A Distributed Linda-like Kernel for PVM

Antony Rowstron, Andrew Douglas and Alan Wood
The University of York, Heslington, York, YO1 5DD, UK.

Abstract

In this paper we describe the implementation of a Linda-like system on top of PVM. The kernel was initially developed for a Transputer based system[4]. For flexibility the system was ported to PVM. This paper outlines how the York kernel works, how it currently performs with respect to a similar system and possible future developments for the PVM version.

1 Introduction

Linda[2] is a communication model for parallel processes. It is also often described as a co-ordination language, based on the idea that any parallel program has two parts: computation and coordination[6]. Linda is always embedded within a *host* language which provides the ability to do computation.

The Linda model only provides communication between different processes involved in a (concurrent) computation. The communication is provided via a shared *tuple space*, which can be thought of as a shared bag into which tuples are inserted and withdrawn by the user processes. The tuple space acts as a logical shared memory with an associative lookup mechanism. This means that there is the need to have some sort of *kernel* which acts as a controller of the tuple space. User processes then communicate with the kernel, sending it commands which represent the Linda primitives and tuples. These Linda primitives are described in Section 2.

A distributed kernel avoids the tuple space becoming a bottleneck. If there were many user processes all issuing commands to a single kernel process, the speed of the entire system is controlled by the speed of that process.

2 Linda

The Linda model uses the exchange of tuples as the means of process coordination. A tuple is an ordered sequence of heterogenously typed objects. For example, the tuple *[1,true,"Bloggs"]* contains an integer, a boolean value and a string. When a tuple is to be retrieved from a tuple space a template is provided by the user in order to allow the tuple to be found. For example, the template {*?int,?boolean,"Bloggs"*} is one of many templates which matches the example tuple. Here we use the notation *?type* to indicate the type of the the field at that position. Therefore, this template matches *any* tuple which has an integer as its first field, a boolean value as its second field and the string “Bloggs” as its third field.

The exact syntax of tuples, templates and primitives depends largely on the syntax of the host language. The Linda model provides four[1] basic operations to manipulate tuples stored in tuple spaces:

out (*tuple***)** This places the given tuple into the tuple space.

in (*template***)** This attempts to match the tuple template with a tuple in the tuple space and return it. If a match does not exist the operation *blocks* until a suitable tuple becomes available (if ever). If the match succeeds, the matched tuple is removed from the tuple space.

rd (*template***)** This is the same as **in** except the matched tuple is not removed from the tuple space — a copy is returned.

eval (*tuple***)** This creates so-called *active* tuples — each element of the tuple is evaluated *concurrently* and, when complete, the resulting tuple of values is placed in a tuple space. This is Linda’s mechanism for spawning new processes. In the system described here, the `eval` primitive is implemented by mapping it on to the `pvm_spawn` function.

The Linda model is intended to be an abstraction, and as such is independent of any specific machine architecture. This has meant that alternatives and extensions to the basic Linda model have been proposed and investigated. The extensions that are currently supported in the York kernel are:

Multiple tuple spaces The various ways in which multiple tuple spaces may be added has been discussed for some time. Currently, York has adopted the idea that each tuple space is independent of any other tuple space. There are however other ways of implementing multiple tuple spaces, such as using a hierarchical structure [5, 7]. The addition of multiple tuple spaces is achieved by incorporating a *tuplespace* type and a primitive to create a new tuple space within the model.

collect primitive A new tuple space primitive[1], `collect()`, has been proposed which subsumes the `inp` and `rdp` primitives. Given two tuple space handles (`ts1` and `ts2`) and a tuple template, the primitive `collect(ts1,ts2,template)` *moves* tuples that match the template in `ts1` to `ts2`, returning a count of the number of tuples transferred.

copy-collect primitive This is another new tuple-space primitive[8], which has recently been proposed in the light of work on the implementation of many different algorithms in Linda. This primitive is very similar to `collect` but it is non-destructive. Hence it *copies* all the tuples that match the tuple template to a separate tuple space rather than *moving* them. As with `collect` it returns a count of the number of tuples copied.

[1] There are two other primitives in the standard Linda model, `inp` and `rdp`, which are predicate versions of `in` and `rd` which return either a tuple or ‘false’. We do not use them because they are semantically problematic[1].

3 Why implement a logical shared memory using PVM?

There are several advantages and disadvantages of implementing a Linda-like system on top of PVM. Linda offers a simple and uncomplicated way to allow parallel programs to communicate. The Linda model allows spatial and temporal decoupling of user processes. Because a tuple space is a persistent shared memory user processes involved in the same computation need not necessarily execute at the same time, as the only way for two user processes to communicate is via a tuple space. Also, each user process need not know about any other user processes. This is in marked contrast to the message passing style adopted by PVM where in order to send a message to another process the sending process must at least be aware of the destination process. This means that any two user processes that communicate can not be spatially decoupled. This also implies that the source and destination processes can not be temporally decoupled.

The disadvantages are primarily ones of speed (especially implementational efficiency) and non-determinism. There has been some work looking at how to optimise a Linda program by optimising the use of Linda primitives[3] in order to increase performance. Non-determinism within the model means that it is not possible to predict the performance of some user programs, and can mean that execution times for a program can vary widely from one execution run to the next. However, by adding extra synchronisation it is often possible to reduce, if not remove, this problem.

4 The Implementation

The kernel is implemented as a set of *tuple space servers*. A single tuple space server is normally placed on every processor that is available. Each tuple space server is responsible for a subset of all the tuples. There is no replication of the same tuple in the kernel.

In order to keep the communication to a minimum the user processes always decide which tuple space server a tuple or template should be sent to, by means of a hashing algorithm. This selects a tuple space on the basis of the length of the tuple and type and value of its first field. The tuple is then sent to that particular tuple space server with a tag indicating the Linda primitive (which can only be an `out`).

If the hashing algorithm is given a template then, using the same information, it will again return a single tuple space server, and the template is sent with a tag indicating the Linda primitive. However, sometimes only the type of the first field of a template is known, so the hashing algorithm cannot say exactly which tuple space server should be used. Therefore it returns a set of tuple space servers that could be used. In order to allow the retrieval of a tuple, any template that could match it must have the tuple's tuple space server in its set of possible tuple space servers. From this set of possible tuple space servers one is chosen. In the original version this was done by picking one at random. However, in the PVM version a check is first made to see if any of the tuple space servers reside on the same processor as the user process. If there is then it is chosen, else one is chosen at random. The template is then sent to that tuple space server with a tag that indicates the primitive and a list of the other tuple space servers to which that request could have been sent. The user process then waits for a reply from the tuple space server.

Initially, a tuple space server tries to complete all requests locally. This is always possible for an `out`. If the request was to find a tuple and a tuple that matches the given tuple does not exist locally and a list of tuple space servers was provided by the user process, then the tuple space server will coordinate the search with the listed tuple space servers. In the case of a `collect` or a `copy-collect`, the tuple space server will have to communicate with all tuple space severs in given set. If a request was made for a blocking primitive (`in` or `rd`) and no tuple that matches the template can be found either locally or on another tuple space server, then the request is stored on a list of blocked requests. If a tuple arrives that satisfies the request, it is removed from the blocked request list and the tuple is returned to the user process.

5 System performance

We have performed a number of tests comparing the Glenda[2] system and the York Linda Kernel. We present here the results of a common test used when evaluating Linda systems, the ping-pong test. This test shows the speed of a single tuple space server.

The ping-pong test uses two processes. The first process produces a tuple which is placed in a tuple space. This process then proceeds to read a different tuple from the tuple space. Meanwhile the second process first attempts to read a tuple (the one which was created by the first process) and then places a different tuple in the tuple space (which will be read by the first process). This embodies the idea of a single tuple being passed between the two processes.

Implementation	Configuration		Execution Time
	Indy 1	Indy 2	(in seconds)
York Linda Kernel	ping/pong	kernel	0.93
	ping	pong/kernel	0.99
	pong	ping/kernel	1.04
		ping/pong/kernel	0.99
Glenda	ping/pong	gts	2.11
	ping	pong/gts	1.89
	pong	ping/gts	1.81
		ping/pong/gts	1.33

Table 1: Ping-Pong Timings

Table 1 shows the time taken to do 100 full ping-pong cycles[3] where the tuple contains only an integer. The test was performed on two Silicon Graphics Indy worksta-

[2]Glenda is available by anonymous ftp from **seabass.st.usm.edu**. The version used in these tests was version 0.91

[3]In other words, 400 tuple space operations (One cycle is: `out` → `in` → `out` → `in`).

tions running PVM 3.3.7. The table shows what processes were running on which machine. Where `kernel` is the York tuple space server process and `gts` is the Glenda system tuple space server. As can be seen in every configuration the York tuple space server is faster than the Glenda system.

However, the real advantages of our kernel can only be really appreciated when many tuple spaces servers are being used. At this point we are getting concurrent execution of the Linda primitives. In order to demonstrate this the the ping-pong test was expanded to run two ping-pong tests in parallel using a network of four Silicon Graphics Indys. The results are shown in Table 2.

No. of tuple space servers	Average Execution Time
1	1.39 seconds
2	1.02 seconds

Table 2: Multiple Ping-Pong Timings

Table 2 shows clearly that the time taken for the ping-pong processes to execute is greater when one tuple space server is being used. This is because the tuple space server is acting as a bottleneck. However, when two tuple space server are being used each ping-pong test uses a different tuple space server, so the execution time is the same as when one ping-pong process is executing with one tuple space server.

6 Conclusion and future work

We have briefly outlined our implementation of a Linda-like kernel using PVM. There are a number of areas where further work would be beneficial. Currently in our C implementation the syntax of the primitives and the format of tuples and templates are particularly awkward. These syntax problems would be solved with the development of of a preprocessor (as used in the Glenda system). A pre-processor would also offer the possibility of different hashing algorithms being developed by the pre-processor to ensure that tuples are evenly distributed across the tuple space servers.

Currently just one hashing algorithm is used. The development of other hashing algorithms is important because the more evenly the tuples are distributed the less likely an individual tuple space server is to become *saturated*[4]. When saturation occurs that particular tuple space server then becomes a bottleneck, potentially slowing the entire kernel down.

A number of performance optimisations could be made to the tuple space server itself. For example, the manipulation of the tuples within the tuple space server could be improved. In this version tuples are stored as lists whereas the use of a *hashing tree* would improve insertion and search speed. This is significant because if there are thousands or millions of tuples the search time can become significant.

When user processes leave tuples in a tuple space they remain there[5] because the

[4]The point where a time spent by a request in a tuple space servers queue is significant.

[5]Unless, of course, another user process is able to remove them.

kernel is persistent. This causes a build up of unwanted tuples over time. Therefore, we have been considering ways of adding garbage collection to the kernel. Although this garbage collection can be done implicitly, we are considering initially allowing a user process to explicitly say that a particular tuple space should be destroyed.

7 Acknowledgements

During this work Antony Rowstron was supported by a CASE studentship from British Aerospace Military Aircraft Division Ltd and the EPSRC of the UK. Andrew Douglas was supported by an EPSRC grant (No. GR/J12765).

References

[1] P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[2] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[3] N. Carriero and D. Gelernter. A foundation for advanced compile-time analysis of Linda programs. In *Languages and Compilers for Parallel Computing*, Lecture Notes in Computer Science Volume 589, pages 389–404. Springer-Verlang, 1991.

[4] A. Douglas, A. Wood, and A. Rowstron. Linda implementation revisited. In P. Nixon, editor, *Transputer and occam developments*, Transputer and occam Engineering Series, pages 125–138. IOS Press, 1995.

[5] D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe. Volume II: Parallel Languages*, pages 20–27. Springer-Verlang, Lecture Notes in Computer Science Volume 366, 1989.

[6] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[7] S.C. Hupfer. Melinda: Linda with multiple tuple spaces. Technical Report YALEU /DCS/RR-766, Yale University, 1990.

[8] A. Rowstron, A. Douglas, and A. Wood. Copy-collect: A new primitive for Linda. *Submitted to Parallel Processing Letters*, 1995.