IMPLEMENTING TUPLE SPACE ON TRANSPUTER MESHES

CRAIG RICHARD FAASEN

Degree awarded with distinction 27 June 1991

Research Report submitted to the Faculty of Science, University of the Witwatersrand, Johannesburg, towards a partial fulfilment of the requirements for the degree of Master of Science

Johannesburg 1991

ABSTRACT

This report describes and evaluates an implementation of the Linda tuple space abstraction on Transputer networks. There is evidence that suggests a need for a new programming methodology to support Transputer-based applications, and Linda, as an attractive and elegant alternative to existing methodologies, has great potential for this role. The research focuses on the implementation of a particular tuple space model, intermediate uniform distribution, on Transputer meshes. The objective of the research is to ascertain the extent of the communication overheads inherent in the implementation and hence evaluate the feasibility of the approach. The overheads are measured relative to message passing performance on native Transputer networks, and are shown to be significant. It is concluded that although the specific tuple space model is not ideally suited to Transputer-based systems and the implementation, as it stands, is too inefficient to be of practical use, the approach requires further exploration in order to exhaust its full research potential.

DECLARATION

I declare that this research report is my own, unaided work. It is being submitted in partial fulfillment of the requirements for the degree of Master of Science in the University of the Witwatersrand, Johannesburg. It has not been submitted before for any degree or examination in any other University.

Craig Richard Faasen

14th day of February, 1991

To my parents and two elder brothers

CONTENTS

PREFACE

This report constitutes the research requirement for the degree of Master of Science by course-work (60%) and research report (40%). The aim of the research is to investigate the communication overheads associated with the implementation of the tuple space abstraction defined by the Linda parallel programming language on networks of Transputers. A particular tuple space model, intermediate uniform distribution, has been implemented on a mesh of Transputers, and this system is analyzed and evaluated in the course of the investigation.

The preliminary literature survey for this research was conducted in August – November 1989, and the implementation design started in February 1990. The testing and verification of the implementation was completed in October 1990. As stated previously, the objective of the research is to investigate the overheads inherent in the specific tuple space implementation – it was not the aim to develop a fully-fledged Linda system. The result of this investigation is intended to shed light on the feasibility of a full Linda implementation (using the selected tuple space methodology) on networks of Transputers.

Various sections within this document have appeared elsewhere in a slightly different form. Section 1 (*Introduction*) features excerpts from [Faasen 1989b and 1990a], and section 2 (*The Linda Paradigm*) is based heavily on a technical report that appeared in June 1990 [Faasen 1990d], as are the first two sub-sections within section 3 (*Tuple Space on Distributed-Memory-Systems*). The overview of the Transputer and occam that appears in Appendix 1 has most of its source in [Faasen 1990a].

I extend my thanks and gratitude to my supervisors, Conrad Mueller and Scott Hazelhurst, for their time, interest and enthusiasm, and for their valuable contributions to many fruitful discussions during the course of this research. Thanks to Yale University's Department of Computer Science, notably Nick Carriero, Jerry Leichter, Steven Ericsson Zenith and Venkatesh Krishnaswamy, for answering various questions and providing access to a significant amount of literature. I would also like to thank various members of the Linda Users and North American Transputer Users Groups for their responses to queries posted to the respective bulletin boards. Thanks to Ian Sanders for his comments on the content of this report, and to Derek Nitch for his remarks regarding the introductory section. Finally, I would like to thank to Lauren Bertola for her support and encouragement, and for her comments on aspects of the system's analysis.

With regard to minor points concerning notation, notice that

- the terms *Transputer* and *transputer*, and *Occam* and *occam* appear interchangeably within the relevant literature. This report adopts the notation prevalent in early documentation i.e. *Transputer* and *occam* (the latter is used synonymously with *occam 2*)
- to distinguish between the names of the Linda primitive operations and their English language counterparts, the former appear in a distinctive font – e.g. in and out.
- Tables, Figures and sub-sections that appear in the Appendices are numbered *A.x.y*, where x is the number of the Appendix and y the respective sequence number.

LIST OF TABLES

LIST OF FIGURES

SECTION 1

1.0 INTRODUCTION

Manufacturers of conventional (i.e. sequential) computers are constantly striving to improve the performance of their hardware in a bid to keep up with ever-increasing processing demands. However, as technologies approach the limits imposed by the speed of light, so these efforts yield improvements of successively less significance [INMOS 1989]. The solution to the problem of modelling real-world systems lies in the domain of parallel processing – dividing up a problem among a number of processors and solving the sub-problems in parallel. Not only does this approach increase processing speed, but it provides the means to express realistic solutions to inherently parallel problems. This is an important issue given that most real-world systems involve some degree of concurrency. In a parallel environment, processes can execute and function independently of each other. It is obvious, however, that there must be some means of coordinating the processes in such a way that they can cooperate and interact with each other. There are a wide range of such synchronization and communication mechanisms – e.g. semaphores, conditional critical regions, monitors, remote procedure calls and message passing. These mechanisms are critically reviewed by Faasen [1989b], and are found, in general, to be applicable to specific domains (e.g. semaphores are obvious candidates for shared-memory models, whereas message passing is highly suited to distributed-memory systems). There is, however, a more recent and far more generalisable mechanism than those listed above. This pertains to the Linda programming paradigm, and is known as *generative communication*.

Linda is a paradigm for high-level parallel programming, the basis of which is a global, logically shared *tuple space* (TS). TS is a form of associative memory that is accessible to all processes running within a parallel program. These processes communicate by manipulating the TS – i.e. by inserting and retrieving data objects (*tuples*). Processes therefore have no direct interaction with each other. Instead, process synchronization and communication are achieved via tuple space operations, providing an entirely new, conceptually simple approach to parallel programming. Inherently, a shared-memory architecture appears most suited to supporting Linda efficiently (since tuples within TS are accessible to every process within the system) and there a number of such implementations in existence [Carriero 1987]. However, distributed-memory architectures are not excluded from the sphere of potential Linda target machines. Indeed, these systems are regarded as an important area of application as they are generally cheaper, less complex and more scalable than shared-memory hardware [Ahuja *et al.* 1988].

The Transputer is representative of such distributed-memory architectures, offering supercomputer performance for a fraction of the cost [INMOS 1989]. There is a close relationship between the Transputer and its programming model, occam. This means that problems to be solved using the hardware can be expressed naturally and elegantly (and efficiently) in occam [Pountain 1989]. However, because of the close relationship between the hardware and the software formalism, the style of programming is closely coupled to the specific hardware topology. Consequently, the underlying processor configuration plays a prominent role in algorithm design. There is a steep learning curve associated with the effective and efficient utilization of Transputers (this statement is based largely on personal experience in teaching the use of occam and the Transputer; however, Pountain [1990] and Rabagliati [1990b] both allude to the difficulties imposed by the model of communication).

This suggests a need for a new programming paradigm to support Transputer-based applications. Linda, as an elegant and conceptually simple alternative to existing methodologies, is a strong candidate for this role. Furthermore, Linda programs are portable [Scientific 1989] – they can be run with little or no modification on any architecture on which the tuple space abstraction and communication kernel are resident. Consequently, there is much to be gained by the provision of a Transputer-based target machine.

The implementation of a globally accessible tuple space on Transputers poses a number of problems. As detailed in Appendix 1, the Transputer is a message passing distributed-memory architecture, with processor interconnection via point-to-point synchronous links. Hence, there is the problem of maintaining distributed data over independent local memories, and also having to deal with a potential communications bottleneck. Nevertheless, there are a number of Transputer-based Linda implementations in existence, although, as will be seen later in this report, the approach taken in this research is unique.

The objective of the research is to investigate the communication overheads imposed by a particular tuple space model, the intermediate uniformly distributed scheme, on Transputer meshes, and hence evaluate the feasibility of such an implementation. In support of this investigation, the model has been implemented in occam 2 on meshes of Transputers. This implementation, termed *X-Linda*[1], is the vehicle of the investigation; ultimately, the research is concerned with ascertaining the communication overheads inherent in this system. It should be pointed out that X-Linda is not a fully-fledged Linda implementation. The design and functionality of the model are restricted to meet the needs and requirements of the investigation.

The structure of this report is intended to lead the reader in a systematic fashion into the heart of the research. The document progresses from introductory information on Linda and tuple space methodologies into the design and implementation of X-Linda, and then details the analysis and evaluation of the system. The actual content of the document is briefly overviewed below.

A logical place to commence this report is with Linda itself. A broad overview of Linda is given in section 2, illustrating the concepts of tuples and TS, and describing the primitive operations that may be used to manipulate the TS. The programming methodology is introduced, focussing on the replicated worker model and Linda's characteristic uncoupled style, and the related advantages and benefits offered by Linda are discussed. As indicated earlier, although there is strong support for the implementation of Linda on shared-memory architectures, distributed-memory implementations remain an important area of application. Since this research is concerned with the implementation of tuple space on a distributed-memory system (i.e. a network of Transputers), it is important to be aware of existing tuple space models and methodologies that have been applied to distributed-memories. Section 3 presents various models of tuple space that can be fitted on to distributed-memory, and describes a custom-built distributed-memory system that has been designed specifically to support intermediate uniformly distributed tuple space (the Linda Machine). The design of the Linda Machine is particularly relevant, since it has, to a large extent, influenced the design of X-Linda. The section also examines existing tuple space implementations on Transputers, illustrating how the various tuple space models have been adapted and applied to this specific environment. The uniqueness of the X-Linda approach is also highlighted.

---

[1] X-Linda ? An abbreviation of Xputer-Linda, derived from Transputer-Linda

The X-Linda implementation is introduced in section 4, re-emphasizing the need for an alternative Transputer-based programming methodology as motivation for the research direction in general. Since the research is implementation oriented, it is relevant and important to describe the underlying hardware and development environment, and details pertaining to the specific computing platform (*viz.* a Parsytec SuperCluster) and its development system are given here. The section then focuses on the fundamental design and specification of X-Linda, motivating the choice of tuple space model (intermediate uniform distribution). The intention here is simply to present the overall design and structure of X-Linda (i.e. without entering into low-level implementation techniques). To this end, the remainder of the section gives a non-technical description of the storage of tuples within TS and the format of tuples and templates; the choice of TS primitives provided by the implementation is also justified. Finally, the structure of the individual nodes that comprise the system is described. A more detailed account of the implementation of the TS primitive operations under X-Linda is given in section 5, highlighting various problematic issues and design decisions that had to be addressed in order to realize the successful implementation of these operations. These considerations are important since they have a direct impact on the overall efficiency of the system. Furthermore, the very existence of a substantial design effort can be regarded as the first indication of the unsuitability of the specific tuple space model on Transputers. This section also examines the design and structure of the system at the process level, illustrating the function and interaction of the individual processes that, collectively, comprise X-Linda.

The actual purpose of the research – to ascertain the communication overheads associated with X-Linda – is addressed in section 6. A series of experiments and tests are presented that have been designed to measure the extent of these overheads (relative to message passing performance on native Transputer networks). It is shown in this section that, in general, the communication overheads imposed by the system are significant, and that the overall overhead of the implementation (attributable to the collective effects of communication, TS search, synchronization, process scheduling and set-up) is excessive. The following section, 7, describes and evaluates two algorithms (numerical integration and matrix multiplication) that have been implemented under X-Linda in order to observe the efficiency of real applications running on the system. Although the relevance of this section to the scope of the research may not be immediately obvious, the results presented here complement those detailed in the previous section. Section 6 details specific overheads and inefficiencies and, here, the collective influence of these effects on the overall efficiency of the system are illustrated.

There is a good deal of potential for future research with regard to the X-Linda project, primarily with regard to enhancing the system and its performance. Section 8 discusses a number of such strategies, addressing the needs of, firstly, distributed-memory Linda implementations in general, and, secondly, those specific to X-Linda. It is concluded in section 9 that although the specific tuple space model is not ideally suited to Transputer-based systems, the approach requires further exploration in order to exhaust its full research potential.

The Appendices delve into some enlightening areas that are not considered to be within the direct scope of this research, but which do provide supplementary information that is related to and supportive of various issues covered in the course of this research.

Appendix 1 – For readers unfamiliar with the Transputer and its native language, occam, a non-technical overview is given in Appendix 1 (and, for those who are familiar with the subject area, this Appendix includes a description of the soon-to-be released H1 Transputer).

Appendix 2 – Appendix 2 gives details of the design and results of the various experiments that were devised as base tests against which the communication overheads of the X-Linda implementation are ascertained. The experiments are all concerned with evaluating the rate of data transmission over various Transputer networks and are obviously fundamental to the analysis of X-Linda.

Appendix 3 – The measurement of the CPU utilization on each node within the system also features extensively in the analysis of X-Linda, and Appendix 3 describes how these utilization figures are derived.

Appendix 4 – Appendix 4 illustrates the top-level program structure of X-Linda, highlighting the use of harnesses that permit the system to be either simulated on a single processor or physically distributed over a network of processors.

Appendix 5 – The process-level functionality and interaction of the X-Linda implementation is diagrammatically illustrated in Appendix 5, showing the suitability of the occam programming model in the design of the system.

Appendix 6 – Section 7 of this document deals with the design, implementation and analysis of two specific example programs, and the full code listings are located in Appendix 6. The primary motivation for including these listings is to illustrate the X-Linda programming style. Furthermore, these listings reflect a very rare programming methodology, i.e. the use of Linda primitives embedded in occam 2 programs.

Appendix 7 – The last Appendix constitutes a short discussion on the semantics regarding the order in which tuples are added to tuple space. This issue is briefly touched on in section 5.1.1.1 in the context of the implementation of the out operation under X-Linda, and is expanded upon here simply in the context of an interesting side-issue.

Finally, to conclude this document, a comprehensive list of Linda-related literature is given. This list features more than ninety references, and should be of use to researchers currently involved with Linda and especially to new-comers to the area.

SECTION 2

2.0 THE LINDA PARADIGM

This research is concerned with the implementation of the tuple space abstraction as defined by the Linda parallel programming language. It is necessary, then, to describe the concept of tuple space, and to discuss the operations that may be used to manipulate the tuple space. This section, much of which has its source in [Faasen 1990d], presents a broad overview of the Linda paradigm, concentrating on the following areas :

1. The Tuple Space Model
An overview of Linda which focuses on the concepts of tuples, TS and TS operations.

2. Programming Methodology
The issues of the replicated worker model and Linda's characteristic uncoupled style are discussed. It is shown that the approach differs greatly from, and has several advantages over, conventional parallel programming methodologies.

3. Advantages of Linda
Here, the fundamental motivation for this research is given by means of illustrating the benefits offered by Linda. In effect, this sub-section is answering the (unasked) question "What is the point of a Linda-like implementation ?".

Readers familiar with the concept of tuple space can safely skip out the first of these sub-sections. The other two sections should, it is hoped, maintain a general level of interest, whether or not the idea of tuple space is a new one to the reader. Finally, a short discussion on the Linda paradigm is given, emphasizing the fundamental difference between this approach and existing parallel programming models.

2.1 TUPLE SPACE MODEL

"The abstract computation environment called "tuple space" is the basis of Linda's model of communication." [Gelernter 1985, p. 82]

This section gives an introductory overview of Linda, and then examines the ideas of tuples and tuple space in some detail. Linda is a paradigm for high-level parallel programming that is centred around the concept of a global, logically shared tuple space. The simplest way to think of tuple space is as an unordered "bag" containing pieces of information (tuples). Processes communicate by adding tuples to and retrieving tuples from this globally accessible space. The model is described by Carriero and Gelernter [1989] as one of *generative communication* – communication between processes is achieved through the generation of data tuples and the creation of processes is done by means of generating live tuples. Communication and process creation are therefore closely related. Indeed, one of the Linda objectives is to do away with the distinctions between synchronization, communication and process creation [Gelernter 1989a]. Building a Linda program entails inserting process tuples into the TS, which then generate further process tuples.

Linda is not a complete language in itself. It comprises a small number of primitive operations that provide mechanisms for process coordination and creation within the base language that they are injected into [Gelernter 1988]. Coordination and communication between processes is implicitly achieved through the effects of the Linda primitives executing against the TS. The model of parallel processing is

orthogonal to the base language – Linda deals only with process creation and coordination and is divorced from all computational issues. Consequently, Linda can potentially be embedded in *any* language.

Central themes underlying the Linda model are –

- Distributed data structures – Ahuja *et al.* [1988] define a *distributed data structure* as being "a data structure that is directly accessible to many processes simultaneously". Any tuple in the TS can be accessed by any process in the system – tuples are therefore distributed data structures. Distributed data structures are not supported in the majority of parallel programming languages. Instead, shared data is normally encapsulated inside *manager processes*, which are responsible for making this data accessible to user processes [Ahuja *et al.* 1986].
- Replicated worker processes – identical worker processes are "stamped out" as they are needed. The replicated worker model is discussed in section 2.2.1.
- Spatial and temporal uncoupling – processes have no direct interaction with each other. Uncoupling is addressed in section 2.2.2.

The Linda model is vividly described by Ahuja *et al.* [1988] as "a swarm of tuples, some passive and some active, grows, shrinks, and maintains internal coordination by generating and consuming more tuples".

2.1.1 TUPLES

Tuples can be thought of as being shared pieces of information – they can assume one of two forms :

1. Data tuples

A data tuple is an ordered collection of data objects known as *fields*. A tuple field can either be an *actual* or a *formal* parameter (termed the field's "polarity" [Zenith 1990b]). An actual field contains some physical value – for example "A", "hello", 100, 3.142 or TRUE. A formal parameter specifies a typed variable name, for example char letter, string message, int max, float pi or bool status. Communication between processes is in effect achieved by the transfer of data from actual to formal tuple fields. For clarity, formal parameters are frequently preceded by an interrogation mark – for example, ?int max. The actual data types that data fields can assume is largely dependent on the host environment – records and arrays are, for example quite permissible. Gelernter [1985] formalizes the structure of tuples as $(N,P_2,...,P_N)$, where N is an actual parameter, and $P_2,...,P_N$ are either actual or formal parameters. The first field is referred to as the tuple "name", and is defined as a character string (the specification of the tuple name has not been strictly adhered to in later and current implementations). A tuple comprising two actuals and one formal field might appear as ("hello", ?int i, 3.142). The specification of formal parameters does have another variation – Ahuja *et al.* [1986] state that "the annotation **formal** may precede an already-declared variable to indicate that the programmer intends a formal parameter". For example, ("A", ?int i) is equivalent to ("A", formal i), given that i has been previously declared to be of type integer.

2. Live tuples

Live tuples invoke computation – they are "process tuples under active evaluation" [Ahuja *et al.* 1988]. These tuples, in their simplest form, comprise the name of a process to be executed. Live tuples are discussed in more detail with regard to the eval operation (section 2.1.3).

2.1.2 TUPLE SPACE

Tuple space is the basis of communication in Linda [Gelernter 1985]. The Linda model does not presume the presence of an underlying shared-memory architecture. The TS is however a logically shared object memory whose *semantics* reflect a

physically unordered underlying component [Carriero *et al.* 1986]. TS can be thought of as a logically shared resource or memory space into which tuples can be deposited or, conversely, be withdrawn. Programmers are able to assume that they are using a physically shared-memory, even if none actually exists. It must be emphasized that the TS is a content addressable, associative memory [Ahuja *et al.* 1983] – the mapping from logical tuples to their physical addresses is a function of the Linda virtual machine [Gelernter 1985]. Tuples are not physically addressable – they are selected on the basis of the structure and content of their fields. Locating a tuple is based on a form of pattern matching, and involves the process of *associative look-up* [Gelernter 1988]. TS is accessible to *any* process in the system. All tuples existing within the TS are distributed data structures – they are available to all processes, but bound to none. An extension to the tuple space concept, that of multiple tuple spaces, is discussed by Gelernter [1989a].

2.1.3 Tuple Space Operations

In order to modify a tuple, it is necessary to withdraw it from the TS, modify the data, and then re-insert it back into the TS. Processes wishing to access the TS may do so via four primitive operations :

1. out (tuple)

Adds a tuple to the TS; for example, the statement out("A",1) adds the tuple ("A",1) to TS. Out does not block; the process inserting the tuple continues executing without waiting for the tuple to be accessed by a reader process.

2. in (template)

Withdraws a *matching* tuple from the TS. The actual parameters of the tuple are assigned to the formal parameters of the template. For example, if there is a tuple ("A",1) in the TS, the statement in("A",?int i) will remove the tuple and assign 1 to the integer variable i. Note that the fields pertaining to an in statement need not necessarily include formal parameters – the statement in("A",1) will remove the tuple from the TS, but no data transfer will take place. In is a blocking operation. If a matching tuple is not found, the process issuing the in command suspends until a matching tuple is inserted into the TS. If two or more matching tuples are present in the TS, one of these is chosen arbitrarily.

3. rd (template)

Same as in except that the tuple matching the template is not withdrawn from the TS. A copy of the tuple's fields is simply returned to the process invoking the command.

4. eval (tuple)

Eval is like out, with the exception that the tuple is evaluated after (and not before) it enters the TS. A process is implicitly forked to perform the evaluation. The following example illustrates the effect of the eval primitive, and is adapted from an example given by Zenith [1990b]. Assume that there exists a function F that returns some integer value i. The command eval(F) causes the process F to be placed in the TS and to be executed. The evaluation of F then causes the active tuple (F) to be transformed into the passive tuple (i).

There are two other operations that are predicate variations of in and rd – inp (template) and rdp (template). These operations behave like in / rd but do not block, and return boolean values that indicate whether or not the operation was successful [Carriero 1987]. Notice that inp and rdp are *not* universally accepted primitives :

- it appears that the predicate primitives will not appear in future releases of Linda, but will be replaced by "some form of alternation construct" [Zenith 1990a]
- "... the non-blocking operations are difficult to implement efficiently, hence poor choices as primitive operations in a language intended to support efficient programs; and are, in any case, unnecessary" [Leichter 1989].

Notation for Tuples and Templates
For the sake of clarity, in this document tuples are generally identified by a letter of the Greek alphabet, e.g. α, and matching templates by that letter with a bar above, e.g. $\bar{\alpha}$. This notation is typically used in conjunction with a primitive operation – e.g. out(α).

2.1.4 Matching Tuples

Tuples in TS are accessed by their logical names as opposed to any form of physical address [Ahuja *et al.* 1986]. In order to withdraw a particular tuple from the TS, a process must issue a matching template. A match occurs if the values and field types supplied in the template are identical to those of a tuple present in the TS. The rules for matching a template against a tuple are given formally by Carriero [1987] :

1. the number of arguments must be the same
2. the field types must match
3. data fields (i.e. actual parameters) must be the same
4. formal fields in the tuple can not match formal fields in the template – i.e. formals can only match actuals.

Aside – The Efficiency of Tuple Matching
Intuitively, one might assume that the tuple matching process is a source of great inefficiency in any Linda implementation – i.e. given that a template must be sequentially matched against the contents of what, in essence, is an unordered list of tuples. This is not necessarily the case – ways of speeding up the matching process re discussed in section 8.1.2.

2.2 Linda Programming Methodology

"A parallel program in Linda is a spatially and temporally unordered bag of processes, not a process graph." [Ahuja et al. 1986, p. 26]

Writing parallel programs in Linda is centred around the concept of replicated workers operating on distributed data structures. In this section, the replicated worker model – and a consequence of this approach, *viz.* an uncoupled style of programming – is discussed. It is shown that the Linda approach differs greatly from, and has several advantages over, conventional parallel programming methodologies.

2.2.1 Replicated Worker Model

A central issue in Linda's programming methodology is that of the replicated worker [Ahuja *et al.* 1988]. Sub-processes are created by "stamping" out identical copies of a single process as opposed to creating distinct sub-processes (obviously, this is not the *only* way to write parallel programs – it is, however, a popular and effective approach). Conventional network-style parallelism relies on *partitioning*, where the program is partitioned amongst the processes. Linda employs replication as opposed to partitioning – this has the advantages of :

- scaling transparently – the same program will work the same way, only faster, as more processors are added [Carriero and Gelernter 1988a]
- preventing needless context switching – each processor runs a single process; hence the number of processes only increases with the number of processors [Ahuja *et al.* 1986]
- dynamic load balancing – worker processes look for tasks that need to be computed and tasks are consequently distributed among the available workers at runtime [*ibid.*].

The crux of the Linda programming methodology revolves around a set of distributed data structures that are manipulated by a set of identical worker processes [Ahuja *et al.* 1988].

2.2.2 UNCOUPLED STYLE

A prime objective of Linda is to remove the coupling between parallel processes, and hence reduce program complexity [Ahuja *et al.* 1986, Clayton *et al.* 1990]. It is desired that processes be able to produce and release data without concern for which processes will use that data. Similarly processes should be able to consume data without caring who produced it. A producer's progress should not be restricted by that of a consumer [Ahuja *et al.* 1988]. This feature allows communication patterns to be modified transparently and dynamically at run-time and also encourages asynchronous communication. Furthermore, it supports dynamic load balancing by virtue of the fact that "evaluator processes" can select sub-tasks to compute, based on dynamic supply and demand [*ibid.*]. Linda supports two forms of uncoupling :

1. *spatial uncoupling* – sending processes do not care which processes will receive their information, and vice-versa
2. *temporal uncoupling* – there are no synchronization constraints involved in transmitting data since sending processes do not block.

Linda is consequently "fully distributed in time and space" [Gelernter 1985]. Linda achieves its uncoupled nature via the TS and processes do not interact directly with each other. Uncoupling and dynamic scheduling are accommodated by having all transactions operate on distributed data structures in the TS. The Linda programming model has been described as an "unconnection machine" [Bjornson *et al.* 1987]. Whereas models such as occam and the Connection Machine characteristically bind parallel processes tightly together, Linda processes have no direct impact on each other.

2.3 ADVANTAGES OF LINDA

"... a large class of problems... can exploit the benefits of Linda as an easy and elegant way of parallel programming, offering portability and scalability." [Borrmann and Herdieckerhoff 1989, p. 158]

The philosophy underlying Linda has, at this point in the document, been examined at some length. The focus is now directed towards the advantages offered by the paradigm. The following is an overview of the favourable aspects of Linda that are either shown or claimed in the literature, given as justification for the research direction in general.

1. *Portability* – A Linda program can be run without modification on message passing systems, shared-memory machines and local area networks [Scientific 1989]. The issue of portability is further discussed by Gelernter [1988].
2. *Scalabilty* – The replicated-worker model scales transparently [Ahuja *et al.* 1986]. A program developed to execute on a single processor can be executed (with zero or minimal modification) on a multi-processor system.
3. *Orthogonality* – Since Linda is orthogonal to the base language, it can be embedded in existing languages with minimal effort or modification [Chorus 1989a]. Carriero and Gelernter [1989] describe this feature as Linda's ability to "coexist peacefully with any number of base languages and computing models".
4. *Ease of Use* – Ahuja *et al.* [1986] claim that under Linda, writing a parallel program is no more difficult than writing a sequential one. This is largely due to Linda's characteristic uncoupled style. Williams *et al.* [1989] illustrate the fact that programs can be written without the need to consider
   - the identities of the source and destination processors involved in communication

- the architecture underlying the application
- the synchronization and coordination of processes.

As a result, programmers are able to concentrate on actual algorithms as opposed to implementation details. Another issue worthy of mention is the fact that in order to use Linda, it is not necessary to learn the features and idiosyncrasies of an entirely new language. Apart from the extra TS operations, the host language is unchanged. Programmers familiar with languages like FORTRAN and C can address communication, synchronization and creation of processes in a uniform manner [Kahn and Miller 1989]. It is relevant to note that personal experience gained in the course of this research undoubtedly supports the claim that Linda simplifies parallel programming.

5. *Dynamic Load Balancing* – The replicated-worker model naturally provides distribution of tasks between available processors at run-time [Ahuja *et al.* 1986]. In this model, worker processes look for tasks that need to be computed – hence tasks become distributed among the available workers.
6. *General Development Tool* – Linda is a good general purpose development tool [Gelernter 1988]. Carriero and Gelernter [1988a] argue that
   - Linda can be applied to and subsequently solve "real" problems. This is supported by the wide range of application examples that Linda has been used in
   - Linda solutions are conceptually simple to understand
   - Linda programs exhibit "real" speed-up.

   Linda caters for coarse, medium and fine-grained parallelism [Carriero and Gelernter 1989]. Linda also has applications in the construction of operating systems [Gelernter 1985]. The low level communication kernel of the XTM system discussed in section 3.3.1 is based on Linda. Applications in information management are described in detail by Gelernter [1989b].
7. *Power and Expressiveness* – Linda has greater power, expressiveness, simplicity and elegance than existing models of parallel processing [Gelernter 1985, Carriero and Gelernter 1989].

Finally, in a more philosophic vein, Gelernter [1988] states that "Linda is a practical system; it is also an attractive and evocative thought tool".

2.3.1 Criticisms

Obviously, every programming paradigm must have its faults. Gelernter [1985] discusses some general weaknesses of Linda –

- Linda does not provide as much as other languages. Consequently, it is up to the programmer to implement features such as calling routines and queue management
- the implementation of Linda in a distributed environment is potentially complex
- generative communication could place massive overheads on network communication systems
- Linda offers no TS security. There is no protection mechanism to prevent unauthorized processes accessing and modifying the TS (this point is also addressed by Shapiro [1989]).

Davidson [1989] cites a number of other criticisms :

- the run-time overhead is substantial
- retrieval of information from the TS is a potential bottle-neck
- Linda's characteristic of spatial uncoupling is "potentially dangerous". Davidson claims that programmers should be forced to consider processor synchronization in the interests of "complete and accurate" programming.

A final comment that is worthy of note is by Kahn and Miller [1989]. They state that "Linda's practicality rests upon global compiler optimizations". Obviously, a major task of any Linda compiler is to optimize the tuple matching process and Linda's dependency upon an efficient compiler can be regarded as a drawback.

The purpose of this research is not to attempt to prove or disprove the above claims. It is simply concluded that there appear to be sufficient benefits offered by the model to merit further investigation.

2.4 DISCUSSION

There is currently no single programming model that can offer a complete solution to the complexities and problems that are inherent in the field of parallel programming. Linda does, however, address many of the key issues in this area. Linda is a relatively young paradigm, having first been comprehensively described in 1983. The model is significantly different from existing approaches to parallel programming (other models and approaches to process synchronization and communication are examined by Faasen [1989b]). The programming style associated with Linda is both simple and powerful. The advantages of the uncoupled replicated worker model have been illustrated, and Linda does reduce the burden of writing parallel programs. Linda addresses a very important issue in the debate on whether it is better to develop new parallel languages, or to modify existing sequential ones. Much research is being done on developing entirely new language models (i.e. models that are, in themselves, a complete language), and it is important and necessary that this trend continues. However, for those who argue against the tremendous expense involved in developing, learning and utilizing a completely new paradigm, Linda offers an ideal solution. The implementation of Linda's globally accessible TS on distributed-memory architectures is dealt with in the following section.

SECTION 3

3.0 TUPLE SPACE ON DISTRIBUTED-MEMORY SYSTEMS

"The power of the VLM [Virtual Linda Machine] will be realised only if the communication kernel can be implemented efficiently." [Gelernter 1985, p. 103]

The fact that every tuple within the TS is a distributed data structure (i.e. is accessible to any node in the system) is strong support for the implementation of Linda on a shared-memory architecture. Indeed, Ahuja *et al.* [1988] state that "Linda is, of course, inherently a shared-memory model". However, distributed-memory systems are cheaper, less complex and more scalable than shared-memory machines [*ibid.*]. Linda's logically shared-memory is sufficiently coarse (i.e. the units of storage are tuples, not bytes) to be implemented on distributed-memory [Ahuja *et al.* 1986]. Furthermore, shared-memory is not suited to local area networks, which are an important area of application. Since this research is based on the implementation of tuple space on Transputers (i.e. on a distributed-memory architecture), it is important to illustrate the various models of tuple space that can be fitted onto distributed-memory. The purpose of this section is three-fold – firstly, to present general approaches to implementing tuple space on distributed-memory systems; secondly, to describe an approach featuring custom-built distributed hardware to support tuple space (the Linda Machine); and, lastly, to illustrate specific approaches taken in the implementation of Linda on Transputer-based architectures. The content and motivation for these areas of investigation are briefly outlined below.

1. General Approaches

Two notable approaches to implementing tuple space on distributed memories are *distributed hashing* and *uniform distribution*. These methodologies are introduced, and the emphasis of the investigation is directed towards a scheme known as *intermediate uniform distribution*. It is shown that intermediate uniform distribution is arguably the most efficient and elegant of the approaches; hence the motivation for implementing X-Linda under this scheme. Intermediate uniform distribution is covered in some detail since it is the foundation of X-Linda. A thorough understanding of this methodology is a prerequisite for understanding, evaluating and appreciating the X-Linda approach.

2. The Linda Machine

The Linda Machine is an example of the implementation of tuple space on dedicated hardware. A description of this system is particularly relevant, since

- it, like the X-Linda system, features an intermediate uniformly distributed tuple space on a mesh of processors
- the design of this system greatly influenced that of X-Linda.

3. Transputer-Based Implementations

A range of existing Transputer-based Linda implementations are presented in order to illustrate the variety of approaches taken, and to illustrate the uniqueness of X-Linda. *None* of the implementations reviewed utilize the intermediate uniformly distributed methodology. Some comments on the relative strengths and weaknesses of the approaches are given, and reasons for the absence of the intermediate uniformly distributed scheme are discussed.

3.1 General Approaches

Implementing tuple space on a machine that lacks physically shared-memory poses a number of implementation problems [Gelernter 1985, Carriero 1987]. Various approaches to implementing a logically shared tuple space across distributed memories have been proposed – the most significant of these are :

1. Hashing

Carriero *et al.* [1986] describe a scheme based on *distributed hash tables*. Tuples are stored on unique nodes in the system, where the identity of the node responsible for any given tuple is hashed from the tuple's fields. Hashing is economical [Bjornson *et al.* 1987] since tuples do not need to be replicated, and there is never a need to broadcast data over the network. Hashing has been successfully implemented on a number of systems. The scheme may not, however, be optimal for message passing systems since, in the worst cases, the distances that a message must travel between nodes will cause a severe communication overhead. Hash-based approaches are described in more detail by Carriero [1987] and Ahuja *et al.* [1988].

2. Uniform distribution

The uniform distribution of tuples across the TS is described below.

3.1.1 Uniform Distribution

Gelernter [1985] and Ahuja *et al.* [1988] describe a technique that attempts to distribute the TS evenly over the nodes in the system. Each node has an "out-set" and an "in-set". Tuples to be inserted into the TS are sent to the nodes that make up the out-set, and tuple requests (templates) are broadcast to those in the in-set. Uniform distribution covers a wide range of implementation possibilities.

Global TS – at the one extreme of the scheme, tuples injected into the TS are passed to and subsequently stored in *every* node in the system. An in or rd operation would therefore necessitate a requesting node to perform a search of its own tuple memory. Note that tuple deletion requires a network-wide operation – Fleckenstein and Hemmendinger [1989] propose a methodology to address this issue : the node that outs a tuple is that tuple's owner; nodes wishing to withdraw the tuple must get permission from the owner. A Linda kernel for the AT & T Bell Labs S/Net has been implemented using the global TS approach [Carriero 1987, Carriero and Gelernter 1986] with a hashing scheme for computing physical tuple addresses within a node's tuple memory. The concept of Global TS is illustrated in Figure 3.1, which shows storage of tuples (denoted α) and templates (denoted $\overline{\alpha}$) over the nodes in a system.

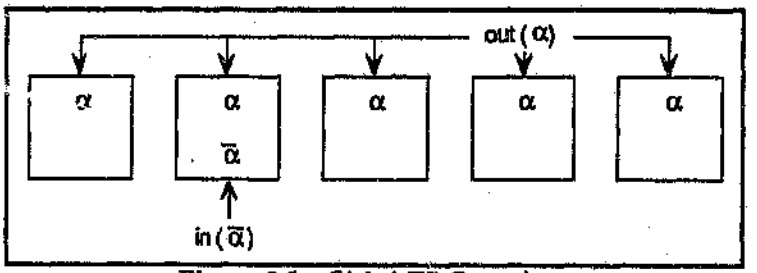


Figure 3.1 : Global TS Operations

Local TS – at the other extreme, tuples could be stored only in the memories of the nodes performing an out operation. In this case, a global network search would be necessary in order to locate a specific tuple, as illustrated in Figure 3.2. This approach simplifies the maintenance of TS consistency.

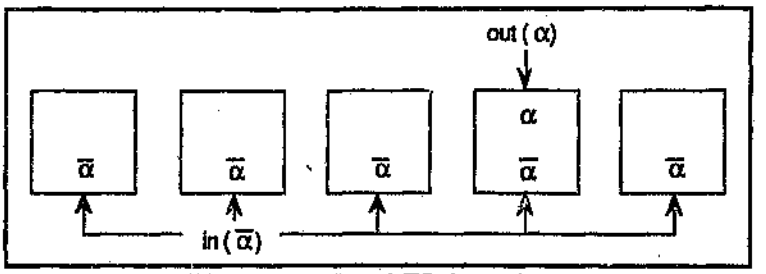


Figure 3.2 : Local TS Operations

3.1.1.1 Intermediate Uniform Distribution

This scheme is mid-way between the local and global strategies. Assuming that there are *k* nodes in the system, both the in-set and out-set comprise √k nodes and must intersect. Since the number of nodes involved in the insertion and withdrawal of a tuple is only 2√k (√k for an out and √k for an in), "the intermediate scheme is provably optimal even if we consider the entire spectrum of uniform distribution schemes." [Ahuja *et al.* 1988]. An interesting description of how TS operations are implemented under the intermediate uniformly distributed scheme is given using the concepts of *tuple beams* and *inverse beams* [*ibid.*] – also known as *out-threads* and *in-threads* [Gelernter 1985]. Tuples in TS (injected using an out statement) are represented as a tuple "beam" across a row of nodes (i.e. the out-set). When a node requires to locate a tuple (executing an in command), it "flashes an inverse beam along a column". If two matching beams intersect, the desired tuple is returned to the requesting node, and both beams disappear. This is depicted in Figure 3.3.

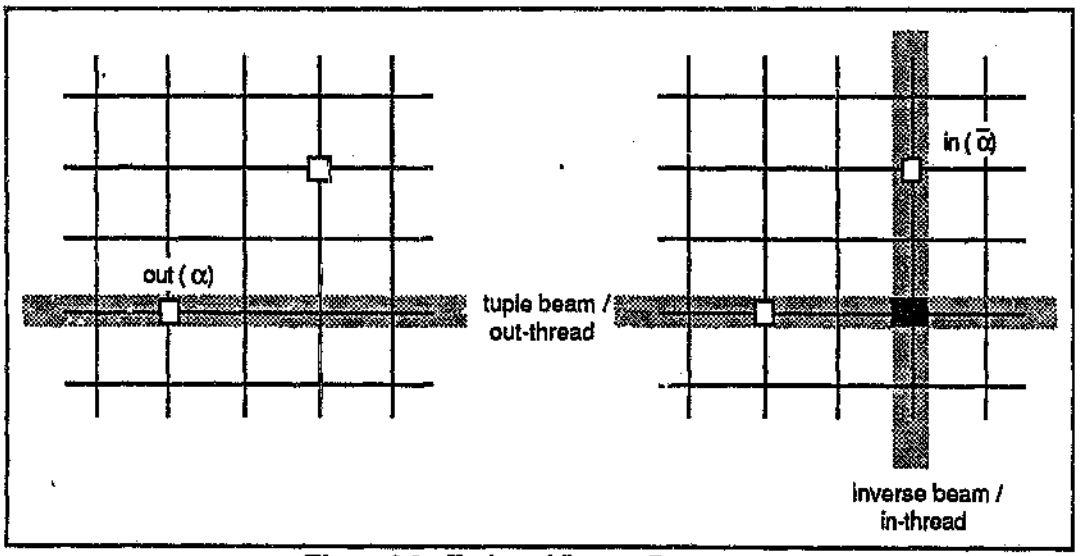


Figure 3.3 : Tuple and Inverse Beams

The uniform distribution scheme suggests itself as an effective way of sharing the communication load among the nodes in the system [Ahuja *et al.* 1988]. Furthermore, it may be generalisable to bigger systems and is an extremely effective way of implementing tuple space over distributed-memory systems [Bjornson *et al.* 1987]. Finally, a "well designed" uniformly distributed implementation has the advantages of [Ahuja *et al.* 1988] :

- making tuple reads "cheap and convenient"
- supporting parallelism among individual TS operations
- being well-suited to hardware and low-level protocol support.

Hardware Topology

Intermediate uniform distribution lends itself to a particular topology – a W^D wrap-around hypercube [Gelernter 1985] or, stated more simply, a $\sqrt{k}$ by $\sqrt{k}$ grid of processors [Ahuja *et al.* 1988]. This topology comprises a grid of processors where

- the number of rows is the same as the number of columns, i.e. the grid is square
- the first node in any row/column is linked to the last node in that row/column.

In this document, the term *mesh* is used synonymously with the above description. Using this topology, a node's out-set is defined as the nodes on its row of the grid and the in-set as the nodes that make up a column. This is illustrated in Figure 3.4, with reference to a 4x4 mesh.

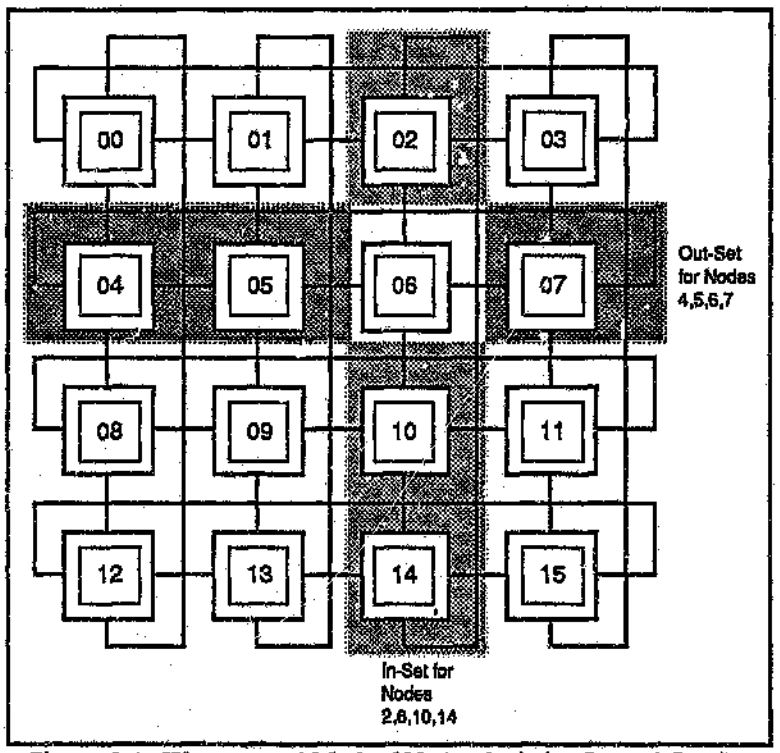


Figure 3.4 : Wrap-around Mesh of Nodes depicting In- and Out-Set

Each node maintains a store of tuples and tuple requests. An out operation causes a tuple to be sent to all nodes along a row and an in or rd operation causes a template to be sent to the nodes in a column. A tuple match occurs when a particular node is in possession of both a tuple and a matching template. The tuple is sent to the requesting node and, in the case of an in operation, the tuple is deleted from the TS. Notice that if broadcast buses are used for network interconnection, transmitting a tuple to a node's respective in- or out-set can be accomplished in a single operation.

Maintaining TS Consistency

The issue of maintaining tuple space consistency under the intermediate uniformly distributed scheme is addressed in the next section in the context of the Linda Machine (section 3.2.2)

3.2 THE LINDA MACHINE

The Linda Machine is an attempt to implement TS and TS operations on a custom-built distributed-memory architecture. It will be seen in due course that the design of the Linda Machine has greatly influenced the design of the X-Linda system. Consequently, althougn the Linda Machine is itself not within the scope of this research, it will be briefly overviewed here. The design of the Linda Machine was first comprehensively detailed by Ahuja *et al.* [1988] who described the system as "a parallel computer that has been designed to support the Linda parallel programming environment in hardware". A prototype version of the Machine has been built and, since fairly recently, been functional [V. Krishnaswamy, personal communication, Nov. 1990]. The objectives of the Linda Machine are to provide a system that is

- easily programmable
- efficient – although the emphasis of the project concerns ease of programming, it is shown that existing Linda implementations (i.e. on conventional hardware) exhibit massive communication overheads; there is a need to utilize a dedicated communications processor.

The system comprises a network of *Linda nodes*. These nodes in turn consist of a computation processor and a Linda co-processor (termed the *Linda Engine*) that is responsible for TS communication and management. TS in the Linda Machine is implemented under the intermediate uniformly distributed scheme on a mesh of Linda nodes. The nodes are connected via broadcast buses – as a result, tuples can be sent to their respective in- and out-sets in a single operation.

3.2.1 THE LINDA ENGINE

The Linda node, as stated above, comprises a computation processor and a Linda engine – obviously, this section is only concerned with the structure and operation of the Linda engine. The Linda engine comprises a tuple memory, a working store, an operations controller (consisting of an IN- and OUT-processor) and interfaces to the buses and computation processor. This is illustrated in Figure 3.5.

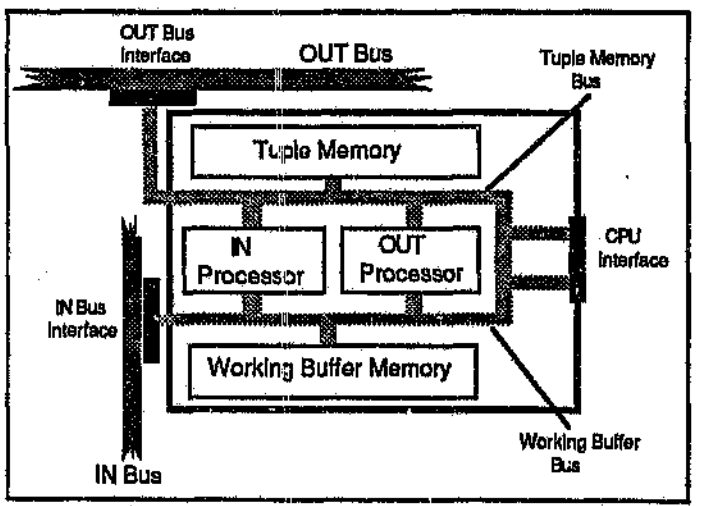


Figure 3.5 : Structure of the Linda Engine

The tuple memory, working store and operations controller are very briefly described below.

Tuple Memory

The tuple memory must not only store the data fields of the tuples, but also, for the purpose of tuple matching, information pertaining to the size and type of these fields. The representation must therefore treat tuples as self-contained data objects.

Tuples are linked together in "tuple groups" -- i.e. groups of tuples having similar field characteristics.

Working Store

The working store holds two queues -- the *in-request*, which stores pending in requests (either from the computation processor or from other nodes), and the *out-request* which stores out requests from the computation processor.

Operations Controller

The operations controller comprises the IN-processor and the OUT-processor which operate on the in-request and out-request queues respectively.

Operation

A TS operation is invoked and serviced as follows :

- the operation is initiated by the computation processor
- in the case of an out request, the operations controller contends for the out-bus and writes the tuple to *identical memory locations* within the nodes in the out-set (the X-Linda system also employs this strategy -- refer section 4.3.1.1)
- in the case of an in request, the operations controller broadcasts the request to all nodes in the in-set (these nodes store the request in their in-queues) and matches the template against the tuples in its tuple memory. If any node in the in-set (including the one invoking the request) resolves a match, it must send the tuple field information to the requesting node and delete the tuple (the protocols for maintaining TS consistency are discussed below).

3.2.2 Maintaining TS Consistency

In any Linda system, the maintenance of TS consistency is a key issue [Gelernter 1985], and X-Linda is no exception. This aspect is fundamental to the design of the X-Linda system, and, without doubt, the majority of the implementation effort (i.e. design, verification and modification) was dedicated to this problem. With respect to an intermediate uniformly distributed TS, the biggest problem is with regard to the in operation -- if two nodes issue identical tuple requests, only one must succeed in retrieving the data and deleting the tuple. Network protocols must be implemented in order to make certain that TS operations are carried out correctly and to ensure TS consistency. Ahuja *et al.* [1988] discuss the network protocols that are proposed in the design of the Linda Machine, which employs broadcast buses for node interconnection. The protocols are based on *distributed arbitration* and although they have been formalized with broadcast buses in mind, the concepts are both important and generalisable. Specifically, the protocols address the following problems :

1. Inserting Tuples into TS

Two or more nodes in the same out-set attempting to add a tuple to TS simultaneously causes a problem since tuples are written to *identical* memory addresses of all nodes in the out-set. Nodes wishing to install tuples must contend for the out-bus -- the successful node then writes the tuple to the other nodes.

2. Withdrawing Tuples from TS (a)

If more than one node in the *same* in-set simultaneously issue an in request for the same tuple, only one must succeed. Nodes wishing to withdraw tuples must contend for the in-bus in order to retrieve a tuple, and then contend for the out-bus to delete it.

3. Withdrawing Tuples from TS (b)

Two or more nodes in *different* in-sets simultaneously wishing to resolve a match on the same tuple is slightly less complicated than the above situation. It is necessary for a node to win ownership of the out-bus before the tuple can be retrieved and subsequently deleted.

The Linda Machine project is actually the topic of a Yale University Ph.D thesis, and a 16-node prototype is currently in existence [V. Krishnaswamy, personal communication, Nov. 1990]. When the Machine was initially proposed, it was expected that performance would far outstrip existing implementations since

- communications and TS management are handled by a dedicated processor
- TS search operations are microcoded
- the intermediate uniformly distributed scheme reduces the length of data broadcasts.

It was also indicated that the Machine would scale far more efficiently than existing systems. Actual performance figures tend to support these expectations. A 12-node implementation exhibited performance "comparable to current shared-memory implementations and vastly superior to distributed-memory versions of Linda"[1] [N. Carriero, personal communication, Nov. 1990]. Furthermore, the results indicate that this performance will scale comfortably to several hundred nodes. As will be seen in the next section (regarding Transputer-based Linda implementations), the design of custom-built hardware specifically to support the Linda paradigm appears to hold promise as an effective approach to implementing the model on distributed memories.

3.3 Transputer-Based Linda Implementations

There are a variety of Linda systems that have been implemented on Transputer-based computing platforms which appear both in the commercial market and in academic research environments. These systems are overviewed in this section to illustrate the different approaches taken in implementing tuple space on Transputers, and, where possible, comment is given on the merits and defects of the approaches. The following implementations are addressed :

1. a system featuring specialized hardware and software support for Linda
2. a Linda sub-system implemented on a ring of Transputers
3. two implementations based on a distributed hashing methodology
4. a system built on top of the Helios operating system.

Notice that the X-Linda approach (i.e. intermediate uniformly distributed tuple space on a mesh configuration) is not included in the above list. This absence is addressed in the conclusion to this section in a brief discussion on the subject of Linda on Transputers.

3.3.1 Specialized Hardware and System Services

Cogent Research Inc. have developed a commercially available Linda system, the XTM. The XTM features a specialized Transputer-based architecture, and supports a Linda-oriented operating system. This approach is important since it is an example of the trend towards the use of specialized hard- and software support in order to implement Linda efficiently. The XTM is described in more detail below.

3.3.1.1 Hardware

The entry-level system (the XTM Workstation) comprises two IMS T800 Transputers, each of which has access to 4 MBytes of RAM. The "backbone" of the system is the XTM Resource Server [G. Brand, personal communication, Nov. 1989], consisting of a 16-slot backplane, a 32-bit communications bus and an intelligent crossbar switch. XTM Compute Cards (also comprising two IMS T800 Transputers) can be inserted into available slots in order to provide additional processing power. A single resource server is capable of linking up to 15 workstations (i.e. 30 processors) together. The layout of the XTM Resource Server [Hayes 1988] is depicted in Figure 3.6.

[1] Extracted from V. Krishnaswamy's thesis defence announcement

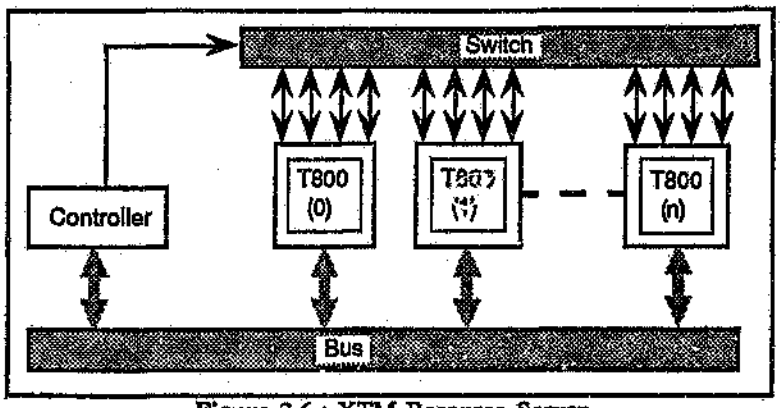


Figure 3.6 : XTM Resource Server

All the Transputers in the system share the communications bus, and, in addition, the 4 links of each processor are connected to the crossbar switch. The switching system provides dynamic system reconfiguration. Typically, a processor requiring to transmit information will send a request through the bus to the switch controller. The controller then establishes a direct connection between the two processors, allowing data transfer to take place without affecting the communication of any other processor. The specifications of the XTM Workstation, Resource Server and Compute Card are detailed in Cogent [1989].

3.3.1.2 Operating System

The XTM operating system, QIX, is a distributed system, and provides multiple tuple spaces, system level programming and dynamic reconfiguration. Each processor runs a small kernel that handles memory management, process creation, communication and synchronization. QIX is implemented on top of Kernel Linda, a low-level communications backbone, as shown in Figure 3.7.

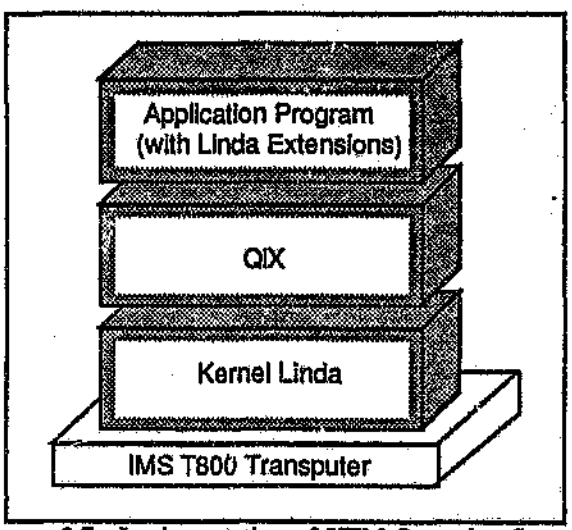


Figure 3.7 : Implementation of XTM Operating System

Kernel Linda and QIX are described in more detail below.

Kernel Linda

Kernel Linda [Leler 1990, Cogent 1990] provides a low-level inter-process communication mechanism, and is designed for system level programming. It is a version of Linda that provides a underlying communication structure on top of which

general Linda implementations can be built. Kernel Linda is a completely general system. It was designed neither for implementation on a specific type of hardware, nor to support a particular language. Although tuples comprise only a single key field, the system does provide extensions to Linda –

- multiple tuple spaces (a tuple field may itself be a tuple space)
- a set of language independent data types (since it does not only provide support for a specific language class).

Implementation of Tuple Space

Tuple space is implemented using a hashing approach. A tuple space name is referred to as a *dictionary*, and tuples within dictionaries comprise a single key / value field. Notice that a value can itself be a structured data type (e.g. an array) and can, in effect, comprise many values. Associated with each dictionary is a distributed pointer (or "locator"), that specifies the identity of the processor responsible for that object. Linda Kernel primitives and operations are described in detail in Cogent [1990].

QIX

QIX [Leler 1990] is a parallel, server-based operating system designed for Transputer-based systems. A small replicated kernel is implemented on each processor. QIX is built on top of Kernel Linda, and provides the lowest level system facilities : memory management, process creation and synchronization, device drivers and communication (through Kernel Linda). Higher level operating system functions are provided by servers that share resources via Kernel Linda. QIX has the advantages of

- providing a UNIX compatible environment
- providing Linda extensions
- supporting familiar languages.

The overheads associated with the implementation are due to the hashing function, TS searches and semaphore locking. Leler points out that this overhead is not excessive (about 10% for "reasonable" amounts of data).

Comment

This approach shows a great deal of promise. It is quite obvious that much effort has gone into the specification, design and creation of the product, and the benefits of the system are apparent. As stated earlier in this section, it is maintained that a "specialized" approach such as this is important, perhaps even necessary, for the efficient implementation of Linda on Transputer-based architectures.

3.3.2 Ring-Based Linda Subsystem

Zenith [1990b] describes a Linda subsystem implemented on a ring of Transputers. The philosophy underlying the design is that Transputers are cheap and easily available. Consequently, as long as the processors are capable of achieving their designated requirements, the actual amount of processor utilization is immaterial. Like the XTM system described above, this approach, to some extent, is based on a specialized architecture – the system comprises two types of *dedicated* processing nodes :

- Tuple space machines (TSMs) which make up the Linda subsystem
- Computation nodes (CNs) which, as their name implies, are responsible for handling a program's computational requirements.

Two CNs are attached to each TSM which are connected in a ring configuration. The prototype described features 5 TSMs, 10 CNs and one processor dedicated to providing monitoring functions. The system configuration is shown in Figure 3.8 (extracted from [Zenith 1990b]).

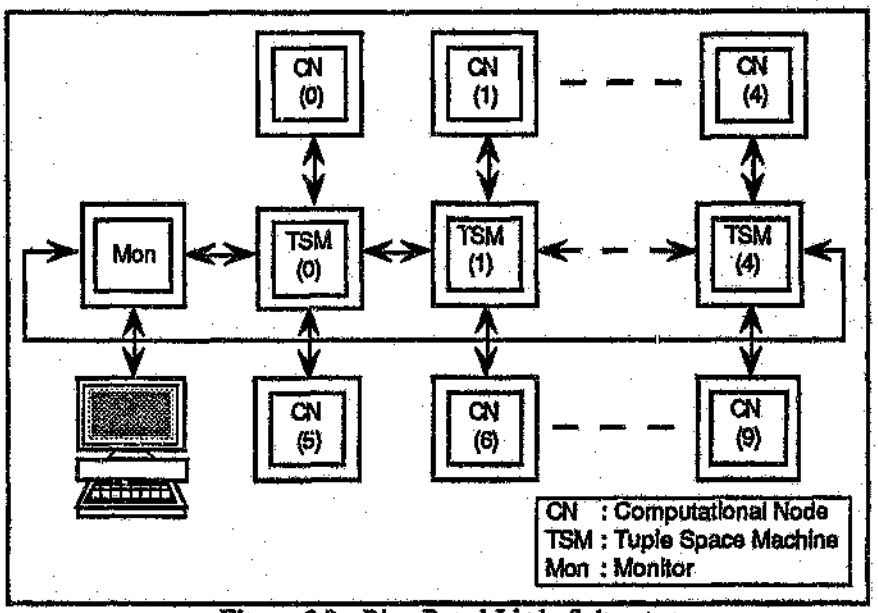


Figure 3.8 : Ring-Based Linda Subsystem

The tuple space is distributed evenly over the Linda subsystem by means of distributed hash tables (distributed hashing is discussed below in section 3.3.3). This, in effect, means that the tuple space is divided into distinct subsets on the basis of tuple structure (i.e. the number, type, order and polarity of fields), and these subsets are distributed among the TSMs.

Comment

The system described above is of interest since it

1. like the XTM discussed in section 3.3.1 features dedicated hardware (although to a lesser extent)
2. utilizes the popular distributed hashing methodology.

The implementation did not advance past the prototype phase [S. Zenith, personal communication, Nov. 1990], which is regrettable since the approach appears to hold promise.

3.3.3 Distributed Hashing

The scheme whereby tuples are stored on unique nodes in the system and the identity of the node responsible for any given tuple is hashed from that tuple's fields features prominently in Linda-related literature. This approach appears to be fairly popular in general (i.e. not only with respect to Transputer-based implementations). In addition to Zenith's approach described above, distributed hashing is also used in the following implementations :

1. The Chorus ComputeServer [Chorus 1989b, Williams, *et al.* 1989] is a commercially available multi-user Linda-oriented implementation that features a distributed hashing scheme. A cluster of 1 to 16 IMS T800 Transputers is attached to an Apple Macintosh network, and access to the computing cluster by users on the network is provided by a dedicated 32-bit I/O processor. Unfortunately, performance figures for this system are not available.
2. Tonsing [1989] describes another hash-based approach, implemented in 3L Parallel C. Tuples and templates within each node's tuple memory are stored in linked lists. These lists are accessed via arrays of pointers which are indexed by a function of the hashing algorithm. Consequently, the hash function, in addition to returning a specific processor identity, also specifies the location in the processor's

local memory where the tuple is stored. The eval primitive is implemented using the normal task-creation facilities provided by Parallel C.

Comment
Specific details (especially with respect to performance figures) are not available for the above implementations. All that can be said at this point is that distributed hashing is a well-understood, tried and tested approach – and has been utilized in non-Transputer-based distributed implementations. It does appear, then, to be a fairly "safe" approach. However, it is suspected that, with regard to Transputer-based implementations, there is much potential for a communications bottle-neck (for example, if many nodes simultaneously attempt to retrieve one specific tuple).

3.3.4 HELIOS-BASED IMPLEMENTATION

Wentworth [1990] and Clayton *et al.* [1990] describe a Linda system implemented on a network of Transputers running under the Helios [King and Powell 1990] operating system. The system, known as *Rhoda*, employs a centralized TS approach where TS is implemented on a dedicated server, access to which is provided transparently by Helios. Although the TS is centralized, it is partitioned into sub-groups of related tuples in an attempt to reduce the overhead of tuple matching (it is pertinent at this point to emphasize the fact that X-Linda does not employ this technique – as described in sections 4.3.1.2 and 6.7, the matching process is implemented simply by means of a linear TS search). A pre-processor has been developed to analyze TS usage in a program in order to

- sub-divide the TS into disjoint sub-sets
- give an indication of where in the network the TS groups and components of the application program should be physically located.

Details of the techniques employed in the analysis are given by Wells [1990] and Clayton *et al.* [1990]. It is indicated that future research will be aimed at decentralizing the TS and distributing it over a number of processors. Rhoda can be implemented on any Transputer network configuration. The system has been tested on relatively small networks (up to 16 nodes), and the results in terms of achieved speed-up for certain applications have been most promising.

Comment
Intuitively, one would feel that a centralized approach is not a good idea for Transputer-based systems (because of the potential for communication bottle-necks). Furthermore, one might expect that the overheads imposed by Helios would have some effect on the efficiency of the implementation. However, as indicated above, the system has yielded impressive results; it is expected that the partitioning of TS into sub-groups is a major factor contributing to the overall efficiency of the system. It will be of great interest to see what effect the proposed decentralizing of tuple space has on the performance of the system.

3.3.5 DISCUSSION

It has been seen in this section that there are a number of Transputer-based Linda systems in existence, built around a variety of tuple space models. There is also evidence of other related work :

- Topologix Inc. have developed Transputer-based add-on boards for Sun workstations to support the Linda model [Gelernter 1988]
- it is indicated that collaboration between Yale University and INMOS Ltd. has been proposed in connection with a Transputer-based Linda project [N. Carriero, personal communication, Oct. 1989]
- it is reported [various personal communications] that research in the area is being / has been conducted at Edinburgh University, The University of Minnesota and Cornell University.

Perhaps the most promising of these approaches are those that feature specialized hardware and software support, although one can obviously not disregard the successes of the other implementations (for example, Rhoda). It is not easy to conclusively state that a specific approach is superior to any other. Both Linda and the Transputer are relatively young entrants to the field of parallel processing, and there is a clear need for further investigative research in this regard. The X-Linda approach (i.e. intermediate uniformly distributed tuple space on a mesh configuration) is conspicuous by its absence from the approaches reviewed here, although a similar approach has been recently proposed at the University of Copenhagen [H. Kristensen, personal communication, August 1990]. The fact that the intermediate uniformly distributed approach has not been pursued is probably due to a reluctance to implement in- and out-sets via the relatively slow point-to-point links of the Transputer. Nevertheless, this illustrates the uniqueness and originality of the X-Linda approach, the design and structure of which is addressed in the next section.

SECTION 4

4.0 THE X-LINDA APPROACH

It is worthwhile at this point to re-examine the purpose of this research. The aim of the research is to investigate the communication overheads associated with the implementation of a particular tuple space methodology on networks of Transputers. The implementation, referred to as *X-Linda*, is the vehicle through which the investigation is conducted – this system is analyzed and evaluated in order to ascertain the extent of these overheads. This section introduces the X-Linda implementation – i.e. an intermediate uniformly distributed tuple space implemented in occam 2 on a mesh of Transputers. The motivation, objective, implementation environment and fundamental design of X-Linda are covered as follows :

1. The Need for a New Programming Paradigm

The motivation for the research is presented in the context of the need for a new programming methodology for Transputers, and justification for choosing the Linda approach is given.

2. Implementation Environment

The implementation environment is described with reference to the physical computing platform and the software development system.

3. X-Linda Design

Finally, the basic or fundamental design of X-Linda is introduced by describing the tuple space model and corresponding hardware configuration. It should be kept in mind that X-Linda is not a fully-fledged Linda system. As discussed in this section, the range of TS primitives and the structure of the tuples themselves are restricted – i.e. limited to the needs of this research. Finally, the structure of the individual nodes in the system is overviewed. Notice that this sub-section does not delve deeply into implementation details – the implementation of the TS operations is covered in section 5.1, and the low-level specification of the system is detailed in section 5.2.

4.1 THE NEED FOR A NEW PARADIGM

"As soon as you want to distribute the processes onto different transputers for real parallelism, though, you find you are limited because the chip has only four communication links. This constraint severely cramps the way you can write programs ..." [Pountain 1990, p. 3]

"One limitation of existing Transputer networks is the need to match algorithms to the interconnectivity in a specific machine. This means the software is not readily ported to other machines with different link topologies." [Rabagliati 1990b]

Since their inception in the mid-1980s, the use of Transputers has steadily become more widespread in industrial and academic spheres. Similarly, occam enjoys high regard as an effective vehicle for expressing problems on Transputers (Pountain [1989] describes occam as "a safe, elegant, and efficient way to program transputer networks"). As discussed in Appendix 1, occam supports the process model of concurrency, and the software formalism is closely coupled to the physical architecture of the Transputer (INMOS [1988a] refer to the "architectural relationship" between the programming model and the hardware). This means that problems to be solved using the hardware can be simply and naturally expressed in occam. It also means, however, that the programmer must have a clear idea of the architecture underlying a specific application. The hardware configuration plays a prominent role in algorithm design [Rabagliati 1990b]. Not only is the programmer restricted by

the physical number of communication links [Pountain 1990], there is the added burden of having to explicitly identify source and destination processors involved in communication, and the responsibility of processor synchronization and coordination. Furthermore, algorithms designed for specific hardware topologies are not easily portable to different configurations [Rabagliati 1990b]. The occam model is based on that of CSP [Hoare 1978]; i.e. process communication occurs via synchronous message-passing. Consequently, the sending process is blocked until the receiver is ready to accept the message. This conflicts with the notion of spatial uncoupling addressed in section 2.2.2 – i.e. that a producer's progress should not be restricted by that of a consumer. Bal *et al.* [1989] state that the synchronous model of communication has a "major impact" on the style of programming. Finally, Bjornson *et al.* [1987] criticize the tight binding of parallel processes within the occam model.

It is felt that there is a need to tear down the process / processor coupling inherent in the style of programming, and unburden the programmer from the restrictions imposed by the model of communication – i.e. a need for a programming methodology that will complement the power and availability of Transputers. This, of course, is where Linda comes in. It may be argued that there is no need to specifically use occam. There are a variety of other languages that can be integrated into the development environment, and there are a number of operating systems running on Transputers that effectively "hide" the underlying hardware configuration from the user. Nonetheless, it is maintained that there is a need for a programming methodology that

- is conceptually simple
- is portable
- from the programmer's point of view, is topology independent
- does not obscure the raw processing power of the Transputer.

It has been illustrated in previous sections that Linda offers an attractive alternative to existing models of concurrency. The objective of this research, then, is to investigate the feasibility of implementing a particular tuple space model on Transputers. In ascertaining this feasibility, the focus is directed at a specific aspect of the implementation – analyzing the communication overheads of the model relative to native Transputer networks.

4.2 Implementation Environment

This research is *implementation* oriented – it has, as its foundation, an implementation of tuple space on a Transputer-based system. Consequently, it is both relevant and important to describe the underlying hardware and development environment.

4.2.1 Computing Platform

The computing platform (*viz.* a Parsytec SuperCluster) and development environment are described briefly below. Some detail is also dedicated to illustrating the use of the host Transputer, as this processor is included in an out-set of the X-Linda mesh. The SuperCluster series [Parsytec 1989a] comprises a range of multi-user Transputer-based systems that are reconfigurable and expandable. Inter-Transputer connectivity is achieved via a *Network Configuration Unit* (NCU), an electronically configurable unit based on the IMS C004 programmable crossbar switch. Although communicating through a C004 does incur some overhead, the delay is negligible. The specific SuperCluster model used in this research features 16 T800[1] Transputers, each of which has access to 1 Mbyte external RAM. Notice

[1] IMS T800-G20S – 20 MHz clock speed, 50 ns cycle time

that the link speeds of the processors are set at 10 Mbits/second[2]. The interconnection of the processors via the NCU is illustrated in Figure 4.1.

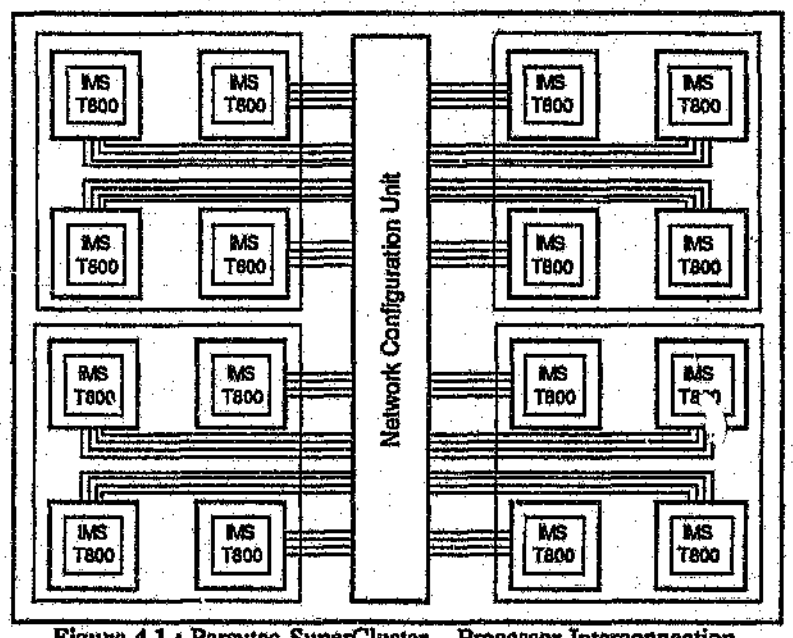


Figure 4.1 : Parsytec SuperCluster – Processor Interconnection

4.2.2 The Host Transputer

A stand-alone Transputer, termed the *host* Transputer is used to develop and compile the programs that will be ultimately loaded on and executed by the processors in the SuperCluster. This Transputer resides on a board that typically has several Mbytes of RAM, and the board (for this specific implementation environment) is plugged into a conventional AT-type machine. Link 0 of this Transputer is connected to the AT's keyboard and screen, and one of the other links (known as the *primary* link) is connected to a Transputer within the SuperCluster. It is possible to connect another of the host's links (the *secondary* link) to another processor in the SuperCluster (a maximum of 2 links from the host into the SuperCluster are permitted – the host's remaining link is unused). The processor that is attached to the secondary link has the identity 1024 – it is worth noting this point here since processor 1024 is referred to in various places in this document. The connection of the host Transputer's links is illustrated in Figure 4.2.

[2]It was necessary to reduce the rate from 20 Mbits/second to support applications not related to this research

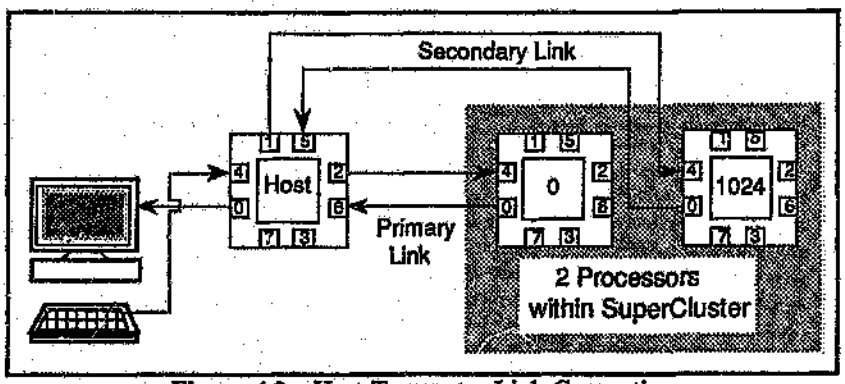


Figure 4.2 : Host Transputer Link Connection

A final point that should be noted with regard to the host Transputer is that it was only possible to connect both the primary *and* the secondary link to a T414 Transputer – the secondary link malfunctioned when used with a T800. This is important since

1. as shown in the next Figure (4.3), the X-Linda system requires the use of the secondary link
2. the T414 is a slower processor that the T800.

With regard this last issue (i.e. that the T414 is slower), it is shown in Appendix 2 that, transmitting sized of arrays of integers, the T414's rate of transmission is 1.29 times slower than the T800. The reason for this is multi-faceted :

- INMOS [1988a] quote a maximum bi-directional data rate of 2.4 Mbytes/second per link for the T800, and 1.6 Mbytes/second for the T414
- the T414 must receive an entire byte before it can send an acknowledgement, whereas the T800 employs an overlapped link protocol and sends an acknowledgement after it has received the first 2 bits of the message
- the connection between the T414 and the SuperCluster is through 2 ICUs, each of which has a signal delay of several ns [Parsytec GmbH, personal communication, September 1990].

4.2.3 Development System

X-Linda was developed under MultiTool [Parsytec 1989b], a Transputer development environment based heavily on the INMOS TDS (Transputer Development System). It features a fully integrated editor and tool-set, and provides the means to edit, compile, configure, load and run occam 2 programs. The version utilized, MultiTool 5.0, is fundamentally equivalent to the IMS D700D TDS, and supports network analysis and debugging facilities, and IMS C004 [INMOS 1988a] link switch configuration.

4.3 X-Linda Design and Specification

This section motivates and describes the fundamental design of X-Linda. The implementation of the tuple space model (intermediate uniform distribution) is discussed, and the structure of tuples and templates defined. The choice of TS primitive operations provided by the system is also explained. Finally, the design of the nodes within the system is described.

Before launching into the design of the system, it is important to be aware of the inherent difficulties associated with the implementation of globally accessible tuple space on Transputer meshes. To this end, it is worth restating the observations made in this regard in section 1. Firstly, there is the problem of maintaining distributed data over independent local memories (this is discussed in section 3.2.2 in the context of maintaining TS consistency within the Linda Machine). Secondly,

there is a problem that is related to the point-to-point nature of Transputer interconnection – i.e. that of a potential communications bottle-neck. This has a significant effect on the transmission of information over the in- and out-sets. The use of synchronous, point-to-point communication links implies that data must be passed consecutively along the nodes in the sets (i.e. it is not possible to broadcast information across an entire set in a single operation, as is the case with communication buses).

4.3.1 Tuple Space Model

Tuple space is implemented under the intermediate uniformly distributed scheme on a mesh of Transputers. Intermediate uniform distribution is discussed in detail in section 3.1.1.1 – all the concepts described in that section apply directly to X-Linda (e.g. the definition of in- and out-sets and the hardware topology are unchanged). This particular scheme was chosen for the reasons that it is

- simple and elegant
- featured in a "state-of-the-art" Linda implementation (the Linda machine – refer section 3.2)
- unique within the sphere of existing Transputer-based Linda implementations.

The tuple space model has been implemented on meshes of 4, 9 and 16 Transputers – for the sake of clarity, the 16 Transputer case (i.e. 4x4 mesh) is depicted in Figure 4.3.

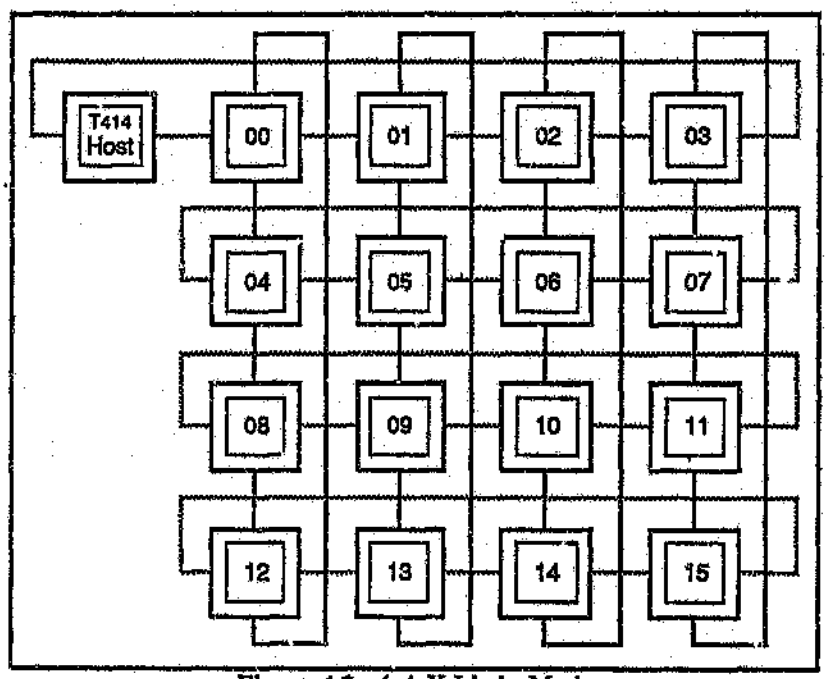


Figure 4.3 : 4x4 X-Linda Mesh

Notice the presence of the T414 host Transputer in the above Figure (all the rest of the nodes are T800s). As indicated in section 4.2.2, it is necessary to maintain a link between the host Transputer and those in the network. As a result, this host Transputer must be included in the top row of the mesh. This means that the out-set corresponding to this row includes an extra (slow) processor; as shown in section 6.2.2, the presence of this extra processor has quite a significant effect on the performance of the system. For the purposes of clarification, notice that Node 3's physical identity is 1024 (i.e. this is the processor that is connected to the host's secondary link).

4.3.1.1 Tuple Storage

A major influence of the Linda Machine on X-Linda is seen in the way that individual tuples are stored in the *same* TS addresses of all the nodes within a specific out-set. This technique was taken a step further and also applied to templates which are also stored in the same locations over the nodes in the in-sets. The need for this technique is evident with regard to maintaining TS consistency – most notably with respect to tuple / template deletion (using this technique, only an address is required for deletion – otherwise, a search would have to be made for the required tuple or template). Given that there are k nodes in the mesh, each in- and out-set comprises $\sqrt{k}$ nodes. The tuple and template queues on each node are sub-divided into $\sqrt{k}$ "buckets" – i.e. every node maintains a bucket that belongs to each of the other nodes in its respective in- or out-set. As a result, it is possible to ensure that every tuple or template issued from a specific node will occupy the *identical* location in the tuple and template request queues within *all* of the nodes in the sets. The storage of tuples across an out-set is illustrated in Figure 4.4 – this shows the TS structure of the nodes within an out-set of a 3x3 mesh. In the example shown, the bucket size is 128 entries and, consequently, each node's TS comprises 384 entries.

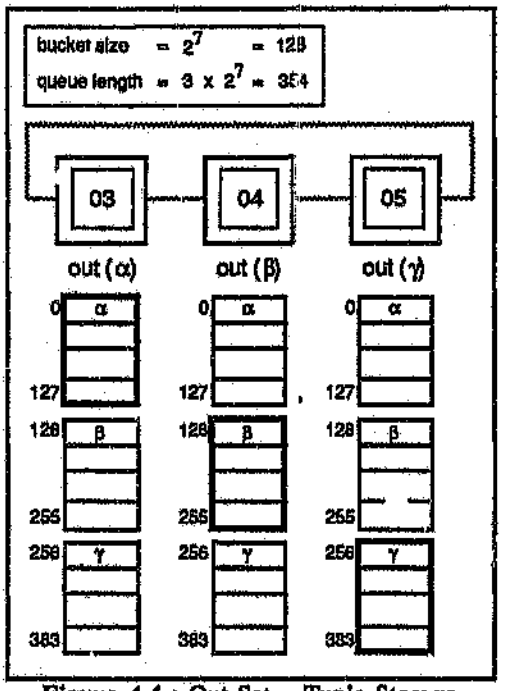


Figure 4.4 : Out-Set – Tuple Storage

The above Figure shows the storage of three tuples, α, β and γ, outed from Nodes 3, 4 and 5 respectively. Notice that each node "owns" a specific bucket, and that the tuples are stored in identical locations across the out-set.

Similarly, the storage of templates within an in-set is illustrated in Figure 4.5. Again, the example chosen reflects an in-set pertaining to a 3x3 mesh, and shows the addition of three templates, $\overline{\alpha}$, $\overline{\beta}$ and $\overline{\gamma}$, invoked from each of the nodes in the in-set respectively. In exactly the same way as for the out-set, it should be obvious that each node is assigned a specific bucket, and that the templates are stored in the same locations over the in-set.

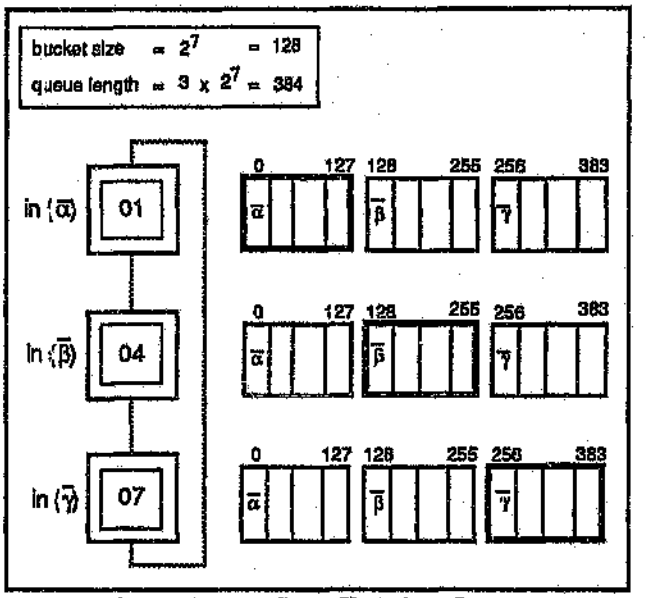


Figure 4.5 : In-Set – Template Storage

4.3.1.2 Tuple Structure

This research is concerned only with the communication overheads associated with the implementation of TS – it was desired that the intricacies and added complexity of tuple matching not be included in the investigation. It is therefore required that

- the tuple-matching process be kept as simple as possible (a simple linear search is utilized for this purpose)
- tuple fields be able to assume different lengths for the purpose of ascertaining the overheads with respect to the size of the tuple.

To meet this specification, tuples and templates need only comprise two fields – a tuple name (for matching) and a data field (of modifiable length). For ease of implementation, the tuple name is defined as a single 32-bit integer, and the data field comprises an array of integers. This is illustrated on Figure 4.6.

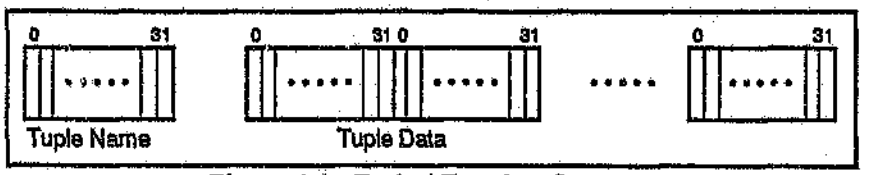


Figure 4.6 : Tuple / Template Structure

Notice that the name is *always* an *actual* parameter; it always contains some physical value, irrespective of whether it pertains to a tuple or a template. Tuple matching is therefore performed against a single key field – without attempting to draw too close an analogy, it is pertinent to note that the S/Net implementation [Carriero and Gelernter 1986] and, as described in section 3.3.1.2, Kernel Linda [Leler 1990] both feature a single key field. A tuple's data field is likewise always an actual parameter. Conversely, the data field for a template is always *formal* (i.e. a variable name that is assigned a value when the template name is successfully matched against a corresponding tuple). The above specification may appear overly restrictive. Indeed, under X-Linda it is not possible to capture the full "flavour" of the Linda programming methodology (keeping in mind, of course, that this was never the intention). However, as illustrated in the example programs discussed in section 7, it is still possible to program algorithms in a fairly elegant and expressive way. This, it is felt, is indicative of the power and potential of the Linda paradigm.

4.3.1.3 TS Operations

In order to investigate the communication overheads of the model, it was only necessary to provide the out, in and rd operations. Eval and the predicate operations were not implemented, as discussed below :

eval – The presence of this primitive would achieve little with regard to ascertaining the communication overhead of the implementation.

inp, rdp – Similarly, the predicate operations, inp and rdp, were not considered. In section 2.1.3, it was indicated that these operations are *not* universally accepted primitives. Furthermore, the feasibility of their inclusion within a distributed environment is highly debatable. Leichter [1990] argues against the implementation of inp and rdp in a distributed environment, claiming that their inclusion in such systems either causes inefficiency or introduces "bizarre semantics".

4.3.1.4 Link Directions

It is relevant to note the direction of communication along the in- and out-sets. From the definition of the in- and out-sets, it is evident that the links between the nodes in the X-Linda mesh must, at least, carry tuples, templates and requests to delete these. Now, as shown in Figure 4.7, the Transputer's communication links are bi-directional.

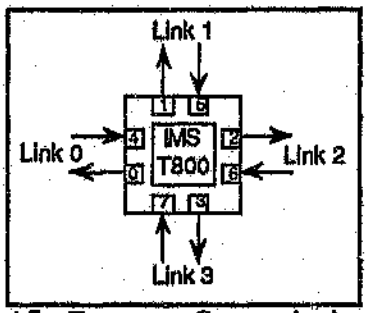


Figure 4.7 : Transputer Communication Links

Hence the traffic need not all be transmitted in a single direction. The implementation utilizes this bi-directionality to reduce the amount of one-way network traffic over the links. The specific link directions used to transmit the traffic are shown in Figure 4.8.

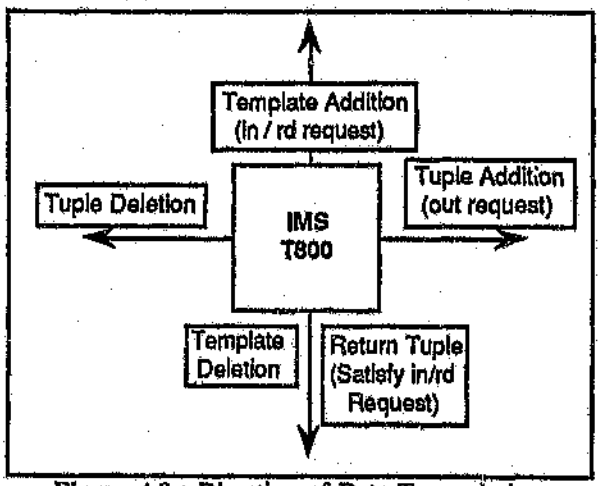


Figure 4.8 : Direction of Data Transmission

Examining the number of links that need to be traversed in order for a template to be sent along an in-set to a row containing the desired tuple, and then for that tuple to be transmitted back to the requesting node, it should be noted that, *on average*, this number is identical irrespective of whether one- or two-way communication is utilized.

4.3.2 X-LINDA NODE

Each Transputer within the mesh executes a number of inter-communicating, concurrent processes, each of which is dedicated to some specific function. For the sake of clarity, the Transputers are referred to as *X-Linda nodes* to make a distinction between them and the *Linda nodes* that are specific to the Linda Machine described in section 3.2. Recall that the Linda node comprises a Linda Engine and a computation co-processor, and that the Linda Engine has dedicated hardware components that are responsible for tuple memory, TS management and the processing of TS operations. The Linda Engine also has interfaces to the in- and out-buses. This arrangement was very influential in the design of the X-Linda node, to the extent that it (the X-Linda node) can, in a sense, be regarded as a *software* implementation of the Linda node. The X-Linda node features dedicated software processes which are responsible for the functions of computation, tuple storage and TS management, the processing of primitive operations and providing an interface into the in- and out-sets. The structure of the X-Linda node is shown in Figure 4.9.

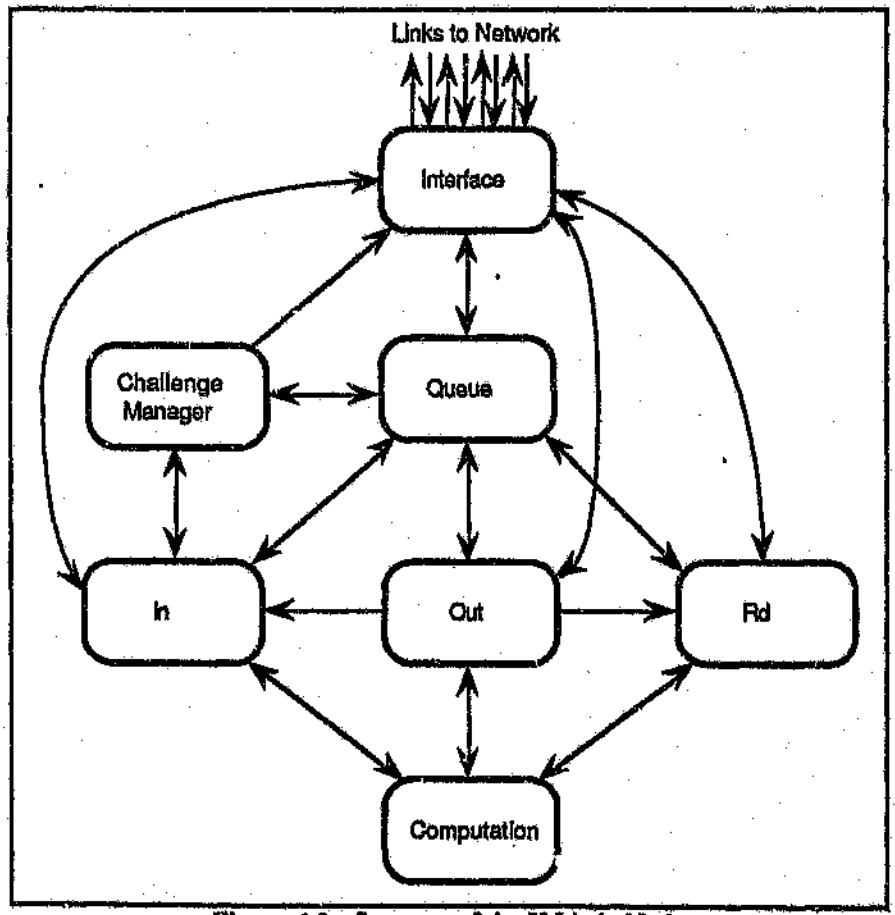


Figure 4.9 : Structure of the X-Linda Node

Linda programs are launched within the Computation process – i.e. the primitive operations are invoked from here. The In, Out and Rd processes control the pro-

cessing of the TS operations, received both locally (i.e. from the Computation process) and externally (received via the Interface process). Tuple space (i.e. the tuple and template queues) is stored on the Queue process – all tuple space addition, deletion and matching is done here. The Interface process is responsible for :

1. receiving tuples and templates from the In, Out and Rd Processes, and distributing them to the in- and out-sets
2. the reverse operation – i.e. receiving information from the in- and out-sets, and sending it on to the internal processes.

Finally, the Challenge Manager is an extra process dedicated to handling the requests for tuple ownership and subsequent deletion associated with the satisfaction of an in operation.

Aside – The Applicability of Occam 2

It is worth commenting on the applicability of occam 2 in the implementation of the X-Linda node. Occam, as stated previously, supports the process model of concurrency. Hence, it provides a very natural and elegant way of expressing the above process interaction. It must be said that occam *does* support an elegant programming model. However, as mentioned before, in a physically distributed environment, this model is too tightly coupled to the underlying hardware.

The intent of this section was to outline the structure and operation of the X-Linda node, and to illustrate the influence of the Linda Machine on its design. Specifications of the implementation of the out, in and rd operation, and a more detailed account of the X-Linda node's internal structure is given in the following section.

SECTION 5

5.0 IMPLEMENTATION DESIGN

The purpose of this section is to illustrate the implementation of the tuple space primitive operations under X-Linda, and, by examining the design and structure of the system at the process-level, to show how this functionality is achieved through the operation and interaction of the respective modules. These issues are addressed as follows :

1. Implementation of the TS Primitives
Shows how the primitive tuple space operations (i.e. out, in and rd) are implemented under X-Linda, and highlights various problematic issues that had to be addressed in order to successfully realize the implementation of these operations.

2. Process-Level Design
Describes the operation of the host processor and the X-Linda node at the process level, illustrating the functionality and interaction of the processes that comprise the system. A note on the system's software specifications, regarding the programming methodology and the storage requirements is also given here.

5.1 IMPLEMENTATION OF THE TS PRIMITIVES

This section describes the design considerations regarding the specification and implementation of the TS primitive operations. It is of importance to note these considerations as they have a direct influence on the efficiency of the system. Furthermore, it is of interest to observe how TS consistency is maintained in the course of processing these primitives. It will be seen in this section that a good deal of design effort was required in order to realize the successful implementation of the TS primitives and the protocols for maintaining TS consistency. This is attributable to the fact that point-to-point communication links are not an ideal medium for the implementation of in- and out-sets. The implementation of the primitive operations is covered as follows :

1. Out operation – an overview of the out operation is given, and some detail regarding a very important design strategy (i.e. the blocking of the processor invoking the out) is presented. The processing of the out request and some tuple matching considerations are discussed.

2. Rd operation – as for the out primitive, the processing of the rd operation is discussed in general. The satisfaction of rd requests is described, and an interesting aspect of request satisfaction (i.e. *multiple* request satisfaction) is presented.

3. In operation – a large part of this section is devoted to the explanation of the "challenge" process (i.e. when two nodes simultaneously issue in requests for the same tuple, they must contend to ascertain which one has the right to delete and retrieve the tuple). As for the rd operation, the process of satisfying the request is described, and the issues of multiple request satisfaction and subsequent tuple restoration are discussed.

5.1.1 OUT OPERATION

Tuples outed by an application program are stored in local tuple space (i.e. on the processor where the program is running) and then sent to the out-set of that node. The node blocks until the tuple has traversed the entire out-set (the reason for this blocking requirement is discussed below). The tuple is added to each node in the set and is matched against any pending templates on that node – if a match is found, the satisfaction of the associated in or rd request is invoked. The following issues are discussed in more detail below :

- blocking the processor until the tuple has traversed the entire out-set

- the general processing of out requests
- matching tuples against templates.

5.1.1.1 Traversal of the Out-Set

Consider an out-set comprising three nodes (0, 1 and 2) connected via point-to-point serial links in a ring as shown in Figure 5.1.

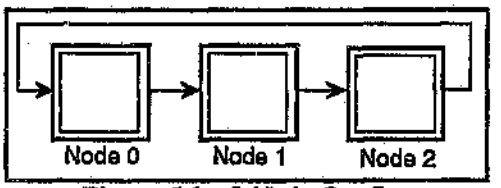


Figure 5.1 : 3 Node Out-Set

The execution of out (α) causes the tuple α to be sent around the out-set and stored in the local memories of each node. Assume that Node 0 outs two tuples, α_1 and α_2, consecutively. There are two ways [Faasen 1990b] in which the addition of these tuples to the out-set can be handled :

1. No Delay between Transmissions

If no delay between the operations is specified as soon as α_1 is transmitted from Node 0 and received by Node 1, α_2 can also be forwarded to the out-set – i.e. α_2 will be sent the the out-set *before* α_1 has been stored in every node in the set. This is shown in Figure 5.2.

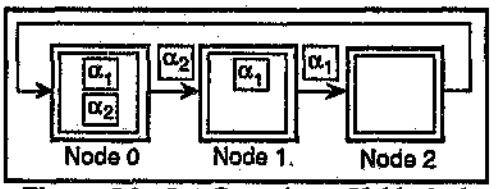


Figure 5.2 : Out Operation – Unblocked

As a side issue, notice that, depending on the specification of the implementation, it is possible that α_2 may "overtake" α_1. This leads to a query regarding the semantic equivalence of out(α_1) ; out(α_2) and out(α_2) ; out(α_1), which is addressed in Appendix 7.

2. Blocking the Transmission

Alternatively, it is possible to force the transmission of α_2 to be delayed until α_1 has been stored in every node in the out-set – i.e. to wait until the tuple returns to the node that invoked the out. This is shown in Figure 5.3.

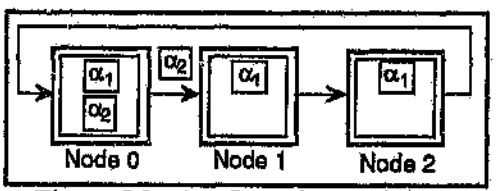


Figure 5.3 : Out Operation – Blocked

Both strategies satisfy Linda's semantic specifications equally well. The first approach was implemented in earlier phases of X-Linda. However, this unrestrained form of transmission understandably resulted in the out-set becoming saturated, and consequently caused deadlock. Hence, the second strategy was adopted. The

effect of forcing the processor to block *does* have a significant effect on the efficiency of the out operation (refer section 6.2.1) – however, a proposal for reducing the effect of blocking is discussed in section 8.2.2.

5.1.1.2 Processing the Out Request

Locally Invoked

A locally invoked out request can be either a normal out operation (i.e. invoked by an application program) or the restoration a tuple satisfying an in request (a single in request may be satisfied at more than one location, resulting in multiple tuples being returned to the requesting node – as detailed in section 5.1.3.4, the "extra" tuples must be re-inserted into TS). In either case, the processing of the request is the same. The tuple is added to local TS and matched against pending templates in the request queue. If no match is found, the tuple is forwarded to the out-set. In the case of a match against a rd request, the tuple is *still* forwarded to the out-set – rds are non-destructive, and the tuple must as usual be added to TS – and the satisfaction of the request invoked. A successful match against an in request causes the tuple to be deleted from the local TS before it is returned to the application program that is requesting it.

Externally Invoked

Nodes receiving external out requests add these tuples to their TS and attempt to match them against their local request queues. If no match is found, the tuple is passed to the next node in the in-set – otherwise, the associated tuple is sent via the in-set to the node that issued the request.

5.1.1.3 Matching Tuples against Templates

Rd Requests

As stated above, a tuple added to a local TS must be matched against the templates in the request queue. It is worth noting that if a match is found against a rd request (as opposed to an in request), it is necessary to test for further matches. Rds are non-destructive; hence, a single tuple may match *all* pending rd requests in the request queue. Therefore, on successfully locating a rd request, the matching process must be successively repeated until

- all matching rd templates have been located, or
- an in request is encountered.

Successful matches are satisfied as discussed in section 5.1.2.2 below.

In Requests

Obviously, the procedure regarding an in request is different – only one matching in request in the request queue can be satisfied by a single tuple. These requests are satisfied as discussed in section 5.1.3.3 below.

5.1.2 RD OPERATION

A rd operation causes a template to be stored in the local memory of the node from which it is invoked, and then to be transmitted to all nodes that make up the in-set. The template is added to the request queues within these nodes, and is matched against the tuples present in the tuple space. A successful match causes the associated tuple to be returned to the requesting node, and the request to be deleted from the nodes within the in-set. It is possible that the same rd request may be satisfied at more than one node. In this case, the requesting node will receive more than one tuple – it must accept only the first tuple it receives, and discard all others. Below, the following issues are discussed in more detail :

- the processing of the rd request
- the satisfaction of requests and deletion of templates
- multiple request satisfaction.

5.1.2.1 Processing the Rd Request

Locally Invoked

Local rd requests are appended to the request queue of the processor invoking the operation and matched against the tuples resident in the local TS. If no match is found, the request is then transmitted to the other nodes in the in-set. If a match is found, the request obviously need not be sent to the in-set as the request can immediately be satisfied (this procedure is different with respect to an In request – refer section 5.1.3.1).

Externally Invoked

Nodes receiving external rd requests add these templates to their request queues and attempt to match them against their local TS. If no match is found, the template is passed to the next node in the in-set – otherwise, the associated tuple is sent via the in-set to the node that issued the request.

5.1.2.2 Satisfying the Request

When a template is successfully matched, that template is deleted from the local request queue. If the request was invoked locally, no further action is necessary. On the other hand, the satisfaction of an external request necessitates

- the deletion of the request from the entire in-set
- returning the associated tuple to the requesting node.

The node satisfying the match must transmit a delete command to the nodes in the in-set. On receipt of this command, these nodes remove the template from their request queues. Notice, however, that this is slightly more complicated than simply deleting the entry. As is illustrated in Figure 5.4, an attempt may be made to delete a template that has not yet been added to the request queue (recall from section 4.3.1.4 that requests are sent *upwards* and delete commands *downwards*).

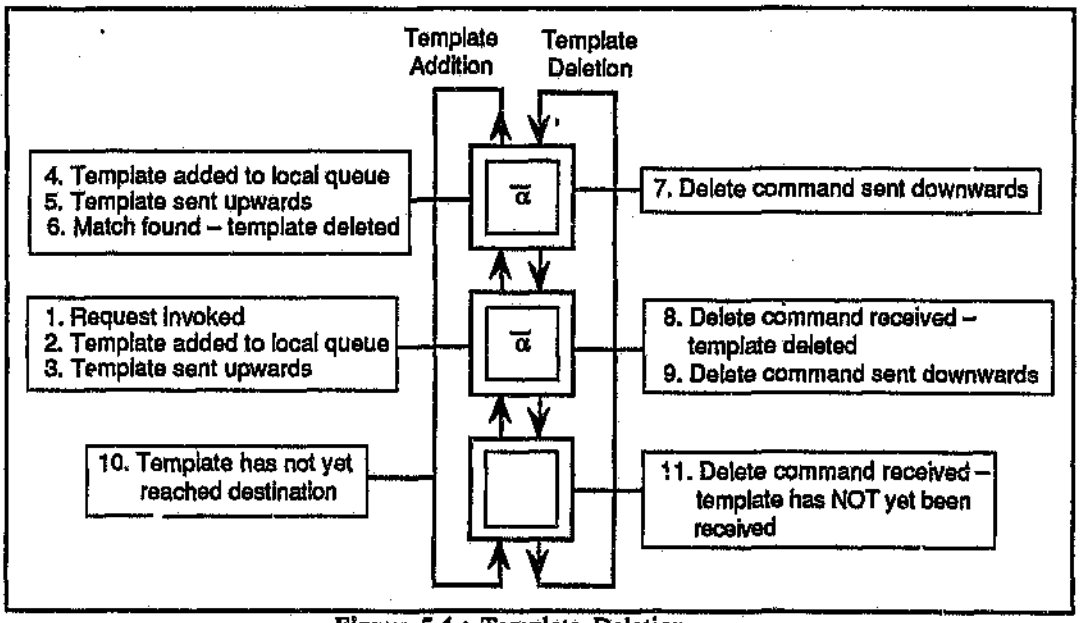


Figure 5.4 : Template Deletion

Consequently, before a template can be deleted, a check must be made that it actually exists. If not, the state of the entry in the queue corresponding to that template is set to "delete pending" – when the template finally arrives and is added to the queue, it will immediately be removed. Notice that the processor that invoked the request is unable to re-use the specific address of the template until the delete message has traversed the in-set. This prevents a situation whereby a new template may be incorrectly deleted due to a "delete pending" state.

Tuples that satisfy templates are returned to the requesting node via the in-set.

Satisfaction by an Out Operation
As discussed in section 5.1.1, a pending template present in the request queue can be matched by a tuple added to the processor's local TS. If the template was locally invoked the request is satisfied locally – otherwise, the tuple is passed via the in-set to the requesting processor.

5.1.2.3 Multiple Satisfaction of Requests

It is possible that a rd request may be satisfied at more than one node. Figure 5.5 shows a situation where Node 0 issues a template, and then Nodes 2 and 8 *both* issue matching tuples. This causes matches to be found at Nodes 0 *and* 6, and hence causes multiple request satisfaction.

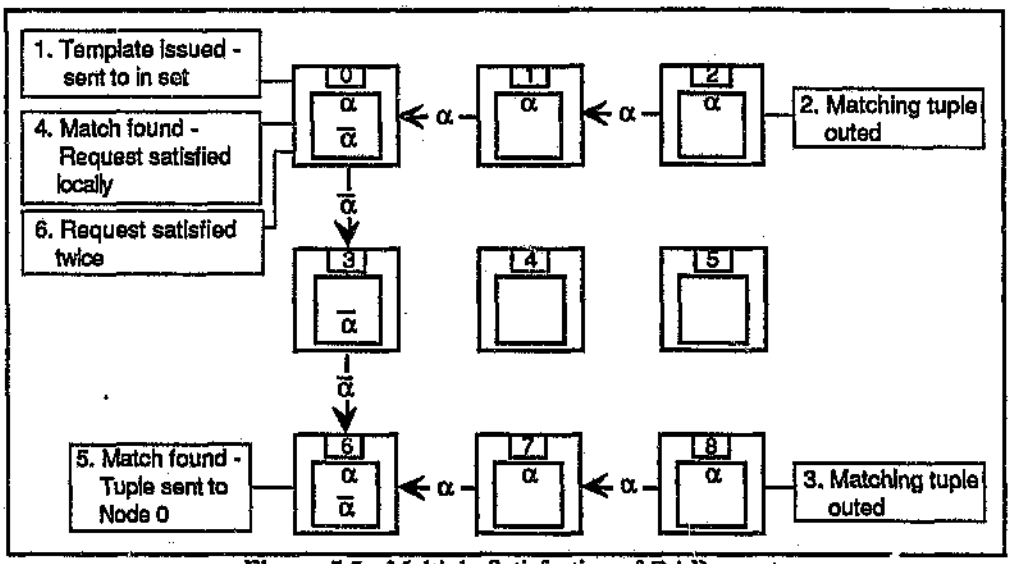


Figure 5.5 : Multiple Satisfaction of Rd Request

In this case, the requesting node will receive more than one tuple. It must accept only the first tuple it receives, and discard all others. The strategy for detecting and consequently discarding multiple request satisfies is straight forward. It is based on a method of two counters – invoked_count and satisfied_count. When a request is invoked, invoked_count is incremented. This count is stored with the actual template in the respective request queues. When that template is successfully matched and the associated tuple returned to the requesting node, invoked_count is transmitted with the tuple. A node receiving a tuple in response to a rd request increments its satisfied_count and then tests whether invoked_count = satisfied_count. If the count values are different, the request has already been satisfied, and the tuple is discarded.

5.1.3 IN OPERATION

The processing of an in operation is similar to the rd in that the template is stored locally and in the memories of nodes in the in-set; and is matched against the tuples in the respective tuple spaces. However, on a successful match, the associated tuple is deleted from the entire out-set before it is returned to the requesting node. Nodes requiring to satisfy an in request must contend for the tuple in question. If there is more than one request on the same tuple, only one of these may succeed in deleting the tuple and returning it to the requesting processor. When a request has been sat-

isfied, the template is removed from the nodes in the in-set. It was shown in section 5.1.2.3 that a template may find a match at a number of nodes simultaneously, causing multiple request satisfaction. The requesting node must, as for a rd operation, accept only the first tuple it receives in response to the request. Notice, however, that any other tuples that are received cannot merely be discarded. These tuples are deleted from TS by the node satisfying the in request, and the requesting node must therefore re-out the tuples into TS. The following issues are discussed below.

- the processing of the in request
- the challenge process
- the satisfaction of requests, and deletion of templates
- multiple request satisfaction and tuple restoration.

5.1.3.1 Processing the In Request

Locally Invoked

Local in requests are appended to the request queue of the processor invoking the operation and matched against the tuples resident in the local TS. If no match is found, the request is then transmitted to the other nodes in the in-set. Notice that, in the case of successful match, the template must *still* be transmitted to the in-set. It is possible that another node may also request the tuple in question; hence templates must *always* be sent to the in-set as the node invoking the request may lose a tuple challenge and have to find another tuple elsewhere. When a match is found, the tuple cannot be simply deleted from TS and consumed. The node must "challenge" the other nodes in the out-set in case they too are attempting to satisfy a request on the same tuple.

Externally Invoked

Nodes receiving external in requests add these templates to their request queues and attempt to match them against their local TS. If no match is found, the template is passed to the next node in the in-set – otherwise, the satisfaction of the request can be invoked. As discussed above, the node must invoke a challenge before it may delete and return the tuple.

5.1.3.2 The Challenge Process

The issue of tuple contention was briefly mentioned above. This is a very real problem. If two nodes simultaneously issue a request for, and successfully locate the same tuple, only one may be permitted to retrieve that tuple and delete it from TS; the other must withdraw from contention and seek another tuple. Take the example whereby a tuple is present in the top row of the mesh. Now, if two nodes in different in-sets simultaneously issue requests for that tuple, a match will succeed in two locations – this is illustrated in Figure 5.6.

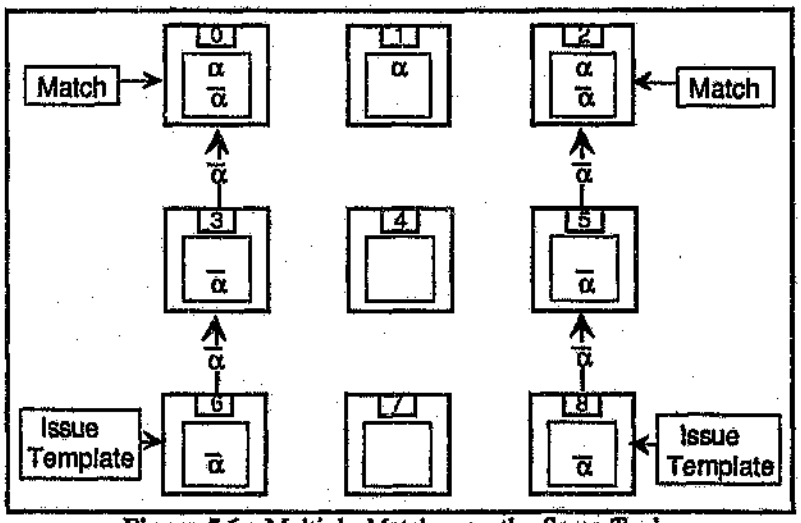
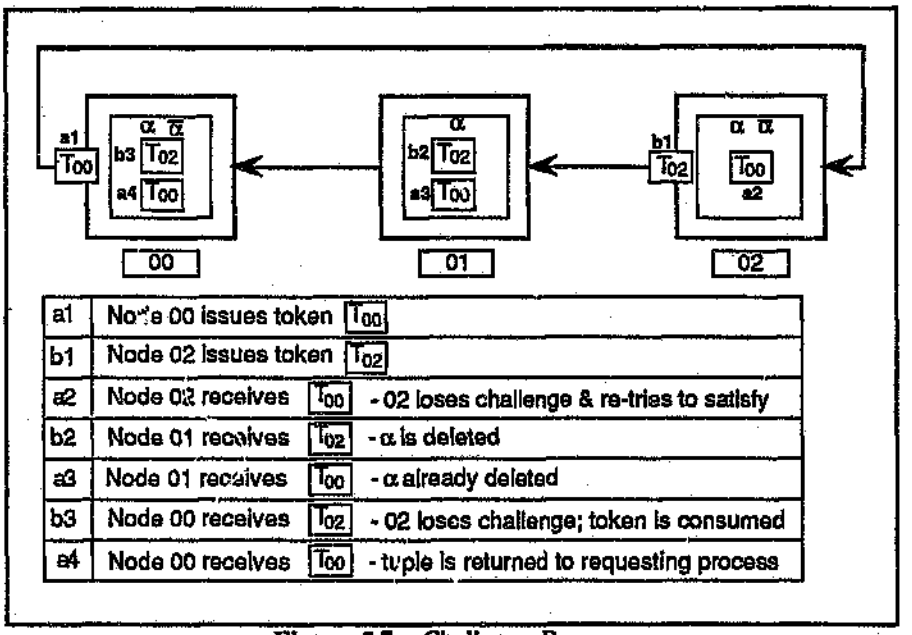


Figure 5.6 : Multiple Matches on the Same Tuple

On finding a match, a node must "fight" for ownership of the associated tuple before it can be deleted from the TS and returned to the requesting node. This is implemented using a simple and efficient strategy. When a match is found, the node sends to the out-set a *challenge token*, which essentially contains the address (i.e. location in TS) of the tuple to be deleted and the identity of the node invoking the challenge. A node receiving a token from a foreign processor tests whether or not it itself is attempting to satisfy an in request on the same tuple. If not, it deletes the tuple from its local TS and passes the token on. On the other hand, if it too is satisfying a request on the same tuple, it must contend the challenge – i.e. one of the nodes must lose out and withdraw from contention. The strategy used in determining the outcome of a challenge is simple. The node with lowest identity wins the challenge; if the identity of the node receiving the token is *less* than that of the node that issued the token, it wins the challenge; otherwise it loses. The procedure associated with winning or losing challenges is outlined below :

- *Winning a challenge* – If the node wins the challenge, it consumes the token (i.e. the losing node's challenge "dies").
- *Losing a challenge* – Conversely, on losing the challenge, the node deletes the tuple from its local TS, passes the token on, and then re-attempts to find a match for the unsatisfied template. Effectively, the node has now withdrawn from contention with regard to the tuple that it originally tried to claim.

Once a token returns to the node that issued it, it is guaranteed that the associated tuple has been deleted from TS, and can now be returned to the requesting node. The idea is expressed by means of an example in Figure 5.7.



a1	Node 00 issues token	T_{00}	
b1	Node 02 issues token	T_{02}	
a2	Node 02 receives	T_{00}	- 02 loses challenge & re-tries to satisfy
b2	Node 01 receives	T_{02}	- α is deleted
a3	Node 01 receives	T_{00}	- α already deleted
b3	Node 00 receives	T_{02}	- 02 loses challenge; token is consumed
a4	Node 00 receives	T_{00}	- tuple is returned to requesting process

Figure 5.7 : Challenge Process

The contention strategy (i.e. based on the identity of the conflicting nodes) is somewhat arbitrary, and obviously favours nodes on the left hand side of the mesh. The strategy is, however, simple to implement and, since this issue does not directly relate to the objective of the research, was considered acceptable to use.

Other Possible Challenge Strategies

Some alternative methods for determining the outcome of tuple contention are discussed below. They are mentioned here for the sake of interest, and, in section 8.2.8, are briefly re-visited in the context of future research.

1. Honouring Tuple Ownership

If two nodes are challenging for the same tuple, and that tuple was outed by one of the nodes in contention, then that node will automatically win the challenge. The practicality of the approach would depend on the frequency with which application programs tend to request tuples that they themselves initially put into TS. This strategy was implemented in earlier phases of X-Linda, but was ultimately discarded for the sake of simplicity.

2. Fairness Policies

As stated previously, the X-Linda tuple contention scheme is biased towards nodes with low processor identities - i.e. nodes on the left-hand side of the mesh. It would be of interest to investigate a "fairer" policy that would distribute the power to win challenge requests. Instinctively, a random selection policy may seem attractive. If two nodes are contesting for the same tuple, a random decision could be taken locally to determine whether or not a specific node should win or lose the conflict. This is, of course, nonsensical, since both nodes might win ownership of the tuple on this basis. A strategy that received serious consideration (and which was in fact implemented in earlier phases of the system) was that of a virtual identity scheme. Every processor is given a Virtual ID, initially set to the processor identity. The strategy for determining the outcome of a challenge is then similar to that described previously, but uses the Virtual ID instead of the processor identity. A node wins ownership of the tuple if its Virtual ID is less than that of the external (challenging) node's Virtual ID. Now, if there are k nodes in the system, a node's Virtual ID could be incremented by $\sqrt{k}$ whenever it wins a challenge. This would guarantee that the node would lose the next challenge, providing a fairer challenge

scheme. The strategy is obviously suited to the situation whereby every node issues a single tuple challenge at a time. However, a node can simultaneously be responsible for several (up to a maximum of √k) challenge requests. The complexity of maintaining a consistent Virtual ID in this situation prohibited the inclusion of the scheme.

Deleting Tuples
It is worth commenting on the complexity involved in deleting tuples from the out-set – the operation is not at all straight forward. The first problem is exactly that which was described with respect to deleting templates – i.e. it is possible that a tuple will not yet have been installed on a node when the command to delete that tuple is received. This is solved in exactly the same way as for template deletion, using a "delete pending" state. The second problem is related to this one. Referring back to Figure 5.7, notice that, with regard to Node 1, the following situation occurs :

a3	Node 01 receives T00 - α already deleted

Now, when Node 1 receives this token, how does it know that α has already been deleted. It cannot simply apply the rule that when a delete command is applied to a tuple that does not exist, the state of that tuple must be set to "delete pending". Hence, it is necessary to store extra identifying information with the tuple address to prevent this situation from occurring.

5.1.3.3 Satisfying the Request
When a template is successfully matched, that template is deleted from the local request queue. It must also (even in the case of a local request) be deleted from the entire in-set (as was seen previously, in requests are *always* sent to the entire in-set). Furthermore, the associated tuple must be either returned to
- the application program in the case of a local request, or
- the requesting node via the in-set for an external request.

The same procedure for deleting templates that was described for the rd operation (refer section 5.1.2.2) is utilized here. Notably, care must again be taken against deleting a template that has not yet been added to the request queue, and the "delete pending" state is used in this regard.

Satisfaction by an Out Operation
Pending templates present in the request queue that are matched by a tuple added to the processor's local TS are processed as described above.

5.1.3.4 Multiple Satisfaction of Requests
With respect to the rd operation, it was described in section 5.1.2.3 how a request could be satisfied at more than node – obviously, the same applies to in requests. The same strategy as described for rd requests (i.e. using two counters – invoked_count and satisfied_count) is employed. Notice, however, that a tuple that has already been received can no longer merely be discarded. It will have been deleted from the TS, and must therefore be *restored* as described below.

Tuple Restoration
The process of tuple restoration is best explained by means of a simple example. Referring to Figure 5.8, assume that a tuple is present in the out-set comprising Nodes 3, 4 and 5, and another tuple with the *same* name exists in the out-set comprising Nodes 6, 7 and 8. Now, if Node 0 issues a template that matches both of these tuples, a match will be found at Nodes 3 *and* 6. The tuples will be deleted from the respective out-sets, and *both* tuples will be returned to Node 0.

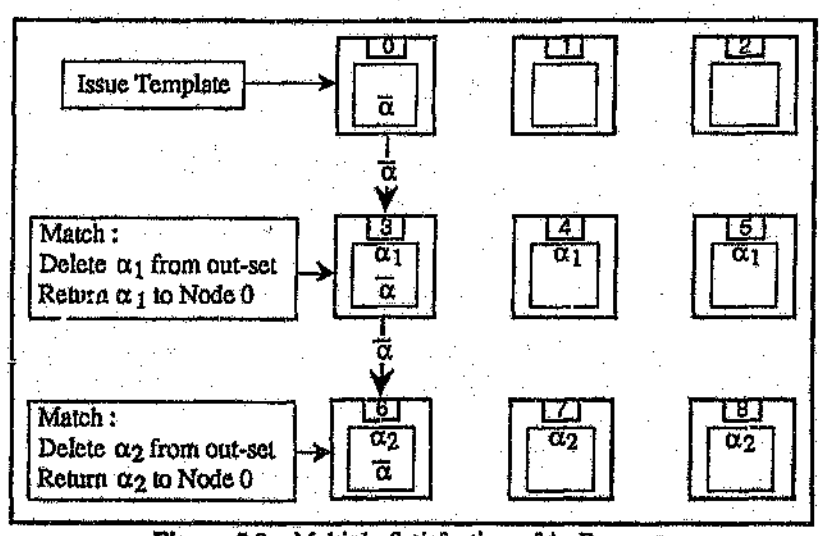


Figure 5.8 : Multiple Satisfaction of In Request

Node 0 will therefore receive two tuples in response to its request. Assume that it receives α_1 first – this tuple is then consumed by the application program that invoked the request. However, on receipt of α_2, Node 0 must *re*-out this tuple since it has (incorrectly) been deleted from Node 6's out-set. Notice that this scheme can actually enhance the performance of the in operation. If Node 0 now requests another tuple with the same name, this request will be satisfied locally.

5.1.4 Discussion

The design considerations detailed in this section have illustrated that the implementation of the TS operations was far from trivial. The greatest factor contributing to the overall complexity stems from the need to maintain TS consistency, a problem obviously common to all systems that are required to maintain distributed data. It is claimed that the TS primitives and protocols for maintaining TS consistency have been implemented in a simple and effective fashion. However, it is worth pointing out that their successful implementation ultimately required a great deal of design effort, verification and modification. This is perhaps indicative of the fact that in- and out-sets are not well suited to implementation over the Transputer's communication links. It is also relevant to note that, given the design considerations outlined in this section, the implementation of the predicate operations (inp and rdp) would be exceedingly difficult, justifying the comments in section 4.3.1.3 regarding their unsuitability to distributed-memory implementations.

5.2 PROCESS-LEVEL DESIGN

The overall design and structure of X-Linda is given below in terms of the operation and functionality of the various modules that comprise the system. It is not the intention to give a blow-by-blow account of the occam programs themselves; instead, the focus is at the process level. It should by now be evident that X-Linda is, in essence, a collection of intercommunicating concurrent processes. In this section, the purpose and interaction of these processes is described. The design of the system is systematically addressed with respect to the Host process and the X-Linda node :

1. Host process – the host process is that which resides on the *host* Transputer – i.e. the extra T414 processor that is included within the top row of the mesh. The primary functions of the host process – to act as an intermediate node and to provide monitoring facilities – are described in the context of the two concurrent processes that provide these operations.

2. X-Linda node – the operation of each Transputer within the X-Linda mesh is given with respect to the dedicated software processes that reside on each processor (i.e. Interface, In, Out, Rd, Queue, Computation and Challenge Manager). The information is neither technical nor detailed, and is intended simply to provide an overview of the function of the processes.

5.2.1 THE HOST PROCESS

As indicated in section 4.3.1, the host Transputer functions primarily as an intermediate node in the network, simply passing on information that it receives from the nodes that comprise the top row of the mesh. A secondary function of the processor is to provide monitoring routines that enable the local TS on each node to be inspected. These two functions are implemented by means of parallel, intercommunicating processes as shown in Figure 5.9 (recall from section 4.2.2 that the processor that is attached to the host processor's secondary link has the identity 1024).

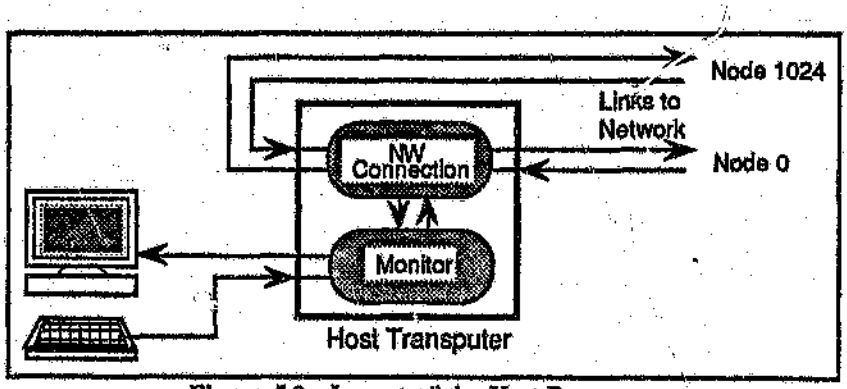


Figure 5.9 : Layout of the Host Processor

The operation of the Monitor and Network (NW) Connection processes are described in more detail below.

5.2.1.1 Monitor Process

The Monitor process provides the means to inspect the tuple space by accessing the tuple and template queues for selected processors. The screen shown in Figure 5.10 is displayed to the user – this shows the arrangement of the X-Linda mesh, relevant system dimensions, and various TS inspection options.

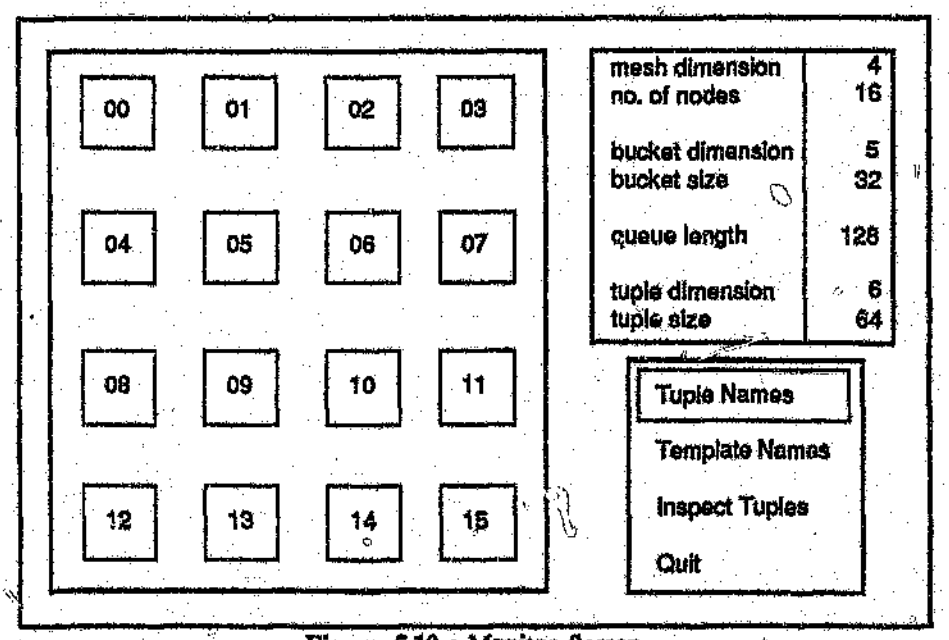


Figure 5.10 : Monitor Screen

TS Inspection

For any specified node identity, the Monitor can be invoked to display

- the names of the tuples present in the TS
- the names of the templates queued on that node
- the contents of the data fields of tuples.

The above requests are relayed to the NW Connection process and then sent to the required processor. The information returns to the Monitor from the network via the NW Connection process, and is displayed in a suitable format.

Aside – Routing Strategy

The algorithm for routing monitor messages from the host to a specified processor and back again is simple. The design and implementation of an optimal routing strategy was not considered to be necessary for the purpose of monitoring. The message is passed along the top row of the mesh until it reaches the column that the specified node comprises (unless, of course, this node is itself present in the top row). The message is then passed down the column until it reaches its destination. This route is reversed in order to return the TS information to the host. The route taken in accessing the TS on processor 7 in a 3x3 mesh is shown in Figure 5.11.

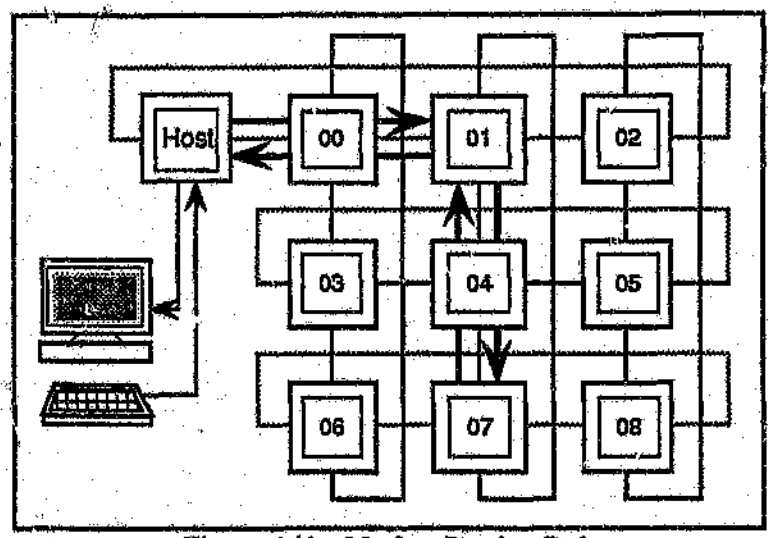


Figure 5.11 : Monitor Routing Path

5.2.1.2 NW Connection Process

The NW Connection process is responsible for

- providing a link between Node 0 and Node 1024
- passing monitor information between the Monitor process and the network.

The process loops continually, testing for input on its relevant channels and re-directing this input as required. This operation is illustrated below :

```
DO Forever[1]
   ALTernative
      • Receive Monitor information from Node 0
        Send to Monitor Process
      • Receive X-Linda information from Node 0
        Send to Node 1024
      • Receive X-Linda information from Node 1024
        Send to Node 0
      • Receive Monitor request from Monitor Process
        Send to Node 0
END DO
```

5.2.2 The Linda Node

Recall from section 4.3.2 that the processors within the X-Linda mesh are referred to as X-Linda nodes. These nodes execute a number of inter-communicating, concurrent processes, each of which is dedicated to some specific function – i.e. computation, tuple storage and TS management, the processing of primitive operations and providing an interface into the in- and out-sets. The structure of the X-Linda node is shown again in Figure 5.12.

[1] "We pick 'FOREVER' to be a very long time, equal to the largest number that can be stored on the machine the universe is being simulated on, or the length of time until the Last Judgement, depending on your religion." D. McDermott, *A Temporal Logic for Reasoning about Processes and Plans*, Cognitive Science (6), 1982, 123

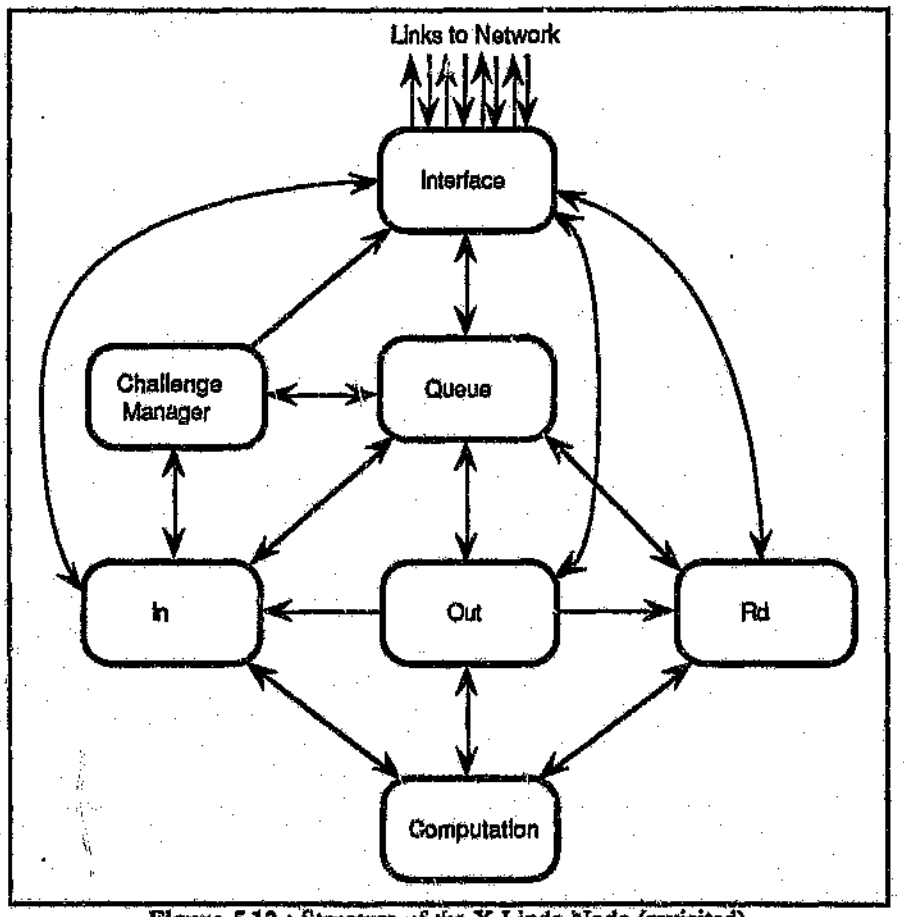


Figure 5.12 : Structure of the X-Linda Node (revisited)

Below, the function of these processes is briefly outlined, illustrating how the overall functionality of the system is achieved via the interaction of the processes. The description follows a diagrammatically bottom-up order – i.e. starting with the Computation process and ending with the Interface (a more detailed diagrammatic representation and the structure and interaction of these processes is given in Appendix 5). To conclude the overview, some comment is given on the need for buffering and for atomicity of operations within the X-Linda node, and the suitability of the occam 2 programming model as a vehicle for expressing the above process interaction is briefly re-addressed.

5.2.2.1 Computation Process

The Computation process hosts the application programs that are executed under X-Linda. TS operations are invoked via procedure calls within the process, from where control is transferred to the relevant TS primitive process (i.e. In, Out or Rd). The processing of TS requests is briefly outlined below :

1. Out Requests – tuples outed from the application program are transmitted to the Out process. Once the tuple has traversed the out-set, an acknowledgement is sent back from the Out process to the procedure from which the out was invoked – recall that the application program is blocked during this time.

2. Rd Requests – rd requests are sent to the Rd process. When a successful match is located, the associated tuple is returned to the procedure invoking the request, and consumed by the application program.

3. In Requests – as for a rd request, In requests are transmitted directly to the In process. Information returned from the In process can be one of two types :

1. a requested tuple – as for the rd, the tuple is consumed by the application program that invoked the request
2. a requested tuple that has *already* been received. It was illustrated in section 5.1.3.4 that a single in request can be satisfied at more than one location, and that the "extra" tuples must be restored to TS. These tuples are simply re-directed to the Out process for addition to TS in the normal way.

5.2.2.2 Out Process

The Out process is responsible for receiving out requests (i.e. tuples) invoked both locally and externally. It must add these tuples to TS, and depending on whether or not there exists a matching template, transfer control to the appropriate process (i.e. In or Rd). Locally invoked tuples are received from the Computation process and external tuples from the Interface. These tuples pass through a pair of buffer processes before being processed (buffering is necessary to avoid network saturation). The tuples are sent to the Queue process for addition to TS, where they are also matched against pending templates. If a match is found, the Out process transmits the tuple to the In or Rd process (depending on the type of the template) for subsequent satisfaction. Notice that

1. if a match is found against a rd request, an attempt must be made to locate further matches
2. locally invoked tuples must be sent to the Interface for addition to the out-set.

5.2.2.3 Rd Process

Locally and externally invoked rd requests are handled by the Rd process. As for the Out process, local requests are received from the Computation process, and external requests arrive via the Interface. In both cases, the associated request is directed to the Queue process for addition to the template queue and subsequent matching against the tuple queue. If a match is successfully located, the Rd process returns the requested tuple to the Computation process in the case of a local request, or to the Interface for transmission to the requesting node in the case of an external request. When the Rd process receives a tuple in response to a request that is satisfied at an external location, it tests whether or not the request has already been satisfied (as described in section 5.1.2.3). If the request has not yet been satisfied, the tuple is sent to the Computation process, and a command to delete the associated template from the in-set is sent to the Interface – otherwise, the tuple is discarded.

It was seen above that tuples matched against rd requests by the Out process are transmitted to the Rd process. These tuples are redirected from the Rd process in the normal way (i.e. to the Computation process or the Interface process, depending on whether a local or external request is to be satisfied).

5.2.2.4 In Process

The function and operation of the In process is similar to that of the Rd. As before, the In process receives locally invoked In requests from the Computation process, and external templates from the Interface. These requests are sent to the Queue process for addition to the template queue and for matching against the tuples in TS. Notice, however, that the In process does not directly act upon successful matches. When a match is found within the Queue process, it (the Queue process) sends the tuple to the Challenge Manager process for addition to the list of "claimed" tuples, and the Challenge Manager issues a challenge token to the out-set. When the In process receives a locally invoked challenge token, it knows that it has won ownership of the tuple, and can delete and return that tuple to the requesting process accordingly. On the other hand, on receipt of an externally invoked token, the In process queries the Challenge Manager process to determine whether or not the tuple in question has also been claimed locally. If not, the token is simply passed on to the

next node in the out-set. However, if there is tuple contention, this must be resolved using the processor identity scheme detailed in section 5.1.3.2. It is worth noting the added complexity associated with a node losing a tuple challenge; when this occurs, the tuple must be re-requested. Re-requesting is done by simply re-adding the template to the list within the Queue process. Notice, however, that if the template in question was originally invoked by an external node, and if that request has subsequently been satisfied, it is possible that the template entry may since have been deleted, or even overwritten by a new request. Hence care must be exercised when re-inserting the template into the list.

Otherwise, the operation of the In process is much the same as the Rd, i.e. with respect to redirecting tuples returned in response to request satisfaction, processing requests satisfied by the Out process and issuing template deletion commands. The processing of requests that have been satisfied more than once is, however, slightly different. The extra tuples are not simply discarded (as done by the Rd process). Instead, they are sent to the Computation process from where a normal out operation sends them to the Out process and subsequently into TS.

5.2.2.5 Challenge Manager Process

The Challenge Manager process is responsible for maintaining the list of tuples that a specific node has claimed and requires to delete in fulfilment of an In request. A single node can at any time be responsible for $\sqrt{k}$ such tuples, given that there are k nodes within the mesh. Tuples matched against In requests within the Queue process are sent to the Challenge Manager and added to the list of claimed tuples. The Challenge Manager then sends a challenge "token" (essentially the TS address of the tuple) to the Interface for transmission to the out-set. When the In process receives a token issued from an external node, it sends this token to the Challenge Manager where it is matched against the claimed tuples in the list. If a match is found, the Challenge Manager informs the In process that tuple contention has occurred, and the In process resolves this conflict as described in section 5.1.3.2. Alternatively, on receiving a locally issued token from the In process, the Challenge Manager simply removes the corresponding tuple from its list (this tuple has been successfully claimed and is of no further interest).

5.2.2.6 Queue Process

Local TS is maintained within the Queue process in the form of a tuple list and a template list. Tuples and templates comprise the 32-bit integer fields shown in Table 5.1, and the lists are simply implemented as arrays of these fields.

Tuples		Templates
Name		**Name**
Source		Source
Sequence		Sequence
State		**Type**
Data (integer array)		State

Table 5.1 : Storage of Tuples / Templates

The fields shown above in non-bold font represent extra identification information required by the system. Source is the identity of the node issuing the tuple or template, Sequence is the corresponding sequence number of the TS operation, State holds the state of the particular entry (e.g. locked, free, etc.) and Type identifies the template as being an in or a rd request.

The operation of the Queue process is simple. Its primary function is to receive requests to add tuples and templates from the Out, In and Rd processes. In general, these items are added to their respective lists and matched against the opposite list

(i.e. tuples against templates and vice versa), and the result of the match is returned to the process invoking the addition. The Queue process must also be able to receive and act upon requests on specific tuple space entries – for example, to lock or delete a specific tuple. Finally, the Queue process is responsible for accessing and retrieving the TS information requested by the Monitor process residing on the host Transputer. Notice that access into the Queue process is restricted by a first-come-first-served policy. When the process receives a request, it blocks out all other interactions until that request has been completed; this is to avoid the contents of TS being accessed by a number of processes concurrently.

5.2.2.7 Interface Process

It should by now be apparent that the Interface process simply redirects traffic from the processes within the node out to the in- and out-sets, and vice versa. It is, however, worth noting that internal structure of the Interface is quite complex. The process comprises 13 concurrent sub-processes – these sub-processes are dedicated to serving

1. each of the Transputer's 4 input and output channels
2. traffic from each of the Out, In, Rd and Challenge Manager processes to the in- and out-sets, and, obviously, traffic in the reverse direction
3. the monitor routing strategy (i.e. redirecting requests for TS inspection from the Monitor process resident on the host Transputer to the required node, then into the Queue process to retrieve the information and, finally, back to the host processor).

5.2.2.8 Discussion

Each of the process modules described above comprise a number of sub-processes (in all, each X-Linda node comprises over 40 concurrent processes) – a diagrammatic overview of the interaction of all of these sub-processes is given in Appendix 5. In the design of the X-Linda node, there were two recurring needs that had to be addressed :

1. Buffering – overall, 20% of the sub-processes running on each node are dedicated to the provision of buffering. The problem of saturation (both over the network and within the nodes themselves) is very real, and could only be effectively resolved by providing process buffers.

2. Atomicity – this problem is more subtle than that described above. Consider the sequence of a node receiving a template, testing for a match against the tuples in TS and then invoking the satisfaction of the request if a match is found. In between these operations being completed, it is possible that some event, for example the addition of a new tuple or template, may occur. If this is allowed to happen, the results can be disastrous (for example, the satisfaction of a request might be invoked from *both* the Out and the In processes). Consequently, it was necessary to enforce atomicity into the processing of tuple space requests to ensure the completion of a particular operation before starting the next.

As discussed in previous sections, the intention in designing the X-Linda node was to provide a software implementation of the Linda Machine node. It is maintained that this objective has been achieved – and, more than this, the objective has been attained in an *elegant* fashion. The design of the X-Linda node is both logical and well-structured, and there is a great deal of modularity inherent in the system.

The Applicability of Occam 2 (revisited)

In section 4.3.2, comment was made on the applicability of occam 2 in the implementation of the X-Linda node. Occam provides a very natural and elegant way of expressing the process interaction detailed in this section – the programming model is *extremely* suitable for this role.

5.2.3 Software Specification

The design and specification of the X-Linda implementation has been detailed at some length in this section. A related issue, the programming methodology applied in the construction of X-Linda, is briefly overviewed below, illustrating an adherence to "good" occam programming style. Finally, the storage requirements of the system (of both source and memory-resident code) are detailed.

5.2.3.1 Programming Methodology

The National Transputer Support Centre, Sheffield [NTSC 1990] recommends the adherence to the following conventions in the interests of enhancing occam programming style –

- correct use of the folding editor as means of implicitly commenting programs, providing logical grouping of blocks of code and providing a natural approach to top-down program development
- the use of channel protocols as a means of providing security to occam programs
- the use of abbreviations for the purposes of improving readability, performance and automating security checking
- providing for easy implementation on different hardware configurations – i.e. the ease with which the number of processors on which the program in running can be increased
- the use of procedures and input/output conventions simply in the interests of enhancing program readability.

It can be claimed with confidence that the programming methodology applied in the implementation of X-Linda closely adheres to the above conventions.

Aside – X-Linda on more than 16 Nodes

Regarding the above point pertaining to increasing the number of processors on which the system is running, it is very easy to switch between 4, 9 and 16 Transputers. However, *it is not known* whether or not X-Linda will function on meshes greater than 4x4. It would be trivial to load the system onto a 5x5 (say) mesh. However, the system has *not* been tested on meshes larger then 4x4; it is quite possible that doing so would require the implementation of additional buffering to prevent network saturation.

It is of interest to note that there is a version of X-Linda running on a single Transputer. The initial development of the system was done on one Transputer, where the physical processors and hard links were simulated in software. The simulated version is not a separate system as such – the X-Linda modules are simply "attached" to a harness that, for the simulated system, launches the nodes in parallel on a single Transputer, and, in the distributed case, physically *places* the nodes and associated communication links on independent processors. Hence, the two versions run *identical* code and have been shown to produce the same results for various application programs. Notice that simulating a 4x4 mesh of processors on a single Transputer causes more than 600 concurrent processes to be launched on that Transputer. The structure of the simulated and distributed versions of X-Linda is shown in Appendix 4.

5.2.3.2 Storage Requirements

The storage requirements pertaining to X-Linda's source and executable code are detailed below. This gives some indication of the magnitude of the system, and also highlights the excessive memory requirements.

Source Code

X-Linda comprises close on 120 source files that occupy more than 300 Kbytes of disk storage. The actual number of lines and bytes taken up the source modules is depicted in Table 5.2.

Process		Lines	Bytes
Host Process		310	9799
	NW Connection	258	7937
	Monitor	1029	28077
		1597	45813
X-Linda Node		1416	44668
	Interface	1799	54049
	Queue	1658	52[illegible]
	Out	705	20589
	In	1545	47140
	Rd	992	28157
	Computation	417	9611
	Challenge Manager	523	15245
		9055	271513
Applications			
	(1) Testing	429	8041
	(2) Efficiency	1572	29324
	(3) Examples	665	13160
		2666	50525
Total		13318	367851

Table 5.2 : Sizes of Source Code Modules

Assuming that blank lines, comments, etc. take up 40% of the source code (which is fairly realistic), the actual implementation is close on 8000 lines long – i.e. a reasonably large system.

Memory Requirements

To get some idea of X-Linda's actual memory requirements, consider the naked system (i.e. *without* the extra storage needed by application programs) running on a 4x4 mesh. Keeping in mind that that each node in the system has access to only 1 Mbyte RAM, some maximum tuple and tuple space specifications are given in Table 5.3, using the following notation :

m – tuple dimension $= \log_2 N$
N – tuple size $= 2^m$
d – bucket dimension $= \log_2 B$
B – bucket size $= 2^d$
S – size of TS $= 4B$

Tuple Length		Bucket Size		size of TS	Storage
m	N	d	B	S	(Kbytes)
4	16	9	512	2048	986
7	128	6	64	256	832
10	1024	3	8	32	923

Table 5.3 : Tuple Space Storage Requirements

The three primary factors contributing to these excessive storage needs are detailed below.

1. Storage of Tuples/Templates

It was shown in section 5.2.2.6 that a significant amount of extra information is required to store tuples and templates (for example, the identification of the node in-

voking the associated operation). Obviously, these extra fields have a significant effect on the amount of storage required, especially when "large" tuple spaces are required (the reduction of the amount of extra information is briefly dealt with in section 8.2.1).

2. Length of Request Queue

For ease of implementation, the lists that hold tuples and templates respectively on each node were specified to be of the same dimension. Now, since the number of requests that can be pending within the system at any given time is bounded by the square root of the number of nodes, it is obvious that the request queue requires fewer entries than the tuple queue. This would consequently reduce the overall storage requirements (this aspect is further addressed in section 8.2.3).

3. Replication of Storage

Each "major" occam process resident on a node – recall that there are 7 such processes : Interface, Computation, In, Out, Rd, Queue and Challenge Manager – is a self-compiled unit. This implies that each process must have its own self-contained storage space. Hence, some degree of storage replication is inevitable (for example, *each* of these processes must have sufficient storage space for (large) tuples). More than this, recall from section 5.2.2.3 that each of these major processes are composed of a number of smaller concurrent sub-processes. Now, it is usually the case that the majority of these sub-process also require access to a specific data structure. To prevent parallel access to this data, each sub-process must be able to access its own copy of the data, and hence must cater locally for the storage of the data.

To conclude this section, it is appropriate to make mention of an implicit and as yet unstated objective of the X-Linda project – i.e. to establish whether or not it is *possible* to implement intermediate uniformly distributed TS on meshes of Transputers. Obviously, the success of this objective has by now been made apparent, and this section, it is hoped, has illustrated the suitability of the design specification in achieving this.

SECTION 6

6.0 ANALYSIS OF EFFICIENCY

This section focuses on the actual research question concerning the communication overheads associated with the X-Linda implementation. The analysis takes the following form :

1. Scheduling Overhead – although not directly related to communication overheads, this is important as it illustrates a source of inefficiency brought about by the scheduling of the numerous processes that reside on the X-Linda node.

2. Out Operation – the overhead of the out operation, focussing on the overhead of out-set traversal, the communication overhead of tuple transmission, the effect of network traffic on the operation and the amount of CPU utilization involved in processing the operation.

3. Rd Operation – the communication overhead of the rd operation, and the effect of network traffic on the processing of the operation.

4. In Operation – the extra overhead of the in operation relative to the rd, and the amount of CPU utilization required to process an in operation. This section also investigates the overhead of tuple contention (i.e. challenging).

5. Data Exchange – the overhead of interchanging information between two processors, relative to

- an X-Linda emulation
- a shortest-path approach on a mesh of processors.

6. Sink Algorithm – the efficiency of a sink algorithm running under X-Linda relative to the same algorithm implemented on a native mesh of processors.

7. TS Search – although not related to communication overhead, this investigation does highlight the inefficiency of X-Linda's tuple matching process, and shows the effect of the scheduling overhead on this operation.

8. Review – an overview of the best and worst case communication overheads.

Notice that the following points apply to the experiments conducted in the section :

1. All the experiments were conducted on a 4x4 mesh – the 2x2 and 3x3 cases were not considered. Recall that the nodes within a 4x4 mesh are arranged and numbered as shown in Figure 6.1.

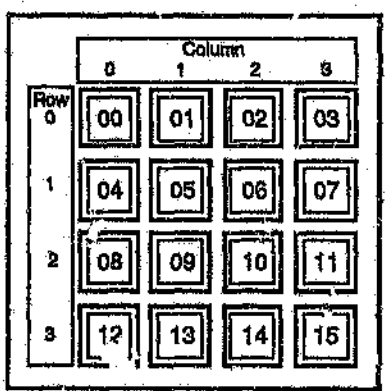


Figure 6.1 : 4x4 X-Linda Mesh

2. The "base" tests (i.e. relative to which the overheads are evaluated) are detailed in Appendix 2.
3. The tests were run a number of times, and the results averaged out to reduce experimental error.

4. The following notation is used throughout –
   N – size of tuple (i.e. number of 32-bit integers)
   m – $\log_2 N$
   Time – Total time in microseconds to perform the experiment
   Rate – Transmission rate = Bytes / Time = 4N / Time.
5. With regard to the evaluation of in and rd, it was required that the extra overhead of the matching process not enter into the results. To this end, the experiments to test these operations were implemented with the smallest possible amount of tuple space. It cannot be guaranteed that the effect of the matching process has been completely eliminated – however, it can be safely assumed that this effect is negligible in relation to the overall communication times.
6. Given that there are k nodes is the mesh, each node has a unique identity, Processor_ID, in the range 0..k-1 (i.e. as shown in Figure 6.1 above). Many of the algorithms presented in this section involve a Processor_ID.

A Note on Communication Requirements

It is, of course, important to be aware of the extra communication imposed by X-Linda on the transmission of tuples and templates. Table 6.1 shows the information that is associated with the transmission of tuples, template requests and tuples that are returned as a result of request satisfaction. All the fields are 32-bit integers unless otherwise specified.

Tuple	Template	Tuple (Request Satisfy)
1. protocol tag	1. protocol tag	1. protocol tag
2. Tuple-Index	2. Template-Index	2. Template-Source
3. Tuple-Source	3. Template-Source	3. Template-Index
4. Tuple-Name	**4. Template-Name**	**4. Tuple-Data** (array)
5. Tuple-Data (array)	5. Template-Count	5. Tuple-Name
6. Home-Id		6. Tuple-Source
7. Tuple-Sequence		7. Template-Count

Table 6.1 : Data Requirements for Tuple / Template Transmission

The fields highlighted in bold-font in the above Table are "necessary". These fields represent the minimum amount of information required in tuple or template transmission; the other fields represent extra "implementation" information. Now, it is important to note that the base experiments (i.e. on native Transputer networks) against which X-Linda's communication overheads are evaluated are concerned only with the transmission of the *minimal* data. Hence, the communication overheads discussed in this section pertain to

- the *extra* information that must be transmitted over the links
- the various delays imposed by the system (the overheads pertaining to scheduling, synchronization and set-up are discussed in the course of this analysis).

To conclude this section, the inherent overheads and inefficiency of the system are discussed, and comment is given on general weaknesses of the implementation and method of evaluation. The overheads imposed by the design and structure of the X-Linda node are addressed and an attempt is made to evaluate the significance of all of the associated system overheads. Finally, a note on the simple and elegant way in which the experiments were implemented under X-Linda is given.

6.1 Process Scheduling Overhead

The overhead of scheduling the 40 odd concurrent processes that reside on each processor can be evaluated simply by measuring the amount of CPU utilization on each processor, given that there is *no* application program running (i.e. on the "naked" system). It was found that the CPU utilization associated with running a

naked implementation was 32% (details regarding the derivation of the CPU utilization figures are given in Appendix 3). This figure of 32% is important, since it

1. represents the lower limit of the range of CPU utilization – i.e. the utilization figures quoted throughout this section should be regarded relative to this base value
2. indicates that there is a high scheduling overhead associated with the system – i.e. "doing nothing" is expensive under X-Linda and a significant amount of processor utilization is taken up simply in the running of the system.

Conversely, it is also necessary to place an upper limit on the range of utilization. A test was conducted whereby each node executed a busy loop. As was expected, each node registered 100% CPU utilization – hence, the derivation of the upper limit against which other utilization figures obtained can be related.

The effect of this scheduling overhead obviously has a significant impact on the overall efficiency of the system. The effect on TS searching in particular is discussed in section 6.7.

6.2 OUT OPERATION

Section 5.1.1 described the approach taken in implementing the out operation, where the node invoking the out waits until the tuple has traversed the entire out-set (i.e. until the tuple returns to the node that outed it). This obviously is the cause of great inefficiency, since the processor invoking the operation is forced to remain idle for a certain amount of time. This section describes experiments that were undertaken to ascertain the amount of overhead involved in processing an out. There are four aspects of the operation that are of particular interest in this evaluation :

1. The extra overhead (i.e. delay) incurred by forcing the node to block until the tuple has traversed the out-set
2. The communication overhead imposed by the system as a whole. Given that the tuple *does* traverse the out-set, how long does this traversal take relative to a native occam 2 implementation of the same operation ?
3. The effects of *other* nodes in the out-set. If the other nodes are busy and creating network traffic, how does this affect the performance of the out operation ?
4. The extra amount of processor utilization needed to process an out operation.

The experiments performed to investigate these issues are described below.

6.2.1 OVERHEAD OF OUT-SET TRAVERSAL

1. Objective

To ascertain the amount of delay imposed by forcing the node invoking an out operation to delay until the associated tuple has traversed the out-set.

2. Design

Two specific tests were performed to measure the time taken for an out operation to complete, given that

1. the tuple traverses the out-set
2. the tuple is simply added to the local node's TS – i.e. not including the time taken for the tuple to be sent to the nodes in the out-set.

3. Results

The tests were performed only for a tuple of one size; 2^{10} integers. The results of the tests are shown in Table 6.2.

Tuple Length		Time		
m	N	With Traversal (1)	No Delay (2)	(1) / (2)
10	1024	36224	3392	10.68

Table 6.2 : Overhead of Out-Set Traversal

4. Observation

Blocking the node invoking the out operation until the associated tuple has traversed the out-set has a significant effect. For the specific size of tuple used in this experiment, the node invoking the operation is forced to remain idle for close on 11 times longer than if it simply added the tuple to local TS and continued processing. This obviously has a great effect on the overall efficiency of the system. A technique for reducing this overhead is discussed in section 8.2.2.

6.2.2 Communication Overhead

1. Objective

To ascertain the communication overhead associated with the out operation – i.e. the overhead of outing a tuple over the out-set relative to a native occam 2 implementation of the same operation.

2. Design

The time taken to complete an out operation was compared with the time taken for an occam 2 program to transmit an integer array around a ring of Transputers (this latter experiment is detailed in section A2.4 (Experiment 4)). Notice that, with regard to the X-Linda experiment, the time taken to complete the out operation was measured at *every* node. The values obtained at each specific node were (as can be expected) fundamentally identical, and the averages of these values are presented below. The values pertaining to nodes in the top row of the mesh must, however, be treated separately, since this row includes the (extra) T414 processor.

3. Results

The tests were performed with tuples (i.e. integer arrays) of various dimension. The results are depicted in Table 6.3, where

- The values for nodes 0..3 are presented separately, since they form the top row of the mesh
- the Base figures correspond to the transmission of an array of integers around a ring of processors using occam 2 (these figures have been extracted from Table A2.6 – notice that the base figures corresponding to Row 0 were measured with an extra T414 in the ring).

Row 0 (Nodes 0..3)					
Tuple Length		Time	Rate		
m	N	X-Linda	X-Linda (1)	Base (2)	(2) / (1)
4	16	2352	0.0272	0.1200	4.41
6	64	4528	0.0565	0.1200	2.12
8	256	13392	0.0765	0.1200	1.57
10	1024	48768	0.0840	0.1200	1.43

Rows 1-3 (Nodes 4..15)					
Tuple Length		Time	Rate		
m	N	X-Linda	X-Linda (1)	Base (2)	(2) / (1)
4	16	2043	0.0313	0.1814	5.80
6	64	3584	0.0714	0.1814	2.54
8	256	10133	0.1011	0.1814	1.79
10	1024	36224	0.1131	0.1814	1.60

Table 6.3 : Out Operation – Communication Overhead

4. Observations

The amount of communication overhead incurred in the transmission of tuples of substantial size is not excessive (for tuples comprising 1024 integers, X-Linda is approximately 1.5 times slower than the base experiment). It is, however, interesting to note the large overhead associated with the transmission of "small" tuples (i.e. about 5 times slower for tuples comprising 16 integers). It is expected that some experimental error has crept into these results (working with very low times

is error-prone). However, this still indicates the presence of a large "set-up" overhead, attributable to

- adding the tuple to local TS, and
- the transmission of the tuple through the various *internal* processes within the X-Linda node before it can be actually sent to the out-set (the issue of the set-up overhead is further discussed in section 6.9.2).

6.2.3 Effect of Network Traffic

1. Objective

To ascertain the effect of network traffic on the out operation. Given that the other nodes in the out-set are busy and making use of their hardware links, how does this affect the tuple's traversal of the out-set.

2. Design

Four nodes (0, 4, 8 and 12) were selected for comparison – they simply performed the out operations in the normal way. The rest of the processors generated network traffic by continually executing ins and outs.

3. Results

Table 6.4 shows the average execution times for out operations invoked from the selected nodes *without* network traffic and with traffic generated as described above. The test was performed for tuples of various dimension, and, as before, the values for Node 0 are presented separately.

Node 0				
Tuple Length		Time		
m	N	No Traffic (1)	Traffic (2)	(2) / (1)
4	16	2368	6336	2.6757
8	256	13440	20672	1.5381
10	1024	48832	78400	1.6055
			Avg	1.95

Nodes 4, 8, 12				
Tuple Length		Time		
m	N	No Traffic (1)	Traffic (2)	(2) / (1)
4	16	2043	6059	2.9657
8	256	10133	13376	1.3200
10	1024	36224	41024	1.1325
			Avg	1.81

Table 6.4 : Out Operation – Effect of Network Traffic

4. Observations

The effect of network traffic, although hardly negligible, is not overly excessive. Notice again that it is with respect to the smaller tuples that the worst overheads are experienced.

Aside – The Effect of Program Behaviour

It is interesting to note, as a side issue, the effect the overall behaviour of the system has on this test. As a variation, an experiment was set up whereby

- Nodes 0, 4, 8, 12 behave as before (i.e. invoke out operations)
- the other nodes generate network traffic. However, instead of simply looping around performing ins and outs on specific tuples, the following algorithm was used (recall that each node in the system has a unique identity, Processor_ID, in the range 0..k-1).

```
define integer array Tuple.Data :
out (Processor_ID, Tuple.Data)
DO i = 0 FOR k
    j <- (k - i - 1)
    in (j, Tuple.Data)
    out (j, Tuple.Data)
END DO
```

This algorithm causes all the nodes to fight for specific tuples, withdraw them from TS and then re-insert them again. It so happens in this specific example that Node 15 "loses out" much of time – i.e. it continually has to re-issue challenges to win a specific tuple, and, ultimately, *all* the outed tuples end up on this node. Now, this has quite an effect on the out operations performed by the other nodes. Using a tuple of dimension 4 (i.e. 16 integers), it was found that the out invoked from Node 12 took more than 1.6 times longer to complete than those invoked from Nodes 4 and 8. Node 12 is in the same out set as Node 15, and, since Node 15 is *very* busy (and generating much network traffic), it slowed down the processing of the operation on Node 12. This is just a small example that illustrates the complexity involved in analyzing Linda program behaviour, which is further addressed in sections 7.4.3 and 8.3. The above algorithm was also run on its own on each of the nodes. Using a tuple of dimension 6 (i.e. 64 integers), the utilization at each node for the execution of this example ranged from 53% to 69%, with an average utilization of 59%. This figure is relatively low, given that the nodes were busy for the duration of the test – i.e. one would expect figures closer to 100%. However, the fact that the utilization is fairly low indicates that the processors are idle for reasonably large amounts of time. This is due to

- the fundamental communication overheads inherent in the system – i.e. the delays incurred in transmitting tuples and tuple requests
- the fact that the specific example chosen forces a great deal of challenges on specific tuples. Many nodes end up losing challenges and have to wait until the tuple is re-issued before they can re-challenge (and possibly lose yet again).

6.2.4 Processor Utilization

1. Objective

To ascertain the CPU utilization associated with processing an out operation.

2. Design

As was done in the previous experiment, four nodes (0, 4, 8 and 12) were selected to perform the out operation – the other nodes were idle. The percentage CPU utilization for each processor was measured for a tuple of dimension 6 (64 integers).

3. Results

The percentage processor utilization for Nodes 0, 4, 8 and 12 (Test Nodes) and the rest of the processors (Others) is shown in Table 6.5.

Tuple Length			% CPU Utilization		
m	N		Test Nodes (1)	Others (2)	(1) – (2)
6	64		47	32*	15

*32 % is the base (i.e. "idle") limit

Table 6.5 : Out Operation – Processor Utilization

4. Observations

1. An extra 15% CPU utilization is needed by the processors invoking the out operation.

2. The rest of the processors (i.e. Others) are *not* completely idle – they must store the tuples in their local TS, and pass the tuples onto the other nodes in the out-set. However, as their CPU utilization is the same as that of an idle processor, it can be deduced that storing and passing on the tuples is very inexpensive.

6.2.5 COMMENT

It has been has shown that processing an out operation is reasonably expensive. The strategy of blocking the node invoking the operation inflicts a tremendous amount of overhead, and the communication overheads involved in transmitting tuples is far from negligible. This serves to emphasize a problem that is inherent in X-Linda – that the implementation of out-sets over point-to-point synchronous links results in a great deal of inefficiency. Obviously, there is a need for communication *buses* in this regard (i.e. as used in the implementation of the Linda Machine). Naturally, this is impossible to achieve using the conventional point-to-point links of the Transputer.

6.3 RD OPERATION

The following series of tests were designed to

1. ascertain the communication overheads associated with the rd operation
2. illustrate the effect of network traffic on this operation.

To measure the time taken to locate and return a tuple, a single tuple was outed from a selected node on the mesh. Each node then, in turn, issued a rd request for that tuple. Node 4 was selected to out the initial tuple – this decision was somewhat arbitrary, and was based primarily on a desire for the selected node not to reside on the outermost rows of the mesh (i.e. it should reside on a row that is "central"). For the sake of clarity, this is illustrated in Figure 6.2.

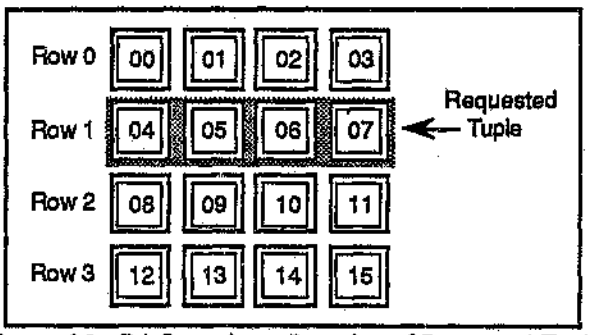


Figure 6.2 : Rd Operation – Location of Requested Tuple

The experiments performed to investigate the communication overhead and effects of network traffic are outlined below.

6.3.1 COMMUNICATION OVERHEAD

1. Objective

To ascertain the communication overhead associated with the rd operation relative to a native occam 2 implementation of the same operation. The way the "same operation" was implemented is described below.

2. Design

As described previously, a tuple was outed from Node 4, and then all the nodes in the mesh consecutively issued rd requests for this tuple. Recall from section 4.3.1.4 that a template request is sent *up* a column, and the requested tuple is returned *down* that column. To ascertain the communication overhead, the *base* test was designed to emulate this behaviour. The base test is described in section A2.3 (Experiment 3) – it entails

1. transmitting a single integer to a node in the row where the tuple is stored – i.e. simulating the sending of the request up the in-set
2. returning an array of integer data – i.e. simulating the retrieval of the tuple.

The distances that the messages have to travel are obviously relevant; these are shown in Figure 6.3 below.

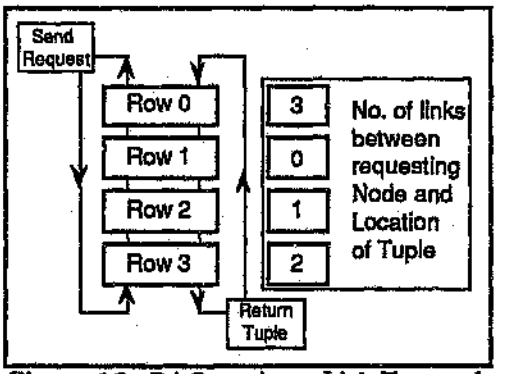


Figure 6.3 : Rd Operation – Link Traversal

3. Results

As can be expected, nodes within each row of the X-Linda mesh yielded fundamentally identical results. These results have therefore been averaged and the figures are presented for the rows as opposed to individual nodes. Table 6.6 shows the times for the retrieval of tuples of various dimension. The Base times listed below have been derived from Table A2.3.

Tuple Length	Row	Time		
		X-Linda (1)	Base (2)	(1) / (2)
m = 4	0	3008	263	11.4373
N = 16	1*	384	-	-
	2	1856	84	22.0952
	3	2432	174	13.9770
			Avg	15.84

Tuple Length	Row	Time		
		X-Linda (1)	Base (2)	(1) / (2)
m = 10	0	28800	16801	1.7142
N = 1024	1*	2752	-	-
	2	14912	5367	2.7785
	3	21840	11106	1.9665
			Avg	2.15

*Requested tuple stored here

Table 6.6 : Rd Operation – Communication Overhead

4. Observations

1. The overheads associated with small tuples (i.e. of dimension 4 in the above Table) are extremely high. These results may be error-prone and should not be taken too literally. *However*, this is again indicative of the massive set-up overheads associated with the implementation.
2. With respect to larger tuples, the results are more reasonable, although the X-Linda implementation is still, on average, more than twice as slow as the base experiment.

6.3.2 Effect of Network Traffic

1. Objective
To ascertain the effect of network traffic on the rd operation. Given that the other nodes in the system are busy and there is network traffic, how does this affect the retrieval of tuples ?

2. Design
Four nodes (0, 4, 8, and 12) were selected for comparison. They simply issued rd requests on a tuple that, as in the previous experiment, was outed from Node 4. The rest of the nodes generated network traffic by continually executing ins and outs.

3. Results
Table 6.7 shows the average execution times of the rd operation for the selected nodes *without* network traffic (extracted from Table 6.6), and *with* traffic generated as described above.

Tuple Length	Node	Time		
		No Traffic (1)	Traffic (2)	(2) / (1)
m = 4				
N = 16	0	3008	5664	1.8830
	4*	384	1600	4.1667
	8	1856	3936	2.1207
	12	2432	4704	1.9342
			Avg	2.53

Tuple Length	Node	Time		
		No Traffic (1)	Traffic (2)	(2) / (1)
m = 10				
N = 1024	0	28800	34944	1.2133
	4*	2752	4992	1.8140
	8	14912	19392	1.2894
	12	21840	27776	1.2727
			Avg	1.40

*Requested tuple stored here – note excessive effect on local requests

Table 6.7 : Rd Operation – Effect of Network Traffic

4. Observations
As for the out operation (section 6.2.3), the effect of network traffic here is not overly excessive. It is, however, worth noting the large overhead imposed on the retrieval of a local tuple. The processing of a local rd request without the effects of traffic is extremely fast relative to the processing of external requests; hence, local requests are far more susceptible to the effects of interference.

6.3.3 Comment

As with the previous experiment (i.e. regarding the out operation), these results exhibit the need for communication via buses as opposed to point-to-point links. In fact, the situation in this case is worse, since, on a 4x4 mesh, the successful completion of a rd operation requires, at worst, 6 transmissions (i.e. 3 to send the request and 3 to return the tuple). Unfortunately, there is no evident way of reducing the communication requirements of the rd operation under X-Linda.

6.4 In Operation

The communication overhead associated with the transmission of an in request and the return of the respective tuple is obviously identical to that of the rd operation. However, it should be obvious that the completion of an in operation takes longer than a rd, since, for every successfully matched in request, a challenge must be sent around the out-set before the tuple can be returned to the requesting node (the challenge process is detailed in section 5.1.3.2).

The objectives of the series of tests described in this section are to evaluate
1. the extra overhead involved in processing an in request (relative to a rd operation)
2. the amount of CPU utilization associated with processing an in request
3. the effect of losing a challenge (i.e. having to re-request a specific tuple).

The approach taken in these experiments is, in essence, identical to that used for the evaluation of the rd operation (refer section 6.3) – Node 4 outs a tuple that is, in turn, requested by all the nodes in the system.

6.4.1 Extra Overhead

1. Objective
To ascertain the extra overhead associated with an in request (i.e. relative to a rd operation).

2. Design
The experiment was performed in exactly the same way as for the evaluation of the rd (refer section 6.3.1). Notice that, in this evaluation, base (i.e. native occam 2) figures were *not* required, since the results are presented relative to those obtained for the rd operation.

3. Results
The times taken to complete an in operation are presented in Table 6.8 below, with the corresponding figures obtained for a rd operation (extracted from Table 6.6).

Tuple Length	Row	Time		
		rd (1)	in (2)	(2) / (1)
m = 4	0	3008	5354	1.7799
N = 16	1*	384	2618	6.8177
	2	1856	4042	2.1778
	3	2432	4711	1.9371
			Avg	3.18

Tuple Length	Row	Time		
		rd (1)	in (2)	(2) / (1)
m = 10	0	28800	31363	1.0890
N = 1024	1*	2752	6317	2.2954
	2	14912	17370	1.1648
	3	21840	24426	1.1184
			Avg	1.42

*Requested tuple stored here – note excessive overhead on local requests

Table 6.8 : In Operation – Extra Overhead

4. Observations
1. As expected, the in operation exhibits some extra overhead relative to the rd. The magnitude of this extra overhead is not excessive, and, as has been seen before, the retrieval of small tuples incurs a greater overhead than the retrieval of larger tuples.
2. Notice the excessive extra overhead associated with local requests (i.e. Row 1 above). Again, this is due to the fact that local rd requests are very fast, and imposing any sort of delay on the processing of a local request has a significant effect.

6.4.2 Processor Utilization

1. Objective
To ascertain the amount of CPU utilization in processing an in request.

2. Design
Using the same approach as before (i.e. all nodes consecutively request a tuple outed from Node 4), the CPU utilization was measured at each node.

3. Results

Recall from section 6.1 that the "idle" (i.e. minimum) CPU utilization was measured to be 32%. The average percentage utilization for each *row* of the mesh is given in Table 6.9, relative to the minimum utilization figures.

Tuple Length	Row	% CPU Utilization		
		Actual (1)	Idle (2)	(1) – (2)
m = 6 N = 64	0	45	32	13
	1*	56	32	24
	2	48	32	16
	3	46	32	14
	Avg	48.75	32	16.75

* Requested tuple stored here

Table 6.9 : In Operation – Processor Utilization

4. Observations

The amount of CPU utilization is similar to that for an out operation (47% – refer Table 6.5). It is significant to note that the utilization on Row 1 is higher than on any of the other rows. Obviously, this is due to the fact that the requested tuple is stored here, and these nodes are responsible for satisfying the requests; hence, they would be expected to do more work.

6.4.3 Effect of Challenging

1. Objective

To ascertain the extra overhead associated with the challenge process – i.e. the amount of delay incurred when a node loses a challenge and has to re-issue a tuple request.

2. Design

As before, tuples were outed from Node 4. Two experiments were conducted where Nodes 14 and 15 then issued requests for tuples such that

1. no challenge would be necessary – i.e. the nodes requested different tuples
2. a challenge would occur – i.e. both nodes requested the same tuple.

This is illustrated in Figure 6.4 below.

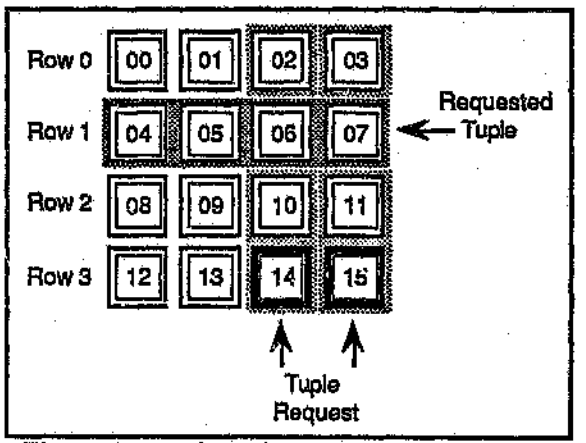


Figure 6.4 : In Operation – Forcing a Challenge

The test in which a challenge was forced to occur was structured so that, on receipt of the respective template, Node 6 would win the challenge and, consequently, Node 7 would lose and have to locate another tuple. Hence, Node 15 would be forced to wait longer than Node 14 to receive the requested tuple.

3. Results

Table 6.10 shows the times taken for Nodes 14 and 15 to complete their in requests. The values are given for the cases where, firstly, no challenge was necessary, and, secondly, when a challenge occurred.

N = 256			
	Time		
Node	No Challenge (1)	Challenge (2)	(2) / (1)
14	10227	11290	1.1039
15	10010	12544	1.2531

N = 1024			
	Time		
Node	No Challenge (1)	Challenge (2)	(2) / (1)
14	24819	27200	1.0959
15	24614	31667	1.2865

Node		Avg
14		1.10
15		1.27

Table 6.10 : In Operation – Effect of Challenging

4. Observations

The challenge process has a significant effect on the nodes involved, irrespective of whether a node wins or loses the challenge. The overhead associated with processing and ultimately winning a challenge is around 10%, while losing a challenge costs a significant 27%.

6.4.4 COMMENT

Not much more can be said about the efficiency of the in operation. Everything that was concluded about the rd operation (section 6.3) is applicable here. It is, however, worth commenting on the fact that the implementation of the tuple deletion protocol is very efficient (i.e. a single traversal of the out-set is needed both to claim ownership of a tuple and to delete it). However, resolving tuple contention is quite expensive. As detailed in section 5.2.2.5, a node receiving a challenge token must search linearly through its own list of "claimed" tuples in order to identify whether or not a challenge must be resolved.

6.5 DATA EXCHANGE BETWEEN PROCESSORS

This experiment investigates the overhead associated with sending a message from a source node to a destination node and back again – i.e. communication between two specific nodes. Achieving the message interchange under X-Linda is simple. Assume that the source node has the identity Source_ID, and the destination has the identity Dest_ID. Then, running the following code on the source and destination processors respectively will cause information to be exchanged :

```
-- source node
define integer array Tuple_Data :
out (Dest_ID, Tuple_Data)
in (Source_ID, Tuple_Data)
```

```
-- destination node
define integer array Tuple_Data :
in (Dest_ID, Tuple_Data)
out (Source_ID, Tuple_Data)
```

The data exchange was performed between a selected source node and *all* the other nodes in the mesh. To remain consistent with the previous experiment, Node 4 was selected as the source node. This node communicated with and received information from all the other processors in turn, and the timing information was gathered here. It is interesting to note that the experiment was designed such that Node 4

also communicates with itself – i.e. it outs a tuple, retrieves that tuple, re-outs it and, finally, re-requests it again. The algorithms running on Node 4 and the other nodes are depicted below (assuming again that each node in the system has a unique identity, Processor_ID, in the range 0..k-1) :

```
-- Node 4
define integer array Tuple_Data :
define integer array time_taken :
DO i = FOR k
    clock ? start
    out (i, Tuple_Data)
    in / rd (4, Tuple_Data)
    clock ? finish
    time_taken [i] <-- (finish MINUS start) * 64 -- microseconds
    --- Delay -- so that the tests do NOT overlap
END DO
```

```
-- all other nodes
define integer array Tuple_Data :
in / rd (Processor_ID, Tuple_Data)
out (4, Tuple_Data)
```

As can be seen from the code segment above, the experiment was tested using both the in and rd operations to retrieve the desired tuples.

Comment – Base Test Comparison

Given the experiment running under X-Linda as described above, some comment must be given on what this algorithm should be tested against – i.e. what should be the base test relative to which the overhead of the implementation can be evaluated ? There are 3 obvious candidates, discussed below :

1. Direct Communication Between Processors

This is the most trivial comparison. Since the X-Linda algorithm, in effect, causes information to be exchanged between two Transputers, it might be argued that the base test should simply constitute transmitting data between two adjacent processors – i.e. across a single link connecting the two Transputers. Support for this argument stems from the idea that, given a *dynamically* reconfigurable network, such a direct connection could be made at any instant that two processors need to communicate. Evaluating the X-Linda implementation against this base test does, however, have serious flaws :

1. It would achieve very little. It is quite obvious that the X-Linda approach is doing more than simply exchanging information between adjacent processors. Hence, we would expect to see a significant amount of overhead; but the magnitude of this overhead would be, to all intents and purposes, meaningless.
2. Dynamically reconfigurable networks are very seldom, if at all, used in practice. The intricacies and overheads associated with setting up and tearing down physical links between processors precludes this approach from standard applications.

Hence, this test is, effectively, worthless as a base comparison. It is not dealt with again in this section.

2. Replicating the Behaviour of the X-Linda Approach

A far more reasonable comparison would be to, using occam 2, replicate *all* of the data transmission needed to effect the exchange under X-Linda. This would give an indication of the *communication* overheads incurred by the system – i.e. the delays in communication imposed purely by running the algorithms on top of X-Linda. This experiment was conducted (section A2.5 – Experiment 5), and a comparison is detailed in section 6.5.1 below.

3. Communicating Over a Mesh Configuration
Another comparison that is worthy of consideration is to use a mesh configuration, and, using occam 2, exchange information between two processors over the *shortest* possible path. The comparison would highlight X-Linda's *routing* overheads. Given that X-Linda and the occam 2 base test are both implemented on a mesh, the comparison would illustrate the difference between using in- and out-sets and the shortest possible path to achieve the data exchange. This experiment is detailed in section 6.5.2 below.

6.5.1 Replicating the Behaviour of X-Linda

1. Objective
To ascertain the communication overheads associated with exchanging information between two processors under X-Linda. This entails taking the data transmissions necessary to complete the operation and comparing the time taken to perform these transmissions using occam 2.

2. Design
This particular base test is described in section A2.5 (Experiment 5). X-Linda's data transmission requirements needed to effect the exchange of information between two nodes were replicated using occam 2. It was shown in this experiment that, from Node 4, either 4 or 8 transmissions are required to send a message to a destination node and back again. For the sake of clarity, the respective transmission times for integer arrays of various dimension (extracted from Table A2.8) are shown again in Table 6.11 :

		Time		
Transmissions	Nodes	N=16	N=256	N=1024
4	0, 5, 6, 7, 8, 12	335	5356	21423
8	1, 2, 3, 9, 10, 11, 13, 14, 15	670	10712	42846

Table 6.11 : Data Exchange – Number of Transmissions

3. Results
From Table 6.11 above it can be seen that the lowest communication requirements (4 transmissions) pertain to the nodes in the second row of the mesh (i.e. Nodes 5, 6 and 7), and the nodes in the first column of the mesh (i.e. 0, 8 and 12) -- obviously Node 4 is not included in these groups. All the rest of the nodes require 8 transmissions to complete the exchange. The actual figures obtained from running the X-Linda implementation reflect this pattern. For every row of the mesh, the time taken to exchange data with Node 4 is the lowest for the first node, and the other nodes in the row all reflect a similar communication time. Hence, in presenting the results, the times pertaining to the last three nodes of any row are presented as an average. The results are given in Table 6.12 under the headings of :

rd -- times taken using the rd operation to retrieve tuples
In -- times taken using the in operation to retrieve tuples
Base -- base test times, extracted from Table A2.8.

N = 16		Time				
Row	Node	rd (1)	in (2)	Base (3)	(1)/(3)	(2)/(3)
0	0	4527	5552	335	13.51	16.57
	1..3	6482	11048	670	9.67	16.49
1	4	2498	4578	–	–	–
	5..7	4080	8156	670	6.09	12.17
2	8	6024	5716	335	17.98	*17.06
	9..11	6246	10689	670	9.32	15.95
3	12	5212	5495	335	15.56	16.40
	13..15	6184	10689	670	9.23	15.95
				Avg	11.6	15.8

N = 256		Time				
Row	Node	rd (1)	in (2)	Base (3)	(1)/(3)	(2)/(3)
0	0	15943	15664	*356	2.98	*2.92
	1..3	28431	34330	10712	2.65	3.20
1	4	11183	13592	–	–	–
	5..7	15448	21055	10712	1.44	1.97
2	8	20757	15612	5356	3.88	*2.91
	9..11	25194	31132	10712	2.35	2.91
3	12	17981	15608	5356	3.34	*2.91
	13..15	25263	31166	10712	2.36	2.91
				Avg	2.7	2.8

N = 1024		Time				
Row	Node	rd (1)	in (2)	Base (3)	(1)/(3)	(2)/(3)
0	0	53862	56231	21423	2.51	2.62
	1..3	97959	102931	42846	2.29	2.40
1	4	39168	42599	–	–	–
	5..7	52324	56142	42846	1.22	1.31
2	8	67584	49166	21423	3.15	*2.30
	9..11	85498	90501	42845	2.00	2.11
3	12	60672	49171	21423	2.83	*2.30
	13..15	85617	90583	42846	2.00	2.11
				Avg	2.3	*2.2

* in faster than .3

Table 6.12 : Data Exchange – Overhead (1)

4. Observations

1. On average, the X-Linda implementation is, at best, more than twice as slow as the base experiment. This illustrates the overhead on the *extra* communication that must be transmitted with tuples and templates, and also the effect of the scheduling overhead.
2. The overhead associated with the exchange of small tuples is huge. This is a common trend exhibited throughout the results presented in this analysis.
3. In some instances, using the in operation is faster than using the rd. This is due to the fact that, in these cases, templates are issued prior to the matching tuples. When a locally outed tuple is matched against a pending in template on a *local* request queue, there is no need to challenge since it is impossible that any other node will try to claim this particular tuple (i.e. the request is satisfied locally). Hence, in these cases, the processing time for rds and ins *should* be equivalent. The fact that they are not is simply due to experimental influence.
4. Notice the results pertaining to Node 4 in the above Table. It is evident that there is a reasonable expense associated with a node communicating with itself. This is an interesting aspect, since a "clever" Linda pre-compiler *should* be able to eliminate this sort of behaviour.

6.5.2 Communicating Over a Mesh Configuration

1. Objective

To compare the overhead of data interchanges between two processors under X-Linda against the shortest path approach – i.e. given a mesh configuration, taking the optimal route between the source and destination nodes. This comparison exposes the overhead of utilizing in- and out-sets on a mesh configuration relative to transmission over the shortest possible path in order to effect information exchange between two nodes.

2. Design

The X-Linda algorithm has already been described in this section. The base test (i.e. communication over the optimal routes) is described in section A2.6 (Experiment 6).

3. Results

The only way to meaningfully compare the X-Linda implementation with the base test is to use an average communication rate – i.e. sum the individual times pertaining to each node and calculate an average rate from this. Table 6.13 shows the average communication rates for tuples of various dimension, using the following notation :

Total Time – Total of the times taken to exchange information for individual destination nodes, excluding the time pertaining to Node 4
Avg Time – Total Time / 15 (i.e. excluding Node 4)
Rate – 4N / Avg Time (i.e. bytes / microsecond)
Base – The rate pertaining to the base test (extracted from Table A2.9)

The results corresponding to the use of rd and in to retrieve tuples are presented separately

rd						
Tuple Length		Time		Rate		
m	N	Total Time	Avg Time	X-Linda (1)	Base (2)	(1) / (2)
4	16	84738	5649.20	0.0113	0.1629	14.38
8	256	314987	20999.13	0.0488	0.1629	3.34
10	1024	1146315	76421.00	0.0536	0.1629	3.04

in						
Tuple Length		Time		Rate		
m	N	Total Time	Avg Time	X-Linda (1)	Base (2)	(1) / (2)
4	16	138509	9233.93	0.0069	0.1629	23.50
8	256	399934	26662.27	0.0384	0.1629	4.24
10	1024	1175040	78336.00	0.0523	0.1629	3.12

Table 6.13 : Data Exchange – Overhead (2)

4. Observations

1. This experiment, which focuses on the overheads associated with routing, is a sterner base comparison than the previous experiment. Here, X Linda is, at best, about 3 times slower than the base test – in the previous experiment, this factor was only two-fold. It can be concluded that the routing performed under X-Linda in order to effect data exchange is vastly inferior to an optimal routing strategy on a mesh of processors.
2. Notice that, yet again, the overhead associated with small tuples is massive. It is felt that these figures are, to some extent, influenced by experimental error. Nonetheless, they are indicative of large set-up overheads.

6.5.3 COMMENT

These experiments have illustrated the overall effects of

- the extra communication necessary to transmit tuples and requests
- the inferiority of routing via in- and out-sets.

It is maintained that the extra communication *can* be reduced (refer section 8.2.1). However, under X-Linda, the routing strategy cannot be altered, and the inefficiency of implementing in- and out-sets over point-to-point communication links cannot be helped.

6.6 OVERHEAD OF A SINK ALGORITHM

A "sink" is a node in a distributed system that collects information from all the other nodes in the system. This experiment describes the X-Linda implementation of a sink, and evaluates the efficiency of the implementation relative to the same operation on a native mesh configuration.

6.6.1 EXPERIMENT

1. Objective

To ascertain the efficiency of a sink algorithm implemented under X-Linda. By comparing the time to execute the X-Linda algorithm with an algorithm implemented on a mesh of Transputers, an indication of X-Linda's communication *and* routing overheads can be obtained.

2. Design

Node 4 was, to retain consistency with the previous experiments, selected to be the sink. It simply collected in information transmitted from all the other nodes in the system. The X-Linda implementation of this algorithm is trivial :

```
-- Node 4
define integer array Tuple_Data :
DO i = 0 FOR k, i ≠ 4
   in / rd (i, Tuple_Data)
END DO
```

```
-- all other nodes
define integer array Tuple_Data :
out (Processor_ID, Tuple_Data)
```

The base test relative to which the above algorithm is evaluated is detailed in section A2.7 (Experiment 7). This test consisted of the implementation of a sink on a mesh configuration whereby Node 4 issued requests to and received information from all the other nodes in the mesh, using the *shortest possible path.*

3. Results

As for the previous experiment, the most meaningful way to compare these results is by using average transmission rates. Again, the following notation is used :

Total Time	–	Time to perform the sink algorithm
Avg Time	–	Total Time / 15 (i.e. excluding Node 4)
Rate	–	4N / Avg Time (i.e. bytes / microsecond)
Base	–	Rate calculated as above for a native occam 2 implementation on a mesh configuration (extracted from Table A2.10).

The results pertaining to the retrieval of tuples of various dimension are shown in Table 6.14. Note that the results obtained with the use of either a rd or an in to retrieve the tuples are presented separately.

rd						
Tuple Length		Time				
m	N	Total Time	Avg Time	Rate (1)	Base (2)	(1) / (2)
4	16	28749	1916.60	0.0334	0.3259	9.76
10	1024	232981	15532.07	0.2637	0.3259	1.24

in						
Tuple Length		Time				
m	N	Total Time	Avg Time	Rate (1)	Base (2)	(1) / (2)
4	16	62084	4138.93	0.0155	0.3259	21.08
10	1024	278591	18572.73	0.2205	0.3259	1.48

Table 6.14 : Sink Algorithm – Overhead

6.6.2 Comment

The overheads of the sink algorithm are significantly less than those associated with data exchange (previous experiment – refer section 6.5). Indeed, for tuples of dimension 10, the results are quite reasonable. This is, to some extent, to be expected, since this experiment is really only concerned with one-way communication. The overhead of the previous experiment, which involved two-way communication, was 2.5 and 2.1 times greater with the use of the rd and the in operation respectively for tuples of dimension 10.

6.7 TS Search Time

A great source of X-Linda's inefficiency lies in the tuple matching process. A template is matched against the tuples in TS by means of a linear search (and the same process is used to match tuples against templates in the request queue). It is of interest to determine how expensive this search operation is and to observe the effect of the scheduling overhead (i.e. to compare the search time with the time taken for the match process running *by itself* on a processor).

6.7.1 Experiment

1. Objective

To ascertain the worst case overhead of searching TS linearly, and to evaluate the effect of the process scheduling overhead on this operation.

2. Design

The time taken to match a tuple against an *empty* request queue was measured – i.e. the *worst* case search time was evaluated. The experiment was tested for bucket dimensions of various size (recall from section 4.3.1.1 that, given there are k nodes in the system, a specific node's TS is sub-divided into $\sqrt{k}$ buckets). The experiment was run under X-Linda, and the match process was then also executed individually in order to ascertain the effect of the scheduling overhead.

3. Results

The times taken to carry out the searches are shown in Table 6.15, using the following notation :

d – bucket dimension = $\log_2 B$
B – bucket size = 2^d
S – size of TS = 4B (i.e. 4 buckets per node)
Base – time taken to search the TS given that the match process is the *only* process running on the Transputer

Bucket		Size of TS	Time		
d	B	S	X-Linda (1)	Base (2)	(1) / (2)
6	64	256	1344	960	1.4000
8	256	1024	5632	3904	1.4426
10	1024	4096	22720	15808	1.4372
				Avg	1.43

Table 6.15 : TS Search Time

Applying linear regression with respect to S and Time yields the following relationships –

X-Linda : Time = S x 5.57 - 74.67
Base : Time = S x 3.87 - 42.67

6.7.2 COMMENT

The time to search TS linearly is naturally O(size of TS) – obviously, a linear search is not an optimal way of locating a specific tuple. It is worth noting that, given an empty TS, the overhead of tuple or template addition includes this extra expense. Notice too that the scheduling overhead increases the search time by a factor of approximately 1.4 – i.e. a significant amount.

6.8 REVIEW

To get some overall feel for the overheads exposed in this section, it is useful to collectively review the results, and, in particular, examine the best and worst case figures. This review is concerned only with communication overheads, and the results pertaining to other aspects (such as CPU utilization and the effects of network traffic) are not dealt with here. The figures of interest are those that reflect the magnitude of the respective overheads relative to the base experiments (i.e. obtained by dividing the results of the X-Linda experiments by those obtained in the base tests). The best and worst case results are presented in Table 6.16, under the following headings (the Table numbers identify the source of the information) :

Out – Table 6.3 (average of Row 0 and Rows 1-3)
Rd – Table 6.6
In – Table 6.8 (X-Linda) and Table 6.6 (Base)
Data Exchange (1) – Table 6.12
Data Exchange (2) – Table 6.13
Sink – Table 6.14

	Best	Worst
Out	1.52	5.11
Rd	2.15	15.84
In	2.20	27.08
Data Exchange (1)		
Rd	2.3	11.6
In	2.2	15.8
Data Exchange (2)		
Rd	3.04	14.38
In	3.12	23.50
Sink		
Rd	1.24	9.76
In	1.48	21.08

Table 6.16 : Best and Worst Case Overheads

These results are presented graphically in Figure 6.5 (Out, In and Rd) and Figure 6.6 (Data Exchange and Sink).

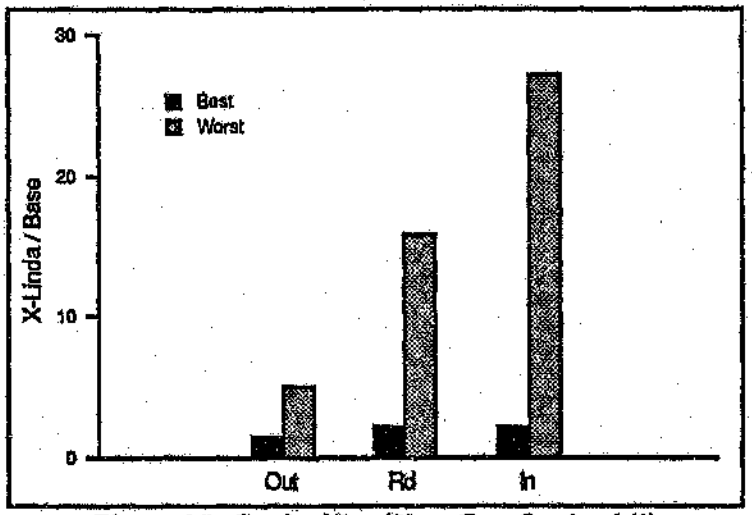


Figure 6.5 : Graph of Best/Worst Case Overhead (1)

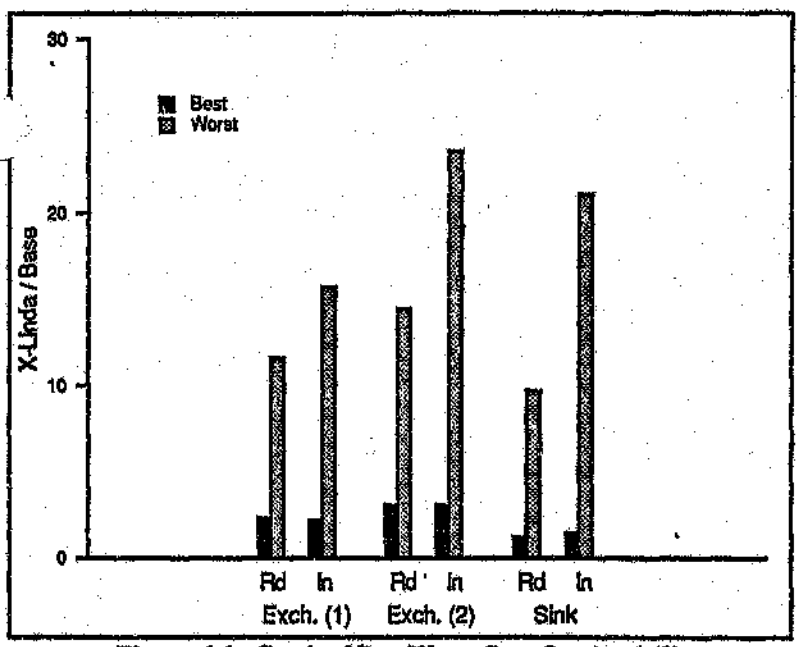


Figure 6.6 : Graph of Best/Worst Case Overhead (2)

The implications of these results are discussed below.

6.9 DISCUSSION

This section has given a great deal of insight into X-Linda's inherent communication overheads and inefficiency, although it is difficult to make a conclusive statement regarding the severity of these overheads. From Table 6.16 and Figures 6.5 and 6.6, it is obvious that the worst case communication overheads (generally associated with the transmission of "small" tuples) are indeed excessive. However, the overheads pertaining to larger tuples (i.e. the best case analysis) are not nearly as severe. Without doubt, X-Linda does impose a significant amount of overhead on the processing of tuple space operations (at best, the rd and in operations are more than twice as slow as the base experiments, and the out is approximately 1.5 times slower). However, given the fact that, according to the Linda philosophy, it

is acceptable to trade some processing performance against the gains provided by the programming paradigm [Ahuja *et al.* 1988], it can be argued that the communication overheads (at least, in the best case situations) are reasonable. This is not to say that X-Linda is the answer to programming Transputer networks. Apart from the communication overheads, there are a host of other overheads associated with the implementation. The process scheduling overhead and inefficiency of TS search have already been discussed, and the synchronization constraints and "set-up" overheads are dealt with in sections 6.9.1 and 6.9.2 respectively.

To conclude this section, comment is given on the
1. weaknesses inherent in the implementation and this evaluation
2. "set-up" overheads induced by the structure of the X-Linda node
3. severity of the respective overheads
4. ease of implementation of the experiments used in the above analysis.

6.9.1 Weaknesses of the Implementation and Evaluation

In general, the weaknesses of the system (i.e. the way in which TS and the primitive operations have been implemented) and consequently the ways in which it can be improved are discussed in section 8 in the context of future research. However, it is worth pointing out here that a fundamental weakness concerning the efficiency of the implementation lies in the synchronization of all the processes residing on the X-Linda node. By way of example, consider a TS primitive process (i.e. In, Out or Rd) wishing to lock a particular entry in TS. In general, a request is sent to the Queue process to perform the operation, and the process invoking the command is blocked until an acknowledgement is received from the Queue process indicating that the operation has been completed. Now, to some extent, this form of blocking synchronization is essential for ensuring atomicity of operations – however, it is felt that this aspect has been carried too far in X-Linda. Returning to the above scenario, consider what happens when the Queue process is busy, adding a tuple to TS for example. Access to the Queue process is restricted by a first-come-first-served policy to ensure that the contents of TS are not accessed in parallel. Hence, the process wishing to lock a tuple space entry must wait until the current operation is complete (i.e. the addition of the tuple and the matching against the template queue) *and* also until its own command has been processed before it can continue. This small example is representative of a whole host of other such synchronization issues that have a similar effect on the performance of individual processes. It is not immediately obvious specifically how the synchronization constraints can be reduced. However, an in-depth investigation of the process interaction would undoubtedly reveal a number of unnecessary constraints.

It is also possible to direct some criticism at the method of evaluating the system (i.e. the way in which the overheads have been ascertained). The analysis has been successful in that it has illustrated the overall overheads and inefficiencies of the system. However, it has told us very little of the specifics of these overheads – i.e. the precise overheads attributable to scheduling, the need to transmit the extra information required to support the system, the routing strategies, the overly restrictive synchronization constraints, etc. (an intuitive idea of the severity of these respective overheads is given in section 6.9.3). There is also some uncertainty regarding the applicability of the base test experiments. Exactly what should the overheads of X-Linda be measured relative to ? The answer to this question is not clear; the best method of evaluating, for example, the efficiency of the In operation is not obvious. Finally, it should be re-emphasized that the results of the base experiments pertaining to the transmission of small amounts of data are prone to error, and can at best be regarded as an approximation.

6.9.2 Structure of the X-Linda Node

As was discussed at some length in a previous section (5.2.3.1), the X-Linda node has been implemented with adherence to the principles of good occam 2 programming style – i.e. the extensive use of concurrent, self-contained, communicating processes. This, however, is the cause of what has been in this section referred to as "set-up" overhead. Recall from section 5.2.2 that the X-Linda node comprises 7 "major" processes (Computation, In, Out, Rd, Interface, Queue and Challenge Manager). Now, take the example of an out operation. Outs are invoked from application programs within the Computation process. The tuple is passed from the application program to a sub-process within Computation that handles out operations. From here, it is sent to the Out process, and then to the Queue process for addition to TS. Finally, the tuple is sent to the Interface process where it travels via a number of internal sub-processes to the Transputer's physical links along which it is ultimately transmitted to the out-set. This traversal of the internal software processes may appear insignificant. However, these processes are themselves time-slicing between various sub-processes, and a significant delay may be incurred between the time that the operation is invoked and when it finally reaches the physical links. Hence, there is evidence of some "set-up" overhead – and this overhead is associated not only with the out operation, but also with the issuing and satisfaction of requests, etc.

6.9.3 Significance of Respective Overheads

At this point, it is appropriate to pass comment on what are believed to be the most serious of the system overheads. The specific overheads that have been revealed in this section are those of communication, process scheduling, TS search, synchronization and set-up. With respect to the communication overhead (i.e. the overhead involved in processing the out, in and rd operations), it is fairly obvious that the greatest cause of inefficiency lies in the blocking of the node invoking the out until the tuple has traversed the out-set. In Table 6.2, it is shown that this causes the addition of the tuple to the out-set to take almost 11 times longer than simply adding it to local TS. Conversely, the implementation of the in and rd primitives is very efficient (this is especially so with regard to the the in, whereby a single traversal of the out-set is required to claim *and* delete a specific tuple).

Evaluating the severity of each of the respective overheads is obviously not easy. However, based partially on the results presented in this section and those pertaining to the execution of the example programs given the next (section 7) – but relying more heavily on intuition – the magnitude of the individual overheads (i.e. in terms of their effect on the efficiency of the system) are estimated to assume the following ranking :

1. TS search
2. communication
3. synchronization
4. process scheduling
5. set-up

The inefficiency of TS search is probably the most significant of these overheads, and it is reasonable to assume that, together, TS search and communication are responsible for the vast majority of the overall overhead. Notice, however, that the value in isolating the respective overheads is questionable since the overheads are generally all closely inter-related. For example, the overheads attributable to process scheduling and synchronization have a direct influence on all of the other overheads.

Ease of Implementation of Experiments

As a side issue, it is worth noting the technique employed in obtaining the majority of the test results presented in this section. Control of the testing and the collection of the relevant results were done using a Linda-like methodology, and, in further support of the claim that Linda simplifies parallel programming (refer section 2.3), this methodology will be briefly discussed here. It is common in the analysis of distributed systems that each node in the system should carry out a specific test procedure, and that these tests should *not* run concurrently (to avoid the effects of the individual tests interfering with each other). Furthermore, there must be some way of, at the conclusion of all the tests, accessing the results obtained at the nodes. Using conventional parallel programming methodologies, this can be a non-trivial exercise – under X-Linda, the process was simple. Assuming again that each node in the system has a unique identity, Processor_ID, in the range 0..k-1, each node executes the following piece of code :

```
Define integer array Tuple_Data :
in (Processor_ID, Tuple_Data)
--- Perform Test
Tuple_Data [Processor_ID] <-- result of test
out (Processor_ID + 1, Tuple_Data)
```

This algorithm is simple, powerful and elegant. The testing process is invoked by placing the tuple (0,Tuple_Data) in TS. Node 0 ins this tuple, performs its test, writes the result to the 0th element of Tuple_Data, and outs the tuple (1,Tuple_Data) – now Node 1 can initiate its test. This sequence continues until all the nodes have, in turn, performed their tests. Finally, there will be left in TS the tuple (k,Tuple_Data), where Tuple_Data [i] contains the test result pertaining to Node i. As indicated previously, this synchronization of processors and subsequent collection of results would not nearly be so straight forward on a native Transputer system, and this simple example highlights the power and elegance of the Linda approach.

As is discussed in the conclusions to this document (section 9), it is maintained that, despite the inefficiency and overheads exposed, the X-Linda approach requires a great deal more investigation. Possible ways of enhancing the efficiency are addressed in section 8, and there is great potential for future research in this area. The following section ties all of the results presented in this section together by evaluating the efficiency of application programs running under X-Linda – i.e. by illustrating the severity of the *overall* effect of these overheads on the performance of the system.

SECTION 7

7.0 EXAMPLE PROGRAMS

This section details a selection of example programs that were implemented under X-Linda in order to

1. test and verify the system (obviously, the verification was not based purely on the results of these example programs – a wide range of algorithms were run through the system in order to verify the maintenance of TS consistency under various conditions and circumstances)
2. observe the efficiency (or, as it turned out, the lack thereof) of the system.

Specifically, algorithms to perform numerical integration, matrix multiplication and sorting were designed and implemented. These algorithms (with the exception of the sort) are presented and evaluated in some detail below. Although the description and analysis of the problems is not directly within the scope of this research, it is maintained that they have great value in illustrating :

- that "real" problems can be solved using X-Linda
- how the X-Linda programming methodology differs from conventional Linda approaches
- certain aspects of the model that are the cause of vast inefficiency.

The above points are sufficient motivation for including the algorithms here. Furthermore, it is believed that the introduction of some degree of "practicality" into the research is of interest and importance. It must be noted at the outset of this section that, unlike the experiments conducted in the previous section, the overhead of TS search has a direct influence on the execution of these programs. The algorithms require a certain amount of TS storage, and the inefficiency associated with searching this space linearly (discussed in section 6.7) obviously has an effect on the overall efficiency of the execution.

Host Process

In all of the X-Linda algorithms illustrated below, it should be noted that one specific node in the mesh was chosen to perform a double function – i.e. that of the host and of a worker process. This is because the host *Transputer* in the system is itself not equipped to perform X-Linda operations (this is discussed in detail in section 8.2.5). Selecting a specific node to perform this double function was somewhat arbitrary – with the restriction that the node chosen should not exist within the top row of the mesh (i.e. should not suffer the extra overhead of communicating through the host Transputer). For consistency, the first node in the second row of the mesh (i.e. the node with identity equal to the dimension of the mesh) was chosen throughout to act both as the host and as a worker process.

The algorithms are evaluated with respect to their execution time and the percentage CPU utilization. This analysis highlights aspects of interest with regard to the efficiency of the system. To conclude this section, some observations are made regarding

- the cause of the system's apparent inefficiency
- the ease of design and implementation of the algorithms
- general program behaviour.

The full code listings for these examples are located in Appendix 6.

7.1 NUMERICAL INTEGRATION

Numerical integration is the type of problem that must surely be a favourite with marketers of distributed systems as a means of illustrating the efficiency of their particular system. The parallelization of this algorithm gives rise to an implementation that features massive computation with minimal communication requirements. Hence, *any* distributed system can be expected to give impressive results for this problem. The example is included here since

1. it is a good example of a "perfect" parallelizable algorithm that is sure to show impressive speed-up
2. it illustrates the implementation of a "real" problem under X-Linda.

7.1.1 ALGORITHM

It was decided to implement numerical integration using the Trapezoidal Rule, since this method is exceptionally simple.

Trapezoidal Rule

The following definition of the Trapezoidal Rule has its source in Spiegel [1974].
To evaluate the integral

$$\int_a^b f(x).dx$$

sub-divide the interval [a, b] into n equal parts of length $\Delta x = (b-a)/n$.
Denote $f(a+k\Delta x) = f(x_k)$ by y_k, $k = 0, 1, 2, ..., n$. Then

$$\int_a^b f(x).dx \approx (\Delta x/2) \times (y_0 + 2y_1 + 2y_2 + ... + 2y_{n-1} + y_n)$$

The parallelization of this algorithm is straight forward. Each worker is simply given a sub-range of the integral to compute, and the integral is then calculated as the sum of these sub-results.

X-Linda Algorithm

The algorithm, as implemented using X-Linda, is given below – a full code listing for this example is given in Appendix 6. Given that the mesh comprises k nodes, then each processor is uniquely assigned an identity, Processor_ID, in the range 0..k-1. The host process, which, as mentioned previously, is implemented on a worker node, outs the limits of the sub-ranges that each worker will be responsible for :

```
define n, a, b :
delta_x <-- (b-a)/n
sub_interval <-- n/k
DO i = 0 FOR k
    interval_start <-- i * sub_interval
    out (i, [a, delta_x, sub_interval, interval_start])
END DO
```

The workers receive the sub-ranges corresponding to their Processor_IDs, perform the calculation, and out the sub-result. Recall from section 2.1.1 that a "?" preceding a variable name indicates a *formal* parameter, which is assigned the value of the corresponding *actual* parameter in the requested tuple. The operations carried out by the workers are specified as follows :

```
in (Processor_ID, [?a, ?delta_x, ?sub_interval, ?interval_start])
result <-- 0
DO k = interval_start FOR sub_interval
    result <-- result + 2f(a + k.delta_x)
    -- if k = 0, replace 2f(...) by f(...)
END DO
out (Processor_ID + 1000, result)
-- note naming convention - to distinguish data tuples from result tuples
```

The host then collects and sums the tuples with name "Processor_ID+1000" :

```
result <-- 0
DO i = 0 FOR k
    in (i + 1000, ?sub_result)
    -- recall : result tuples are named "i + 1000"
    result <-- result + sub_result
END DO
```

It must also compute and add y_n, and multiply the total by $\Delta x/2$.

7.1.2 EVALUATION

The efficiency of the algorithm running under X-Linda is evaluated below with respect to execution time and CPU utilization.

7.1.2.1 Execution Time

The algorithm was tested for a specific example, where

$f(x) = (x / 2^{13}) - 1$
$n = 2^{22}$
$a = -n/2 = -2^{21}$
$b = 3n/2 = 3 \times 2^{21}$

Notice that exceedingly large values for n, a and b were chosen to maximize computation. Using 4 processors, each node must evaluate the function 2^{20} times, whereas using 16 processors, the function is computed 2^{18} times. The algorithm was implemented in a variety of environments :

Sequential – Sequential occam 2 implementation on a single T800
Simulated – 4x4 X-Linda mesh – *simulated* on a single T800
2x2 – 2x2 X-Linda mesh – physically distributed (4 processors)
4x4 – 4x4 X-Linda mesh – physically distributed (16 processors).

Table 7.1 shows the time taken (in microseconds) for these algorithms to execute. The relative speed-ups (obtained by dividing the figures corresponding to the top row of the Table by those down the left-hand side) are also shown.

Time			Speed-Up			
			Sequential	Simulated	2x2	4x4
Sequential	37451520		1			
Simulated	*59113280		0.63	1		
2x2	12785645		2.93	4.63	1	
4x4	4975514		7.53	11.88	2.57	1

* Includes time to draw Monitor Screen

Table 7.1 : Numerical Integration – Execution Times and Speed-Up

Some interesting information can be drawn from these results :

1. Simulated is 1.6 times slower than Sequential. Even disregarding the extra time taken to write to the screen, this still indicates some inherent inefficiency
2. 2x2 is 2.9 times faster than Sequential. Calculating the efficiency [Quinn 1987] of the algorithms as $\frac{\text{Time (Sequential)}}{4 \times \text{Time (2x2)}} \times 100$
   we see that the algorithm is ~73% efficient
3. 4x4 is 7.5 times faster than Sequential, and 2.6 times faster than 2x2. Notice, however, that the efficiency (relative to Sequential) is, in this case, only 47%.

The fact that some significant speed-up was attained for this algorithm is no great cause for celebration. The nature of the example chosen suggests that speed-up is inevitable – i.e. something would have to be *seriously* wrong for speed-up *not* to occur. Conversely, notice that the efficiency of 4x4 is a poor 47% – this, with the results that are presented with the next example (matrix multiplication) illustrates the model's inherent *lack* of efficiency. Factors contributing to this lack of efficiency are discussed in 7.4.1.

7.1.2.2 CPU Utilization

It is of interest to examine the percentage CPU utilization associated with the execution of this algorithm (the derivation of the CPU utilization figures is explained in Appendix 3).

16 Processors (4x4 mesh)

Table 7.2 shows the percentage CPU utilization for 16 processors (i.e. 4x4 mesh). Notice that the structure of the Table reflects the mesh itself – the identity of the node appears above the CPU utilization figure.

		Column			
		0	1	2	3
Row	0	00 99	01 99	02 99	03 98
	1	04 89	05 94	06 92	07 89
	2	8 80	09 82	10 76	11 74
	3	12 78	13 69	14 67	15 64

Table 7.2 : Numerical Integration – Processor Utilization (4x4 mesh)

In this case, the host process was resident on Node 4 – i.e. this node was also responsible for putting the interval information into TS and collecting the results. Notice that the utilization figure for Node 4 applies only to the worker process. This figure would be significantly lower if the execution of the host process was also taken into account. Some interesting observations can be made with regard to these figures :

1. The percentage CPU utilization progressively decreases over the processors in the mesh. The reason for this is obvious – recall that the host process outs the sub-intervals sequentially; i.e. starting with the information for Node 0 and ending with Node 15. Hence, Node 0 can start its computation almost immediately, whereas Node 15 has to wait for a reasonable amount of time before its information becomes available.
2. The utilization on Row 1 (Nodes 4, 5, 6 and 7) is relatively high. These nodes retrieve tuples that are stored *locally*, and therefore do not have to wait for the information to be sent from external sources. The (relatively) low utilization on Node 4 is inexplicable. It has been stated already that the execution of the host

process was disregarded, and the utilization was measured only with respect to the worker. When the worker was invoked, it should have found its requested information locally, and immediately started computation – i.e. one would expect a higher degree of CPU utilization.

3. The utilization in Column 0, excluding Node 4 (i.e. Nodes 0, 8 and 12) is relatively high. There is a good reason for this. The templates issued from these nodes are sent to the in-set, which, for these nodes, includes Node 4 (the host process). Hence, as soon as the requested tuples are outed by the host, they are matched against pending templates in the local (i.e. Node 4's) request queue and satisfied immediately. Therefore, there is no delay in waiting for the tuple to traverse the nodes in the out-set before finding a match.

4 Processors (2x2 mesh)

In the case of the 2x2 mesh, *all* 4 processors measured 100 % CPU utilization (!). This reflects the effect of the communication overheads. For meshes of larger dimension, the communication overhead increases significantly, and processors spend more time waiting for information (i.e. idle).

7.2 MATRIX MULTIPLICATION

Matrix multiplication is an example that is frequently cited in the literature (e.g. [Ahuja *et al.* 1986 and 1988], [Carriero *et al.* 1986] and [Wentworth 1989]) as a means of illustrating how a "real" algorithm can be implemented using Linda. Indeed, it has come to be regarded as a "classic" Linda problem. It was decided to included this example here since :

1. as indicated above, it is well-known and understood in the context of Linda-based implementations
2. like the previous example, it serves to illustrate the implementation of a "real" problem on the X-Linda system.

7.2.1 ALGORITHM

The standard, or classic approach to this problem is simple to describe – hence, although this is not directly relevant, it is outlined briefly below. As will be seen later, it is of interest to compare the "classic" approach with the X-Linda equivalent.

"Classic" Algorithm

This algorithm is based on those presented by Ahuja *et al.* [1988] and Wentworth [1989]. Assume that two nxn matrices, A and B, are to be multiplied together to create a resultant nxn matrix, C. The host process outs the rows of A and columns of B into TS as follows :

```
DO i = 0 FOR n
    out ("A", i, <ith row of A>)
    out ("B", i, <ith column of B>)
END DO
```

It also outs "instructions" telling the worker processes that inner-product operations must be performed on these rows and columns :

```
DO i = 0 FOR n
    DO j = 0 FOR n
        out ("compute IP", i, j)
    END DO
END DO
```

The workers loop forever, reading in the rows and columns, computing inner-products, and outing the corresponding results :

```
DO forever
    In ("compute IP", ?i, ?j)
    rd ("A", i, ?A_row)
    rd ("B", j, ?B_col)
    ip <-- Inner_Product (A_row, B_col)
    out ("result", i, j, ip)
END DO
```

To collect in the results of the computations, the host simply Ins all of the "result" tuples (irrespective of the order in which they become available) :

```
DO k = 0 FOR (nxn)
    In ("result", ?i, ?j, ?ip)
    C [i,j] <-- ip
END DO
```

X-Linda Algorithm

Since X-Linda tuples comprise only 2 fields (refer section 4.3.1.2), *viz.* an integer name and an array of integer data, it is obvious that the algorithm as described above could not be directly implemented. The problem had to be modified to fit the constraints of the tuple structure. The algorithm is outlined below, and the full code listing may be found in Appendix 6. Notice that, once again, one worker node must double as the host. As in the classic approach, the host process outs the rows of A and the columns of B :

```
DO i = 0 FOR n
    out (i, <ith row of A>)
    out (i + n, <ith column of B>)
    -- use "i + n" to distinguish between the name of tuples pertaining to A and B
END DO
```

The strategy at each worker is as follows. The workers rd the *entire* B matrix, and then, for each selected row of A, compute and out an *entire* row of the result matrix. Although this differs vastly from the classic approach, it is probably the most logical and efficient method available given the tuple structure constraints. The strategy for determining which row of A should be accessed by a specific worker is not complicated. Initially, each worker Ins the row that corresponds to its processor identity (as in the previous example, assume that the mesh comprise k nodes, where each processor is uniquely assigned an identity, Processor_ID, in the range 0..k-1). After each computation, each worker simply adds k to the index that identifies the row to be accessed. Hence, the workers are guaranteed that they will access distinct rows of the A matrix. This is expressed by the following :

```
-- read entire B matrix
DO i = 0 FOR n
    rd (i + n, ?B [i]) -- recall : ith column of B is named "i + n"
END DO

-- computation
A_Index <-- Processor_ID
DO WHILE A_Index < n
    rd (A_Index, ?A_row)
    DO i = 0 FOR n
        C_row [i] <-- Inner_Product (A_row, B [i])
    END DO
    out (2n + A_Index, C_row) -- ith row of C is named "2n + i"
    A_Index <-- A_Index + k
END DO
```

To collect in the results (i.e. the rows of the C), the host ins the tuples with the name "2n + i" :

```
DO i = 0 FOR n
    in (2n + i, ?C [i])
END DO
```

Comparing this (i.e. X-Linda) algorithm with the classic approach, we see that, *in essence*, they are similar. The classic approach, however, exhibits a far finer grain of parallelism, and the storage requirements of the X-Linda solution are far higher.

7.2.2 Evaluation

The efficiency of the algorithm running under X-Linda is discussed below, again with respect to execution time and CPU utilization.

7.2.2.1 Execution Time

As will be seen in this section, the efficiency of the system is appalling – the algorithm takes *longer* to execute as *more* processors are added ! A specific example was used as a test case – the multiplication of two 32x32 matrices – on meshes of 4 and 16 processors. From the description of the algorithm given earlier, it should be obvious that, in the 4 processor case, each worker computes 8 rows of the result matrix and, in the 16 processor case, each worker is responsible for only 2 rows. This algorithm was also implemented sequentially on a T800 Transputer, and on a 2x2 X-Linda mesh *simulated* on a single T800. As for the previous example, the following notation is used to define to various algorithms :

Sequential – Sequential occam 2 implementation on a single T800
Simulated – 2x2 X-Linda mesh – *simulated* on a single T800
2x2 – 2x2 X-Linda mesh – physically distributed (4 processors)
4x4 – 4x4 X-Linda mesh – physically distributed (16 processors).

Table 7.3 shows the time taken (in microseconds) for these algorithms to execute. The relative "speed-ups" (obtained by dividing the figures corresponding to the top row of the Table by those down the left-hand side) are also shown.

Time			Speed-Up			
			Sequential	Simulated	2x2	4x4
Sequential	345728		1			
Simulated	*1505984		0.23	1		
2x2	566379		0.61	2.66	1	
4x4	1487526		0.23	1.01	0.38	1

* Includes time to draw Monitor Screen

Table 7.3 : Matrix Multiplication – Execution Times and Speed-Up

The inefficiency of the system is self-evident – the results of Table 7.3 can be restated as follows :

1. Simulated is 4.4 times slower than Sequential. Although the time taken to write to the screen must be taken into consideration, this still gives an indication of the inefficiency of the model running on a single processor
2. 2x2 is 1.6 times slower than Sequential *and* 2.6 times *faster* than 4x4
3. 4x4 is
   - 4.3 times slower than Sequential
   - approximately the same as Simulated
   - 2.6 times slower than 2x2.

Factors contributing to these poor results are outlined in section 7.4.1.

7.2.2.2 CPU Utilization

Some interesting observations can be made with regard to the respective CPU utilization of the processors for this example.

16 Processors (4x4 mesh)

Table 7.4 shows the percentage CPU utilization for 16 processors (i.e. 4x4 mesh).

		Column			
		0	1	2	3
Row	0	00 38	01 46	02 46	03 46
	1	04 74	05 74	06 74	07 74
	2	08 40	09 48	10 48	11 48
	3	12 39	13 47	14 47	15 47

Table 7.4 : Matrix Multiplication – Processor Utilization (4x4 mesh)

As in the previous example, the host process was resident on Node 4. There are 3 important observations that can be made regarding these figures :

1. The utilization of the nodes in Row 1 of the Table (i.e. Nodes 4, 5, 6 and 7) is far higher than for any of the other rows. This is because these nodes are able to access the matrix information in TS *locally* – they do not have to wait for the requested tuples to be sent from other nodes in the mesh.
2. The utilization of the nodes in Column 0, excluding Node 4 (i.e. Nodes 0, 8 and 12) is significantly less than any of the other utilization figures. It is suspected that this is because the templates issued from these nodes travel through Node 4, and, since Node 4 is the busiest node in the mesh, these requests take longer to be satisfied.

   Note : in the previous example, Nodes 0, 8 and 12 exhibited relatively high utilization. This is a prime example of how different program behaviour can have a dramatic effect on processor efficiency. In the previous example, the host did very little work (i.e. outed a relatively small number of small tuples); in this example, it does far more.

3. Row 0 (Nodes 0, 1, 2 and 3) has the lowest CPU utilization. Although the difference is not significant, the fact that this row also comprises a (slow) host T414 Transputer and therefore suffers the extra communication overhead obviously has some effect.

4 Processors (2x2 mesh)
Consider now the case of the 2x2 mesh. Table 7.5 shows the percentage processor utilization :

		Column 0	Column 1
Row	0	**0 0** 42	**0 1** 53
	1	**0 2** 75	**0 3** 85

Table 7.5 : Matrix Multiplication – Processor Utilization (2x2 mesh)

Here, the host processor was implemented on Node 2. The trend of the figures is similar to those for the 4x4 mesh and will not be re-analyzed. It is important, however, to note that these figures are proportionally *higher* than those for the 4x4 mesh. This is significant, and is again indicative of the system's communication overheads.

7.3 SORTING
A sort algorithm, based on a distributed dimensional collapse [Faasen 1987], was also implemented using X-Linda. The algorithm had previously been implemented on a native Transputer network in occam 2, and showed significant speed-up as more processors were utilized. Little would be gained by presenting the algorithm and analyzing the result of the execution under X-Linda here. Suffice to say that, as for the matrix multiplication example, the execution time *increased* with the size of the mesh. Observations regarding the implementation and efficiency of the example programs are given below.

7.4 OBSERVATIONS
Some factors contributing to the inefficiency of the algorithms executing under X-Linda are presented below, and general note regarding the design and implementation of the problems is given. A short discussion on general Linda program behaviour is given in conclusion.

7.4.1 FACTORS CONTRIBUTING TO INEFFICIENCY
The following issues are all related to the general lack of efficiency exhibited in execution of the algorithms. It is important to note these factors, since they highlight general sources of inefficiency within the system as a whole.
1. Tuple Matching
It was indicated at the outset of this section that the execution of these algorithms is influenced by the inefficiency of the tuple matching process. It is not unreasonable to assume that the utilization of TS partitioning (as featured in, among others, the Rhoda implementation discussed in section 3.3.4) would have a significant effect on the overall efficiency of the system.
2. Host Processor
As has been described at length in various places throughout this document, the top row of the X-Linda mesh incorporates an extra processor – a (slow) T414. Hence, communication through this row incurs extra overhead.

3. Host Process
In all the examples described, the host process is implemented on a worker node – i.e. this node performs a double function. This is guaranteed to affect the efficiency of the system.
4. Using ins instead of rds
The in operation is slower than the rd (refer section 6.4.1). In the interests of efficiency, rds could, in most instances, have been used instead of ins. This was not done in order to conserve some of the true "flavour" of Linda programming. Furthermore, ins "clean up" TS – had rds been used instead of ins, a tremendous amount of extra TS storage would have been necessary.
5. Overheads incurred by the size of the Mesh
Increasing the dimension of the mesh (for example, from 2x2 to 4x4) causes

- the length of the in- and out-sets to be increased. Hence, the times taken to perform an out operation and to locate and retrieve specific tuples are increased accordingly
- the amount of local tuple space on each node (and, consequently, the amount of search time required to locate a tuple) to be increased. This is especially relevant with regard to the matrix multiplication example. For this specific example (i.e. multiplying two 32x32 matrices), each node maintains a local tuple bucket of 128 entries. Recall from section 4.3.1.1 that local tuple space is subdivided into $\sqrt{k}$ buckets, where $\sqrt{k}$ is the number of nodes in a particular out-set. A node's local TS for a 2x2 mesh therefore comprises 256 tuples, where nodes in 4x4 mesh maintain 512 entries. Hence, the worst case search time in the 4x4 case is double that of the 2x2 case.

7.4.2 Ease of Implementation
Disregarding the efficiency issues, it is worth making the point that the actual design and implementation of the algorithms under X-Linda was *extremely* easy. Personal experience with distributed systems, and, in particular, Transputer-based systems has shown that the issues of processor synchronization and communication add a large degree of complexity to algorithm design. The experience of programming under X-Linda has, without doubt, verified the claim (refer section 2.3) that Linda, in general, eases the burden of writing parallel programs, and provides an easy and natural approach to parallel algorithm design and implementation.

7.4.3 Program Behaviour
The behaviour of Linda application programs on a given system is a field of study on its own, and has much valuable research potential. A thorough understanding of program behaviour would give important insight into system design. Merely touching on the surface of this issue, it is of interest to compare the percentage CPU utilization figures for the numerical integration example (Table 7.2) with those for the matrix multiplication example (Table 7.4). The two algorithms are somewhat similar in that the host outs information, the workers receive this data, perform some computation and out a result, and the host collects the sub-results. The percentage CPU utilization figures associated with the two algorithms are significantly different – i.e. they are behaving in fundamentally different ways. The whole issue of program behaviour is of great importance, and it is believed that this area will feature prominently in future Linda-oriented research (this topic is addressed briefly at the end of the next section in the context of future research).

SECTION 8

8.0 ENHANCEMENTS AND FUTURE RESEARCH

There is a good deal of potential for future research with regard to the X-Linda project. In this section, a number of ways of enhancing the system and its performance are discussed; these enhancements obviously fall into the scope of future research. The proposals for improving the system are presented under two broad categories :

1. enhancing the performance of general distributed-memory Linda implementations – obviously, these approaches are by default applicable to the X-Linda system itself
2. enhancements specific to the X-Linda implementation – here, approaches to generally improving the system as well enhancing its performance are discussed.

Finally, a short discussion regarding future research in the area of analyzing program behaviour is given.

8.1 GENERAL ENHANCEMENTS

There are two obvious ways of enhancing the system performance in general –

1. using dedicated processors to handle communications and tuple space management
2. speeding up the matching process (obviously, this issue is applicable to *any* Linda implementation – not specifically distributed-memory systems).

These issues are discussed in more detail below.

8.1.1 DEDICATED HARDWARE SUPPORT

It is evident that the primary sources of X-Linda's inefficiency are the overheads of communication and the matching process, and it is reasonable to assume that these problems are common to the majority of distributed-memory implementations. A technique that can be employed to speed up matching is discussed in section 8.1.2. However, the greatest gain in efficiency would necessitate the utilization of dedicated processors to handle communication and TS management. The Linda Machine (refer section 3.2) makes use of dedicated processing power – and this approach has been addressed in connection with a proposed Transputer-based implementation (section 3.3.2). Adopting the philosophy related to the latter proposal (i.e. that Transputers are cheap and easily available), the approach is certainly feasible. It is envisaged that each node in the mesh would actually comprise 2 processors – one dedicated to communicating with the in- and out-sets, and one on which TS and TS management would be implemented (or, going a step further, implementing a large TS over a *number* of processors). This would greatly reduce the overheads of communication and of the matching process. It is maintained that this approach *must* receive further investigation. The benefits of the programming model, it is believed, far outweigh the extra cost of providing processing power.

8.1.2 SPEEDING UP THE MATCHING PROCESS

In section 2.1.4, the fact that TS search may inherently seem to be a source of great inefficiency was briefly mentioned. There are, however, ways of speeding up this process. Leler [1990] describes such a method; that of dividing TS into sub-sets. A Linda preprocessor can be built in order to identify the usage of tuples in a particular program. Tuples with similar characteristics can be grouped together in memory, hence reducing the amount of TS searching at run-time. This technique of effectively dividing the TS into subsets is utilized in the Rhoda implementation (refer section 3.3.4). Preprocessors can not, however, be used in conjunction with inter-

preted languages, nor with compiled languages where the programs that manipulate the TS are compiled independently.

8.2 ENHANCING X-LINDA

The scope for providing enhancements to the X-Linda implementation is vast. Some obvious ways in which the system could be improved (besides the general strategies covered above) are discussed below and it is hoped that these proposals will be addressed in the course of future research. The issues discussed below relate both to improving the efficiency of the system, and to providing a more useable system.

8.2.1 REDUCING STORAGE AND COMMUNICATION

The storage and communication requirements of the system are huge. A detailed analysis of these requirements is needed in order to ascertain what information is actually necessary (i.e. opposed to extra or redundant information that is present as a result of ease of implementation or oversight). Take, for example, the storage and communication requirements pertaining to a single tuple. The extra information required to store a tuple is shown in Table 5.1, and Table 6.1 details the extra information that is needed in the transmission of a tuple. Some of this extra information can, undoubtedly, be discarded. For example, storing the identity of the node issuing the tuple is redundant since this can be computed from the tuple's location in tuple space. This is just one small example of how the amount of extra information can be reduced. No doubt a re-write of X-Linda would expose many more such instances.

8.2.2 UNBLOCKING THE OUT OPERATION

In section 5.1.1.1 the technique of blocking the processor invoking an out operation until the associated tuple had traversed the out-set was discussed. This was done to prevent network saturation and subsequent deadlock. This has a dramatic effect of the efficiency of the out operation. It is difficult to eliminate the need for blocking altogether – however, it is certainly possible to reduce it. One possible way of doing this would be to provide network buffering for a reasonable number of outed tuples. Then the consecutive outing of tuples up to this limit would be unhindered (i.e. the processor would not block). Only once this limit was reached (and if the previous tuples had not yet been through the entire out-set) would blocking be enforced to prevent saturation. This would have a significant effect on the overhead of the out operation. Alternatively, S. Hazelhurst [personal communication, Jan. 1991] suggests a scheme whereby the node invoking the out would be allowed to continue processing, and only block when attempting to perform a subsequent out operation.

8.2.3 REDUCING THE LENGTH OF THE TEMPLATE QUEUE

It was seen in section 5.2.3.2 that the length of the template queue on any given node is defined to be of the same length as the tuple queue. This is obviously not necessary, and much of the system's efficiency is lost through the searching of this overly-long list. The number of requests that can be pending within the system at any given time is bounded by a function of the number of nodes. Hence, the template queue could be greatly shortened. This would consequently reduce the overhead of the matching process, as well as the overall storage requirements.

8.2.4 PIPELINING TRANSMISSION

When passing messages of significant length between Transputers, it is possible to, instead of sending and receiving the entire message in its entirety, break up the message into segments, or packets. This is of great importance with regard to pipelined configurations, where a message must be passed consecutively along the

processors in the pipe. By breaking up the message into smaller segments, it is possible for a Transputer to *concurrently* receive and transmit packets on its hardware links. Lakier [1989b] illustrates that this has a *tremendous* effect on the overall rate of communication through a pipeline of processors. This technique has an obvious application in X-Linda. The effect of using the approach in the transmission of tuples over the out-set and in returning tuples to requesting nodes over the in-set would be of great interest.

8.2.5 Invoking TS Operations from the Host

The host Transputer is not capable of invoking Linda primitive operations. As described at length at section 5.2.1, its primary function is to act as an intermediate node and provide a link between the first and last processors in the top row of the mesh. For the sake of clarity, this is depicted again in Figure 8.1.

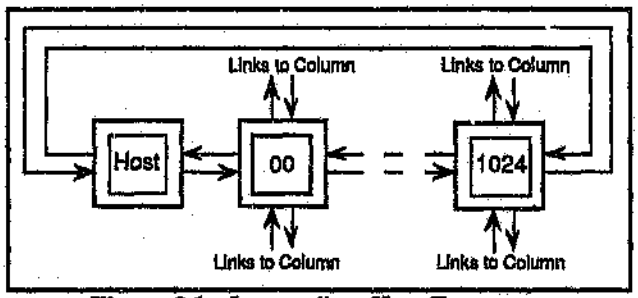


Figure 8.1 : Intermediate Host Transputer

Since the host processor is not part of any inverse beam (i.e. column of the mesh), it is not possible to invoke in or rd operations from this node. However, it is proposed that the system could be enhanced by having Node 0 carry out these operations on the host's behalf. Templates would then be sent from the host to Node 0 and processed in the normal fashion. Tuples returned to Node 0 as a result of a request from the host being satisfied would then be passed back accordingly. Although this scheme would necessitate Node 0 handling more TS operations and network traffic, it should prove to be more advantageous than the current scheme whereby the implementation of applications that exhibit typical master / slave behaviour necessitates one processor in the mesh functioning both as the host and as a worker. Apart from providing greater efficiency, this feature would also provide a more effective user interface. Currently, the results of a computation can only be observed by inspecting the TS of the node on which the host process is implemented (TS inspection is performed via a monitoring routine – refer section 5.2.1.1). By incorporating the host Transputer into the system, the user would have the means to both provide input to and directly observe the output from a particular application program.

Aside – Configuring the Host out of the Mesh

Conversely, it is also feasible to consider dynamically configuring the host Transputer *out* of the X-Linda mesh [C. Mueller, personal communication, Dec. 1990]. The presence of this extra processor in the top row of the mesh does have an effect on the efficiency of the system. Consequently, it may be desirable to load the application programs from the host onto the network, and then dynamically connect Processor 0 to Processor 1024, effectively blocking the host out. Once the programs have terminated, the link from Processor 0 to the host could be re-established in order to access the results of the computation. This approach, although potentially beneficial to the overall efficiency of the system, is unfortunately rather clumsy.

8.2.6 Using Arbitrary Numbers of Nodes

This idea is closely related to that of invoking TS operations from the host processor. The system has been designed for use on a $\sqrt{k}$ by $\sqrt{k}$ mesh of processors. Obviously, this is an ideal grid dimension since it allows the tuple and inverse beams to be of equal length. However, it would be useful if the system could still operate on configurations where the number of processors available was not necessarily a perfect square. This could be achieved in a fashion similar to that described above. Take, for example, a configuration comprising only 5 nodes. These nodes could be configured as shown in Figure 8.2.

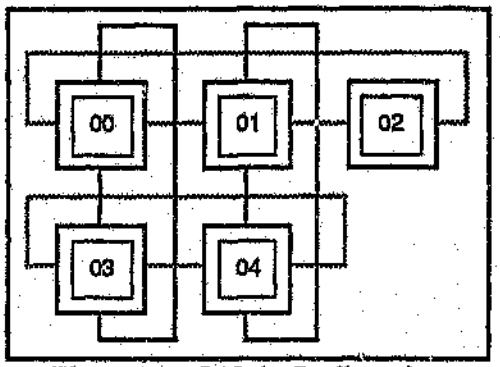


Figure 8.2 : 5 Node Configuration

Templates invoked from Node 2 would now be passed to Node 1 and processed there, and tuples satisfying these requests would then be redirected accordingly.

Inherent in both of the schemes described (i.e. invoking TS operations from the host processor and catering for configurations of arbitrary size) is the problem that specific nodes would incur extra storage and communication overheads. The effect of these overheads is difficult to estimate, and would require implementation analysis to be accurately determined.

8.2.7 Multiple Application Programs

X-Linda permits only a single application process to be active on a processor at any one time. It should by now be obvious that the Transputer supports concurrent process execution, and, as described in Appendix 1, task switching is controlled by microcode (i.e. is extremely fast). Hence, it is feasible to envisage multiple application programs running on the same processor. Although Ahuja *et al.* [1986] maintain that an advantage of the replicated worker model (refer section 2.2.1) lies in the fact that each processor executes a single process, eliminating the need for context switching, it is felt that Transputers are an exceptional case. As described in section 5.2.2.1, application programs are executed from within a Computation process, and the Linda primitives invoked from here are transmitted to and handled by dedicated processes (i.e. the In, Out and Rd processes). This is illustrated in Figure 8.3.

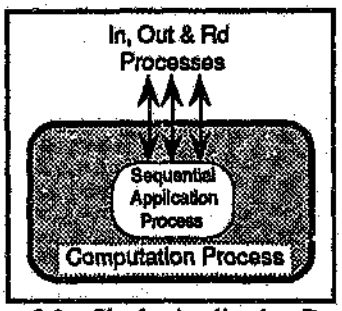


Figure 8.3 : Single Application Program

An extension to the system would be to permit multiple application processes to be invoked concurrently on each node. This would allow the programmer to take advantage of occam's process model of concurrency, and therefore provide greater freedom in providing a more powerful means for expressing problems. It is envisaged that these multiple programs could then be multiplexed through a single controlling process in order to communicate with the In, Out and Rd processes, as shown in Figure 8.4.

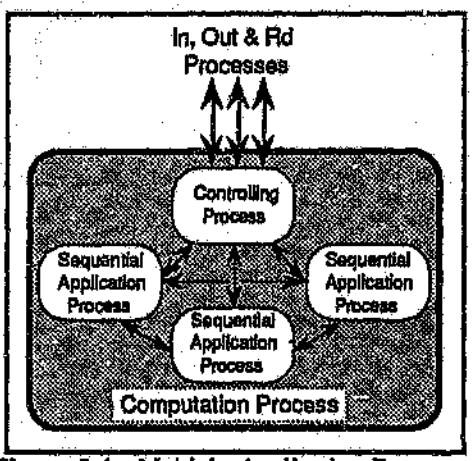


Figure 8.4 : Multiple Application Programs

A tremendous amount of modification to the system would not be needed to implement multiple application programs. One obvious consideration that does come to mind, however, is the fact that every template would need to have associated with it the identity of the sub-process that invoked it, so that satisfied tuple requests could be returned to the appropriate requesting processes.

8.2.8 The Tuple Challenge Process

Alternative strategies for determining which node should win a tuple "challenge" (i.e. when two nodes simultaneously attempt to satisfy In requests on the same tuple) were discussed in section 5.1.3.2. This topic has obvious potential for future research, and should receive further investigation.

8.2.9 Link Directions

It was shown in section 4.3.1.4 that the tuple space requests and operations are transmitted *bi-directionally* (e.g. templates are sent *up* the in-sets; tuples returned in response to request satisfaction are passed *down* the in-sets). It was also pointed out that, *on average*, the number of links traversed is identical irrespective of whether one- or two-way communication is utilized. Nevertheless, it would be of

interest to test other directional strategies and ascertain what effect (if any) these would have on the efficiency of the system.

8.3 DISCUSSION

The scope for enhancing X-Linda and the possibilities for future research are vast. As is discussed in the next section at some length, it is felt that, despite the inefficiencies and overheads of the system, the potential for development and improvement should not be neglected. The future research directions considered in this section have focussed on enhancing or improving the implementation. However, a further direction that is important, even critical, to future Linda research in general is that of *program behaviour*. This issue has been touched on in sections 6.2.3 and 7.4.3, and it is worth re-emphasizing the importance of this area. It is felt that currently not enough is known about the behaviour of Linda programs in general. An intensive investigation into this topic would prove invaluable in foreseeing the implementation *requirements* of any proposed system.

SECTION 9

9.0 CONCLUSIONS

In conclusion, it is worthwhile to re-examine the motivation and objective of this research. It is indicated in this report that although occam does provide an elegant and efficient means of programming Transputer networks, the software formalism is very closely coupled to the hardware. This causes the underlying processor topology to become an integral part of the design and implementation of parallel algorithms. On the other hand, Linda is a conceptually simple, topology independent programming model; and, more than this, Linda programs are portable. It is therefore claimed that there is much that can be gained by the implementation of the Linda model on Transputer networks. The objective of this research is to investigate the communication overheads imposed by a specific tuple space model on Transputer meshes, and hence evaluate the feasibility of such an implementation. The X-Linda system was created to conduct the investigation, and the research is ultimately concerned with ascertaining the communication overheads inherent in X-Linda. Below, a review of the report is conducted, and observations regarding the findings of the research given.

9.1 REVIEW

To place the document in perspective, it is useful to briefly review the content up to this point. The intention has been to lead gradually into the crux of the research, introducing Linda and tuple space methodologies, then describing the design and implementation of X-Linda and, finally, delving into the analysis and evaluation of the system. To start with, section 2 describes the Linda paradigm, illustrating the concepts of tuples and TS, and describing the primitive operations that may be used to manipulate TS. This section is in itself useful, as it collates information located across a broad range of reference material and presents a readable overview of the relevant issues. Importantly, it is shown that Linda processes have no direct interaction with each other. Instead, process synchronization and communication are achieved via tuple space operations, providing an entirely new, conceptually simple approach to parallel programming. The benefits of the style of programming are introduced, and the advantages of the paradigm discussed. It is concluded in this section that although there is no one programming model that can offer a complete solution to the complexities and problems inherent in parallel programming, Linda does address many key issues in the area.

Issues pertaining to the implementation of TS on distributed-memory systems are considered in section 3, and two specific approaches (hashing and uniform distribution) presented. Much emphasis is given to intermediate uniform distribution, illustrating that it is an elegant and conceptually simple approach (and hence its use in the X-Linda implementation). The Linda Machine, a distributed-memory system designed specifically to support intermediate uniformly distributed tuple space, is described, illustrating the advantages of using custom-built hardware. The design of the Linda Machine is particularly relevant, since it has, to a large extent, influenced the design of X-Linda. This influence is apparent in various places throughout the document. Existing Transputer-based Linda implementations are also examined, illustrating that such implementations are feasible and, by its absence, the uniqueness of the X-Linda approach. A further point raised within the context of these Transputer-based implementations is the applicability of dedicated hardware to support Linda.

The X-Linda implementation is introduced in section 4, motivating the project in terms of a desire for a new programming methodology. X-Linda's implementation environment (i.e. computing platform and development system) is described, emphasizing the complications imposed by communication between the host Transputer and the network. The fundamental design and specification of the system are outlined in this section, motivating the choice of tuple space model by virtue of its simplicity and elegance, and its uniqueness within the sphere of existing Transputer-based Linda implementations. A major influence of the Linda Machine is shown in the storage of tuples and templates under X-Linda (i.e. the use of identical TS locations across the in- and out-sets). The design of the individual nodes within the system is described with reference to the design of the Linda Machine, and it is shown that, essentially, the X-Linda node is a software implementation of the Linda node. The applicability of occam 2 in the design and implementation of the X-Linda node is discussed, emphasizing the value of a natural and elegant means of expressing the required process interaction. The section also deals with the structure of tuples under X-Linda, illustrating the influence of the research needs.

Section 5 delves deeper into the design specification, examining the actual implementation of the TS primitive operations (i.e. in, out and rd). Various problematic issues and design considerations are addressed. The importance of detailing these considerations lies in the fact that they have a direct impact on the overall efficiency of the system, and, furthermore, they illustrate the maintenance of TS consistency in the processing of the primitives. It is also important to note that the very existence of a substantial design effort can be regarded as an indication of the unsuitability of the specific tuple space model on Transputers. This section also examines the overall design and structure of the system at the process level, illustrating the function and interaction of the individual processes that, collectively, comprise X-Linda. The design of the system is addressed with regard to the Host process and the X-Linda node by means of an overview of the operation of the various modules within the system. A discussion on two fundamentally important aspects of the design (i.e. the provision of buffering and atomicity) is given, and the applicability of occam 2 to the overall design is readdressed. It is maintained that the decision to model the X-Linda implementation on the Linda Machine is well justified, and that the design objective (i.e. to provide a software implementation of the Linda Machine) has been achieved. In addition, it is claimed that the objective has been attained in an elegant fashion. To conclude the section, the programming methodology applied in the construction of X-Linda is briefly overviewed, illustrating an adherence to "good" occam programming style. Finally, the (excessive) storage requirements of the system and the causes thereof are detailed.

The actual purpose of the research – to ascertain the communication overheads associated with X-Linda – is covered in section 6. X-Linda is analyzed by means of a comprehensive series of tests and experiments designed to measure the communication overheads of the model (relative to message passing performance on native Transputer networks). The overheads specific to the TS primitive operations are evaluated, and experiments are conducted to measure the overhead of data exchange between processors and of a sink algorithm. In addition, the process scheduling overhead and the inefficiency of the tuple matching process are discussed. It is shown in this section that, in general, the communication overheads of the implementation are significant and that, in addition, the collective influence of all of the associated system overheads is excessive. The following section, 7, describes and evaluates two example programs implemented under X-Linda. The results presented here complement those detailed in the previous section in that they illustrate the collective influence of the various overheads on the overall efficiency of the

system – i.e. emphasizing the extreme nature of the overheads and the resultant inefficiency of the system. Section 7 also gives some interesting insight into the causes of this inefficiency. It is worth restating here that the implementation of the experiments in section 6 and the example programs in section 7 was extremely easy under X-Linda, supporting the claim in section 2.3 that Linda, by virtue of its power and expressiveness, simplifies parallel programming.

The final section in this document, 8, deals with the future research potential of X-Linda and discusses means of enhancing the system and its performance. It is apparent that X-Linda has vast scope for future research and it is maintained that much can be gained in pursuing the ideas dealt with here.

9.2 OBSERVATIONS

It has been shown that the X-Linda implementation suffers significant communication overheads and that the system, as it stands, is too inefficient to be of practical use. However, it is important to keep in mind that X-Linda is not intended to be a fully-fledged Linda system, but was created to provide a means of investigating the communication overheads pertaining to a specific TS model. This is especially relevant with regard to the TS search strategy, the inefficiency of which is illustrated in section 6.7. It is not unreasonable to assume that a redesign of the system, given the knowledge acquired in the course of this research and taking into account the enhancements discussed in section 8 (particularly the schemes for speeding up the matching process), may yield "acceptable" performance. Of course, it would never be possible to achieve performance equal to that provided by native Transputer networks. However, the emphasis of Linda is on ease of programming (i.e. it is acceptable to trade some performance off against the gains of the programming paradigm [Ahuja *et al.* 1988]). With regard to scalability, the approach does not, however, hold promise. Given a mesh of 2^{10} processors, each in- and out-set would comprise 32 nodes, and the corresponding amount of communication necessary to transmit tuples and templates across the respective sets would be significant.

It is concluded that the communication capabilities of the Transputer are not well suited to the efficient implementation of in- and out-sets (of course, this statement will need to be re-addressed with the introduction of the H1 Transputer described in Appendix 1). The fact that a massive design effort, with a great deal of additional modification, was required to successfully implement the TS primitive operations is immediately indicative of this unsuitability, and the extent of the overheads inherent in the implementation illustrates this point in a more obvious way. The overall efficiency of the system is appalling, indicating the cumulative effect of all the associated overheads (i.e. communication, TS search, synchronization, process scheduling and set-up). This, however, does not mean that the approach must be discarded. It was shown that the communication overheads are not necessarily prohibitive and it is felt that by addressing the other overheads listed above, it is possible to develop a usable system. More than this, as indicated in section 6.9.1, there are various weaknesses in the design and implementation of the system. It is, therefore, maintained that, although the tuple space model is not well suited to Transputer networks, the X-Linda approach requires further exploration in order to exhaust its full research potential.

Finally, comment must be made on one aspect of the objective of this research that has been previously stated, but not yet addressed; to evaluate the feasibility of a full Linda implementation based on the X-Linda approach. In favour of this evaluation, there is the elegance of a design that is modelled on the successful Linda Machine project, and the fact that, as a result of this research, it is known that the design can

be applied to Transputer networks. Working against these advantages are the facts that

1. the point-to-point communication links of the Transputer are not ideally suited to the implementation of in- and out-sets, and
2. apart from the resultant communication overheads, there are other inherent overheads that have a significant effect on performance.

Therefore, is a full Linda implementation feasible ? The answer to this question is reservedly affirmative. The system undoubtedly has the potential for development into an efficient and usable product; however, extensive effort would be required to achieve this. Personal opinion favours further development of the system, and, in particular, investigating the use of dedicated hardware support in this regard.

APPENDIX 1

A1 THE TRANSPUTER AND OCCAM

For those unfamiliar with the Transputer and its native language, occam, a brief introduction is given below. The information is based primarily on an overview given by Faasen [1990a] and is non-technical in nature.

The Transputer is a microcomputer developed in the mid-1980s by INMOS Limited. The fundamental components of the Transputer are a 16 or 32-bit processor (CPU), fast local memory and high-speed communication links that provide point-to-point connections between Transputers; all of these components reside on a single chip and can operate concurrently. The IMS T414 Transputer is an example of this basic model – other special purpose members of the Transputer family feature additional circuitry, microcode and interfaces that support a specific task (e.g. disk and memory controllers). The IMS T800[1] Transputer includes an on-chip floating-point unit (FPU). The CPU, memory, communication links and FPU (in the case of the IMS T800) all share a 32-bit data / address bus. An external memory interface is provided to allow access to additional, off-chip memory. A block diagram of the IMS T800 architecture is given in Figure A1.1.

[1] In this Appendix, the term *T800* by default refers to a 20 MHz processor. Where necessary, the notation *T800-20* and *T800-30* is used to distinguish between processors running at 20 and 30 MHz respectively

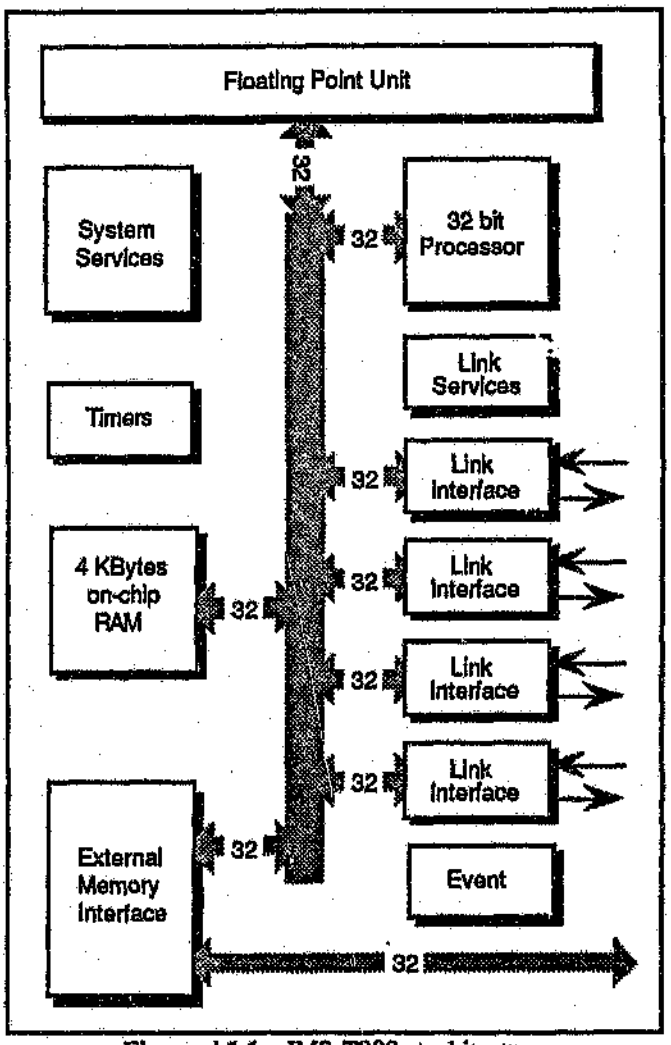


Figure A1.1 : IMS T800 Architecture

Parallel applications invariably involve a high degree of communication and ta' switching. The Transputer utilizes microcode to control these aspects of concurrency. The result is exceptionally fast context switching (the IMS T800-20 can switch between tasks in less than 950 nanoseconds). Consequently, it is possible to achieve real-time multi-tasking performance. Transputers can be utilized as high-speed stand-alone processors; alternatively, they can be easily networked together to provide parallel computing surfaces. The process model of concurrency is supported by both the hardware and the programming model. Problems are described using the software formalism, and, since the there is a close relationship between this formalism and the physical architecture, the implementation in hardware on a configuration that best meets the processing requirements is relatively straight-forward.

A1.1 PERFORMANCE

The Transputer is a fast processor. The IMS T414, released in 1985, is capable of 10 MIPS. The IMS T800-20, announced in 1986, delivers 1.5 MFLOPS of 32-bit IEEE standard arithmetic, and the IMS T800-30 is capable of 2.25 MFLOPS. With regard to floating-point performance, these impressive results are largely due to the fact that the FPU and CPU reside on the same chip and can operate in parallel –

which means that the figures quoted represent sustained as opposed to peak performance.

A1.2 COMMUNICATION

The Transputer has 4 high-speed serial links that provide full-duplex point-to-point communication with other Transputers. Each link has an input and an output channel. A connection between two Transputers is implemented by connecting these channels together via a pair of uni-directional signal lines, as shown in Figure A1.2 :

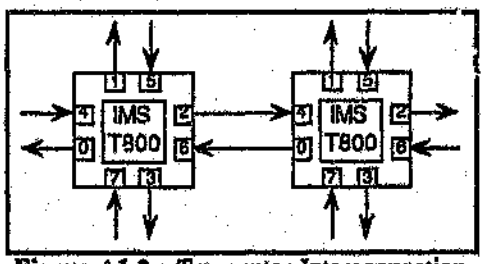


Figure A1.2 : Transputer Interconnection

Data that are sent along a link's output channel are acknowledged on the input channel and process synchronization is provided by a handshaking technique. The IMS T800 allows messages to be pre-acknowledged and each of the links are capable of transmitting data at a rate of 20 Mbits/second, which corresponds to roughly 2.3 Mbytes/second of real data (i.e. excluding control information). The fact that each link has its own DMA controller means that any specific link can operate independently and in parallel with the other three links, the CPU and, in the case of the IMS T800, the FPU. Communication, therefore, does not involve processor overhead. The fact that the speed of the processor is so much faster than that of the links does, however, mean that communication is the bottle-neck in any Transputer-based system and is the limiting factor in efforts to achieve speed-up and processor efficiency. Networks of Transputers can be configured via the communication links to specific topologies – for example, a pipe, ring, mesh or hypercube.

A1.3 OCCAM

Occam is the programming model for the Transputer and was developed concurrently with the hardware. Occam is based on the CSP model [Hoare 1978], and was originally intended as a low-level compiler target for Transputers. However, the language has been successively developed into its current form, occam 2, which is a fully-fledged general purpose programming language with built-in support for concurrency and communication. Occam provides a simple and natural way of describing parallel systems. The model supports concurrent processes and inter-process communication via channels. These processes are used to model Transputers, and the channels to simulate the physical links. Consequently, it is possible to develop and test application programs on a single Transputer and then physically distribute the programs on networks of Transputers.

Further details regarding the hardware specifications can be found in [INMOS 1988a and 1989], and INMOS [1988b], Pountain [1989] and Jones and Goldsmith [1988] cover the specification and usage of occam 2.

A1.4 Looking Ahead – The H1 Transputer

INMOS have announced a new generation of Transputer, the H1, which is due for release in 1991. The following details have their source Pountain [1990] and Rabagliati [1990b]. The H1 will feature enhanced hardware with more powerful operating system facilities to provide a faster and more usable computing platform. The chip itself will run at 50 MHz, and will be capable of 100-150 MIPS and 20 MFLOPS. Again, only 4 communication links will be provided – however, each of the links will be capable of a data transmission rate of 100 Mbits/second (about 5 times faster than at present). On-chip memory will be expanded to 16 Kbytes, and it is significant to note that this memory will be cached.

Perhaps the most radical feature of the H1 is the introduction of *virtual channels*. A separate on-chip communications controller is provided to multiplex any number of logical channels onto the 4 physical links – i.e. the hard links will be shared transparently. A dedicated routing chip, the C104, is also under development for use with the H1. The C104 is a high-performance 32x32 packet-switching exchange which will provide a transparent connection between any two Transputers in a network. The H1 and C104 will ease the burden associated with occam programming, perhaps most significantly in that there will no longer be a need to "PLACE" communication channels at physical link addresses. The programmer may, instead, specify as many channels as necessary which the routing mechanism will interconnect as required.

APPENDIX 2

A2 BASE FIGURE EXPERIMENTS

Details regarding rates of inter-processor data transmission using occam 2 on native Transputer networks are presented in this Appendix. The values obtained from the experiments described below are the base figures against which the X-Linda measurement were compared in order to ascertain the communication overheads of the model (refer section 6). Notice that :

1. All the tests were conducted on the Parsytec SuperCluster described in section 4.2.1. Recall that the link speeds of the processors within the SuperCluster are set at 10 Mbits/second
2. All the experiments are based on the transmission of *sized* arrays of 32-bit integers
3. The experiments were run a number of times, and the results averaged out to eliminate experimental error
4. The following notation is used throughout :
   N – length of array (i.e. number of 32-bit integers in message)
   m – $\log_2 N$
   Time – Time in microseconds
   Rate – Transmission rate = Bytes / Time = N x 4 / Time
5. In general, transmission rates are listed for messages of length between 2^{10} and 2^{17} integers, and an overall average rate is cited. It was often the case that the times pertaining to the transmission of small messages (e.g. 2^4 integers) were too low to be reliable (i.e. times of the order of a few internal clock ticks). Although it is realized that the rate of transmission of small messages is *different* to that for very large messages, it was considered more practical to use the values pertaining to "stabilized" rates (for messages of length 2^{10} words and up, the rates of transmission are very similar)
6. The technique of breaking up messages into smaller packets, and receiving and sending on these packets concurrently was *not* employed (this technique can be used to enhance the speed of transmission through a pipe of processors – refer section 8.2.4).
7. The experiments detailed below exhibit some degree of replication (for example, two different experiments are conducted to investigate one- and two-way communication between processors). Although these sorts of results may have been inferred from other experiments, in the interests of accuracy it was not considered prudent to do this.

A2.1 EXPERIMENT 1
ONE WAY COMMUNICATION BETWEEN PROCESSORS

1. Objective

This experiment, based on that performed by Lakier [1989a], was designed to evaluate the speed of transmitting data from one processor to another. Furthermore, it was desired to compare the rate of communication from a T414 processor to a T800 with that between two T800s (this is of particular significance since the top row of the X-Linda mesh incorporates a T414 host Transputer).

2. Design

The configuration shown in Figure A2.1 was used for this experiment.

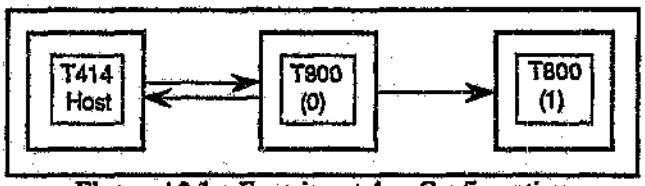


Figure A2.1 : Experiment 1 – Configuration

The experiment was run in two phases – it was necessary to measure the time taken to transmit a message from

1. T414 Host to T800 (0)
2. T800 (0) to T800 (1).

In both cases, the time was obtained as follows, where Link.Out is the name of the link between the source and destination Transputers :

```
clock ? start
Link.Out ! array_size :: array
clock ? finish
time_taken := (finish MINUS start) * 64 -- microseconds
```

3. Results

The communication times and corresponding transmission rates for this experiment are shown in Table A2.1.

Message Len		T414 – T800(0)		T800(0) – T800(1)	
m	N	Time	Rate	Time	Rate
10	1024	7360	0.5565	5696	0.7191
11	2048	14720	0.5565	11456	0.7151
12	4096	29440	0.5565	22912	0.7151
13	8192	58944	0.5559	45888	0.7141
14	16384	117952	0.5556	91712	0.7146
15	32768	235840	0.5558	183488	0.7143
16	65536	471744	0.5557	366976	0.7143
17	131072	943616	0.5556	734016	0.7143
		Avg	0.5560	Avg	0.7151

Table A2.1 : Experiment 1 – Results

4. Observations

1. The rate of transmission between two T800s is 1.29 times that of the rate from a T414 to a T800. Factors contributing to this phenomenon are discussed in section 4.2.2.
2. The rates of transmission can be more meaningfully stated as
   - T414 – T800 (0) : 543 Kbytes/second
   - T800 (0) – T800 (1) : 698 Kbytes/second – this result correlates those obtained by Lakier [1989a].

A2.2 EXPERIMENT 2
TWO WAY COMMUNICATION BETWEEN PROCESSORS (1)

1. Objective

The objective of this experiment was to evaluate the time taken to transmit a message from a source to a destination Transputer, and then back again. Measurements were required for transmission between adjacent processors, and also for the case where communication occurs through intermediate nodes.

2. Design

The configuration shown in Figure A2.2 was used for this experiment.

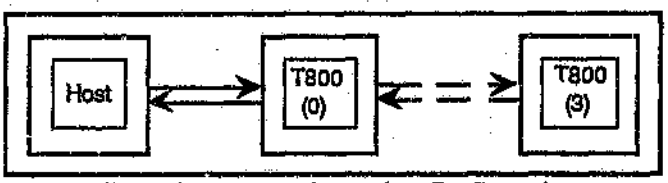


Figure A2.2 : Experiment 2 – Configuration

The time taken for a message to travel from T800 (0) to each of T800 (1), (2) and (3) and back again was measured. Notice that communication from the host is not measured in this experiment. It therefore does not matter whether this is a T414 or a T800 processor.

3. Results

Table A2.2 shows the communication times and corresponding rates of transmission for this experiment. Notice that these transmission rates correspond to the *entire* transmission of the message (from source to destination *and* back again).

Message Len		T800(0) – T800(1)		T800(0) – T800(2)		T800(0) – T800 (3)	
m	N	Time	Rate	Time	Rate	Time	Rate
10	1024	10713	0.3823	21452	0.1909	32633	0.1255
11	2048	21388	0.3830	42873	0.1911	65222	0.1256
12	4096	42752	0.3832	85696	0.1912	130368	0.1257
13	8192	85504	0.3832	171328	0.1913	260672	0.1257
14	16384	170944	0.3834	342636	0.1913	521254	0.1257
15	32768	341888	0.3834	685222	0.1913	1042438	0.1257
16	65536	683750	0.3834	1370400	0.1913	2084800	0.1257
17	131072	1367468	0.3834	2740793	0.1913	4169536	0.1257
		Avg	0.3832	Avg	0.1912	Avg	0.1257

Table A2.2 : Experiment 2 – Results

4. Observations

1. The rate pertaining to T800 (0) – T800 (1) is slightly more (i.e. about 7%) than half the rate of a one-way communication (refer previous experiment – Table A2.1). The reason for this is unclear.
2. The rates shown above in Table A2.2 appear reasonably scalable as more processors are added (i.e. there is an approximately linear relationship between the rates and the number of links traversed).

A2.3 EXPERIMENT 3
TWO WAY COMMUNICATION BETWEEN PROCESSORS (2)

1. Objective

As in the previous experiment, the objective here was to evaluate the time taken to transmit a message from a source to a destination Transputer, and then back again. However, in order to implement a base test against which the in and rd operations could be compared (sections 6.3 and 6.4), the above experiment had to be amended slightly. The message from the source to the host, instead of consisting of an array of integers, now simply comprised a single integer; the return message, however, consisted of an integer array, as before.

2. Design

Refer to Experiment 2 and Figure A2.2. The experiments are identical except for the fact that, here, only a single integer is sent from source to destination.

3. Results

Table A2.3 shows the communication times and corresponding rates of transmission for this experiment.

Message Len		T800(0) – T800(1)		T800(0) – T800(2)		T800(0) – T800 (3)	
m	N	Time	Rate	Time	Rate	Time	Rate
10	1024	5382	0.7611	11136	0.3678	16889	0.2452
12	4096	21459	0.7635	44416	0.3689	67379	0.2432
14	16384	85760	0.7642	177542	0.3691	269350	0.2433
16	65536	343091	0.7641	710105	0.3692	1077126	0.2434
		Avg	0.7632	Avg	0.3688	Avg	0.2438

Table A2.3 : Experiment 3 – Results

4. Observations

As expected, the rates shown above are approximately half those obtained in the previous experiment (Table A2.2).

A2.4 EXPERIMENT 4
COMMUNICATIONS THROUGH A RING OF PROCESSORS

1. Objective

This test was designed specifically for comparison with X-Linda's out operation (section 6.2). Communication times were required for the transmission of messages around rings of various dimension, and it was also desired to observe the effect of including the host T414 Transputer within the ring.

2. Design

The experiment was tested on rings comprising 2, 3 and 4 Transputers – each case was tested with and without the additional host Transputer. The configuration used is shown in Figure A2.3 below, where k represents the number of T800s in the ring.

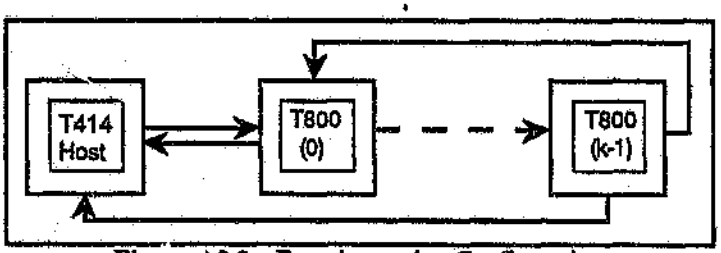


Figure A2.3 : Experiment 4 – Configuration

The message was transmitted from T800 (0), and time taken for that message to traverse the ring was measured. Two specific test cases were used, where the ring

1. did not include the Host – i.e. it comprised only T800 (0) through T800 (k-1)
2. included the Host – i.e. the message had to go from T800 (k-1) through the Host to get back to T800 (0).

3. Results

The results for the various configurations are given in Tables A2.4 – A2.6. For each Table, the number (k) of T800s in the ring is given, and figures are quoted for

1. No Host – not including the T414
2. With Host – including the T414 in the ring.

Notice that the transmission rates quoted correspond to the transmission of the message over the *entire* ring.

k = 2					
Message Len		No Host		With Host	
m	N	Time	Rate	Time	Rate
10	1024	11136	0.3678	22976	0.1783
11	2048	22336	0.3668	45952	0.1783
12	4096	44672	0.3668	91840	0.1784
13	8192	89280	0.3670	183680	0.1784
14	16384	178560	0.3670	367360	0.1784
15	32768	357184	0.3670	734720	0.1784
16	65536	714368	0.3670	1469440	0.1784
17	131072	1428672	0.3670	2938880	0.1784
		Avg	0.3671	Avg	0.1784

Table A2.4 : Experiment 4 – Results (k = 2)

k = 3					
Message Len		No Host		With Host	
m	N	Time	Rate	Time	Rate
10	1024	16832	0.2433	28672	0.1429
11	2048	33664	0.2433	57280	0.1430
12	4096	67328	0.2433	114560	0.1430
13	8192	134656	0.2433	229120	0.1430
14	16384	269312	0.2433	458240	0.1430
15	32768	538624	0.2433	916480	0.1430
16	65536	1077248	0.2433	1832832	0.1430
17	131072	2154432	0.2434	3665792	0.1430
		Avg	0.2433	Avg	0.1430

Table A2.5 : Experiment 4 – Results (k = 3)

k = 4					
Message Len		No Host		With Host	
m	N	Time	Rate	Time	Rate
10	1024	22592	0.1813	34176	0.1199
11	2048	45184	0.1813	68288	0.1200
12	4096	90304	0.1814	136576	0.1200
13	8192	180608	0.1814	273152	0.1200
14	16384	361216	0.1814	546240	0.1200
15	32768	722368	0.1814	1092416	0.1200
16	65536	1444800	0.1814	2184960	0.1200
17	131072	2889536	0.1814	4369920	0.1200
		Avg	0.1814	Avg	0.1200

Table A2.6 : Experiment 4 – Results (k = 4)

4. Observation

Transmitting the messages through the host T414 has a dramatic effect on transmission times. Ideally, the rate of transmission through ring of size k+1 without the host should be same as that for a ring of size k with the host. Table A2.7 shows that this is definitely *not* the case :

k	R_1 Rate – No Host	R_2 Rate – With Host	$R_1(k+1)/R_2(k)$
2	0.3671	0.1784	1.36
3	0.2433	0.1430	1.27
4	0.1814	0.1200	–

Table A2.7 : Experiment 4 – Observations

A2.5 EXPERIMENT 5

INFORMATION EXCHANGE ON A MESH CONFIGURATION (1)

The experiment described below constitutes the one of the base tests against which the X-Linda implementation of data exchange between two processors was compared (section 6.5.1).

1. Objective

To replicate the data transmission required to effect the exchange under X-Linda – i.e. to perform and measure the identical amount of message transmission using occam 2.

2. Design

The total number of tuple transmissions necessary to perform an information interchange between a source node and any destination node comprises the following :

Xmit (1) – Transmission from the source node along its out-set to the *column* containing the destination node
Xmit (2) – Transmission along the column to the destination node
Xmit (3) – Transmission from the destination node along its out-set to the *column* containing the source node
Xmit (4) – Transmission along the column back to the source node

Notice that the transmission of templates has been neglected. This is because, due to the nature of the X-Linda algorithm (refer section 6.5), it is possible to assume that a template will be in place (i.e.resident on a specific node) *before* the matching tuple reaches that node. Hence, the time taken to transmit a template can be disregarded.

The calculation of the total number of transmissions is best illustrated by means of an example. Given that Node 4 is the source node (as was done for the X-Linda implementation), assume that data exchange is to occur between it and Node 11 (the destination node). Figure A2.4 shows the sequence of transmissions necessary to complete the exchange under X-Linda (recall from section 4.3.1.4 that tuples are outed to the right, and are returned downwards).

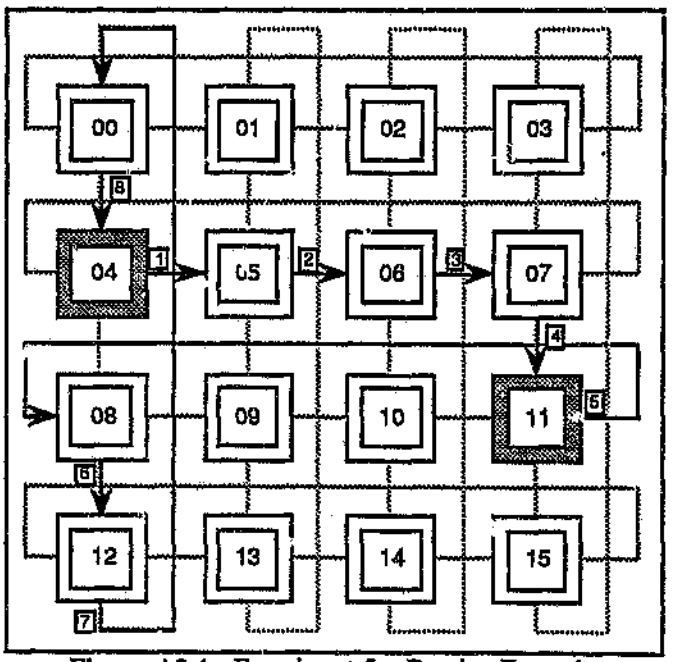


Figure A2.4 : Experiment 5 – Routing Example

From Figure A2.4, we can deduce that the total number of data transmissions needed to exchange data between Nodes 4 and 11 is :

Xmit (1)	Xmit (2)	Xmit (3)	Xmit (4)	Total
3	1	1	3	8

3. Results

In Experiment 2 it was shown that the overall rate of transmission over 4 links is 47800 integers/microsecond (refer Table A2.2). Hence, we can, with justification, deduce the rate over 8 links to be half of this – i.e. 23900 integers/microsecond (this is claimed "with justification" since the results pertaining to different number of links appear to be scalable). Using these results, it is possible to deduce the times taken to perform the data exchange with respect to each of the nodes in the mesh for tuples of various dimension – these are shown in Table A2.8.

Node	Xmit (1)	Xmit (2)	Xmit (2)	Xmit (3)	Total	Time N=16	Time N=256	Time N=1024
0	0	3	0	1	4	335	5356	21423
1	1	3	3	1	8	670	10712	42846
2	2	3	2	1	8	670	10712	42846
3	3	3	1	1	8	670	10712	42846
4	0	0	0	0	0	-	-	-
5	1	0	3	0	4	335	5356	21423
6	2	0	2	0	4	335	5356	21423
7	3	0	1	0	4	335	5356	21423
8	0	1	0	3	4	335	5356	21423
9	1	1	3	3	8	670	10712	42846
10	2	1	2	3	8	670	10712	42846
11	3	1	1	3	8	670	10712	42846
12	0	2	0	2	4	335	5356	21423
13	1	2	3	2	8	670	10712	42846
14	2	2	2	2	8	670	10712	42846
15	3	2	1	2	8	670	10712	42846

Table A2.8 : Experiment 5 – Results

4. Observations

From the communication patterns, it is obvious that the first node in every row (i.e. nodes 0, 8 and 12) of the mesh will have the same communication times – which is the same as that for all the nodes in the second row (nodes 5, 6 and 7). Similarly, the rest of the nodes all have the same communication times.

A2.6 EXPERIMENT 6

INFORMATION EXCHANGE ON A MESH CONFIGURATION (2)

The experiment described below constitutes the second base test against which the X-Linda implementation of data exchange between two processors was compared (section 6.5.2).

1. Objective

To evaluate the rate of data transmission between two processors residing on a mesh configuration over the shortest possible path. Notice that the data transmission is two-way – i.e. an exchange of information.

2. Design

A single node, processor 4, was selected to act as the "controller". Arrays of integers were transmitted from this node to every other node and back again, over the

shortest possible path. Figure A2.5 shows the paths that were used for the experiment. Notice that, although the routes depicted in the Figure are optimal, they are *not* unique; there are a variety of shortest possible paths using a mesh configuration. The routes shown below were chosen simply because they seem to the most logical.

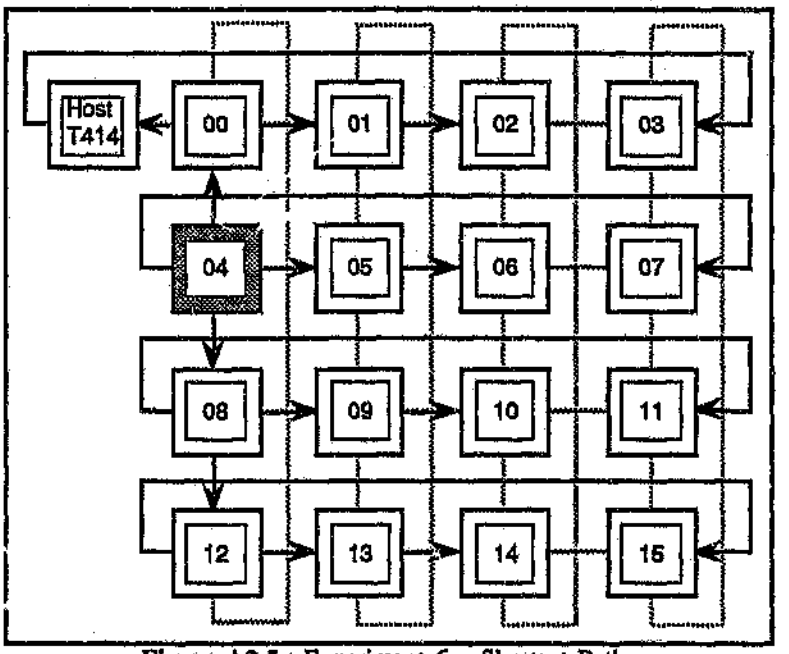


Figure A2.5 : Experiment 6 – Shortest Paths

Notice that Node 3's communication occurs through the host T414. Obviously, this route could have been specified differently. However, to retain some form of consistency with the X-Linda configuration (where communication *does* occur through the host Transputer), it was decided to use the above route. Hence, the rate of transmission to and from this processor can be expected to be relatively slow.

3. Results

The only way that this experiment can be meaningfully with the X-Linda equivalent is to consider *average* data transmission rates; the results pertaining to specific nodes are, on their own, meaningless. Hence, the times taken for the transmission of messages from Node 4 to all the destination nodes and back again have been summed, and an average rate of communication calculated from this. These rates are shown in Table A2.9 for the transmission of integer arrays of various dimension, using the following notation :

Total Time – Total of individual transmission times
Avg Time – Total Time / 15 (i.e. excluding Node 4)
Rate – N x 4 / Avg Time (i.e. bytes / microsecond)

Message Len		Time		Rate
m	N	Total Time	Avg Time	
10	1024	377984	25198.93	0.1625
12	4096	1508160	100544.00	0.1630
14	16384	6029056	401937.06	0.1631
16	65536	24112448	1607497.53	0.1631
			Avg	0.1629

Table A2.9 : Experiment 6 – Results

A2.7 EXPERIMENT 7
IMPLEMENTATION OF A SINK ON A MESH CONFIGURATION

This experiment is the base test against which the X-Linda implementation of a sink is compared in section 6.6.

1. Objective

To evaluate the time taken for the all nodes within a mesh configuration to send information to a single requesting node (the sink).

2. Design

The experiment is almost identical to the previous one (Experiment 6). Node 4 was selected to send a request to all the nodes in the mesh, and these nodes then returned information to this node. The paths over which the processors communicated are identical to those shown in Figure A2.5. The *single* difference between this experiment and the previous one is that, here, Node 4 sends out a *request* for data, as opposed to an entire integer array. Ideally, this request should take the form of a single integer. In practice, 3 integers were required to carry routing information (obviously, using 3 integers instead of 1 has a negligible effect on the overall rate of transmission). The time taken for Node 4 to transmit all of its data requests and to receive messages (i.e. integer arrays) from all the other nodes in the mesh was measured.

3. Results

As in the previous experiment, the results are presented under the headings of :

Total Time – Time taken for all the nodes to receive a request from, and return information to, Node 4
Avg Time – Total Time / 15 (i.e. excluding Node 4)
Rate – N x 4 / Avg Time (i.e. bytes / microsecond)

The results pertaining to the transmission of arrays of various dimension are shown in Table A2.10.

Message Len		Time		Rate
m	N	Total Time	Avg Time	
10	1024	189760	12650.67	0.3238
12	4096	759920	50261.33	0.3260
14	16384	3008512	200567.47	0.3268
16	65536	12028096	801873.07	0.3269
			Avg	0.3259

Table A2.10 : Experiment 7 – Results

4. Observations

As expected, the average rate of transmission for this experiment is almost exactly double that obtained in the previous experiment.

APPENDIX 3

A3 ASCERTAINING CPU UTILIZATION

The percentage CPU utilization figures used throughout this document have been of great value in the analysis and evaluation of the system (refer section 6), and have provided useful insight into the general behaviour of application programs running under X-Linda. This Appendix describes the derivation of the utilization figures used in the analysis, and, for the Transputer enthusiast, an assembly language routine for ascertaining these figures is also given. A comparison of these figures is conducted, and they are shown to perform identically.

A3.1 OCCAM 2 *SUPERVISOR* PROCESS

The method of evaluation used in the X-Linda analysis is based on that developed by Rabagliati [1990a], with some modification. Rabagliati's approach involves a Supervisor process which is run in parallel with the rest of the processes on the Transputer. This process operates in 2 phases, both of which are invoked by sending input via a channel into the procedure :

1. Calibration – this phase evaluates the speed of the scheduling mechanism. It simply increments a counter over a specified time period, and returns the final count value. The calibration phase must be run when the rest of the processes are idle (i.e. waiting) in order to obtain a true calibration value.

2. Timing – when this phase is invoked, the counter is reset and then incremented until the command to end the timing is received. The amount of CPU utilization is obtained by comparing the new count value with the one obtained in the calibration phase – in effect, the procedure actually measures how idle the processor is. The percentage CPU utilization is calculated as follows :

Let MaxIdle be the maximum value of the counter attainable during the timing phase (i.e. based on the value obtained in the *Calibration* phase and on the period of time that the *Timing* phase was active).

Let RealIdle be the actual value of the counter that was obtained during the *Timing* phase.

Then the percentage CPU utilization, is calculated to be ;

((MaxIdle - RealIdle) / MaxIdle) x 100

This value is then returned to the process invoking the Supervisor procedure.

The process described above was slightly modified in order to obtain a more accurate calibration value. The calibration phase was taken out of the procedure, and run on its own – the resultant calibration figure was then "hard-coded" back into the Supervisor process. The Calibrate and Supervisor procedures are presented below. Both processes run two code segments in parallel – one process increments the counter, and the other is the "controller" that is responsible for resetting and returning the count value. The idle counter segment, common to both procedures, is defined as follows :

```
... Idle counter
INT now, then :
SEQ
  Idle := 0

  ... Set initial 'then'
  PRI PAR
    TIMER TIME :
    TIME ? then
```

```
    SKIP

  ... Loop forever
  WHILE TRUE
    SEQ

      ... Get 'now'
      PRI PAR
        TIMER TIME :
        TIME ? now
        SKIP

      ... Increment Idle
      IF                                --"If ... less than a certain number of Ticks -- 20 seems
        (now MINUS then) < 20           -- about right -- SUPERVISOR must be the only process
          Idle := Idle + 1              -- active"
        TRUE
          SKIP

      then := now
```

The Calibrate procedure launches the idle counter process and its controlling code in parallel. The controller resets the counter, and then delays for one second (during which time the counter is incremented by the counting process). It then returns the new count value to the process that invoked it. The listing for this procedure is given below :

```
PROC Calib.Proc (CHAN OF INT start, result)

  ... Declarations
  VAL period IS 15625 :   -- one second
  INT Idle, Calibrate :   -- Idle is accessed in parallel - hence, usage checking must be off

  PAR

    {{{
    ... Idle counter
    }}}

    ... Controller
    WHILE TRUE
      SEQ
        start ? Calibrate

        ... Delay while Idle is incremented
        INT now :
        TIMER TIME :
        SEQ
          Idle := 0

          ... Delay
          TIME ? now
          TIME ? AFTER now PLUS period

          Calibrate := Idle

        result ! Calibrate
:
```

The Calibrate procedure was executed a number of times, and an average count value obtained. This value (157487) was then defined in the Supervisor procedure

to be the maximum count value attainable in one second (i.e. given that the no other processes are running on the Transputer). The Supervisor also launches the count process and a controlling process in parallel. The controller is invoked by the receipt of a '0' on the input channel. It resets the counter, and on receipt of a '1', accesses the new count value (that was, in the interim, incremented by the count process). The percentage CPU utilization is then computed as described earlier, and returned to the requesting process. The listing of the Supervisor process is given below :

```
-- This Proc written by Andy Rabagliati (INMOS)
-- Amended : Craig Faasen - Oct. 1990

PROC Supervisor (CHAN OF INT command, result)

  ... Declarations
  ... Define start, read
  VAL start  IS 0 :
  VAL read  IS 1 :

  ... Calibration
  VAL time.period  IS 15625.0 (REAL32)      :   -- one second
  VAL calibrate    IS 157487.0 (REAL32)     :   -- Obtained using Calibrate procedure
  VAL count.rate   IS calibrate / time.period :

  INT Idle : -- Idle is accessed in parallel - hence, usage checking must be off

  PAR

    {{{
    ... Idle counter
    }}}

    ... Controller
    SEQ

      ... Vars
      INT begin :
      TIMER TIME :

      ... Loop forever
      WHILE TRUE
        INT signal :
        SEQ
          command ? signal
          F

            ... start
            signal = start
              SEQ
                Idle := 0
                TIME ? begin

            ... read
            signal = read

              ... Vars
              INT finish :
              REAL32 Real.Idle, Max.Idle, CPU.util :

              SEQ
                Real.Idle := REAL32 ROUND Idle
                TIME ? finish

                ... Calculate maximum possible count value
```

```
    VAL REAL32 elapsed IS REAL32 ROUND (finish MINUS begin) :
    Max.Idle := elapsed * count.rate
    CPU.util := 0.0 (REAL32)

    ... Calculate percentage utilization
    IF
      Max.Idle <> 0.0 (REAL32)
        CPU.util := ((Max.Idle - Real.Idle) * 100.0 (REAL32)) / Max.Idle
      TRUE
        SKIP

    result ! INT ROUND CPU.util
:
```

A3.2 Assembly Language Routine

Mitchell *et al.* [1990] present a Transputer assembly language routine for estimating processor utilization. This approach, although identical in principle to that discussed above, is the more sophisticated of the two. The fundamental difference between the two methodologies is in the implementation of the idle counter process. Here, an assembly language routine is used for this purpose, and the idle counter is incremented if the low priority process queue is empty. In effect, this criterion is identical to that used in the occam 2 Supervisor approach – however, unlike occam 2, assembly language permits access to the scheduling registers. The idle counter process, IdleTime, is listed below :

```
... SC IdleTime
... COMMENT
This PROC extracted from :
"Inside The Transputer", D. Mitchel    ',
Blackwell Scientific, Oxford, 1990

PROC IdleTime (INT IdleCount, E...    :unt, Semaphore)
  ... Vars
  INT L.IdleCount, L.ExtraCount :
  INT Back.Low.Ptr, FrontLowPtr :

  CUY
    ... Initialize
    LDC 0                    -- initialize counters
    STL L.IdleCount
    LDC 0
    STL L.ExtraCount

    ... Repeat ...
    :REPEAT
    LDL Semaphore            -- read Semaphore
    CJ  .END                 -- if zero terminate
    LDLP FrontLowPtr         -- read low priority queue registers
    SAVEL
    LDL FrontLowPtr          -- and compare front one with MinInt
    MINT
    DIFF
    CJ  .INCIC               -- if queue empty inc idle count
    LDL L.ExtraCount         -- otherwise inc extra count
    ADC 1
    STL L.ExtraCount
    J   .PASS

    ... Increment Idle Count
    :INCIC
    LDL L.IdleCount          -- inc idle count
    ADC 1
```

```
        STL L.IdleCount

        ... Scheduling ...
        :PASS                   -- add to end of low priority
        LDC 2                   -- active queue
        LDLP 0
        STARTP
        STOPP                   -- deschedule this process
        J   .REPEAT

        ... Return Counts
        :END
        LDL L.IdleCount         -- return counts and finish
        STL IdleCount
        LDL L.ExtraCount
        STL ExtraCount
:
```

This procedure is used in very much the same way as the idle counter process is used by the Supervisor. As shown below, IdleTime is launched in parallel with the rest of the processes on the Transputer. On terminating (by setting Semaphore to 0), the procedure returns the value of the idle counter (IdleCount), and also an indication of the time taken up by the execution of the procedure itself (ExtraCount).

```
... Initialize
Semaphore   := 1
IdleCount   := 0
ExtraCount  := 0

PAR
  SEQ
    ... Processes to be evaluated
    Semaphore := 0 -- terminate IdleTime ()

  IdleTime (IdleCount, ExtraCount, Semaphore)
```

As for the Supervisor process, it is necessary to establish a calibration value relative to which the actual IdleCount value can be measured. Again, this is done by running IdleTime by itself for a specified amount of time and hardcoding the resultant count values back into the evaluation procedure.

A3.3 COMPARISON

Both of the approaches were tested using the following evaluation process, where the values of Idle Limit and Busy Limit were varied in order to produce a range of processor utilization figures.

```
... Wait / Busy
clock ? time
clock ? AFTER time PLUS Idle.Limit

SEQ i = 0 FOR Busy.Limit
  SKIP
```

The utilization figures returned using both the Supervisor and IdleTime procedures were exactly the same – i.e. the approaches perform identically. The reason for using the Supervisor in the analysis of X-Linda is simply that the author was exposed

to this approach first. For the sake of interest, the results of the evaluation are listed in Figure A3.1 under the following headings :

Idle – Idle Limit x 1000
Busy – Busy Limit x 1000
% – % CPU Utilization

Idle	Busy	%		Idle	Busy	%		Idle	Busy	%
0	1000	100		10	1000	50		0	1000	100
1.25	875	88		10	875	47		1.25	1000	89
2.50	750	75		10	750	43		2.50	1000	80
3.75	625	63		10	625	39		3.75	1000	73
5.00	500	50		10	500	34		5.00	1000	67
6.25	375	38		10	375	28		6.25	1000	62
7.50	250	25		10	250	20		7.50	1000	58
8.75	125	13		10	125	11		8.75	1000	54
10.00	0	0		10	0	0		10.00	1000	50

Table A3.1 : CPU Utilization

APPENDIX 4

A4 X-LINDA PROGRAM STRUCTURE

In section 5.2.3.1, it was indicated that there are two versions of X-Linda in existence – a simulated system running on a single Transputer and the physically distributed version. It was also noted that both of these versions run identical code since the X-Linda modules are simply attached to harnesses that, for the simulated system, launch the X-Linda nodes concurrently on one processor (these nodes communicate via software channels) and, in the distributed case, physically place the nodes and associated communication channels on separate processors. This Appendix shows the overall structure of the simulated and distributed systems.

A4.1 SIMULATED SYSTEM

This system launches the Host process and the X-Linda nodes on a single Transputer – i.e. the entire mesh of processors is simulated on one processor.

```
... EXE X.Linda.Simulated
... Hard.Channel.Protocols
```

```
... SC Host.Process
... Hard.Channel.Protocols

PROC Host.Process ( CHAN OF INT keyboard,
                    CHAN OF ANY screen,
                    ... Network Links )

  ... SC NW.Connection
  ... SC Monitor

  PAR
    ... Launch NW.Connection
    ... Launch Monitor
:
```

```
... SC T8 X.Linda.Node
... Hard.Channel.Protocols

PROC Linda.Node ( VAL INT Proc.Id,
                  ... Input / Output Links )

  ... Processes
  ... SC T8 Interface
  ... SC T8 Queue
  ... SC T8 Out
  ... SC T8 In
  ... SC T8 Rd
  ... SC T8 Computation
  ... SC T8 Challenge.Manager

  PAR
    ... Launch Processes
    ... Interface
    ... Queue
    ... Out
    ... Rd
    ... In
    ... Computation
    ... Challenge.Manager
:
```

```
PAR -- launch all processes concurrently on a single processor
  ... Launch Host.Process
  ... Launch Linda Nodes (0..k-1)
```

A4.2 DISTRIBUTED SYSTEM

In this version, the Host process is launched on the Host *processor*, and the X-Linda nodes are launched on independent Transputers within the network. Notice that the Host process and X-Linda nodes in this version are *identical* to those in the simulated version.

```
EXE X.Linda.Host
... Hard.Channel.Protocols
```

```
... SC Host.Process
... Hard.Channel.Protocols

PROC Host.Process ( CHAN OF INT keyboard,
                    CHAN OF ANY screen,
                    ... Network Links )

  ... SC NW.Connection
  ... SC Monitor

  PAR
    ... Launch NW.Connection
    ... Launch Monitor
:
```

```
... Links to Network
... Host <--> Processor 0
... Host <--> Processor 1024

SEQ
  ... Launch Host Process
```

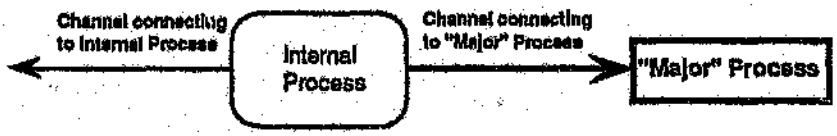
```
... PROGRAM Linda.Network
... Hard.Channel.Protocols

... SC T8 X.Linda.Node
... Hard.Channel.Protocols

PROC Linda.Node ( VAL INT Proc.Id,
                  ... Input / Output Links )

  ... Processes
  ... SC T8 Interface
  ... SC T8 Queue
  ... SC T8 Out
  ... SC T8 In
  ... SC T8 Rd
  ... SC T8 Computation
  ... SC T8 Challenge.Manager

  PAR
    ... Launch Processes
    ... Interface
    ... Queue
    ... Out
    ... Rd
    ... In
    ... Computation
    ... Challenge.Manager
:

... NW.Configuration
PLACED PAR -- physically place the nodes and links on separate processors
  ... Processors (0..k-1)
```

APPENDIX 5

A5 LOW-LEVEL PROCESS DESIGN

In section 5.2.2, the function and operation of the processes that comprise the X-Linda node (i.e. Computation, Out, Rd, In, Challenge Manager, Queue and Interface) were briefly described. This Appendix shows the diagrammatic interaction of these processes, and of the sub-processes resident within each. The intention here is not so much to give the reader a clear and concise understanding of the workings of the X-Linda node, but more to illustrate the manner in which the overall design has been based on low-level process interaction. It is desired that the following illustrations emphasize the fact that the system has been designed with adherence to the principles of good occam programming procedure, and also the suitability of the occam programming model for this role.

The following identification scheme is used in presenting the process design :



A5.1 COMPUTATION PROCESS

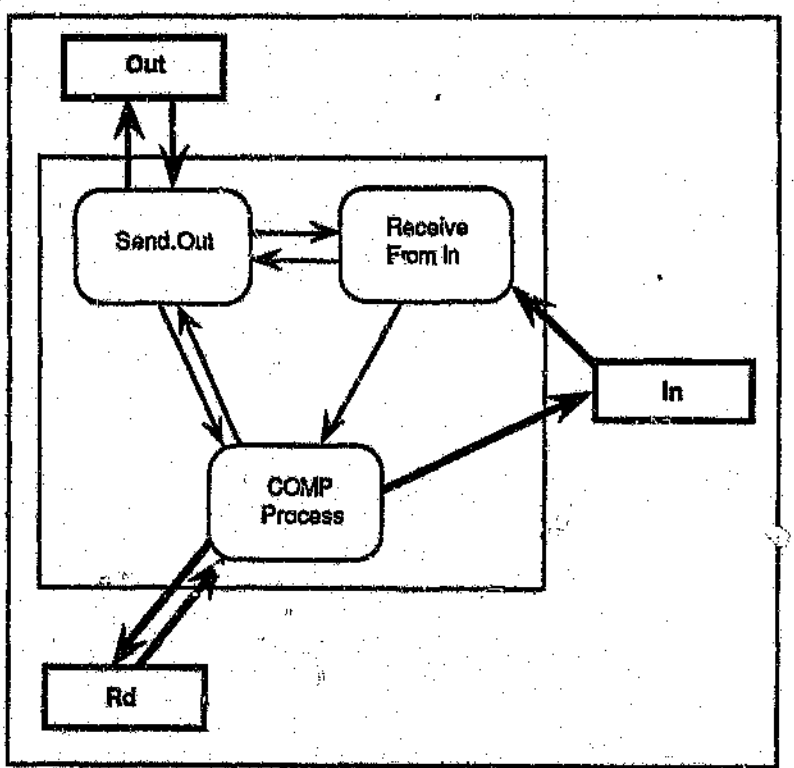


Figure A5.1 : Computation Process – Structure and Interaction

A5.2 OUT PROCESS

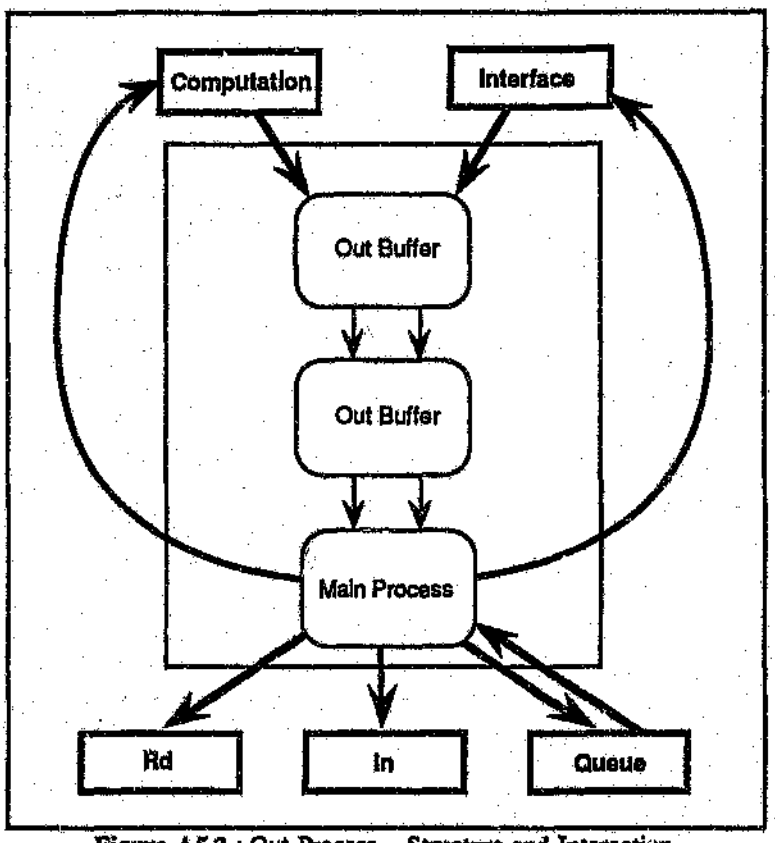


Figure A5.2 : Out Process – Structure and Interaction

A5.3 RD PROCESS

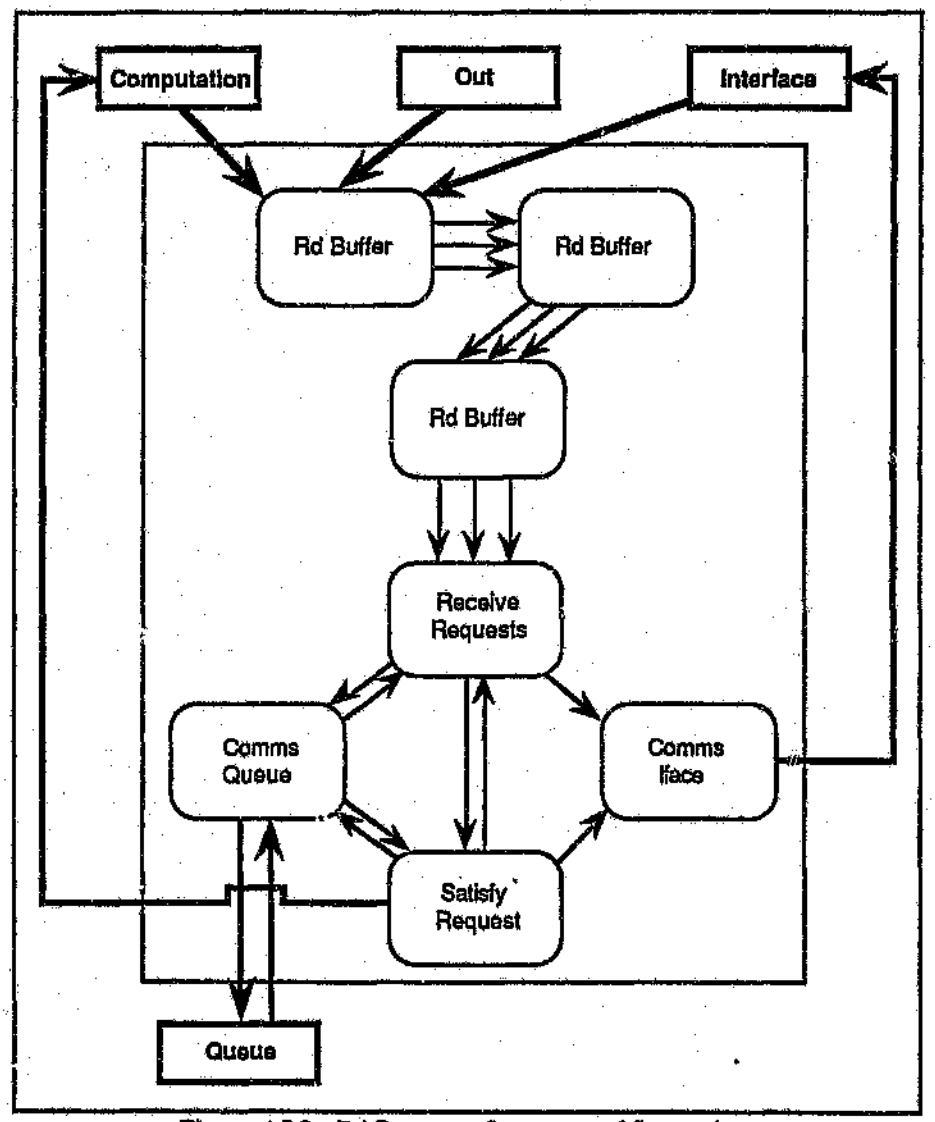


Figure A5.3 : Rd Process – Structure and Interaction

A5.4 In Process

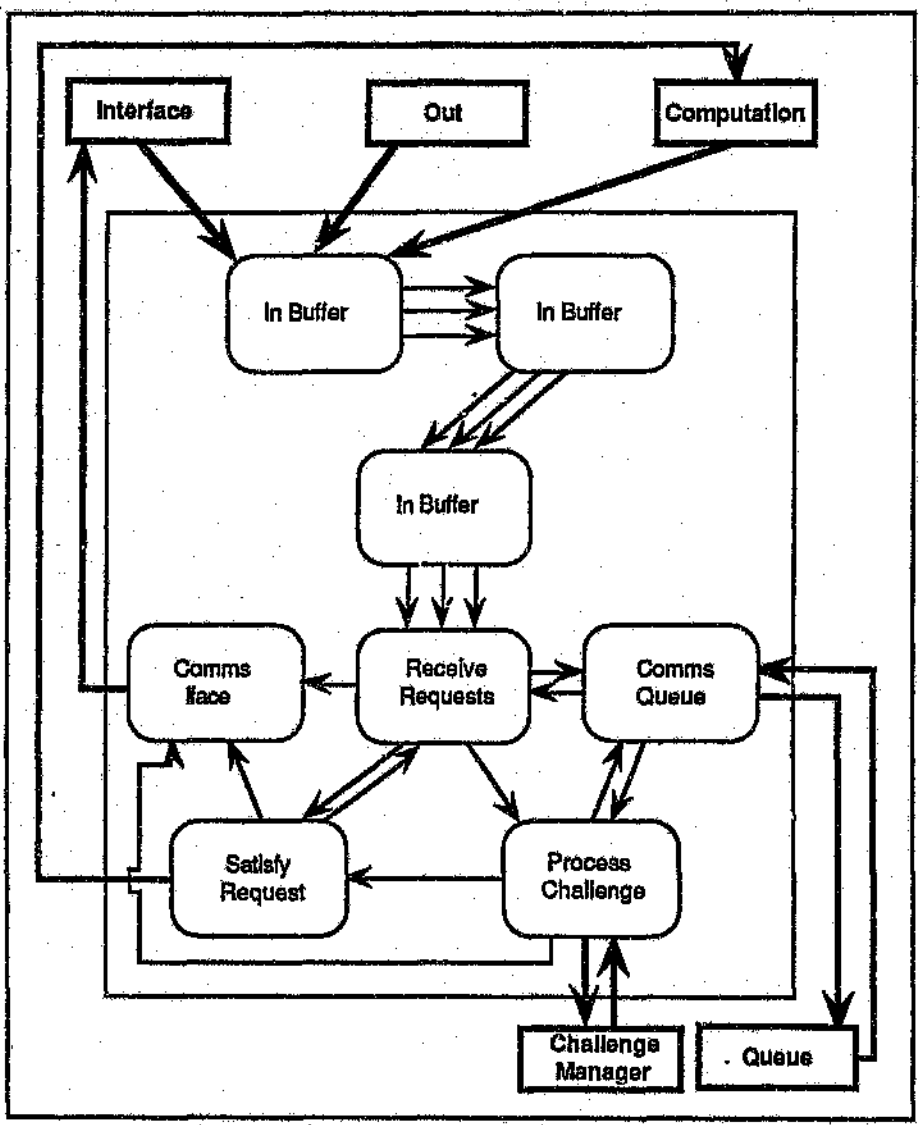


Figure A5.4 : In Process – Structure and Interaction

A5.5 Challenge Manager Process

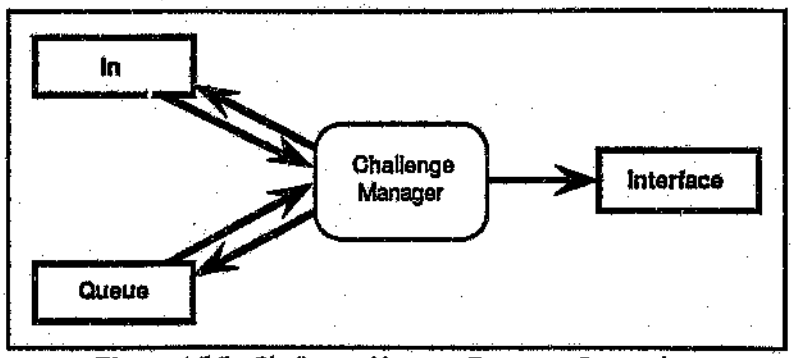


Figure A5.5 : Challenge Manager Process – Interaction

A5.6 Queue Process

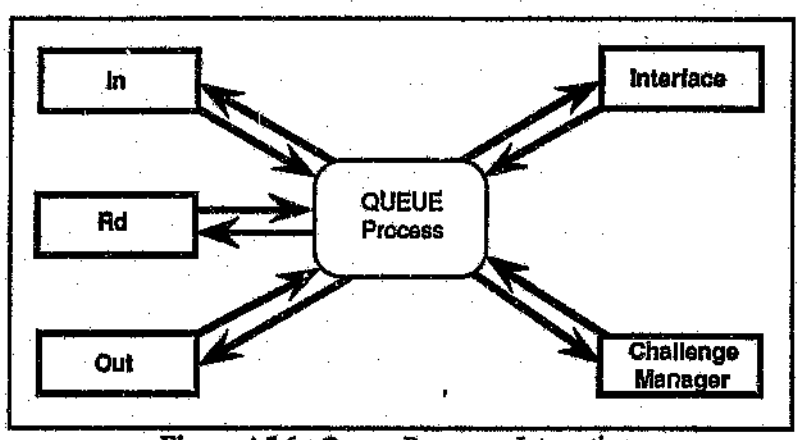


Figure A5.6 : Queue Process – Interaction

A5.7 Interface Process

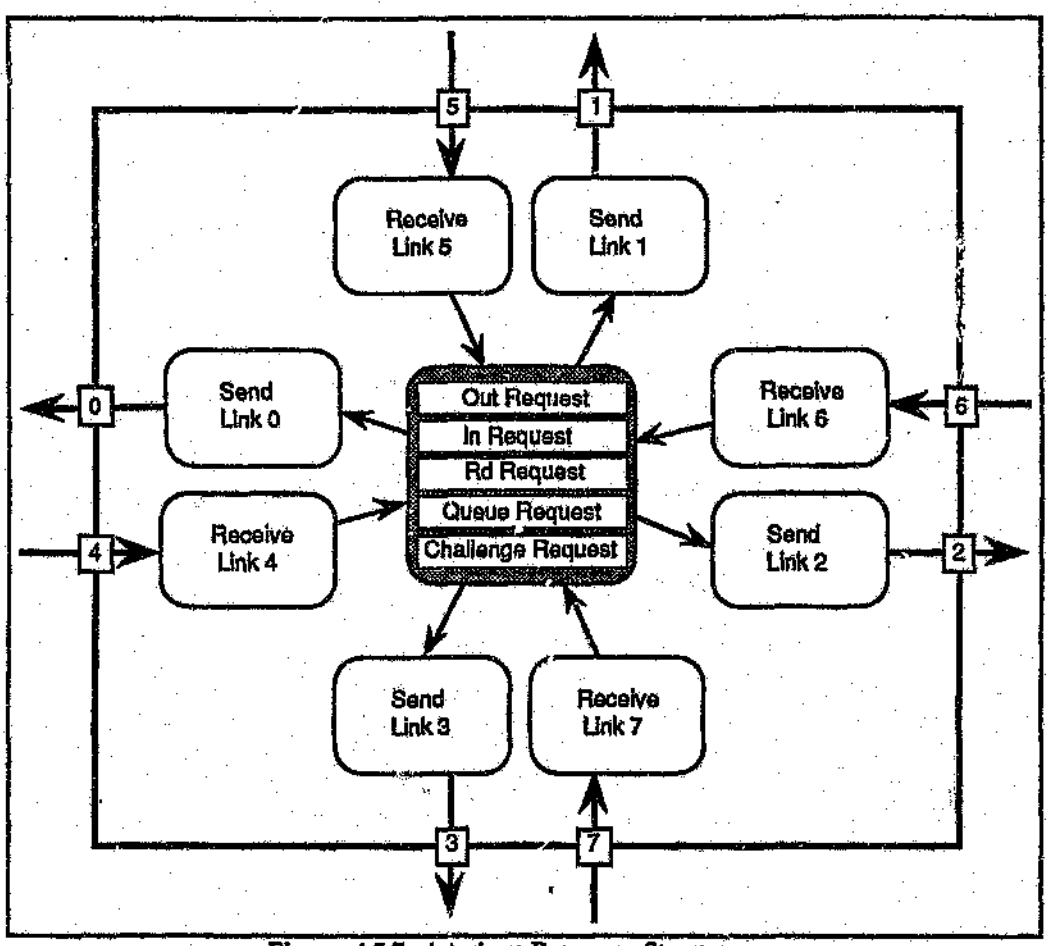


Figure A5.7 : Interface Process – Structure

In Figure A5.7, the central component actually comprises individual sub-processes – the OutRequest, InRequest, RdRequest, QueueRequest and ChallengeRequest sub-processes interact with the Out, In, Rd, Queue and Challenge Manager processes respectively. The illustration has been presented in the above form for simplicity.

APPENDIX 6

A6 EXAMPLE PROGRAMS – CODE LISTINGS

The listings of the example programs (i.e. numerical integration and matrix multiplication) covered in section 7 are given below. This is done largely for the sake of interest – they give some idea of the "feel and flavour" of programming under X-Linda. Furthermore, they illustrate a methodology the must surely be a rarity in the world of programming – the use of Linda primitives embedded in occam 2 programs.

A6.1 NUMERICAL INTEGRATION

The code listing below corresponds to the example described in section 7.1 – i.e. the integration of the function $f(x) = (x / 2^{13}) - 1$ over the interval -2^{21} to $3x2^{21}$, using the Trapezoidal Rule with 2^{22} steps. Recall that, given there are k processors in the system, the worker processes reside on k-1 of the processors; the kth processor functions as both the host and a worker process.

```
VAL tuple.length IS 4 :

... Trapezoidal.Worker
PROC Trapezoidal.Worker ()

  ... f (x)
  INT FUNCTION f (VAL INT x)
  INT result :
  VALOF
    result := (x / 8192) - 1
    RESULT result
:

... Vars
INT a, delta.x, worker.result, sub.interval, start :
[tuple.length] INT Tuple.Data :

SEQ

  ... in sub-range information
  in (Proc.Id, Tuple.Data)
  a             := Tuple.Data [0]
  delta.x       := Tuple.Data [1]
  sub.interval  := Tuple.Data [2]
  start         := Tuple.Data [3]

  ... Compute Integral over sub-range
  worker.result := 0
  SEQ k = start FOR sub.interval
    VAL INT arg IS (a + (k * delta.x)) :
    IF
      k = 0
        worker.result := worker.result + (f (arg))
      TRUE
        worker.result := worker.result + (2 * (f (arg)))

  ... Send out result
  Tuple.Data [0] := worker.result
  VAL index IS Proc.Id + 1000 :
  -- naming convention for sub-results
  out (index, Tuple.Data)
```

```
:

--------------------------------------------------------------------------------

... Trapezoidal.Host
PROC Trapezoidal.Host ()

  ... f (x)
  INT FUNCTION f (VAL INT x)
  INT result :
  VALOF
    result := (x / 8192) - 1
    RESULT result
:

... Consts
VAL n            IS 1 << 22 :
VAL a            IS -(n / 2) :
VAL b            IS (3 * n) / 2 :
VAL delta.x      IS (b - a) / n :-- 2
VAL sub.interval IS n / no.of.nodes :

... Vars
[tuple.length] INT Tuple.Data :
TIMER clock :
INT t0, t1, t2, time.taken :
INT result :
VAL time.tuple IS -99 : -- name of tuple that holds result & time

SEQ

  ... Obtain time to access clock
  clock ? t1
  clock ? t2
  t0 := t2 MINUS t1

  ... Initialize Tuple.Data
  Tuple.Data [0] := a
  Tuple.Data [1] := delta.x
  Tuple.Data [2] := sub.interval

  ... Distribute sub-intervals
  time.taken := 0
  clock ? t1
  SEQ i = 0 FOR no.of.nodes
    SEQ
      Tuple.Data [3] := i * sub.interval
      out (i, Tuple.Data)

  ... Launch Worker (node doubles as Host & Worker)
  Trapezoidal.Worker ()

  ... Receive sub-results
  result := 0
  SEQ i = 0 FOR no.of.nodes
    VAL index IS i + 1000 :
    -- naming convention for sub-results
    SEQ
      in (index, Tuple.Data)
      result := result + Tuple.Data [0]

  ... Perform final computation
  result := result + f (a + (n * delta.x))
  result := result * (delta.x / 2)

  clock ? t2
  time.taken := time.taken + (t2 MINUS (t1 MINUS t0))
```

```
        ... Send out result & time taken
        Tuple.Data [0] := result
        Tuple.Data [1] := time.taken * 64
        Tuple.Data [2] := -1
        Tuple.Data [3] := -1

        out (time.tuple, Tuple.Data)
:
```

```
... Launch Host & Worker processes
SEQ
    IF

        ... Proc.Id = mesh.dim - Worker doubles as Host
        (Proc.Id = mesh.dim)
            -- Note : must not be on the row with the Host Transputer
            -- i.e. the host should not be slowed down by extra communication
            Trapezoidal.Host ()

        ... Otherwise, Worker
        TRUE
            Trapezoidal.Worker ()
```

A6.2 MATRIX MULTIPLICATION

Section 7.2 examined the multiplication of two matrices under X-Linda, and the code listing for this example is shown below. As for the previous algorithm (i.e. numerical integration), the workers reside on k-1 of the processors, while the last processor doubles as a worker and a host process.

```
VAL array.dim IS 32 :

... PROC Matrix.Worker
PROC Matrix.Worker ()

    ... Inner.Product ()
    INT FUNCTION Inner.Product (VAL [array.dim] INT A.row, B.col)
        INT Result.Prod :
        VALOF
            SEQ
                Result.Prod := 0
                SEQ i = 0 FOR array.dim
                    Result.Prod := Result.Prod + (A.row [i] * B.col [i])
            RESULT Result.Prod
    :

    ... Vars
    ... A-row, C-row
    [array.dim] INT A.row, C.row :

    ... B-matrix
    [array.dim][array.dim] INT B.col :

    INT A.index :

    SEQ

        ... Read all B rows
        SEQ i = 0 FOR array.dim
            VAL INT B.index IS array.dim + i :
            -- naming convention for B-columns
            rd (B.index, B.col [i])
```

```
        A.index := Proc.Id

        ... Computation Loop
        WHILE A.index < array.dim
            SEQ

                ... In A-row
                in (A.index, A.row)

                ... Inner Product
                SEQ i = 0 FOR array.dim
                    C.row [i] := Inner.Product (A.row, B.col [i])

                ... out C-row
                VAL C.index IS (2 * array.dim) + A.index :
                -- naming convention for result rows
                out (C.index, C.row)

                A.index := A.index + no.of.nodes
:

--------------------------------------------------------------------------------

... PROC Matrix.Host
PROC Matrix.Host ()

    ... Vars
    TIMER clock :
    INT t0, t1, t2, time.taken :

    ... Matrices
    [array.dim][array.dim] INT A, B, C : -- A * B = C

    ... A-row, B-column
    [array.dim] INT A.row, B.col :

    VAL time.tuple IS -99 : -- name of the tuple that holds "time taken"

    SEQ

        ... Obtain time to access clock
        clock ? t1
        clock ? t2
        t0 := t2 MINUS t1

        ... Initialize Matrix
        SEQ i = 0 FOR array.dim
            SEQ j = 0 FOR array.dim
                VAL k IS (i * array.dim) + j :
                SEQ
                    A [i][j] := k
                    B [i][j] := k

        ... Distribute rows and columns
        clock ? t1
        SEQ i = 0 FOR array.dim
            VAL B.index IS array.dim + i :
            -- naming convention for B-columns
            SEQ

                ... ith row of A
                A.row := A [i]

                ... ith col of B
                SEQ j = 0 FOR array.dim
                    B.col [j] := B [j][i]
```

```
            out (i, A.row)
            out (B.index, B.col)

        ... Launch Worker (this Node doubles as a host and a worker)
        Matrix.Worker ()

        ... Receive results
        SEQ i = 0 FOR array.dim
            VAL C.index IS (2 * array.dim) + i :
            -- naming convention for result rows
            in (C.index, C [i])

        clock ? t2
        time.taken := t2 MINUS (t1 MINUS t0)

        ... Out time taken
        A.row [0] := time.taken * 64 -- microseconds
        out (time.tuple, A.row)
:

... Launch Host & Worker processes
SEQ
  F

    ... Proc.Id = mesh.dim - Worker doubles as Host
    (Proc.Id = mesh.dim)
        -- Note : must not be on the row with the Host Transputer
        -- i.e. the host should not be slowed down by extra communication
        Matrix.Host ()

    ... Otherwise, Worker
    TRUE
        Matrix.Worker ()
```

APPENDIX 7

A7 ORDER OF TS ADDITION

In section 5.1.1.1, the issue of the *order* in which tuples are added to TS was briefly mentioned. This issue is re-examined here, where we consider the question of whether tuples *must* be added to TS in the order in which they are outed. This question (initially raised by Faasen [1990c]) is discussed purely in the context of an interesting side-issue – it has no bearing on the X-Linda implementation (where tuples *are* added to TS in the order in which they are outed). However, the issue does raise some interesting points regarding the semantics of Linda, and, as such, is worthy of deeper investigation.

The specific case that is of interest here is that where it is not possible to distinguish between tuples – i.e. they have the same names (but different data fields). Note that the term *tuple name*, although prominent in early Linda literature, has little meaning in many current Linda implementations. In this Appendix, the term is used in its broadest sense, and the specification that two tuples have the same names simply indicates that the tuples have the necessary characteristics for a single template to successfully match both of them. Under these conditions, it is obvious that the order in which tuples are added to TS may have a significant effect on the on the results produced by an application program. However, as shown below, it is claimed that, *semantically*, this order is irrelevant.

A7.1 AN EXAMPLE SCENARIO

The question of tuple ordering can be expressed by means of a simple example. Consider the following sets of statements :

(1)	out ("A", 1) out ("A", 2)	(2)	out ("A", 2) out ("A", 1)

The execution of (1) *or* (2) has the same effect on TS – i.e. two tuples, ("A", 1) and ("A", 2), are added to TS. Now, if the statement in ("A", ?Int i) is invoked, the order in which the tuples were inserted into the TS is *irrelevant*. Linda's semantics specify that the tuple request will succeed on there being a matching tuple in the TS, and, if there are more than one such tuples, one of these is selected arbitrarily. Consequently, (1) is semantically equivalent to (2) – i.e. the order in which tuples are inserted into TS is not a semantic consideration.

Notice that the above argument assumes that a request was issued *after* the addition of the tuples to TS. Obviously, it is possible that the request could have been invoked *prior* to the insertion of the tuples. However, as shown by Hazelhurst [1990] the semantics of the in operation are the *same* whether the in request precedes or succeeds the insertion of the tuples into TS. Hence, in this instance, (1) and (2) are also equivalent.

A7.2 DISCUSSION

Linda's semantics do not guarantee tuple ordering – this aspect remains the responsibility of the programmer. The order in which the tuples are inserted into TS will obviously have some effect on the execution of the program, as shown in the example below. Assume that ("A",1) and ("A",2) are present in the TS, and the following segment of code is executed :

```
in ("A", ?int i)
in ("A", ?int j)
k<-i-j
```

Depending on the order in which the tuples are added to the TS, k may the assume the values -1 or 1. Linda's semantics allow either result – it is up to the programmer to provide more explicit sequencing if required.

It is interesting to note that Jerry Leichter [personal communication, May 1990] states that, with regard to his own Linda implementation [Leichter 1989], and that of Nicholas Carriero's [Carriero 1987], the order in which requested tuples are returned is *usually* the *reverse* of that in which they were outed. Conversely, the Cogent Research XTM system [Cogent 1990] *guarantees* the order of requested of tuples.

REFERENCES

S. Ahuja, N. Carriero and D. Gelernter [1986]
Linda and Friends, IEEE Computer, 19 (8), August, 26-34

S. Ahuja, N. Carriero, D. Gelernter and V. Krishnaswamy [1988]
Matching Language and Hardware for Parallel Computation in the Linda Machine, IEEE Trans. Computers, 37 (8), August, 921-929

H. Bal, J. Steiner and A. Tanenbaum [1989]
Programming Languages for Distributed Computing Systems, ACM Computing Surveys, 21 (3), September, 261-322

R. Bjornson, N. Carriero, D. Gelernter and J. Leichter [1987]
Linda, The Portable Parallel, Yale Univ. Dept. of Computer Science Research Report 520, February

L. Bormann and M. Herdieckerhoff [1989]
Parallel Processing Performance in a Linda System, Proc. 1989 Int. Conf. Parallel Processing (August 8-12, Penn State University), Vol. 1, 151-158

N. Carriero [1987]
Implementing Tuple Space Machines, Yale Univ. Dept. of Computer Science Research Report 567, December (also a 1987 Yale Univ. Ph.D Thesis)

N. Carriero and D. Gelernter [1986]
The SINet's Linda Kernel, ACM Trans. Comp. Sys., 4 (2), May, 110-129

N. Carriero and D. Gelernter [1988a]
Applications Experience with Linda, in Proc. ACM SIGPLAN PPEALS, ACM SIGPLAN Notices, 23 (9), September, 173-187

N. Carriero and D. Gelernter [1988b]
How to write Parallel Programs : A Guide to the Perplexed, Yale Univ. Dept. of Computer Science Research Report 628, November

N. Carriero and D. Gelernter [1989]
Linda in Context, Comm. ACM, 32 (4), April, 444-458

N. Carriero, D. Gelernter and J. Leichter [1986]
Distributed Data Structures in Linda, Proc. ACM Symp. Princ. Prog. Lang., (January 13-15, St. Petersburg, Fla.), 236-242

Chorus [1989a]
Linda-C Documentation, Chorus Supercomputer Inc, New York,

Chorus [1989b]
The ComputeServer : Developer's Introduction, Chorus Supercomputer Inc, New York,

P. Clayton, P. Wentworth, G. Wells and F. de-Heer-Menlah [1990]
An Implementation of Linda Tuple Space under the Helios Operating System, Rhodes Univ. Dept. of Computer Science Technical Document PPG 90/8, October

Cogent [1989]
XTM Product Specification, Cogent Research Inc., Beaverton, Oregon

Cogent [1990]
Kernel Linda Specification - Version 4.0, Cogent Research Inc, Technical Note 89.17, June

C. Davidson [1989]
Technical Correspondence on "Linda in Context", Comm. ACM, 32 (10), October, 1249-1252

C. Faasen [1987]
Sorting on Transputer Arrays, Research report submitted in partial fulfilment of the requirements for the degree of BSc Honours, Dept. of Computer Science, Univ. of the Witwatersrand, November

C. Faasen [1989a]
Automatic Generation of Occam Configuration Commands, Univ. of the Witwatersrand, Dept. of Computer Science Technical Report 1989-03, September

C. Faasen [1989b]
Linda - An Alternative Approach to Parallel Programming, Technical report submitted in partial fulfilment of the requirements for the degree of MSc, Dept. of Computer Science, Univ. of the Witwatersrand, November

C. Faasen [1990a]
An Introduction to Transputers, Univ. of the Witwatersrand Computer Centre Communique, 165, Autumn, 1-5

C. Faasen [1990b]
Subject : *Out-Set Protocol*, Linda Users Group Bulletin Board, 11 May

C. Faasen [1990c]
Subject : *Re : Linda Semantics Queries*, Linda Users Group Bulletin Board, 26 May

C. Faasen [1990d]
Linda – An Overview and Proposed Transputer-Based Implementation, Proc. Fifth National MSc and Ph.D Computer Science Students Conference, Port Elizabeth, August, 100-116 (also appeared as Univ. of the Witwatersrand Dept. of Computer Science Technical Report 1990-05, June)

C.J. Fleckenstein and D. Hemmendinger [1989]
Using a Global Name Space for Parallel Execution of UNIX Tools, Comm. ACM, 32 (9), September, 1085-1090

D. Gelernter [1985]
Generative Communication in Linda, ACM Trans. Prog. Lang. Syst., 7 (1), January, 80-112

D. Gelernter [1988]
Getting the Job Done, BYTE, 13 (12), November, 301-309

D. Gelernter [1989a]
Multiple Tuple Spaces in Linda, Yale Univ. Dept. Computer Science Research Report, January (also appeared as *Elastic Computing Envelopes and their Operators (Linda 3)*)

D. Gelernter [1989b]
Information Management in Linda, Proc. AI and Communicating Process Architectures (London), July, to appear

F. Hayes [1988]
The Crossbar Connection, BYTE, 13 (12), November, 278-279

S. Hazelhurst [1990]
A Proposal for the Formal Specification of the Semantics of Linda, Univ. of the Witwatersrand Dept. of Computer Science Technical Report 1990-14, October

C. Hoare [1978]
Communicating Sequential Processes, Comm. ACM, 21 (8), August, 666-677

INMOS [1988a]
The Transputer Databook, INMOS Limited

INMOS [1988b]
Occam 2 Reference Manual, INMOS Limited, Prentice Hall, London

INMOS [1989]
Transputer Handbook, INMOS Limited

G. Jones and M. Goldsmith [1988]
Programming in Occam 2, G. Jones and M. Goldsmith, Prentice Hall, London

K. Kahn and M. Miller [1989]
Technical Correspondence on "Linda in Context", Comm. ACM, 32 (10), October, 1253-1255

T. King and J. Powell [1990]
The Helios Distributed Operating System – Fundamentals, Perihelion Software Limited

D. Lakier [1989a]
Message Transmission Times on the INMOS T800-20 Transputer, Univ. of the Witwatersrand, Dept. of Computer Science Technical Report 1989-06, October

D. Lakier [1989b]
Performance of Pipelining Messages Through a Transputer Network, Univ. of the Witwatersrand, Dept. of Computer Science Technical Report 1989-07, November

J. Leichter [1989]
Shared Tuple Memories, Shared Memories, Buses and LANs - Linda Implementations Across the Spectrum of Connectivity, Yale Univ. Dept. of Computer Science Research Report 714, July

J. Leichter [1990]
Subject : *re : Linda Semantics Question*, Linda Users Group Bulletin Board, 18 May

W. Leler [1990]
Linda Meets Unix, Computer, 23 (2), February, 43-45

D. Mitchell, J. Thompson, G. Manson and G. Brookes [1990]
Inside The Transputer, Blackwell Scientific, Oxford, 76-80

NTSC [1990]
A Guide to Occam Programming Style and Software Documentation, National Transputer Support Centre, Sheffield; appeared in SERC/DTI Transputer Initiative Mailshot, May, 37-52

Parsytec [1989a]
SuperCluster Series Hardware Documentation, Parsytec GmbH

Parsytec [1989b]
MultiTool 5.0 Technical Documentation, Parsytec GmbH

D. Pountain [1989]
Occam 2, BYTE, October, 279-284

D. Pountain [1990]
Virtual Channels : The Next Generation of Transputers, BYTE, April, 3-12

M. Quinn [1987]
Designing Efficient Algorithms for Parallel Computers, McGraw-Hill, New York, 43

A. Rabagliati [1990a]
Subject : *How busy is your Transputer*, North American Transputer User's Group Bulletin Board, 23 October

A. Rabagliati [1990b]
Subject : *New Electronics Article, Sept 1990*, North American Transputer User's Group Bulletin Board, 23 October

Scientific [1989]
Linda-C Product Summary, Scientific Computing Associates Inc., New Haven, CT

E. Shapiro [1989]
Technical Correspondence on "Linda in Context", Comm. ACM, 32 (10), Oct. 1989, 1244-1249

M. Spiegel [1974]
Advanced Calculus, McGraw-Hill, London, 84-85

J. Tonsing [1989]
A Linda Implementation for Transputers, Report submitted in partial fulfilment of the degree of BSc (Elec. Eng.), Univ. of Pretoria, October

D. Ushijima [1989]
Sharing Supercomputing Power, MacWorld, June, 83-85

G. Wells [1990]
An Implementation of Linda, Proc. Fifth National MSc and Ph.D Computer Science Students Conference, Port Elizabeth, August, 302-307

P. Wentworth [1989]
Prototyping a Linda System, Proc. Concurrent Computing 89 (5 September, CSIR, Pretoria)

P. Wentworth [1990]
Parallelism via Linda - A Transputer Implementation (Status Report), SACLA Conf., Bloemfontein, June

J. Williams, S. Bogoch and L Bason [1989]
Supercomputing on the Cheap : MPW C-Linda and the Chorus ComputeServer, MacTech Quarterly, Autumn, 60-63

S. Zenith [1990a]
Subject : *Concise description of Linda operations (The Simplicity of Linda)*, Linda Users Group Bulletin Board, 19 April

S. Zenith [1990b]
Linda Coordination Language; Subsystem Kernel Architecture (on Transputers), Yale Univ. Dept. of Computer Science Research Report 794, May

S. Zenith [1990c]
Programming with Ease : Semiotic Definition of the Language, Yale Univ. Dept. of Computer Science Research Report 809, July

LINDA BIBLIOGRAPHY

The following is a comprehensive list of Linda-related literature. The information has been gathered from a variety of sources, with much of the input received in response to requests posted to the Linda User's Group Bulletin Board. The source of the literature has, where possible, been verified – however, in some cases, this was not possible (and in infrequent cases, details have by necessity been left incomplete).

S. Ahuja, N. Carriero and D. Gelernter, *Linda and Friends*, IEEE Computer, 19 (8), Aug. 1986, 26-34

S. Ahuja, N. Carriero, D. Gelernter and V. Krishnaswamy, *Progress Towards a Linda Machine*, Proc. Int. Conf. Comput. Design, Oct. 1986, 97-101

S. Ahuja, N. Carriero, D. Gelernter and V. Krishnaswamy, *Matching Language and Hardware for Parallel Computation in the Linda Machine*, IEEE Trans. Computers, 37 (8), Aug. 1988, 921-929

V. Ambriola, P. Ciancarini and M. Danelutto, *Design and Distributed Implementation of the Parallel Logic Language Shared Prolog*, Proc. ACM/SIGPLAN Practice and Principles of Parallel Programming, 23 (9), 1990, 40-49

P. Bercovitz, *TupleScope User's Guide*, Yale Univ. Dept. of Computer Science, *further details unknown*

P. Bercovitz and N. Carriero, *TupleScope : A Graphical Monitor and Debugger for Linda-Based Parallel Programs*, Yale Univ. Dept. of Computer Science Research Report 782, April 1990

D. Berndt, *C-Linda Reference Manual (DRAFT) Beta Version 2. 0*, Scientific Computing Associates Inc., New Haven, Jan. 1989

R. Bjornson, *A Linda User's Manual*, Scientific Computing Associates Inc., New Haven, June 1987

R. Bjornson, *Experience with Linda on the iPSC/2*, Yale Univ. Dept. of Computer Science Research Report 698, March 1989

R. Bjornson, N. Carriero, D. Gelernter and J. Leichter, *Linda, The Portable Parallel*, Yale Univ. Dept. of Computer Science Research Report 520, Feb. 1987

R. Bjornson, N. Carriero and D. Gelernter, *Linda on Distributed Memory Systems*, Proc. 1988 Workshop on Hypercube Multiprocessors, *further details unknown*

R. Bjornson, N. Carriero and D. Gelernter, *The Implementation and Performance of Hypercube Linda*, Proc. Fourth Conf. Hypercube Concurrent Computers and Applications, March 1989 (also appeared as Yale Univ. Dept. of Computer Science Research Report 690)

S. Bogoch *et al.*, *Supercomputers Get Personal*, BYTE, May 1990

L. Borrmann and M. Herdieckerhoff, *Parallel Processing Performance in a Linda System*, Proc. 1989 Int. Conf. Parallel Processing (Aug. 8-12, Penn State Univ.), Vol. I, 1989, 151-158

L. Borrmann and M. Herdieckerhoff and A. Klein, *Tuple Space integrated into Modula-2*, Implementation of the Linda Concept on a Hierarchical Multiprocessor, Proc. CONPAR '88 (Manchester, U.K.), Cambridge Univ. Press, 1988

M. Braner, *Brenda -- A Tool for Parallel Programming*, Cornell Theory Centre, *further details unknown*

P. Busalacchi, *Linda on a Transputer-Based PC*, Proc. Australian Transputer and Occam User Group, July 1989

N. Carriero, *Implementing Tuple Space Machines*, Yale Univ. Dept. of Computer Science Research Report 567, Dec. 1987 (also a 1987 Yale Univ. Ph.D Thesis)

N. Carriero and D. Gelernter, *The S/Net's Linda Kernel*, ACM Trans. Comput. Syst., 4 (2), May 1986, 110-129

N. Carriero and D. Gelernter, *Linda on Hypercube Multicomputers*, Hypercube Multiprocessors, SIAM, 1986

N. Carriero and D. Gelernter, *How to write Parallel Programs : A Guide to the Perplexed*, Yale Univ. Dept. of Computer Science Research Report 628, Nov. 1988

N. Carriero and D. Gelernter, *Applications Experience with Linda*, Proc. ACM SIGPLAN PPEALS, ACM SIGPLAN Notices, 23 (9), Sept. 1988, 173-187

N. Carriero and D. Gelernter, *Integrating Multiple Tuple Spaces, the File System and Process Management in a Linda-Based Operating System*, Yale Univ. Dept. of Computer Science Research Report, Feb. 1988

N. Carriero and D. Gelernter, *How to Write Parallel Programs: A Survey of the Three Main Techniques*, Yale Univ. Dept. of Computer Science Technical Memo, Aug. 1988

N. Carriero and D. Gelernter, *Linda in Context*, Comm. ACM, 32 (4), April 1989, 444-458

N. Carriero and D. Gelernter, *Technical Correspondence on "Linda in Context"*, Comm. ACM, 32 (10), Oct. 1989, 1255-1258

N. Carriero and D. Gelernter, *Tuple Analysis and Partial Evaluation Strategies in the Linda Precompiler*, Second Workshop on Languages and Compilers for Parallelism, MIT Press, Aug. 1989

N. Carriero and D. Gelernter, *Coordination Languages and their Significance*, Yale Univ. Dept. of Computer Science Research Report 176, *further details unknown*

N. Carriero, D. Gelernter and J. Leichter, *Distributed Data Structures in Linda*, Proc. ACM Symp. Principles of Programming Languages, Jan. 13-15, St. Petersburg, Florida, 1986, 236-242

Chorus Supercomputer Inc., New York, *Linda-C Documentation*, 1989

Chorus Supercomputer Inc., New York, *The ComputeServer : Developer's Introduction*, New York, 1989

P. Clayton, P. Wentworth, G. Wells and F. de-Heer-Menlah, *An Implementation of Linda Tuple Space under the Helios Operating System*, Rhodes Univ. Dept. of Computer Science Technical Document PPG 90/8, Oct. 1990

Cogent Research Inc., Beaverton, *XTM Product Specification*, 1989

Cogent Research Inc., Beaverton, *Process creation in QIX*, Technical Note 89. 3, 1989

Cogent Research Inc., Beaverton, *Kernel Linda Specification -- Version 4. 0*, Technical Note 89. 17, June 1990

C. Davidson, *Technical Correspondence on "Linda in Context"*, Comm. ACM, 32 (10), Oct. 1989, 1249-1252

F. de-Heer-Menlah, *Analyzing Communication Flow and Process Placement in Linda Programs on Transputers*, Proc. Fifth National MSc and Ph.D Computer Science Students Conference, Port Elizabeth, Aug. 1990, 39-45 (also appeared as Rhodes Univ. Dept. of Computer Science Technical Document 90/4, 1990)

C. Faasen, *Linda – An Overview and Proposed Transputer-Based Implementation*, Proc. Fifth National MSc and Ph.D Computer Science Students Conference, Port Elizabeth, Aug. 1990, 100-116 (also appeared as Univ. of the Witwatersrand Dept. of Computer Science Technical Report 1990-05, June 1990)

A. Factor and D. Gelernter, *The Parallel Process Lattice as an Organizing Scheme for Real-time Knowledge Daemons*, Yale Univ. Dept. of Computer Science Technical Memo, March 1988

C. Fleckenstein and D. Hemmendinger, *Using a Global Name Space for Parallel Execution of UNIX Tools*, Comm. ACM, 32 (9), Sept. 1989, 1085-1090

D. Gelernter and A. Bernstein, *Distributed Communications via Global Buffer*, ACM SIGACT-SIGOPS Symp. Principles of Distributed Computing, Ottawa, Canada, Aug. 1982, 10-18

D. Gelernter, *An Integrated Microcomputer Network for Experiments in Distributed Programming*, Ph.D. Dissertation, SUNY, Stony Brook, Dept. Comp. Sci, 1983

D. Gelernter, *Dynamic Global Name Spaces on Network Computers*, Proc. 1984 Int. Conf. Parallel Processing, Aug. 1984, 25-31

D. Gelernter, *Generative Communication in Linda*, ACM Trans. Prog. Lang. Syst., 7 (1), Jan. 1985, 80-112

D. Gelernter, N. Carriero, S. Chandran, S. Chang, *Parallel Programming in Linda*, Proc. Int. Conf. Parallel Processing, Aug. 1985, 255-263

D. Gelernter, *Domesticating Parallelism*, Computer, 19 (8), Aug. 1986, 12-16

D. Gelernter, *Programming for Advanced Computing*, Scientific American, 257 (4), Oct. 1987, 65-71

D. Gelernter, S. Jaganathan and T. London, *Environments as First Class Objects*, Proc. ACM Symp. Principles of Programming Languages, Jan. 1987

D. Gelernter, *Getting the Job Done*, BYTE, 13 (12), Nov. 1988, 301-309

D. Gelernter, *Multiple Tuple Spaces in Linda*, Yale Univ. Dept. Computer Science Research Report, Jan. 1989 (also appeared as *Elastic Computing Envelopes and their Operators (Linda 3)*)

D. Gelernter, *Information Management in Linda*, Proc. AI and Communicating Process Architectures (London), July 1989

D. Gelernter, *The Metamorphosis of Information Management*, Scientific American, 261 (2), Aug. 1989, 54-61

D. Gelernter, N. Carriero and C. Ashcraft, *Is Explicit Parallelism Natural ? Hybrid DB Search and Sparse LDL^T Factorization using Linda*, Yale Univ. Dept. of Computer Science research report, Jan. 1989

D. Gelernter and J. Philbin, *Spending Your Free Time*, Byte, May 1990, 213-219

D. Gelernter, *Ada-Linda : Motivation, Informal Description and Examples*, Yale University Technical Report, *further details unknown*

F. Hayes, *The Crossbar Connection*, BYTE, 13 (12), Nov. 1988, 278-279

S. Hazelhurst, *A Proposal for the Formal Specification of the Semantics of Linda*, Univ. of the Witwatersrand Dept. of Computer Science Technical Report 1990-14, Oct. 1990

S. Hupfer, *Melinda : Linda with Multiple Tuple Spaces*, Yale Univ. Dept. of Computer Science Research Report 766, 1990

S. Jaganathan, *Semantics and Analysis of First-Class Tuple Spaces*, Yale Univ. Dept. of Computer Science Research Report 783, April 1990

R. Jellinghaus, *Eiffel Linda : An Object Oriented Linda Dialect*, Yale Univ. Dept. of Computer Science Undergraduate Thesis, 1990 (also to appear in SIGPLAN NOTICES)

M. Frans Kaashoek, H. Bal and A. Tanenbaum, *Experiences with the Distributed Data Structure Paradigm in Linda*, Workshop on Experiences with Distributed and Multiprocessor Systems (WEBDMS), Usenix Association, Ft. Lauderdale, FL, Oct. 1989, 175-191

K. Kahn and M. Miller, *Technical Correspondence on "Linda in Context"*, Comm. ACM, 32 (10), Oct. 1989, 1253-1255

L. Kale, *Technical Correspondence on "Linda in Context"*, Comm. ACM, 32 (10), Oct. 1989, 1252-1253

V. Krishnaswamy, S. Ahuja, N. Carriero, and D. Gelernter, *Architecture of a Linda Coprocessor*, Proc. 15th Annual Int. Symp. Comput. Arch., 1988

J. Leichter, *The VAX Linda-C User's Guide*, Yale Univ. Dept. of Computer Science Research Report 615, March 1988

J. Leichter, *Shared Tuple Memories, Shared Memories, Buses and LANs – Linda Implementations Across the Spectrum of Connectivity*, Yale Univ. Dept. of Computer Science Research Report 714, July 1989

W. Leler, *PIX, the latest NeWS*, IEEE Computer Science Conference (COMPCON), March 1988

W. Leler, *Linda Meets Unix*, Computer, 23 (2), Feb. 1990, 43-45

W. Leler, *A System-Level Standard Based on Linda*, 1990 NATUG Conference, Santa Clara

W. Leler, *Topology-Independent Programming with Linda*, 1990 Workshop on Standards for MIMD Computers, Abingdon, U. K.

S. Lucco, *A Heuristic Linda Kernel for Hypercube Multiprocessors*, Proc. 1986 Workshop Hypercube Multiprocessors, Sept. 1986

M. Manthey, H. Pedersen and G. Gunnlaugsson, *ARIADNE – A Linda Kernel for Transputer Networks*, Univ. of Aalborg (Denmark) Dept. of Mathematics and Computer Science Technical Report, 1990

S. Matsuoka, *Tuple Space Communication in Distributed Object-Oriented Computing*, Ph.D thesis, Dept. of Information Science, Univ. of Tokyo, *further details unknown*

S. Matsuoka and S. Kawai, *Using Tuple Space Communication in Distributed Object-Oriented Languages*, OOPSLA '88 Proceedings, ACM SIGPLAN Notices, 23 (11), Nov. 1988, 173-187

J. Narem, *DB: A Parallel News Database in Linda*, Yale Univ. Dept. of Computer Science Technical Memo, Aug. 1988

K. Obermeier, *Side by Side*, BYTE, 13 (12), Nov. 1988, 275-283

Scientific Computing Associates Inc., New Haven, *Linda-C Product Summary*, 1989
D. Stemple *et al.*, *Functional Addressing in Gutenberg : Inter-process Communication Without Process Identities*, IEEE Trans. Software Eng., 11, Nov. 1986

E. Shapiro, *Technical Correspondence on "Linda in Context"*, Comm. ACM, 32 (10), Oct. 1989, 1244-1249

G. Singh and A. Rangaramujan, *An Efficient Linda Implementation in a Distributed Memory System*, Computational Mechanics Institute, Southampton UK, Sept. 1990

G. Sutcliffe, J. Pinakis and N. Lewins, *Prolog-Linda : An Embedding of Linda in muProlog*, Univ. of Western Australia, Dept. of Computer Science Technical Report 89/14, 1989

J. Tonsing, *A Linda Implementation for Transputers*, Report submitted in partial fulfilment of the degree of BSc (Elec. Eng.), Univ. of Pretoria, Oct. 1989

D. Ushijima, *Sharing Supercomputing Power*, MacWorld, June 1989, 83-85

G. Wells, *An Implementation of Linda*, Proc. Fifth National MSc and Ph.D Computer Science Students Conference, Port Elizabeth, Aug. 1990, 302-307

P. Wentworth, *Prototyping a Linda System*, Proc. Concurrent Computing 89 (5 Sept., CSIR, Pretoria), 1989

P. Wentworth, *Parallelism via Linda – A Transputer Implementation (Status Report)*, SACLA Conf., Bloemfontein, June 1990

D. Weston, *Translate Sequential Programs to Parallel*, Electronic Design, March 1990

R. Whiteside and J. Leichter, *Using Linda for Supercomputing on a Local Area Network*, Proc. Supercomputing '88, Nov. 1988 (also Yale Univ. Dept. of Computer Science Technical Report 638, June 1988)

I. Williams, S. Bogoch and I. Bason, *Supercomputing on the Cheap : MPW C-Linda and the Chorus ComputeServer*, MacTech Quarterly, Autumn 1989, 60-63

A. Xu, *A Fault Tolerant Network Kernel for Linda*, Technical Report 424, Massachusetts Institute of Technology Laboratory of Computer Science, Aug. 1988 (also a 1988 MIT Master's thesis)

A. Xu and B. Liskov, *A Fault Tolerant, Distributed Implementation of Linda*, Technical Report, Massachusetts Institute of Technology Laboratory of Computer Science, Aug. 1988

A. Xu and B. Liskov, *A Design for a Fault Tolerant, Distributed Implementation of Linda*, Proc. Nineteenth Int. Symp. on Fault Tolerant Computing, IEEE, June 1989, to appear (also published as Massachusetts Institute of Technology Laboratory of Computer Science Programming Methodology Group Memo 65)

S. Zenith, *Linda Coordination Language : Subsystem Kernel Architecture (on Transputers)*, Yale Univ. Dept. of Computer Science Research Report 794, May 1990

S. Zenith, *Programming with Ease : Semiotic Definition of the Language*, Yale Univ. Dept. of Computer Science Research Report 809, July 1990