's standard tuple space operations, permitting unification and Prolog style deduction in the tuple space. Examples of the added capabilities introduced by a deductive tuple space, as well as example applications of Prolog-Linda as a whole, are given.

1 Introduction

The Linda programming paradigm [1] developed by David Gelernter has recently attracted much interest as a method of enabling communication between, and synchronization of, parallel processes. A recent paper [2] compared this paradigm with other approaches to parallelism, specifically with the concurrent logic programming language Parlog86 [3]. Subsequent discussion [4] compared Linda with the FCP family of languages [5]. In Gelernter's reply to this discussion, he noted that "a Linda based parallel Prolog is a potentially more elegant, expressive and efficient alternative to the concurrent logic programming languages."

This paper presents an embedding of the Linda paradigm in Prolog, called Prolog-Linda. Prolog-Linda extends the standard tuple space operations, permitting unification and Prolog style deduction in the tuple space. Two implementations of Prolog-Linda are described, the first on a single processor, the second on a network of processors connected via an Ethernet. The parallelism introduced by these implementations is very coarse, in that each process is a seperate Prolog interpreter. Some applications of this parallelism are discussed.

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 C [6], Modula-II [7], LISP [8] and Joyce [9], 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 only possible 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 to match tuples against. 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 [10] 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 [10] (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.

Tuples are added to and removed from the tuple space using Prolog's database operations, `assertz` and `retract`. Tuple space interrogation is implemented 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 orginal.

4 Implementation

Prolog-Linda has been implemented in muProlog [11] under the UNIX operating system. muProlog provides ways of performing UNIX system calls. These are used to perform process creation and inter-process communication. For the multi-processor implementation, two new UNIX system calls had to be added to muProlog to facilitate inter-machine process creation and communication. The multi-processor implementation runs on a network of Sun SPARC station-1s running SunOS 4.0.3, and connected via an Ethernet. This provides access to a shared file system via Sun's Network File System [12].

In Prolog-Linda the tuple space and associated operations are implemented in a *server* process. Linda operations in *client* processes are translated into requests which are passed to the server. The requests are serviced by evaluating them as Prolog queries in the server. Requests for tuple space operations are simply queries on Prolog procedures which implement those operations. The use of Prolog evaluation to service requests is a general mechanism, and allows <u>any</u> query to be passed to the server for evaluation. For example, clients indicate their termination using this mechanism.

4.1 The mono-processor implementation : Prolog-1-Linda

In Prolog-1-Linda, requests for tuple space operations from all clients are passed directly to the server on a single UNIX pipe, the *request pipe*. Each client process has a *reply pipe* on which results of its requests are returned. This is illustrated in figure 1.

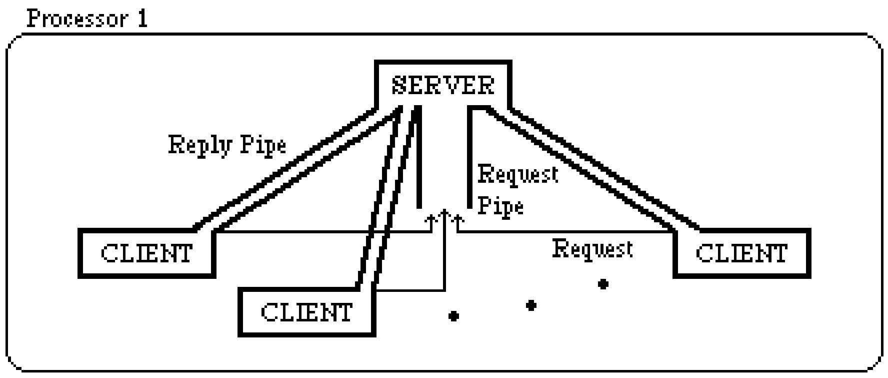

Figure 1. Prolog-1-Linda process and communication structure.

Prolog-1-Linda is started by executing the server. It creates the request pipe and starts the first client using an `eval` operation. The server then repeatedly reads and services requests from the request pipe. The server maintains a record of the number of clients, and terminates when this number drops to 0.

Prolog-1-Linda's `eval` operation takes two arguments : the name of a Prolog source file and a query on that file. A new client is created by forking the server. Before forking, the server creates a reply pipe for the new client, thus permitting both the server and the new client to access both the request and reply pipes. The newly created client removes the server procedures and tuple space from the Prolog database, loads the client procedures and the specified Prolog source, and then evaluates the query. On completion of the query, the client sends requests to the server to close the reply pipe at the server end and decrement the server's record of the number of clients. The client then terminates.

A `rd` operation queries the database, and an `in` operation attempts to retract the required clause. The result of a satisfied `in` or `rd` request is passed back to the requesting client on its reply pipe. If the server is unable to satisfy an `in` or `rd` request, the request is placed on a global wait queue. An `out` operation asserts the supplied clause into the Prolog database and causes the wait queue to be examined for `in` and `rd` requests that may have become satisfiable. These requests are removed from the wait queue and re-evaluated. The `inp` and `rdp` operations return the atom `fail` to the requesting client if the operation cannot be immediately satisfied. This is used in the client to cause the operation to fail.

4.2 The multi-processor implementation : Prolog-N-Linda

In Prolog-N-Linda, requests from clients are passed to the server via a *communicator* process, which resides on the same processor as the server, as shown in figure 2. (The communicator program is the only part of the system not implemented in Prolog. It is written in C.) The communicator reads client requests off a stream socket attached to the client. The `eval` operation, and a special system request, `close_socket`, are executed by the communicator. All other requests are passed to the server on a request pipe. Server replies are returned on a reply pipe, and forwarded to the appropriate client on the client's socket.

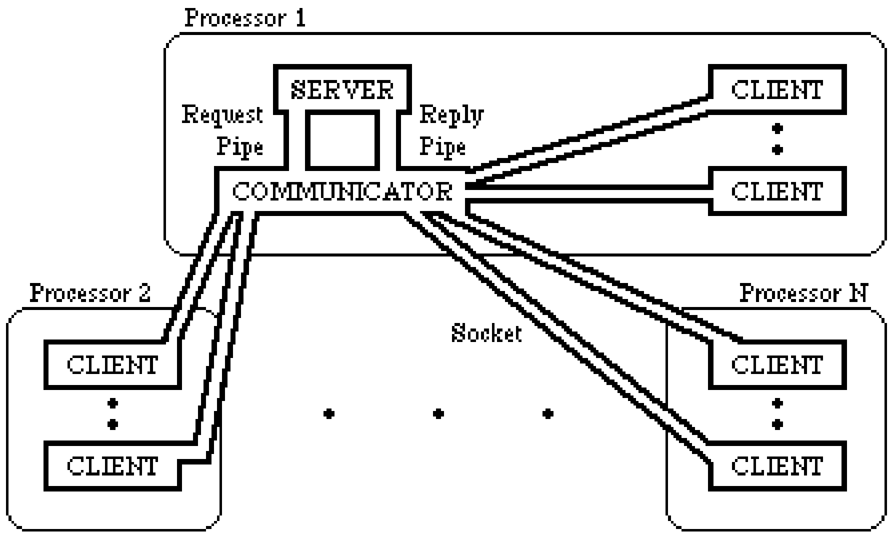

Figure 2. Prolog-N-Linda process and communication structure.

Prolog-N-Linda is started by executing the server. It creates the request and reply pipes, then forks and execs to create the communicator. To create the first client the server sends an `eval` request to the communicator on the reply pipe. This is the only time that the server initiates a Linda operation. The server then repeatedly reads and services requests from the request pipe. Every such request is followed by an integer descriptor, being the descriptor for the socket attached to the client that originated the request. The server also reads the descriptor and appends it to replies, which are written on the reply pipe. This enables the communicator to forward replies to the appropriate client. The generality of the server request evaluation system is especially useful in Prolog-N-Linda, as it enables terminal i/o requests from clients to be serviced. This is in fact the only way that clients can perform terminal i/o, and forces the i/o operations to be atomic.

The first task of the communicator is to read the start-up `eval` request off the reply pipe and execute it. The communicator then waits for client requests on client socket descriptors, and for server replies on the reply pipe. `eval` and `close_socket` requests are implemented by the communicator. The implementation of `eval` is described below. `close_socket` requests come from clients that wish to disconnect themselves from the tuple space, typically when the client is about to terminate. Upon receipt of a close socket request the communicator closes the socket on which the request was received, and decrements its counter of the number of client processes. If there are no client processes left the communicator sends a termination request to the server, and then terminates itself. Other client requests, including `out`, `rd`, `rdp`, `in` and `inp` requests, are passed to the server on the request pipe, followed by the descriptor number of the socket from which the request was read. Server replies and the associated descriptor are read off the reply pipe and the reply is forwarded to the appropriate client.

Prolog-N-Linda's `eval` operation takes three arguments : the name of a processor on which to execute the client, the name of a Prolog source file, and a query on that file. The new client is created by a remote exec of the muProlog interpreter on the specified processor passing the name of the Prolog source file as a command line argument. (The shared file system provides transparent access to files on remote processors.) The descriptor returned by the rexec is then used for communication with the standard input and output of the new client. The communicator writes the Prolog query to the descriptor. The new Prolog interpreter loads the client procedures and the specified Prolog source, then reads the query from its standard input and evaluates it. On completion of the query, the client sends a request to close the socket at the communicator end, and then 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 only viewed as storing data. Much research has been done on the generalisation of relational database to deductive database, in Prolog. [13] provides a good summary of this work. It is a logical step to extend the Prolog-Linda tuple space to a deductive tuple space.

The use of a deductive tuple space means that rules as well as facts may be added to and removed from the tuple space by `in` and `out` operations. `rd` and `rdp` requests may be satisfied by facts, or using rules. Rules are evaluated using normal Prolog deduction, including backtracking. It is possible that a `rd` request may not be satisfied if a sub-query in a rule cannot be solved. An ideal deductive tuple space implementation would keep track of sub-queries whose solution could allow the `rd` request to be satisfied. The subsequent `out` of a rule or fact that could lead to the solution of such a sub-query would then cause the original request to be completely re-evaluated. Complete re-evaluation would be necessary as rules used in the deduction may have been removed. Prolog-Linda is not yet this refined, and simply re-evaluates all waiting `rd` requests after an `out` operation.

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 [13]. Despite the problems associated with a deductive tuple space, it provides facilities that are not available from a standard tuple space :

In Linda it is awkward to simultaneously `rd` tuples of two different signatures. A method suggested in [10] 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 even.

6 Applications

6.1 Automated deduction

The Linda-Prolog system may be used to introduce parallelism into automated deduction systems. The axioms from which deductions are made may be stored in the tuple space, and accessed by several independant client inference engines, using `rd` and `rdp` operations. The inference engines may be attempting the same problem with different methods or angles of attack, or may be attempting distinct problems with the same set of axioms. Lemmas generated at derivation time can be inserted into the tuple space using `out` operations. Subsumed axioms and lemmas may be removed from the tuple space using `in` and `inp` operations.

A second application of Prolog-Linda in automated deduction is to perform syntactic and semantic checks of the derivation in parallel with inference steps. This approach is being used in the GCTP system [14].

6.2 Parallel function evaluation

Prolog-Linda may be used to apply a function to a list of values in parallel. An effective implementation is to use one process to insert function evaluation requests in the tuple space, and for other processes to perform the evaluations. The results are placed in the tuple space for the first process to retrieve. The tuple space holds tuples for values that have been, or are being, evaluated, to prevent duplicate evaluations. An example application is the static evaluation of game positions in the MiniMax algorithm.

```
%==============================================================
%----File 'maplist.pro', which controls function evaluations
%----Start the evaluation clients then send out the
%----evaluation requests
map_list(Input_list,Available_machines):-
     start_evaluators(Available_machines),
     map_each_element(Input_list,Output_list),
     send_request(writeln(Output_list)).
%--------------------------------------------------------------
```

```
map_each_element([],[]).

map_each_element([First_value|Rest_of_values],[First_result|
Rest_of_results]):-
%----Made sure this value is being evaluated
     start_evaluation(First_value),
%----Start the evaluations of the other values before collecting
%----the result for this value
     map_each_element(Rest_of_values,Rest_of_results),
     rd(evaluated(First_value,First_result)).
%-----------------------------------------------------------------
%----If a value is waiting for evaluation then do nothing
start_evaluation(Value):-
     rdp(evaluate(Value)),
     !.

%----If a value has been evaluated then do nothing
start_evaluation(Value):-
     rdp(evaluated(Value,_)),
     !.

%----If a value is being evaluated then do nothing
start_evaluation(Value):-
     rdp(being_evaluated(Value)),
     !.

%----Otherwise, send out an evaluation request
start_evaluation(Value):-
     out(evaluate(Value)).
%-----------------------------------------------------------------
start_evaluators([]).

%----Start each available machine with an eval request
start_evaluators([First_machine,Rest_of_machines]):-
     eval(First_machine,'evaluate.pro',poll_requests),
     start_evaluators(Rest_of_machines).
%=================================================================
%----File 'evaluate.pro', which does the function evaluations
poll_requests:-
     in(evaluate(Value)),
     out(being_evaluated(Value)),
     do_function(Value,Result),
     out(evaluated(Value,Result)),
     in(being_evaluated(Value)),
     poll_requests.
%-----------------------------------------------------------------
```

6.3 Interactive Linda

As Prolog does not differentiate syntactically between code and data, it is easy to implement an interactive front end to the tuple space. This example also illustrates how terminal i/o is implemented for client processes on remote machines.

```
%=================================================================
%----This query is evaluated on a remote host to provide
%----interaction
go:-
     repeat,
%----Send a prompt and a request for input
     send_request(write(?-)),
     send_request(remote_read(interact)),
%----Remove the input from the tuple space and evaluate
     in(interact(Query)),
     Query,
     fail.

%=================================================================
%----This is loaded in with the server procedures to provide
%----remote input
remote_read(Reply_predicate):-
%----Read some input
     read(Input),
%----Build a tuple using the supplied predicate and place into
%----the tuple space
     Tuple =.. [Reply_predicate,Input],
     out(Tuple).
%-----------------------------------------------------------------
```

7 Conclusion

The Prolog-Linda embedding is very natural. The pattern matching and database features of Prolog have been used directly in the embedding. Furthermore, garbage collection and hashing in the tuple space are provided free by the Prolog implementation. This naturalness contrasts with the FCP(↑) implementation described in [4].

The direct use of Prolog features to implement the tuple space and tuple space operations adds useful flexibility to the Linda paradigm. The implementation of formals in tuples is direct, in contrast to other Linda embeddings.

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.

Acknowledgement - Thanks to Nick Lewins for his input on deductive tuple spaces, and comments on early drafts of this paper.

8 References

1. D. Gelernter, *Generative Communication in Linda, ACM Transactions on Programming Languages*, 7(1) (1985), pp 80-112.
2. N. Carriero, and D. Gelernter, *Linda in Context, C. ACM*, 32(4) (1989), pp 444-458.
3. G.A. Ringwood, *Parlog86 and the Dining Logicians, C. ACM*, 31(1) (1988), pp 10-25.
4. E. Shapiro, et. al., *Technical Correspondence, C. ACM*, 32(10) (1989), pp 1244-1258.
5. E. Shapiro, *Concurrent Prolog : Collected Papers, Volumes 1 & 2*, (MIT Press, Cambridge, MA, 1987).
6. D.J. Berndt,*C-Linda Reference Manual, Version 2.0*, (Scientific Computing Associates Inc., New Haven, 1989).
7. L. Borrmann, M. Herdieckerhoff, A. Klein, *Tuple Space Integrated into Modula-2, Implementation of the Linda Concept on a Hierarchical Multiprocessor, Proceedings of CONPAR '88*, (Cambridge University Press, 1988).
8. C.K. Yuen, W.F. Wong, *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).
9. J. Pinakis, C. McDonald, *The Inclusion of the Linda Tuple Space Operations in a Pascal-based Concurrent Language*, (Unpublished manuscript, 1989).
10. J.S. Leichter, *Shared Tuple Memories, Shared Memories, Buses and LAN's - Linda Implementations Across the Spectrum of Connectivity, Ph.D. Thesis*, (Yale University, Yale, CT, 1989).
11. L. Naish, *muProlog 3.2 Reference Manual, Technical Report 85/11*, (Department of Computer Science, University of Melbourne, Melbourne, Australia, 1985).
12. R. Sandberg, *The Design and Implementation of the Sun Network File System, USENIX Association Conference Proceedings*, (1985), pp 119-130.
13. J.W. Lloyd, *Foundations of Logic Programming*, (Springer-Verlag, Berlin, 1987).
14. G. Sutcliffe, *A General Clause Theorem Prover, in Proceedings of 10th International Conference on Automated Deduction, Lecture Notes in Artificial Intelligence* 449, ed. M.E. Stickel (1990), pp 275-276.