An efficient distributed tuple space implementation for networks of heterogenous workstations

Antony Rowstron and Alan Wood
Department of Computer Science, University of York,
York, YO1 5DD, UK

Abstract

The distributed tuple space concept, on which the Linda process coordination model is founded, has given rise to several implementations on parallel machines and networks of heterogenous workstations. However, the fundamental techniques used in there systems have remained largely unchanged from the original Linda implementations.

This paper describes a novel implementation which, using extensions to the original Linda model and recently developed bulk access primitives for tuple spaces, is able to demonstrate 10 to 70 times speed improvements over the best available commercial system. This is achieved dynamically without any compile time optimisations.

1 Introduction

Since the Linda model was invented there has been a constant stream of implementations, for many different platforms. However, these earlier implementations appear to be founded on the same principles used in the original implementations. Here we describe a new technique, for networks of heterogenous workstations, which has been designed differently from traditional implementations and has subsequently demonstrated major speedups over those approaches.

Section 1.1 describes the basic Linda tuple space model, how multiple tuple spaces are added to the model, and the new primitives this can introduce. Section 2 provides a brief guide to other implementations. Section 3 describes the new implementation technique, section 4 describes why this achieves a speedup over other earlier implementations and section 5 demonstrates experimental results to justify our claims.

1.1 The Linda model

The Linda model is now well known and a detailed description can be found in [CG90]. The main primitives are:

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

in(*template*) This removes a tuple from a tuple space. The tuple removed is associatively matched using the template[1] and the tuple is returned to the calling process. If no tuple that matches exists then the calling process is blocked until one becomes available.

rd(*template*) This primitive is identical to *in* except the matched tuple is not removed from the tuple space, and a copy is returned to the calling process.

eval(*active-tuple*) The *active-tuple* contains one or more functions, which are then evaluated in parallel with each other and the calling process. When all the functions have terminated a tuple is placed into the tuple space with the results of the functions as its elements.

Some Linda systems support two other primitives, `inp` and `rdp`. These are non-blocking versions of `in` and `rd`. Instead of blocking they return a value to indicate no tuple was found. Linda is only concerned with communication, and therefore the computation elements of a parallel language are provided by a *host* languages, such as C, C++, ISETL[DRW95], Fortran etc.

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 Linda model have been proposed and investigated. One important suggestion has been the addition of multiple tuple spaces.

1.1.1 Multiple tuple spaces

The incorporation of multiple tuple spaces has been discussed for some time. Schemes based on hierarchies of tuple spaces have been suggested[Gel89, Hup90] as well as one with a mixture of flat and hierarchical tuple spaces[Jen93]. However, for the purpose of the work described here, *how* multiple tuples spaces are added to the model is not important – only the fact they are available.

1.1.2 Additional primitives

When multiple tuple spaces are added to the model, there is a need to consider if new primitives are required to manipulate tuple spaces rather then tuples. For example, it has been proposed[NS93] that a `copy` and a `move` primitive should be added. The `copy` primitive would copy all tuples from one tuple space to another, whilst the `move` primitive would move all tuples from one tuple space to another. Such primitives although appearing simple additions to the model (particularly from a kernel implementors point of view), may not necessarily provide the kind of information that a Linda programmer may require and, indeed, have serious semantic problems.

[1] Sometimes referred to as an *anti-tuple*.

At the University of York we have added the following two primitives, which although superficially similar to `move` and `copy` have important semantic differences. They provide information which is useful to a programmer and overcome fundamental problems in the model.

collect Syntactically[2] the `collect`[BWA94] primitive can be represented as: `collect(ts1, ts2, template)` where `ts1` and `ts2` are two tuple space handles[3] and `template` is a template. The `collect` primitive *moves* tuples in `ts1` that match `template` into `ts2`, returning a count of the number of tuples moved.

copy-collect This new primitive is syntactically and semantically similar to `collect`. `copy-collect(ts1, ts2, template)`[RW96] *copies* all available tuples that match the given `template` in the source tuple space (`ts1`) to the destination tuple space (`ts2`). As with `collect` it returns a *count* of the number of tuples copied.

One justification for `collect` is that it provides information about the number of tuples of a particular format – more information can be found in [BWA94]. The justification for `copy-collect` is that it solves the *multiple* `rd` *problem*. Both a description of `copy-collect` and the *multiple* `rd` *problem* can be found in [RW96].

2 Overview of other implementations

In order to implement a Linda system there needs to be some sort of underlying control system, which we shall call a *kernel*. Kernel implementations fall into one of two categories which we describe as *open* and *closed* implementations. A *closed* implementation is one where all processes that are to interact must be known about at compile time. An *open* implementation is one where processes can join and leave the kernel at will, and therefore can be compiled separately (and indeed can be written in different languages). The usual difference between open and closed kernels is that the closed kernels can use information derived at compile time for optimisations. Open implementations are often considered advantageous because it would be possible to use persistence of tuple spaces to a programmer advantage. A persistent tuple space can effectively be used as a long term store for information that will be required in the future.

The first implementations of Linda were produced by Yale University and were all *closed* systems[Car87, Lei89, Bjo92] using precompilers to gain information about the use of tuples which could be used to distribute tuples intelligently and retrieve them quickly. Since then there have been numerous other implementations largely based on the work of [Bjo92]. Distributed servers provide a mechanism for the kernel to store tuples. Processes then communicate with these servers to retrieve tuples. The exact way in which tuples are stored on

[2] Using a C-Style syntax.
[3] A tuple space handle is a unique handle associated with each tuple space.

to the servers is normally dependent on tuple contents, size and other such information.

To incorporate multiple tuple spaces the kernels normally tag tuples with the name of their containing tuple space. Depending on the implementation requirements this extra field is then either used or not used in the hashing process. We will refer to all implementations that fall in this category as *traditional* implementations.

The implementation described in [NS94] is based on the work of [Jen93] on multiple tuple spaces. It is particularly interesting because they introduce the concept of *special tuple spaces* which allow the programmer to tag a tuple space with a special descriptor, such as: local tuple space, compiled tuple space, replicated tuple space, and so on. This allows a kernel to use this *explicit* information to improve performance. However, the explicit declaration of tuple space "types" seems to complicate the model. It also raises the interesting question of how do you alter the type of a tuple space? If a tuple space is declared as local - can it subsequently change its type to compiled?

3 The novel implementation technique

Our new implementation builds on the concepts of [NS94] except that tuple spaces provide *implicit* information about which processes are likely to consume tuples. The implementation is therefore focused on *tuple spaces* rather than *tuples*. The kernel is able to classify every tuple space as either a *local tuple space* or a *remote tuple space* on the fly. A local tuple space is one which can only be accessed by one process. A remote tuple space is one that many processes can access. Our prototype implementation[4] uses a slightly more restrictive definition, where if a tuple space handle is contained in a tuple which is placed in a remote tuple space then the tuple space referred to has to become a remote tuple space, and once a tuple space is a remote tuple space it can not become a local tuple space. An earlier University of York implementation[DWR95, RDW95] used a "traditional" approach. However, experimental work showed that with the addition of multiple tuple spaces programming styles began to alter in many circumstances. However, the general use of tuple spaces followed a similar pattern. Therefore, not allowing tuple spaces to be local once they have been remote is not currently a problem as it rarely happens.

Figure 1 shows several processes and tuple spaces, and indicates the difference between remote and local tuple spaces.

Once a tuple space is classified, this controls where its tuples are stored. If the classification for a tuple space changes then its tuples are moved to the correct storage place for that tuple space and that format of tuple.

It should be noted that making this distinction between local and remote tuple spaces does not alter the semantics of the Linda model. Therefore, from a user's point of view there is no distinction between a local and remote tuple

[4] Our current prototype implementation is built on top of PVM version 3. For more information on PVM see [SDGM94].

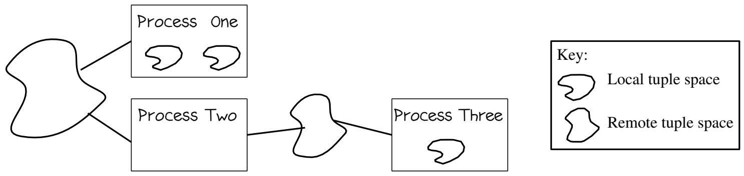


Figure 1: Diagram showing the relationship between local and remote tuple spaces.

space. To this end all local tuple spaces could be treated as remote tuple spaces and the implementation would degenerate into a "traditional" kernel.

Figure 2 shows the general structure of the kernel. The kernel has two distinct sections, a set of *Tuple Space Servers* and a number of *Local Tuple Space Managers*. These are now described in more detail.

Tuple Space Server The Tuple Space Server (TSS) is a stand alone process which acts as a tuple server. It receives tuples and requests for tuples and services them. All remote tuple spaces are stored on a TSS or on a set of TSSs. The precise method of distributing the tuples is not important, and a number of schemes could be used. In this implementation a method similar to that used in the original York kernel is used[DWR95, RDW95]. It should be noted that tuple spaces *never* migrate from one TSS to another, but tuples from the same tuple space may be distributed over several TSSs.

Local Tuple Space Manager The Local Tuple Space Manager (LTSM) is the part of the kernel which is able to *dynamically* (with no inter-process communication) information about which tuples spaces are local and which are remote. The LTSMs therefore control all movement of tuple spaces. A tuple space will only ever migrate from a TSS (or set of TSSs) to a LTSM or vice-versa. The LTSM stores all local tuple spaces which are local to its process. The LTSM is *not* a stand alone process, it merely exists as a set of procedures linked unto each user process – this does not require any form of inter-process communication between the LTSM ad its user process, not even interrupts or signals.

When tuple space operations occur the LTSM checks internally to see if the tuple space is a local tuple space. If it is, then the operation is performed on the tuple space (stored in a data structure within the LTSM). There is no communication with any TSS. If the operation is on a remote tuple space then the LTSM contacts the relevant TSS (or TSSs), and awaits a reply if the operation requires it. Whenever a tuple leaves a LTSM to go to a TSS the tuple is checked to see if it contains a tuple space handle. If any tuple space handles exist then the kernel checks to see if the handle is for a local tuple space. If it is, the tuple space is moved to a TSS or a set of TSSs.

However, the crucial point is that both the operations of `collect` and `copy-collect` are slightly different. Where they are performed depends on

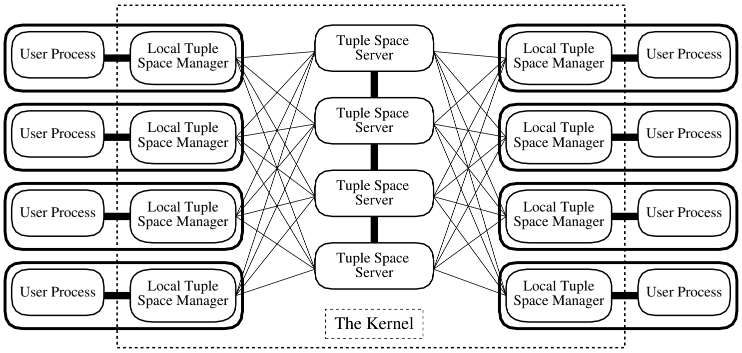


Figure 2: Diagram showing the layout of the kernel.

the status of the two tuple spaces used as shown in Table 1. If the operation requires a tuple space to be moved to a LTSM, the LTSM will send a request for the tuple space to the relevant TSS(s). If the result requires a tuple space to be moved from a LTSM to a TSS then the tuple space is automatically sent by the LTSM (as it is if a tuple leaves the LTSM containing a handle of a local tuple space).

		Source tuple space	
		Local	Remote
Destination tuple space	Local	LTSM	TSS(s) (Result to LTSM)
	Remote	LTSM (Result to TSS)	TSS(s)

Table 1: Table showing where `collect` and `copy-collect` operations should be performed.

When no tuple space movement is needed there is *no* movement of tuples between one LTSM and another, or indeed between one TSS and another. In other words when a `copy-collect` is performed by the TSSs, no tuples will *ever* be moved from one TSS to another.

The only other occasion when the bulk movement of tuples occurs is when a tuple space handle for a local tuple is placed in a tuple in a remote tuple space. In this case the local tuple space referred to will become a remote tuple space and will be moved from the LTSM on which it was resident to a single or set of TSSs.

This gives a brief overview of the implementation. There are many details that we have omitted due to the nature and size of this paper. For example in the implementation we achieve parallelism between user computation and the receiving of tuple spaces in the LTSM. Also we often never need to insert tuples into the local data structure and can read them straight out of the message buffer. All these things help to make the implementation fast and efficient. Now we consider in detail what properties our implementation has that allows it to achieve a speedup over other implementations.

4 Obtaining speed

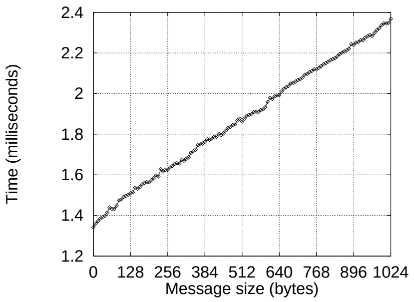
It might seem strange that a kernel that requires the bulk movement of tuples can achieve a speed up over traditional implementations. This happens because of two factors, which we now describe in more detail.

4.1 Packet sizes

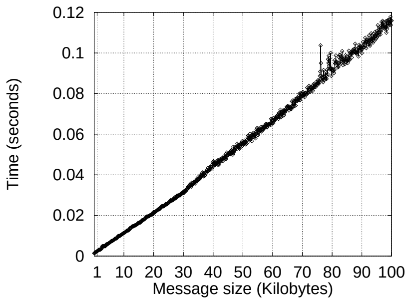
When a tuple space is being moved from a TSS to a LTSM or vice-versa it is possible to control how many tuples are packed into a single packet . By examining the characteristics of network communication it is possible to see exactly how this can increase the performance of the system. The characteristics measured here are derived using PVM, running over two Silicon Graphics Indy workstations connected using a non-dedicated 10 Megabit per second Ethernet.

Figure 3(a) shows the time taken to send messages of between 0 and 1 Kilobyte, and Figure 3(b) shows the time taken to send messages of between 128 bytes and 100K in size. Figure 3(c) shows the bandwidth in megabytes per second that is achievable for messages of sizes between 0 and 1 Kilobyte, and Figure 3(d) shows the bandwidth for messages of between 128 bytes and 100K in size. This clearly shows that as the message size increases the bandwidth increases. A packet size of about 30K seems to produce the best performance. Figure 3(e) shows the time that it takes to send a single byte in a message for messages of sizes between 0 and 1 Kilobyte, and Figure 3(f) shows the time that it takes to send a single byte in a message for messages of between 128 bytes and 100K in size. These demonstrate that the cost of sending bytes significantly drops as more bytes are packed into a message.

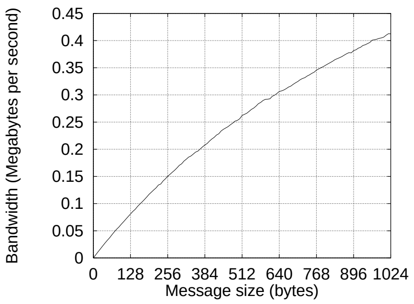
Therefore, the ability to pack several tuples (or indeed several hundred tuples) into a single message allows the message sizes to be increased achieving better performance. In many of the domains where we use Linda it is not uncommon to have tuples containg just a few elements, with a total size of under 50 bytes (in our implementation this would be the size of a tuple with nine integer fields for example). Therefore the cost of sending these little tuples is very large compared with the cost packing them into 30K packets and sending them.



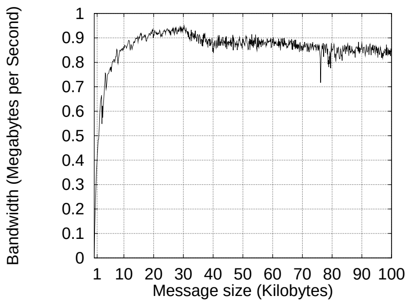
(a) Latency for messages from 0 bytes to 1K in size.



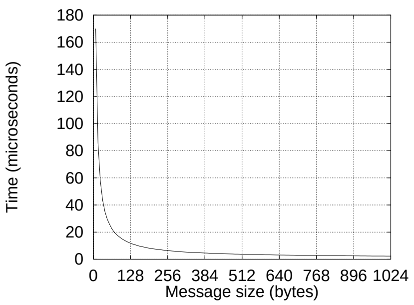
(b) Latency for messages of 128 bytes to 100K in size.



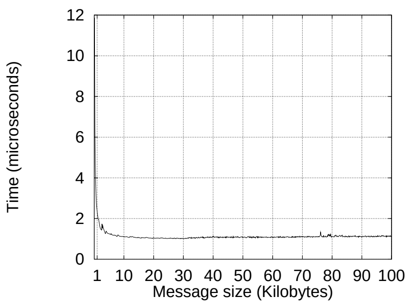
(c) Bandwidth versus messages size for messages from 0 bytes to 1K.



(d) Bandwidth versus messages size for message sizes of 128 bytes to 100K.



(e) The time taken to send a single byte for messages from 0 bytes to 1K.



(f) The time taken to send a single byte for messages size from 128 bytes to 100K.

Figure 3: The Ethernet network profile.

4.2 Less communication

By ensuring that a tuple space is stored locally if it is only accessed by a single process that, we reduce the number of messages that need to be transmitted to the tuple space servers. In traditional kernels every `in` or `rd` would require two messages between the process and the tuple space servers. The first message is from the process to the tuple space server with the specified template, tuple space and any other information the tuple space server may require. If the tuple space server has the tuple it will then return it, requiring the second message. If the tuple space server does not contain the tuple then further communications may be required in order to check that no other tuple space servers have a matching tuple.

In our system if the tuple space is stored in the LTSM, then there is no need for any communication between the LTSM and the TSS(s). Therefore, even if there was no saving on the time to bulk move the tuples as a group and it was only possible to transmit them individually from the TSS to the LTSM, there will be fewer messages if at least 50% of the tuples transferred are required. This is because the number of messages required to move a tuple space of N tuples to the LTSM will be $1 + N$ messages, whereas if the tuples were individually fetched from the tuple space server the number of messages required would be $2N$. If the tuples can be packed into larger messages then the number of messages required will be lower. By looking at the characteristics of actual programs we speculate that in those using `collect` and `copy-collect` it is most likely that all the tuples transferred will be used.

Reducing the number of requests that a TSS gets also has further advantages. It is common in most implementations for individual servers to saturate. That is they receive more messages per second than they can process, so a queue forms. These are often referred to as hot spots. By lowering the number of messages being sent to the TSSs we are lowering the load on each TSS so the chances of it saturating drop.

5 Experimental results

In order to demonstrate the performance of our kernel we compare the performance with the LTSM enabled and disabled. When the LTSM is disabled the kernel degenerates into a "traditional" implementation, where all tuple spaces are always remote tuple spaces and therefore distributed across the TSSs.

For the experiments C was used as the host language. Two Silicon Graphics Indy workstations were used, with a 10 Megabit per second Ethernet non-dedicated connection. We used only two workstations because there is no significant difference in the results using more workstations, and as each of the experiments outlined here involve a single process there seems little point in using more workstations.

Table 2 shows the times (in seconds) taken to perform two experiments. Experiment One shows the time taken to place 1000 tuples in a tuple space

Experiment	One		Two	
LTSM	Disabled	Enabled	Disabled	Enabled
`out`	2.890	2.861	2.769	0.018
`in`	3.224	3.227	3.270	0.017
Total	6.114	6.087	6.039	0.035

Table 2: Local tuple space access versus remote tuple space access.

using `out` and then retrieve them (in any order) using `in`. The tuple space used is forced to be a remote tuple space. As can be seen there is no significant difference in the time taken when the LTSM is enabled or disabled. This is expected, because the tuple space is a remote tuple space. Experiment Two shows the time taken to perform the same experiment as in Experiment One, except the tuple space is not forced to be a remote tuple space. In this case when the LTSM is enabled it detects that the tuple space is a local tuple space and stores it locally, creating a large speed up over the case when the LTSM is disabled.

Experiment	Three		Four		Five	
LTSM	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
`out`	2.802	0.018	2.770	0.018	2.769	2.767
`collect`	n/a	n/a	0.007	0.038	0.007	0.007
`in`	3.303	3.877	3.258	3.670	3.255	0.056
Total	6.105	3.895	6.035	3.726	6.031	2.830

Table 3: Transfer characteristics between local and remote tuple spaces.

Table 3 shows the times (in seconds) taken to perform Experiments Three, Four and Five. Experiment Three shows the time taken to place 1000 tuples in a tuple space using `out`, then place in `UTS`[5] a tuple containing the handle of that tuple space and then retrieve them (in any order) using `in`. This quite clearly shows that the time taken to perform the operation is less when the LTSM is enabled. This is because, when the movement of tuple occurs (as the tuple is placed in the `UTS`), the tuples are packed into larger packets for dispatching to the TSSs. What is important is that it is quite clear that the LTSM *does not* slow the system down when such an operation occurs. Experiment Four shows the time taken to place 1000 tuples in a tuple space using `out`, then `collect` them all into `UTS` and then retrieve them from `UTS` (in any order) using `in`. As one would expect the times are comparable to those of Experiment Three, as essentially the same tuple space "movements" are occuring. Experiment Five shows the time taken to place 1000 tuples in a tuple space using `out` in `UTS`, then `collect` them all into a tuple space and then retrieve them from this tuple space (in any order) using `in`. The tuple space used to store the `collect`ed tuples is a tuple space that is local. This is an important operation, as it represents the

[5] `UTS` is a *universal tuple space*, which all processes have access to.

communications behaviour of the basic operations that a process has to perform to overcome the multiple `rd` problem[RW96].

These results demonstrate the effect that LTSM has on the execution time for a number of specific examples. We have shown that the LTSM does not slow the kernel down, and that when used it provides speed increases for bulk tuple operations.

We now consider how our kernel performs against other kernels. The kernel performs better than the original York kernel[DWR95, RDW95]. It has been shown that the old kernel performs better than Glenda[6][RDW95]. Therefore, this kernel will perform better than Glenda! There is also a commercial version of Linda available, called C-Linda[7][Ass95]. This is one of the implementations that uses a pre-compiler and optimiser and is therefore a closed implementation and is based on the work on implementations at Yale University, USA. It does not support multiple tuple spaces, providing only a `UTS`[8]. Because of the restriction that the program must be "complete" at compile time it is able to do a large number of optimisations, such as determining which processes can consume tuples, how to optimally store and search for tuples. Therefore, persistent kernels are not normally compared to it as they can not normally be so highly optimised. However, our new kernel implementation can, in many circumstances compete with the performance of SCA C-Linda. In order to demonstrate the advantages of our kernel we consider the implementation of an image processing algorithm, the Hough Transform. The Hough transform is an interesting algorithm for many reasons, but primarily because the parallel implementation of it is not trivial using the standard Linda primitives because it suffers from the multiple read problem. We use it as an example because the amount of communication required is high and the amount of computation is not large. Therefore, the time it takes reflects the communication properties of the underlying system.

5.1 The Hough transform

We now briefly consider the basic Hough transform for detecting straight lines, as an example of a real application where the use of the our kernel is faster than the SCA C-Linda.

The Hough transform maps a binary image pixel, (x, y), in the *coordinate space* to a curve in the *parameter space* (or *Hough space*) which can therefore be represented as a set of coordinates in parameter space. The version of the Hough transform which will be considered here is described by the following equation, where (ρ, θ) pairs represent solutions of the equation given a specific (x, y).

$$x \cos\theta + y \sin\theta = \rho \tag{1}$$

[6]A public domain implementation of Linda also running on top of PVM.

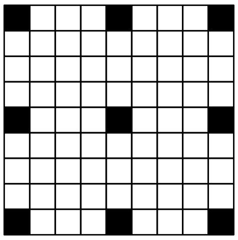
[7]Available from Scientific Computing Associates, One Century Tower, 265 Church Street, New Haven, CT 06510-7010, USA.

[8]This is called the *global tuple space* on C-Linda.

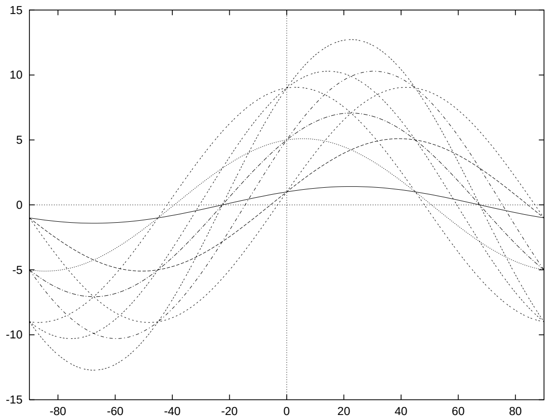
For each pixel (x, y) in the coordinate space the set of (ρ, θ) pairs define a sinusoidal curve in the parameter space. The range of θ is $\pm 90°$ and the range of ρ is D to $-D$ where D is defined as:

$$D = \max(x, y, \frac{\sqrt{2}}{2}x + \frac{\sqrt{2}}{2}y) \tag{2}$$

Given two points in coordinate space, the equation of the line joining them is determined by the point of intersection of their corresponding curves in parameter space — the (ρ, θ) value at the intersection determines an equation of the form of (1). Therefore, in order to detect the straight lines in an image, the parameter space is examined for intersections, and the number of curves that intersect at a particular point is equal to the number of image pixels lying on the line so determined. The Hough transform therefore consists of two stages — the transform from coordinate space to parameter space, and the subsequent processing of parameter space. In this paper only the transformation stage will be considered. Figure 4(b) shows the resultant parameter space for the simple image shown in Figure 4(a). A more detailed description of the Hough transform can be found in [GW87].



(a) Binary image



(b) Parameter space after Hough transform

Figure 4: A simple image and its Parameter space after the Hough transform

It was necessary to use two slightly different approaches to implementing the algorithm, because the SCA C-Linda only has a single tuple space and does not support `collect` or `copy-collect`. However, two implementations of the Hough transform were conceptually similar, with the same number of tuples being processed and a similar number of tuple operations required.

Table 4 shows the execution times for the Hough transform using four Silicon Graphics Indy workstations connected by a 10 Megabit per second Ethernet. As can be seen our kernel with the LTSM enabled is significantly faster than

	SCA C-Linda	Our kernel with LTSM disabled	Our kernel with LTSM enabled
256x256 image 100% pixels set (65536 pixels)	523.20 seconds	670.48 seconds	56.32 seconds

Table 4: Execution time for the parallel Hough transform.

either our kernel with the LTSM disabled and the SCA C-Linda. The speedup achieved against the other versions is presented in Table 5. As can be seen there is a significant speed up against both the versions.

	Our kernel with LTSM enabled speedup against	
	SCA C-Linda	LTSM disabled
256x256 image 100% pixels set	9.3	11.9

Table 5: Speedup of the parallel Hough transform.

Due to the existence of the *multiple* `rd` *problem*[RW96] it is important to compare the time when all the pixels are set, since the time that the implementation using SCA C-Linda takes is independent of the number of pixels set, where as the time that the implementation running on our kernel takes *is* dependent on the number of pixels set. Table 6 shows how our kernel with the LTSM enabled compares over a range of pixel set densities. It quite clearly shows how the set LTSM approach with the use of `copy-collect` to solve the multiple `rd` problem provides significant speed-up.

	SCA C-Linda	Our kernel with LTSM		LTSM enabled speed up against	
		disabled	enabled	SCA C-Linda	disabled LTSM
256x256 image 75% pixels set (49152 pixels)	523.37	559.03	42.61	12.3	13.1
256x256 image 50% pixels set (32768 pixels)	510.92	447.14	26.39	19.4	16.9
256x256 image 25% pixels set (16384 pixels)	514.47	337.91	15.19	33.9	22.2
256x256 image 0% pixels set (0 pixels)	520.59	182.15	7.46	69.8	24.4

Table 6: The advantages of using `copy-collect`.

It should be noted that with many algorithms the SCA C-Linda will perform better than our kernel, simply because it is able to use information derived at compile time for the placement of tuples. An ultimate implementation would blend the two implementations into a single implementation, but it would be a closed implementation.

6 Conclusion

We have presented a novel way of implementing tuple space based systems (Linda) using the locality inherent in the use of multiple tuple spaces which can be calculated cheaply on the fly using implicit information, without making the programmer add any extra information. We have shown that such an approach can lead to significantly faster implementations.

Although the prototype implementation is complete there are many areas where further work is needed. Currently all tuple spaces are classified as either local tuple spaces or remote tuple spaces. This is an effective way of introducing simple locality information to tuple spaces. Work is currently underway aimed at extending this principle to allow a larger classification of tuple spaces, with a view to creating kernels that will allow a large number of computers with a wide geographical distribution to use them.

7 Acknowledgements

The authors wish to thank Andrew Douglas for his helpful comments and suggestions. During this work Antony Rowstron was supported by a CASE studentship from British Aerospace Military Aircraft Division Ltd, and the EPSRC of the UK.

References

[Ass95] Scientific Computing Associates. *Linda: User's guide and reference manual.* Scientific Computing Associates, 1995.

[Bjo92] R. Bjornson. *Linda on distributed memory multiprocessors.* PhD thesis, Yale University, 1992. YALEU/DCS/RR-931.

[BWA94] P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, 1994.

[Car87] N. Carriero. *Implementation of Tuple Space Machines.* PhD thesis, Yale University, 1987. YALEU/DCS/RR-567.

[CG90] N. Carriero and D. Gelernter. *How to write parallel programs: A first course.* MIT Press, 1990.

[DRW95] A. Douglas, A. Rowstron, and A. Wood. ISETL-LINDA: Parallel programming with bags. Technical Report YCS 257, University of York, 1995.

[DWR95] 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.

[Gel89] 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.

[GW87] R. Gonzalez and P. Wintz. *Digital Image Processing*, pages 130–134. Addison Wesley, second edition, 1987.

[Hup90] S.C. Hupfer. Melinda: Linda with multiple tuple spaces. Technical Report YALEU /DCS/RR-766, Yale University, 1990.

[Jen93] K.K. Jensen. *Towards a Multiple Tuple Space Model.* PhD thesis, Aalbrog University, Department of Mathematics and Computer Science, 1993.

[Lei89] J. Leichter. *Shared tuple memories, shared memories, buses and LAN's – Linda implementations across the spectrum of connectivity.* PhD thesis, Yale University, 1989. YALEU/DCS/TR-714.

[NS93] B. Nielsen and T. Sorensen. Implementing Linda with multiple tuple spaces. Technical report, Aalbrog University, Department of Mathematics and Computer Science, 1993.

[NS94] B. Nielsen and T. Slrensen. Distributed programming with multiple tuple space Linda. Technical report, Aalbrog University, Department of Mathematics and Computer Science, 1994.

[RDW95] A. Rowstron, A. Douglas, and A. Wood. A distributed Linda-like kernel for PVM. In J. Dongarra, M. Gengler, B. Tourancheau, and X. Vigouroux, editors, *EuroPVM'95*, pages 107–112. Hermes, 1995.

[RW96] A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 357–367. Springer-Velag, 1996.

[SDGM94] V. Sunderam, J. Dongarra, A. Geist, and R Manchek. The pvm concurrent computing system: Evolution, experiences, and trends. *Parallel Computing*, 20(4):531–547, 1994.