Bulk Primitives in Linda Run-Time Systems

Antony Ian Taylor Rowstron

Submitted for the Degree of Doctor of Philosophy

THE UNIVERSITY *of York*

Department of Computer Science

October 1996

Abstract

This thesis investigates techniques for the efficient implementation of the Linda parallel process coordination model for *open, distributed* computing systems.

The principal focus of the research is on the use of the bulk movement of tuples within open systems which, contrary to intuition, can result in significant efficiency gains for a large class of problems. The emphasis on *open* systems — those in which future history of process creation and deletion cannot be known at compile-time — is due to the current interest in extending the Linda model to encompass widely distributed computing, as exemplified by the 'Network Computer' notion. However, such open systems place severe constraints on the types of optimisation available relative to closed systems — in particular, the very powerful compile-time analysis techniques previously used are no longer feasible.

Methods for the construction of efficient Linda kernels are introduced based on a novel method of *dynamically* classifying tuple spaces according to their locality, which allows the run-time movement of tuple spaces' locations within the distributed kernel. An important consequence of the proposed technique is that it does not require any 'global' information — it works solely on information *locally* available to each component of the distributed kernel. Equally importantly, the scheme is entirely transparent to the programmer, and therefore requires no user-supplied 'hints' or 'pragmas'.

The implemented kernel is fully distributed, consequently tuples within a particular tuple space may be stored on several physical nodes. The kernel supports standard Linda with multiple tuple spaces, the `collect` primitive, and another primitive called `copy-collect`. The justification for the addition of `copy-collect` is the multiple `rd` problem which is described in detail in this dissertation. No acceptable way of overcoming the multiple `rd` problem without the use of the `copy-collect` primitive has been published.

The performance of the implemented kernel is shown to be significantly better than the performance of a kernel that does not use the bulk movement of tuples, and through using a "real-world" example the kernel is shown to provide, under some circumstances, better performance than the best commercially available closed implementation which uses compile time analysis.

Finally, an extension of the concept of classifying tuple spaces is presented, which generalises the concept leading to a detailed proposal for a multi-layer hierarchical kernel, which is more scalable than current traditional implementations.

Contents

List of Figures

List of Tables

List of Programs

Dedication

To my parents, Jean and Peter Rowstron, for their support and help and to my gran, Connie Franks.

Acknowledgements

There seems to be so many people to thank, and if I forget anyone then please forgive me. First I must thank Alan Wood, my supervisor, for helping, encouraging, generally keeping me going for three years, and for reading this thesis and providing many useful comments and suggestions.

I would like to thank Andrew Douglas, a member of the Linda team and always available to give advice, support and to escape to the pub for a drink.

I must thank Simon O'Keefe (plus Jackie and Natalie) and Daniel Kustrin for providing the best office in the department for two years, and for the numerous meals and drinks that we have shared (especially those "last order" outings whilst trying to write up!). I must also thank Graham Finlayson for always been there for a drink - and lively discussion!

Then there is Nick Carriero and David Gelernter for allowing me to spend two brilliant months at Yale University, USA.

I would like to thank the other members of the Linda team at York University, particularly Ronaldo Menezes.

There are the other people in the department who have encouraged, supported and socialised with me; Andrew Finch, John Kennedy, Richard Filer, Paul Zanelli, Simon Moss and Aaron Turner (and the other members of ACAG) and Andrew Hague.

Other people who I wish to thank for generally making my stay at York fun are David Hewitt, John 'Elvis' Hattersly, Zoey Brewer, 'Old Man' Pete Tickle, and numerous others.

I would also like to thank Mike Everret, Colin Neunan and David Parker of British Aerospace Military Aircraft, for providing a CASE grant and allowing me a "free hand" over the work.

Finally, but certainly not least, I would like thank my parents for patiently reading and re-reading this dissertation so, at least, others can have a chance of understanding it!

Declaration

This dissertation is the result of my own work. It is not substantially the same as any dissertation I have submitted for any other qualification at any other university. Some of the contents of this dissertation have been published.

- Chapter 3 has been in part published at the 1st International Conference on Coordination Languages, Italy, 1996[RW96c].
- Chapters 3 and 4 have been in part submitted as an invited paper to a special edition of the Journal of the Science of Computer Programming.
- Chapters 5 and 6 have been in part published at Euro-Par'96[RW96a]. Earlier ground work on the implementations has been published at EuroPVM'96[RDW95].

A number of papers have also been published although not directly associated with the work in this dissertation. A paper on the implementation of mathematical morphology using ISETL-Linda[RW95] and a paper on a set of primitives[RW97] which provide asynchronous tuple space access for tuple spaces being used in distributed environments have been published.

A number of technical reports have also been published based on the work in this thesis[RW96b, RDW96].

Chapter 1

Introduction

Linda was developed during the mid eighties by David Gelernter[Gel85]. The underlying philosophy is that a complete programming model for parallel programs is created by two separate parts, a computation model and a coordination model[GC92]. The computation model is provided by a sequential programming language. The coordination model is needed to allow the sequential computation sections to coordinate and communicate. Linda is a coordination language which supplies a number of operations or primitives that provide the ability for sequential code segments (processes) written in a sequential programming language to communicate. The Linda primitives are normally embedded within the language to be used (the *host* language).

Linda uses a shared structure, called a *tuple space*[1] which is an unordered collection of tuples. A tuple is an ordered collection of elements, with each element having a value and an associated type. Processes insert tuples into the tuple spaces, and other processes can then retrieve those tuples using an associative matching process. The only way that two processes can communicate is via a tuple space, and Linda provides asynchronous communication between processes. A fuller description of Linda is given in Chapter 2. Linda provides a *shared associative memory*, but Gelernter[Gel85] states that:

> *Our new technique* [Linda] *is closest to message passing, but the difference between the two are as significant as the similarities.*

Gelernter argues that Linda and the use of tuple spaces leads to a new paradigm for process communication and coordination called *generative communication*. Generative communication has two characteristics: *communication orthogonality* and *free naming*. Communication orthogonality means that the receiver of a message does not know which process created the message, and the sender of a message does not know which process will receive the message. Free naming refers to the concept that any field within a message (tuple) can be used to retrieve a message[2].

[1]Also referred to as a *bag* or *multi-set*.

[2]This is discussed in Chapter 2 in greater depth.

Linda has a number of properties which make it attractive as a coordination language. The property of communication orthogonality means that it supports processes that are spatially separated and temporally separated[3]. This means that two processes can communicate even if their existence does not overlap (temporally separated), and two processes can communicate without knowing to which process they are communicating, or anything about the address space of the process with which they are communicating (spatially separated). Linda is also a very simple coordination language, requiring only a few primitives to be added to the host language and it is conceptually easy for people to use. These properties have led to the use of Linda in many environments, for many different and varied tasks. The success of Linda has led to the development of at least one commercial version, which in turn has facilitated the use of Linda by a diverse group of people from academics to large Wall Street banks. The Linda primitives have been embedded into many different languages, including C[Car87, Nar89, Lei89], ISETL[DRW95, Has94], Gofer[DRRW96], and Fortran[YFY96]. There are many parallel applications[CG93] that have used Linda, including parallel ray tracing[MM91, BKS91], financial modeling[NB92, CCZ93, Cag93], real-time data fusion[FGK^{+}91], seismic applications[BS92], probabilistic fatigue analysis[SLSC92], and many others. More recently, as open implementations of Linda have been produced, Linda has been used in a number of applications in the domain of Computer Supported Cooperative Work (CSCW), including using Linda to create shared distributed virtual environments for virtual reality systems[Ams95].

All implementations of systems that use the Linda primitives, or more generally shared tuple spaces, fall into one of two categories:

Closed implementations are ones which require information about all the processes which wish to communicate via tuple spaces to be available when the system starts; and

Open implementations are ones which allow all processes to join and leave the system at will, and do not require information about all processes which wish to communicate to be available when the system starts.

All current implementations that are considered closed implementations have a further requirement, that all the source code must be available at compile time. This is because compile time analysis is used to control many aspects of the run-time system, including where processes are placed and how tuples are distributed at run-time. Open implementations allow processes to join and leave freely without requiring them all to be present at compile time. Therefore, the run-time system has to manage the placement of tuples and processes without the aid of information that can be calculated at compile time. In open implementations the communicating processes need not even be written in the same language, and interpreted languages can be used as well as compiled languages.

[3]Also referred to as space uncoupling and time uncoupling respectively.

Closed implementations exist for many different platforms, ranging from high performance parallel computers (such as the Cray T3D) to networks of heterogenous workstations. Closed implementations which use compile-time analysis should provide better performance than open implementations because of their ability to optimise the flow of tuples at run-time using information about which processes can consume and generate tuples. The fundamental implementational approach adopted for both the closed and open implementations has altered little over the years despite the addition of new features to Linda.

Since its original inception Linda has evolved to include many extensions. One of the main extensions has been the addition of multiple tuple spaces[Gel89]. The original model contained a single tuple space, but most modern implementations support the use of multiple tuple spaces. Multiple tuple spaces are important because they allow processes and groups of processes to hide information from other processes. When all processes share a single tuple space the process producing the tuples has no control over which processes can use the tuples (coordination orthogonality). This is not a problem in dedicated programs (typically produced using efficient closed implementations). However, as the use of Linda is considered for more general parallel computing, distributed programming and Internet computing, the ability to hide tuples from other processes becomes necessary. The introduction of multiple tuple spaces has led to the question of whether new primitives need to be added to the model.

In this dissertation the following two issues are addressed:

- The sufficiency or otherwise of the original set of Linda primitives (given multiple tuple spaces), and
- how can the bulk primitives of `collect` and `copy-collect` be implemented efficiently within an *open* Linda implementation?

When multiple tuple spaces were first proposed, only a tuple space creation primitive was proposed[Gel89]. However, the addition of multiple tuple spaces to Linda has inevitably led to proposals of new primitives which rely on multiple tuple spaces. Most appear to have been proposed because they are either easy to implement or “appear obvious”.

The addition of new primitives to the Linda model needs careful consideration. There should be a strong justification for the addition of any new primitives. A number of criteria should be satisfied; one is to demonstrate that the primitive is required because the current primitives are unable to perform an operation satisfactorily; and the second is to demonstrate that the proposed primitive does not simply move the underlying reasons why the operation cannot be satisfactorily performed into the implementation.

Other work at the University of York has shown the need for a new primitive called `collect`[BWA94] which uses multiple tuple spaces. In this dissertation it is proposed that another primitive, `copy-collect`, be added to Linda. Rather than just assume the need for such a primitive, an identifiable operation that is difficult to perform using the current Linda primitives is described (the multiple `rd` problem). The proposed primitive is then shown to solve the problem. Both the `collect` primitive and the proposed `copy-collect` primitive are called *bulk primitives* because they can manipulate more than one tuple in a single operation.

Having shown that the proposed primitive is required the obvious question is how is the proposed primitive implemented? Due to the relationship between `collect` and the proposed primitive, the techniques outlined for the implementation of the proposed primitive also work for the `collect` primitive. Therefore, instead of asking, how is the proposed primitive implemented, the more appropriate question is, how are both these bulk primitives implemented efficiently?

The implementation techniques used in Linda systems have changed little since the first implementations despite the fact that new primitives (with or without justification) and multiple tuple spaces have been added. Thus, when considering how the bulk primitives are implemented, a new approach to tuple space implementation is proposed, which uses *implicit* information provided by the bulk primitives, and by the general use of tuple spaces within a Linda program to create an efficient open implementation supporting the bulk primitives. An implementation for a network of workstations using the implementation strategy outlined is presented with performance figures to support the claim that the implementation strategy is better than traditional approaches.

1.1 Thesis Overview

Chapter 2 describes in detail the Linda model and its attributes. It describes some of the important extensions to Linda, including the addition of multiple tuple spaces and proposed primitives. A review of related and auxiliary work is also presented.

Chapter 3 investigates the *multiple* `rd` *problem*. A multiple `rd` is an operation that Linda is unable to express acceptably. A small example program is used to show the multiple `rd` problem and show how current implementation strategies using the standard Linda primitives to overcome it are unacceptable.

Chapter 4 describes a new primitive for Linda called `copy-collect`. The purpose of the primitive is to overcome the multiple `rd` problem. The example used in Chapter 3 is again used to show how the new primitive solves the problem.

Chapter 5 investigates how a Linda run-time system can be produced which takes full advantage of the implicit information that the bulk primitives and multiple tuple spaces provide. A naive approach to implementing the bulk primitives based on a simple extension of the traditional implementation techniques is first discussed. A novel approach is then presented which uses *implicit* information to move tuples around the system in advance of their actual use by a user process. An actual implementation for a network of workstations is considered.

Chapter 6 the performance of the network implementation is shown using a number of simple examples and an image processing case study. The performance is shown to be better than other implementations including one that uses compile time analysis (a closed implementation).

Chapter 7 is a discussion of how different programming styles can effect the performance of the new approach, and how the generalisation of the techniques described in Chapter 5 can be used to overcome these different programming styles and provide a more general run-time system. A detailed proposal for a more general run-time system is able to cope with more workstations which could be geographically separated (for WAN computing) is presented.

Chapter 8 presents a number of conclusions about the research described in this dissertation. A number of future research questions which have arisen from the work described in this thesis are also presented.

1.2 Contributions

The following contributions have been made in this dissertation:

- experimental study of the limitations of Linda to perform a multiple `rd` operation;
- the addition of a new primitive to Linda , with informal semantics (which can also be applied to the `collect` primitive) to overcome the limitation; and
- a novel run-time system for Linda, providing a number of innovative features:
  - a scheme to manage and perform the dynamic movement of tuples and tuple spaces which is achieved by using *implicit* information provided from Linda programs to achieve better performance; and
  - a detailed description of the structure of a generalised hierarchical kernel, which is scalable beyond the bounds of a local area network of workstations, which utilises dynamic movement of tuples and tuple spaces.

Chapter 2

Related and background work

2.1 Introduction

This chapter presents a detailed review of Linda, followed by a description of some of the proposed (and adopted) extensions to Linda, including the addition of multiple tuple spaces.

A number of important characteristics of Linda are then considered in detail, including the role of the `inp` and `rdp` primitives, `out` ordering, the role of the `eval` primitive, and non-determinism. A detailed description of Linda implementations is given in Chapter 5. An overview of the properties of the implementations used in this dissertation is given in Appendix A.

2.2 Linda

Linda[Gel85] is a process coordination language[GC92] which is based on the idea of *generative communication*. Linda as described here is based on Linda 2[CG89b] with multiple tuple spaces added. The original Linda proposal[Gel85] (Linda 1) was slightly different, and is described in Section 2.2.3.

The fundamental objects of all the versions of Linda are tuples, templates and tuple spaces:

Tuple A tuple is an ordered collection of fields. Each field has a type and a value associated with it. A field with both a value and a type is known as an actual. The same field can be replicated many times within a tuple. The tuple:

$$\langle 10_{integer}, \text{“Hello World”}_{string}, 10_{integer}, 1.0_{float}\rangle$$

is a tuple containing four fields with the type of the field shown as a subscript of the value. The types of the fields are normally restricted by the language into which Linda is embedded. Tuples are placed into tuple spaces and are removed from tuple spaces using an associative matching process.

Template A template is similar to a tuple except the fields do not need to have values associated with them, but all fields must have a type. A field that has only a type and no value is known as a formal, and a template is a tuple which can have formals.

The templates:

$$\langle|10_{integer}, \text{“Hello World”}_{string}, 10_{integer}, 1.0_{float}|\rangle$$

and

$$\langle|10_{integer}, \square_{string}, \square_{integer}, 1.0_{float}|\rangle$$

will match the tuple $\langle 10_{integer}, \text{“Hello World”}_{string}, 10_{integer}, 1.0_{float}\rangle$. In this dissertation the symbol $\square$ in a template is used to indicate that the field is a formal, so it has no value. A template is sometimes referred to as an *anti-tuple*[Car87].

Tuple space A tuple space is a *logical shared associative memory* that is used to store tuples. A tuple space implements a *bag* or *multi-set*, and the same tuple may be present more than once and there is no ordering of the tuples in a tuple space. Originally, Linda 2 had only a single tuple space known as the *global tuple space* (GTS), however in the proposal for Linda 3[Gel89] multiple tuple spaces were introduced. Multiple tuple spaces are now widely adopted, although in many different forms (see Section 2.3.1).

2.2.1 The Linda primitives

Gelernter[GC92] states that any parallel program can be divided into two sections: communication and computation. The communication section is provided by a coordination language, such as Linda and the computation section is provided by a *host* programming language into which the Linda primitives are embedded. There have been many different languages from several programming paradigms which have been used as host languages, including:

- C[Car87, Nar89, Lei89],
- C++,[CCH91],
- Pascal[YFY96],
- Fortran[YFY96],
- Lisp[YFY96],
- Prolog[Cia91, BW91],
- Eiffel[Jel90],

- Scheme[Jag91],
- ISETL[DRW95, Has94], and
- Gofer[DRRW96].

This is not an exhaustive list, and there are many other embeddings using both the languages listed and languages which are not listed.

Processes are written using the host language. Different host languages manage processes in different ways. In some implementations processes are mapped onto functions and in others they are mapped onto separate executable files. Regardless of how a process is created within the host language the different processes can only communicate with each other through tuple spaces using tuples and the Linda primitives. The basic Linda primitives (which are embedded into the host language) are:

out(*ts, tuple*) The `out` primitive places a tuple (*tuple*) into the specified tuple space (*ts*).

in(*ts, template*) The `in` primitive retrieves some tuple from the tuple space that matches the template. If there is no tuple that matches the template then the primitive blocks until a tuple that does match the template is inserted into the tuple space. The matched tuple is removed from the tuple space and returned to the user process.

rd(*ts, template*) The `rd` primitive is similar to the `in` primitive, except the matched tuple which is returned to the user process is *not* removed from the tuple space. As with the `in` primitive when there is no matching tuple available the `rd` primitive blocks until a matching tuple is inserted into the tuple space.

eval(*ts, active tuple*) The *eval* primitive is included within Linda as a means of spawning processes. The `eval` primitive creates a special tuple called an active tuple, which is a tuple which contains one or more fields containing functions that require evaluation in order to provide a value. The functions are evaluated concurrently with the process which performed the `eval` primitive. When a field's function has been evaluated the result is inserted into the active tuple. When all the fields in the active tuple that need evaluating have been evaluated the tuple becomes a passive tuple (like any tuple inserted using the `out` primitive), which can be accessed like any other tuple (see Section 5.3.2).

It should be noted that any primitive which blocks will cause the process which performs the primitive to block.

There are two more primitives which have been proposed by Carriero[Car87]. These are the `inp` and `rdp` primitives, which are non-blocking versions of the `in` and `rd` primitives respectively. If a tuple is not available then instead of blocking, the primitive returns a value to indicate this.

These primitives were not in the original Linda proposal[Gel85], and are often not supported for a variety of reasons (see Section 2.4).

There are currently no widely accepted formal semantics of the Linda primitives, although a number of proposals have been produced[But90, Jen93, CJY95, COW96, BJ96]. Most of the proposals deal with providing a formal semantics for a particular part of Linda. However, with Linda constantly evolving the production of widely accepted semantics is difficult.

2.2.2 Tuple matching

Templates are matched to tuples using an associative match. A tuple $\mathcal{T}$ and a template $\mathcal{M}$ match when:

$$\left(\sharp(\mathcal{T}) = \sharp(\mathcal{M})\right) \bigwedge \left(\forall j \in \{1..\sharp(\mathcal{T})\} : \left(t_j(\mathcal{M}) = t_j(\mathcal{T})\right) \wedge \left((v_j(\mathcal{M}) = v_j(\mathcal{T})) \vee (v_j(\mathcal{M}) = \Box)\right)\right)$$

is true, where $t_j(\mathcal{I})$ is a function which returns the type of field number j of tuple or template $\mathcal{I}$, $v_j(\mathcal{I})$ is a function which returns the value of field number j of tuple or template $\mathcal{I}$, and $\sharp(\mathcal{I})$ is the cardinality of tuple or template $\mathcal{I}$. So, for a match to occur the number of fields in the template must be equal to the number of fields in the tuple, for every field in the tuple the type of the field must match the corresponding field type in the template, and either the value of the fields must be the same, or the template field must be a formal (have no value).

2.2.3 Linda 1

When Gelernter[Gel85] first described Linda, it was in a slightly different form from the Linda that has been described so far in this chapter. The primitives that were proposed in Linda 1 were the `in`, `rd` and `out` primitives. There were no `eval`, `inp` or `rdp` primitives. The other difference was that tuples and templates used the concept of an identifier, which was a field attached to the front of the tuple with a special type and was used as an identification tag. This was necessary because a *tuple* was allowed to contain formals. Matching on formals in tuples was not allowed, so it was potentially possible to insert tuples that could not be matched, unless at least one field was guaranteed to be an actual within every tuple, hence the addition of a identification tag which was always an actual.

The ability to use all actuals present, rather than just the identification tag, within a template in the matching process was called *structured naming*. The properties that *structured naming* provides are the basis of the *free naming* property required for *generative communication*, which was introduced in Chapter 1.

By the time the first implementations were produced by Carriero[Car87] Linda had become Linda 2. The concept of being able to place tuples with formals into the tuple space had disappeared, which meant there was no need for identifier tags, and subsequently these too were removed.

2.3 Linda extensions

Due to the nature of Linda there have been a number of proposals for alterations and additions to it. This has involved the creation of many new coordination languages based on the concepts of Linda, including Bauhaus Linda[CGZ95], Piranha[GJK93], MTS-Linda[Jen93], Laura[Tol95a] and Melinda[Hup90]. There have also been numerous implementations of Linda which provide extra primitives or features, including a special purpose Linda machine[KACG87, ACGK88]. The major suggestions of relevance and interest are now reviewed.

2.3.1 Multiple tuple spaces

The concept of multiple tuple spaces was introduced by Gelernter as part of Linda 3[Gel89]. Linda 3 added a type `ts` and a new primitive `tsc` to Linda 2 to allow for multiple tuple spaces. The idea of adding multiple tuple spaces has led to many different proposals of how multiple tuple spaces could be incorporated within Linda[Hup90, Jen93], and many implementations include multiple tuple spaces in one form or another[DRW95, RDW95, Has94, NS93, Jeo96, Kie96].

Multiple tuple spaces were introduced as an effective way of hiding information. Information within a tuple space can *only* be accessed by those processes that know about the tuple space. As the use of Linda has changed to incorporate different styles of distributed computing the need to hide tuples has become increasingly important to ensure that other processes do not either maliciously or accidently tamper with the tuples that other processes are using.

When multiple tuple spaces are added to Linda there are two important questions: are the tuple spaces first class objects, and what is the relationship between the tuple spaces?

Tuple spaces as first class objects

Making tuple spaces first class objects has been proposed by a number of researchers, including Gelernter[Gel89], Hupfer[Hup90] and Jensen[Jen93]. However, few implementations support tuple spaces as first class objects, although the MTS-Linda[NS94] implementation does and is based on the work of Jensen[Jen93].

In general tuple spaces have not been widely adopted as first class objects[Ass96, DRW95, RDW95, Has94, Jeo96, Kie96]. This is because the ability to manipulate entire tuple spaces as first class objects raises many awkward questions, which have yet to be answered satisfactorily. For example, what happens if a tuple space is removed by one process, whilst another process is blocked on an `in` primitive waiting for a tuple to appear in that tuple space? What happens if a process wishes to perform a tuple operation on a removed tuple space? Can the removed tuple space be manipulated within the user process, and if so how? Gelernter[Gel89] introduces the idea of freezing tuple spaces, and then converting them to other data structures within the user process, which would imply new primitives would be added to enable the conversion to take place.

The relationship between the tuple spaces

In some proposals tuple spaces are hierarchical[Hup90, Gel89], in others they are flat structures and in others a hybrid approach is used[NS94].

If the tuple spaces are first class, then this implies that there will probably be some relationship between tuple spaces, because tuple spaces will subsequently be placeable inside tuples, and will be inserted into other tuple spaces.

Melinda[Hup90] supports only a hierarchical system, where tuple spaces are created within other tuple spaces. In Melinda a tuple space is named by the user, creating the possibility that many tuple spaces can be called the same thing. If this occurs, and an `out` primitive is performed where more than one tuple space could be the destination tuple space, then one is chosen non-deterministically. This has the disadvantage that a process can no longer ensure that tuples it is producing are being received by the intended process, as there is the potential that two unrelated processes both create a tuple space with the same name, and each process has no control over which of the tuple spaces the tuples are being placed in. This can also lead to unintentional deadlocks. For example, consider a process that creates a tuple space, places a tuple into the tuple space, and then retrieves that tuple. If the tuple space it creates is not unique then the process can not guarantee retrieving the tuple it inserted, and could therefore deadlock.

MTS-Linda[NS94] supports hierarchical tuple spaces and flat tuple spaces. The hierarchical tuple spaces are created by allowing processes to create tuple spaces which are considered as belonging to a process. The process can spawn a process within these tuple spaces, and the spawned process sees the tuple space as its parent tuple space. A tuple space which is local to a process can be *duplicated* in the parent tuple space, by placing a tuple within the parent *containing* the tuple space. The flat tuple spaces that MTS-Linda supports are similar to the flat tuple spaces used by Douglas et al.[DRW95, RDW95], SCA Paradise[Ass96], ProSet-Linda[Has94], and PLinda[Jeo96]. Within all these approaches a tuple space is created. A tuple space handle is returned to the process which created the tuple space. The tuple space *handle* can then be passed to other processes in tuples, and if a process has a tuple space handle it is able to access the tuple space. The tuple spaces are not first class objects, just the tuple space handles. Therefore, if two tuple space handles are checked for equality, this checks that they refer to the same tuple space, but does not compare the *contents* of the tuple spaces to which the handles refer. Some of the implementations support the concept of a parent tuple space for a process. This can be seen as the tuple space into which the active tuple which contains the function (process) is placed. If a flat structure of tuple spaces is being used it is common to include one (or more) global tuple spaces.

Tuple spaces used within this dissertation

The work presented within this dissertation requires multiple tuple spaces, but is generally independent of the way in which multiple tuple spaces are related. A flat tuple space structure is assumed, where a tuple space is unrelated to any other tuple space. If two processes are to share a tuple space, then the tuple space handle is passed between the processes through another shared tuple space, or by one process passing the tuple space handle as an argument to the other process when it is spawned. Tuple spaces are assumed not to be first class objects and tuple spaces *cannot* be copied into local data structures within a user processes, the tuple space handles can be seen as pointers to a tuple space. Processes do not have a parent tuple space, but there is a single tuple space which all processes *always* have access to, called the *Universal Tuple Space* (`UTS`). It is assumed that tuple spaces are unique, and therefore, any tuple space handle can only refer to a single tuple space at any one time.

A flat tuple space structure is used because of the issues that were raised in the last two sections, concerning tuple spaces as first class objects and hierarchical tuple spaces.

In order to allow the creation of tuple spaces a primitive is added to the basic Linda primitives (as presented in Section 2.2.1), and a type for tuple space handles is also added.

2.3.2 New primitives

New primitives are normally proposed either to provide better performance or extend the functionality of Linda. An overview is now presented of some of the primitives which have been proposed. They are divided into either primitives which provide more functionality or primitives for performance.

Primitives to provide more functionality

Primitives are often added to Linda when multiple tuple spaces have been incorporated, because the addition of multiple tuple spaces introduces the possibility of new coordination constructs. All the primitives described in this section could be classified as bulk primitives which manipulate more than one tuple at a time.

The first two primitives were added to the standard Linda primitives in MTS-Linda[Jen93, NS93, NS94]. The original description of MTS-Linda[Jen93] does not include either of these primitives, but the implementation[NS93, NS94] of MTS-Linda includes both of them.

- **copy_contents(*ts1, ts2*)** This primitive *copies all* the tuples present in tuple space `ts1` to tuple space `ts2`.
- **move_contents(*ts1, ts2*)** This primitive *moves all* the tuples present in tuple space `ts1` to tuple space `ts2`.

The next primitive was proposed at the University of York by Butcher et al.[BWA94] and was first implemented in the York Kernel I by Douglas et al.[DRW95]. The primitive is seen by the authors as a replacement for and generalisation of, both the `inp` and `rdp` primitives.

- **n = collect(*ts1, ts2, template*)** This primitive moves *all* the tuples that match the template from tuple space `ts1` to tuple space `ts2`, and a count of the number of tuples moved is returned. This primitive is similar to the `move_contents` primitive except it uses a template to match the tuples and returns a count of the number of tuples moved. A more detailed discussion of the `collect` primitive is presented in Chapter 4.

The next primitives are all similar in function, and were proposed by Anderson et al.[AS91]. However, they have never been described as implemented in the published literature. Similar primitives have been considered by researchers at Yale University, which they refer to as `rd*`/`in*`[Car95] and `in`-loops/`rd`-loops[Lei89].

- **rd(*template*)all(*function*)**
- **in(*template*)all(*function*)**
- **rdp(*template*)all(*function*)**
- **inp(*template*)all(*function*)**

  These primitives iterate through all the tuples that match the template, and apply the function to each matched tuple. Whether the tuples are removed or not depends on whether the `in()all` or `rd()all` primitive is being used. The example given in Anderson et al.[AS91] is:

```
/* sum the count field of all tuples matching
the tuple pattern */

int i, sum =0;
..
inp("example 8", "count", ?i)all {sum += i;};
```

  which iterates through all the tuples which match the template $\langle|$“example 8”$_{string}$, “count”$_{string}$, $\square_{integer}|\rangle$, summing the third field. When the primitive terminates the variable `sum` will contain the result.

The next two primitives are suggested as part of Objective Linda[Kie96]. Although not directly using multiple tuple spaces, the primitives can return more than one tuple in a single operation, hence making it a bulk primitive.

- **rd(*min, max, template, timeout*)**
- **in(*min, max, template, timeout*)**

  These primitives are extensions of the traditional Linda `in` and `rd` primitives. The idea is to extend the primitives to deal with multiple tuples, so instead of returning a single tuple a multi-set (tuple space) of tuples is returned. The fields `min` and `max` refer to the minimum and maximum number of tuples the primitive is to return, therefore effectively providing a way to bound the number of tuples matched. A traditional `in` primitive would be emulated by setting both the fields to one. The `timeout` field allows the time that the primitive can block to be controlled. If this field is zero, and the `min` and `max` fields are set to one a traditional `inp` primitive is emulated. The addition of timeouts to a primitive is not necessarily desirable as it is unclear as to exactly *what* is being timed[RW97].

The primitives described in this section are of particular interest because of their potential for overcoming the multiple `rd` problem introduced in Chapter 3. A description of the differences between these primitives and the proposed primitive in Chapter 4 is given in Section 4.6.

Primitives for performance

These primitives have been proposed to increase the performance of Linda systems. In general, it is possible to emulate them using the current Linda primitives. The motivation behind most of these primitives is to allow implementations that do not use compile-time analysis to use some of the optimisation techniques achievable when compile-time analysis is used.

- **wr**

  The motivation for this primitive comes from the use of tuple replication in run-time systems to achieve faster tuple access times, and is proposed by Wells et al.[WC95]. A Linda run-time system must ensure that if a single tuple is placed in a tuple space, that tuple can only be *destructively removed* once. In implementations using compile-time analysis it is possible to check if a tuple is only non-destructively read (in other words only the `rd` primitive is used to access the tuple). If it is known that the tuple is only non-destructively read then the tuple can be replicated as many times as the implementation wants, because the control of replicated tuples is simple. This optimisation is only normally used in closed implementations as all the processes which communicate need to be present at compile time to determine which processes can access the tuple and how they access it. In open implementations the costs of managing replication of tuples usually outweighs the advantages, because of the arbitration needed to ensure only one process can destructively remove a tuple[Faa91].

  Wells et al.[WC95] suggest the addition of the `wr` primitive (or write primitive) which indicates that the tuple will be *mainly* non-destructively read. Semantically, an `out` primitive and a `wr` primitive are the same, both insert a tuple into a tuple space. The idea is that an `out`/`in` pair is cheap and a `wr`/`rd` pair is cheap. However, an `out`/`rd` is more expensive than a `wr`/`rd` and a `wr`/`in` is more expensive than an `out`/`in`. The `wr` primitive provides a hint to the run-time system that the replication of a tuple is acceptable. If a tuple can be replicated then possibly every store of tuples can contain the tuple. Hence, the finding of the tuple within a run-time system is cheap *provided* the tuple does not need to be removed. If the tuple has to be removed then there is significant control required, to ensure that a replicated tuple is not destructively removed more than once. Hence a `wr`/`rd` pair is cheap, but a `wr`/`in` pair is expensive. The onus is then placed upon the programmer to decide which pairings to use to obtain the best performance. It should be noted that if the run-time system decides not replicate the tuple (and treat the operation as an `out` primitive) the semantics of the program will not be altered. Also, whenever an `out` primitive is used a `wr` primitive could be used and the semantics of the program will not be altered.

- **add**

  This primitive was proposed by Carreira et al.[CSS94], and it enables a field within a tuple to be updated in a single action, without the need to remove a tuple, update it and return it to the tuple space. The motivation for this primitive is the observation that a tuple is often used as a shared global counter. In order for a process to increment (or decrement) the global counter, it has to destructively remove the tuple using an `in` primitive and then replace the tuple with the updated counter value using an `out` primitive. The `add` primitive removes the need for the tuple to be returned to the process by allowing a value to be specified which is added to the appropriate field in the tuple. However, Carriero et al.[CG90b] states:

  > *Optimising idioms.*
  >
  > *Tuple space operations are often used in standard patterns which the pre-compiler can detect and the partial-evaluator support with optimised code. One important pattern is the following:*
  >
  > *in(fields);*
  >
  > *out(f(fields));*
  >
  > *That is: remove a tuple, change some of its fields and then re-insert it. A simple case is the atomic update of a counter:*
  >
  > ```
  > in(CounterName, ?value);
  > out(CouterName, ++value);
  > ```

  The premiss is that open implementations cannot perform compile-time analysis. However, the compile-time analysis required to detect this does *not* require all user processes to be present at compile time, or to be present when the run-time system starts executing. This means that the optimisation using compile-time analysis could be incorporated within open implementations effectively. The need to add explicit information to Linda programs by Linda programmers should be avoided unless absolutely necessary.

- **update**

This primitive was also proposed by Carreira et al.[CSS94] and is similar to the `add` primitive. The principle of this primitive is similar to the `add` primitive except a new tuple to replace the old one is specified rather than an increment for a particular field.

The semantics of the `update` primitive are unclear, but as with the `add` primitive the same sort of compile-time analysis should make the optimisations that are suggested by Carriero et al.[CG90b] applicable to this primitive.

In Chapter 5 a deeper discussion about the addition of explicit information to Linda programs is presented.

Miscellaneous primitives

There are some proposals for primitives that are not really classifiable as either performance primitives or primitives which provide extra functionality.

- **cancel** This primitive was proposed by Banville[Ban96], and enables blocked `in` and `rd` primitives to be "unblocked" by another process. The perceived need that justifies the addition of this primitive is for a process blocked on an `in` or `rd` primitive to be able to be "unblocked" by another process. Therefore, if a process performs an `in` primitive using the template $\langle|$"work"$_{string}, \square_{integer}|\rangle$ another process can perform a `cancel` primitive using the template $\langle|$"work"$_{string}, \square_{integer}|\rangle$ and the process blocked will become unblocked. The `in` primitive returns a value to indicate that a `cancel` primitive caused the `in` primitive to become unblocked, not a matching tuple.

  The need for such a primitive appears unclear. Most Linda programmers would use an `out` primitive to place a tuple in the tuple space which matches the template. This is the basis of "poison pill" programming style that is in common use in Linda programming (see Section 4.7). The process that reads the tuple checks to see if the tuple contains a "poison pill" and if so acts accordingly. The `cancel` primitive seems to add nothing, except to be a higher level construct compared to the "poison pill". The process performing the `cancel` primitive has to be aware of the template that the blocked primitive is using, and therefore should be just as able to generate a tuple to match it as a template. After the use of the `cancel` primitive the process which becomes unblocked has to check to see if a valid tuple was found or if a `cancel` primitive unblocked it. It could just as well check to see if the "poison pill" was present within the tuple. The only argument for using the `cancel` primitive is that the process which consumes the tuple containing the "poison pill" may have to replace it in the tuple space, in case other processes are also blocked waiting for a similar tuple, and with a `cancel` primitive this does not have to be done.

The last three primitives (`add`, `update` and `cancel`) are examples of primitives that have been proposed which are potentially unnecessary. It is important that the addition of new primitives to Linda are added because there is a solid and sound justification for their addition.

Within this dissertation the `collect` primitive is used as one of the primitives in conjunction with the primitives described in Section 2.2.1. In general the `inp` and `rdp` primitives are not used, because the `collect` primitive and the primitive proposed in Chapter 4 replace (and generalise) the `inp` and `rdp` primitives.

2.3.3 Piranha and Bauhaus Linda

Two variants of Linda are of particular interest, Piranha and Bauhaus Linda, because they provide more than simply extra primitives and they support new styles of programming based on the concept of shared tuple spaces. Both were developed at Yale University by Gelernter and Carriero.

Piranha

Piranha[GJK93, CGKW93, GK92, CFGK94, CFG93] is a variation of Linda, which uses the same basic primitives as Linda, except the `eval` primitive has been removed. The concept behind Piranha is that during any period the processors (or workstations on a LAN) which are not being used change dynamically. When a user starts using a node he does not wish his node to be slowed by other people's computationally intensive tasks. Therefore, processes are made to migrate from one node to another.

Piranha is designed to support this type of computing, and supports only a master-worker style of parallelism. The user specifies a function that is the worker, and a function that is the master. The user has no control over how many worker processes are executing as the run-time system decides this based on the resources (nodes) available.

When a node becomes busy the worker is terminated. This involves executing a "retreat" function within the worker process and then killing the process. By using the retreat function within a worker process the user is able to ensure that when the worker process terminates nothing is lost, which is normally achieved by placing the current tuple back into the tuple space. It is possible that all the worker processes can be terminated if all the nodes are busy, and then restarted as nodes become available. Piranha does not actually migrate worker processes but rather kills them and then restarts them from the start of the worker function on another node.

Piranha has been successfully used, and is now a commercial product from SCA, who also produce the commercial SCA C-Linda compiler.

Bauhaus Linda

Bauhaus Linda[CGZ95] is in an attempt to address the needs of collaborative working. It is specifically designed for open computing, where agents (processes) will join and leave at will, leaving information within a system to be retrieved later. Bauhaus Linda removes the distinction between tuples and tuple spaces, by introducing the concept of nested multi-sets which are first class objects. It also removes the distinction between tuples and templates, using actuals given within a multi-set as the matching criteria rather than a mixture of actuals and formals as in Linda. As Bauhaus Linda uses only multi-sets the ordering of elements within a multi-set is arbitrary. It also removes the distinctions between passive tuples and active tuples, by making processes first class, and allowing them to move within the multi-set structure.

The initial work appears interesting, and represents the most radical move away from Linda that still uses the same basic access primitives and concepts (shared tuples). However, there are a number of issues that need considering. For example, they currently propose the addition to the host language of operations to manipulate multi-sets which have been removed from the shared multi-set structure. This creates a distinction in the access and use of multi-sets stored locally and those which are shared, and also makes the use of Bauhaus Linda more complex. It also appears simple for processes to remove another process by matching parts of the multi-set in which the process resides. This may not be desirable in real life systems and access controls on nested multi-sets may be required.

2.3.4 The Linda machine

The Linda machine[KACG87, ACGK88] is a parallel computer that was designed to support Linda using specialist dedicated hardware. The machine was made up of a set of *Linda nodes*, with each node containing a general purpose processor (Motorola 68020 processor), general purpose memory, tuple storage memory and a Linda co-processor[KACG88]. The tuples are distributed across the Linda nodes, using an intermediate uniform distribution (see Chapter 5, and Figure 5.1). The Linda nodes are arranged as a two-dimensional mesh. Bus contention is used to control replication of tuples within the Linda nodes, providing a neat solution to managing replicated tuples when using an intermediate uniform distribution.

Predicted results indicated that the performance of the machine would be good, and it was scalable to at least 1024 nodes. Unfortunately, the Linda machine was never completed.

2.3.5 Closing comments

There are many other proposals which have been made but are not related to the work described in this dissertation. These include the proposals for the addition of timeouts on the primitives[Ban96, Kie96], and the extension of the template tuple matching process[Ban96, AS91].

2.4 The `inp` and `rdp` primitives: the need for `out` ordering

As mentioned in the Section 2.2.1, the two primitives `inp` and `rdp` were proposed by Carriero[Car87] and incorporated into the first implementations of Linda. However, the `inp` and `rdp` primitives are not widely supported in other Linda implementations. There are a number of perceived “semantic problems” associated with these primitives which are used as the primary reason for their removal[Lei89], replacement[BWA94] or, when implemented, behaviour which can potentially lead to unintentional program behaviour[Ass95]. The problems associated with these primitives can be considered as *global synchronisation* and *out ordering*. Both these mean that the information the primitives return need not necessarily be useful.

2.4.1 Global synchronisation

The *global synchronisation* problem is that in order to determine the result of a `rdp` primitive, a global synchronisation is required[Lei89, BWA94], which is expensive, and should be avoided if possible. Leichter [Lei89] states that:

> *The non-blocking operations* [inp and rdp primitives], *on the other hand, refer to the complete spatial extent of tuple space:* `inp` *must return true if there is a matching tuple anywhere in the tuple space, and it must return false if there is no matching tuple. These are statements about a slice taken across all tuple space at a given time. To state them at all requires a notion of simultaneous action across tuple space; but given an ability to specify a moment in time, the correctness of* `inp` *returning false can be falsified by the observation of a matching tuple at that moment, anywhere in tuple space.*

The statement that: *the correctness of* `inp` *returning false can be falsified by the observation of a matching tuple at that moment, anywhere in tuple space*; is important. If a user process cannot know that there is a tuple that could match the template used for the `inp` primitive, then the `inp` primitive result is correct. The crux of the problem is, if a tuple space is distributed (over a number of processors), not all sections of the tuple space can be searched *at the same time* unless there is an expensive operation to lock the entire tuple space, perform the search, and then unlock the tuple space, which requires global synchronisation. Why should all sections of the tuple space be searched at the same time? Because one section of the tuple space can be searched, and then a tuple inserted into that section of the tuple space which matches the template. After this has occurred the search of another section of the tuple space is performed, and no matching tuple is found. The `inp` primitive therefore returns false, when in fact during its "search time" a matching tuple was inserted. However, this is not a problem due to the asynchronous nature of Linda. A process does not know what another process has done, unless the processes explicitly synchronise. Therefore, if a process performs an `out` primitive placing the tuple $\langle 10_{integer} \rangle$ into a tuple space, and another process performs an `inp` primitive using the template $\langle | 10_{integer} | \rangle$, the second process does *not* know that the tuple exists, and therefore cannot predict the result and so cannot falsify the result. Hence, the `inp` primitive could return either true or false, and this would be perceived by the process as correct. This would still be correct if the tuple was inserted before the `inp` primitive commenced and the `inp` primitive returned false. However, Leichter notes this can be falsified by the process if *explicit synchronisation between the processes* occurs. This leads to the need for *out ordering*. If `out` ordering is used there is no need to perform a global synchronisation.

2.4.2 `out` ordering

The example processes shown in Program 2.1 (which is based on the example given by Leichter[Lei89]) demonstrate how a process can falsify the result of an `inp` primitive. Let us assume that the tuple space `ts1` is only accessible from the processes shown. Process one places two tuples into the tuple space `ts1`. Process two performs an `in` primitive on one of the tuples and then performs an `inp` primitive on the other. If the `inp` primitive returns false, the user process can falsify the result because it knows that another process has placed a matching tuple into the tuple space by virtue of the explicit synchronisation. Leichter argues that it is possible (and correct) for the `inp` primitive to return false in this situation. This assumption effectively makes the `inp` and `rdp` primitives useless in practice as they can, under such an assumption, always return false regardless of whether a tuple exists. A number of implementations support the `inp` and `rdp` primitives which, given the example program in Program 2.1, would allow the `inp` primitive to return false, most notably SCA C-Linda[Ass95].

Program 2.1 Out ordering example.

```
              Process one                    Process two

        out(ts1, [10]);        in(ts1, |["DONE"]|);
        out(ts1, ["DONE"]));  x := inp(ts1, |[?int]|);
```

Leichter argues that the reason why the `inp` primitive can return false is because the length of time a primitive takes is unknown. The basic argument is that the time taken to send messages to places where tuples are stored is not necessarily equal from all processes. Therefore the tuple which is inserted first could be travelling through the communication system whilst the other tuple is inserted, and the `in` primitive and the `inp` primitive performed. However, this perceived problem is an issue of implementation. Leichter also notes that the problem is further compounded because explicit synchronisation that allows a process to know that a tuple exists can occur through other processes.

The solution to the problem is to use *out ordering*[DWR95, Hup90], which ensures that a *single* process does not complete a subsequent `out` to a tuple space until the previous tuple the process inserted is present and visible within the tuple space to other processes. It is not guaranteeing *out ordering* across several processes – processes can independently insert tuples into a tuple space. The ordering is only guaranteed for a single process. This is perfectly achievable using an acknowledgement message between the run-time system which stores the tuple and the process performing the `out` primitive. Another `out` primitive cannot be performed until the acknowledgement has been received (see Chapter 5). If the system supports *out ordering* then the example in Program 2.1 has only one outcome: process two removes the tuple $\langle ``DONE"_{string}\rangle$ and then the variable `x` is assigned the tuple $\langle 10_{integer}\rangle$.

Is *out ordering* acceptable within Linda? Due to the informal nature of the semantics of Linda it is difficult to tell exactly what an `out` primitive does and when. It appears natural that *out ordering* should be preserved, because the `out` primitive is a non-blocking primitive that inserts a tuple into a tuple space, implying that the tuple should be present within the tuple space when the primitive has been performed. Carriero et al.[CG89b, CG90a] in the description of the `out` primitive state that: `out`*(t) causes tuple t to be added to tuple space; the executing process continues immediately*. This could support the first argument that the tuple is inserted *during* the primitive not *after* the primitive (and this is true of the early shared memory implementations created by Carriero[Car87]), but the notion of "*the executing process continues immediately*" may imply that the tuple can be inserted after the primitive has returned (and this interpretation is assumed by Hupfer[Hup90]). Given the vague semantics attached to the `out` primitive it does not seem unreasonable to assume that the `out` primitive *does* insert the tuple before completing, thus providing *out ordering*, and this can be specified in the formal semantics. This can be supported by considering two issues. Firstly, if a tuple is not inserted before the `out` primitive completes, when *does* a tuple have to be inserted? It would appear in the informal semantics, if *out ordering* is not used, to be a concept of as soon as possible after the primitive has completed, but does this mean that there is no need for an implementation to ever insert a tuple? Secondly, by considering the nature of tuple space access within Linda. Linda provides asynchronous communication *between* processes but the primitives of `in` and `rd` (and `collect`) provide synchronised tuple space access. Once the primitive is initiated a tuple space access occurs and completes before the primitive completes. By providing *out ordering* effectively the `out` primitive is being made synchronous, like the other tuple space access primitives.

This leaves the question of whether *out ordering* should be over all tuple spaces or single tuple spaces. Consider Program 2.2. If *out ordering* is guaranteed over all tuple spaces, rather than just a single tuple space, then the outcome of the program is deterministic with `n` being assigned one. However, if the *out ordering* is not guaranteed over all tuple spaces then the value of `n` is non-deterministic. Because the tuple $\langle \text{“DONE”}_{string} \rangle$ can appear in tuple space `ts2` *before* the tuple $\langle 10_{integer}, 10_{integer} \rangle$ can appear in tuple space `ts1` the `collect` primitive may or may not find the tuple $\langle 10_{integer}, 10_{integer} \rangle$ in tuple space `ts1`.

If *out ordering* is only guaranteed over single tuple spaces, then to make the program deterministic, the same tuple space would have to be used for both the $\langle 10_{integer}, 10_{integer} \rangle$ and $\langle \text{“DONE”}_{string} \rangle$ tuples. Should *out ordering* be guaranteed over all tuple spaces or just individual tuple spaces? If *out ordering* is only guaranteed for a single tuple space, then this would imply that the tuple is *not* inserted before the primitive completes, raising the question again of when a tuple is inserted, after the last tuple was inserted by the same process and before the next tuple is inserted from the same process, but that is all. If the `out` primitive, when *out ordering* is guaranteed, is seen as a synchronous tuple space access primitive, then *out ordering* should be guaranteed over all tuple spaces. This is because when a primitive completes the tuple will be present in the tuple space, hence *out ordering* is guaranteed over all tuple spaces. As a final point, if *out ordering*

Program 2.2 *Out ordering* across multiple tuple spaces.

```
int A(TS ts1, TS ts2)
{
  out(ts1, 10, 10);
  out(ts2, "DONE");
  return 0;
}

void B(void)
{
  int n;
  TS ts1, ts2;

  ts1 = tsc();
  ts2 = tsc();

  eval(ts1, A(ts1,ts2));
  in(ts2, "DONE");
  n = collect(ts1, ts2, 10, 10);
}
```

is used then there is no need to perform a global synchronisation for the `inp` or `rdp` primitives. Each of the sections of the tuple space can be searched independently, without fear of a matching tuple not being found that a process knows exists.

2.5 Host languages used in this dissertation

All examples in this dissertation are given using either ISETL-Linda[DRW95] or C-Linda. A brief description of the syntax of the Linda primitives for each of the embeddings is now given.

2.5.1 ISETL-Linda syntax

ISETL (**I**nteractive **SET L**anguage)[BDL89] is a set based interpreted imperative language. ISETL-Linda is a full implementation of Linda as outlined above, except that it supports only a limited `eval` primitive[1]. The ISETL-Linda used in this dissertation runs on a Meiko Computing Surface 1 with 32 transputers, and uses the York Kernel I run-time system (see Appendix A).

[1]There is another implementation of Linda using ISETL as the host language called ProSet-Linda[Has94]. The main difference between ISETL-Linda and ProSet-Linda is that the latter does not support the `eval` primitive, but provides another mechanism for process spawning (based on futures).

In ISETL-Linda the tuple

$$\langle 10_{integer}, \text{“Hello World”}_{string}, 10_{integer}, 1.0_{float}\rangle$$

is written as

```
[10, "Hello World", 10, 1.0]
```

where the interpreter automatically types the fields for the user. The templates

$$\langle| 10_{integer}, \text{“Hello World”}_{string}, 10_{integer}, 1.0_{float}|\rangle$$

and

$$\langle| 10_{integer}, \square_{string}, \square_{integer}, 1.0_{float}|\rangle$$

are written as

```
|[10, "Hello World", 10, 1.0]|
```

and

```
|[10, ?str, ?int, 1.0]|
```

where, if a field is an actual it is typed automatically, and if a field is a formal it is represented by a “?” followed by a type descriptor. As tuples are first class objects in ISETL-Linda, all Linda primitives that return results representing tuples return tuples. The following keywords are added to ISETL:

- `NewBag`

  This function[2] creates a new tuple space:

  ```
  ts := |{}|;
  ts := NewBag;
  ```

  where both functions create a new tuple space and assign the handle of the created tuple space to the variable `ts`. A tuple space has to be explicitly created before it can be used. Tuple space handles can be passed within tuples, using the type descriptor `bag`.

- `lout`

  This is the `out` primitive. It has the following syntax:

  ```
  lout(tuple_space, tuple);
  ```

  where `tuple_space` is a valid tuple space handle and `tuple` is a tuple.

[2]The function `NewBag` is overloaded with the function $|\{\}|$. Therefore $|\{\}|$ has the same effect as `NewBag`.

- `lin`

  This is the `in` primitive. It has the following syntax:

  ```
  result := lin(tuple_space, template);
  ```

  where `tuple_space` is a valid tuple space handle and `template` is a template. The tuple returned is assigned to `result`.

- `lrd`

  This is the `rd` primitive. It has the following syntax:

  ```
  result := lrd(tuple_space, template);
  ```

  where `tuple_space` is a valid tuple space handle and `template` is a template. The tuple returned is assigned to `result`.

- `lcollect`

  This is the `collect` primitive. It has the following syntax:

  ```
  count := lcollect(tuple_space1, tuple_space2, template);
  ```

  where `tuple_space1` and `tuple_space2` are both valid tuple space handles and `template` is a template. The number of tuples moved from `tuple_space1` to `tuple_space2` is assigned to `count`.

- `leval`

  This is the `eval` primitive. It has the following syntax:

  ```
  leval(tuple_space, active_tuple);
  ```

  where `tuple_space` is a valid tuple space handle and `active_tuple` is an active tuple. In the context of ISETL-Linda an active tuple is said to be the same as a tuple except *no more than one* of its fields is a function to be evaluated concurrently. When the function has been evaluated a tuple is inserted into the tuple space `tuple_space`.

An example interactive session using ISETL-Linda is shown in Figure 2.1. First a new tuple space is created, then a tuple is placed into that tuple space. The tuple is then retrieved, printed and then the two elements of the tuple are printed independently. A more detailed explanation of ISETL-Linda is presented in Douglas et al.[DRW95].

2.5.2 C-Linda syntax

Two versions of C-Linda have been used. One is a commercial version produced by Scientific Computing Associates[3] which will be referred to as SCA C-Linda[Ass95]. The product is based on

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

```
>       ts := |{}|;
>       lout(ts,[10,"HELLO"]);
>       answer := lin(ts,|[?int,?str]|);
>       answer;
[10,"HELLO"];
>       answer(1);
10;
>       answer(2);
"HELLO";
>
```

Figure 2.1: An example of an interactive session using ISETL-Linda.

the work completed at Yale University and is a closed implementation using compile time analysis. The other is a C-Linda which uses the York Kernel II (described in Chapter 5) or the York Kernel I[DWR95], which is an open implementation, and does not use compiler-time analysis. Throughout this dissertation when *any* C-Linda code is being presented it will use a modified syntax of SCA C-Linda[4]. In C-Linda the tuple:

$$\langle 10_{integer}, \text{“Hello World”}_{string}, 10_{integer}, 1.0_{float}\rangle$$

is written as

```
(10, "Hello World", 10, 1.0)
```

where the compiler automatically types the fields for the user. The templates

$$\langle| 10_{integer}, \text{“Hello World”}_{string}, 10_{integer}, 1.0_{float}|\rangle$$

and

$$\langle| 10_{integer}, \square_{string}, \square_{integer}, 1.0_{float}|\rangle$$

are written as

```
(10, "Hello World", 10, 1.0)
```

and

```
(10, ?str, ?var_name, 1.0)
```

[4]See Appendix A for an overview of the main features of each of the implementations/run-time systems.

where, if a field is an actual it is typed automatically, and if a field is a formal it is represented by a "`?`" followed by either a type descriptor or a variable name where the compiler determines the type. Tuples are *not* first class objects in C-Linda, therefore if a type descriptor is used the information is discarded. If the field is a formal and a variable is used then the result for that field is placed into that variable. In the last example above, the second field of the tuple is discarded, but the third field (an integer) is placed into a variable called `var_name`. The first template appears identical to a tuple; whether a collection of values is a tuple or a template is based upon the Linda primitive with which they are associated. The embedding of Linda in C involves the addition of the following functions/procedures to C:

- `tsc`

  This function creates a new tuple space. The prototype for the function is:

  ```
  TS tsc(void)
  ```

  Therefore, the statement

  ```
  ts = tsc();
  ```

  creates a tuple space and assigns its handle to the variable `ts`. The type `TS` is added, and variables of type `TS` are used to store tuple space handles.

- `out`

  This procedure is the `out` primitive. The prototype for the procedure is:

  ```
  void out(TS, ...)
  ```

  Therefore, the statement

  ```
  out(ts, 10, "HELLO");
  ```

  places a tuple $\langle 10_{integer}, \text{“HELLO”}_{string}\rangle$ into tuple space `ts`. There are no limits on the number of elements in the tuple.

- `in`

  This procedure is the `in` primitive. The prototype for the procedure is:

  ```
  void in(TS, ...)
  ```

  Therefore, the statement

  ```
  in(ts, ?my_val, "HELLO");
  ```

  retrieves a tuple in tuple space `ts` which matches the template $\langle | \square_{integer}, \text{“HELLO”}_{string} | \rangle$. When a tuple is found it is split into component fields. In the above example `?my_val` is a formal (of type integer) and the first field of the returned tuple will be assigned to `my_val`.

- `rd`

  This procedure is the `rd` primitive. It has the same syntax as the `in` procedure above, except it is called `rd`.

- `collect`

  This procedure is the `collect` primitive. The prototype for the procedure is:

  ```
  int collect(TS, TS, ...)
  ```

  Therefore, the statement

  ```
  count = collect(ts1, ts2, ?int, "HELLO");
  ```

  moves all the tuples in tuple space `ts1` which match the template $\langle | \square_{integer}$, "HELLO"$_{string} | \rangle$ to tuple space `ts2`. A count of the number of tuples moved is assigned to `count`.

- `eval`

  This procedure is the `eval` primitive. The prototype for the procedure is:

  ```
  void eval(void (*)())
  ```

  Therefore, the statement

  ```
  eval(worker);
  ```

  will spawn the process `worker`. The result of the function `worker` is discarded[5].

2.5.3 Processes and the host languages

In ISETL-Linda the `eval` primitive requires a function that is currently in scope to be used as the function to be evaluated concurrently. Therefore, a process is an ISETL function that is in scope at the time the `eval` primitive is performed.

In SCA C-Linda the `eval` primitive again requires a function to be specified that is in scope within the program performing the `eval` primitive. In York C-Linda the `eval` primitive requires a filename for an executable file. The file is executed, and interacts with the run-time system.

Whether the process is an executable file, or a function which is somehow executed concurrently, they will be referred to as processes. Within this dissertation's example programs processes are always given as though the process is a function.

2.6 Non-determinism and Linda

There are two characteristics of Linda which introduce non-determinism. An understanding of these characteristics and their implications is important for anyone working with Linda, especially if new primitives are to be proposed. These characteristics are:

[5]SCA C-Linda supports a full `eval` primitive that allows many functions to be specified within a single tuple.

Inter-process tuple competition When two or more processes compete for the same tuple, which process receives the tuple is non-deterministic. This is shown in Program 2.3, where a process spawns two processes both of which will compete for the same tuple. One of the processes will obtain the tuple and the other will block (in this case forever). It is impossible to say whether the first process spawned or the second process will retrieve the tuple, the choice is non-deterministic.

Program 2.3 Example showing inter-process tuple competition.

```
worker := func(ts);             $ ts is a tuple space handle
  x := lin(ts,|[?int]|);        $ Get a tuple
  return 0;
end func;

simple := proc();
  local tuple_space;
  tuple_space := NewBag;        $ Create a new tuple space
  leval(ts,[worker(tuple_space)]);
  leval(ts,[worker(tuple_space)]);
  lout(tuple_space,[1]);        $ Put a tuple in the tuple space
end proc;
```

Multiple tuple competition When there is more than one tuple within a tuple space that would satisfy a template the tuple is chosen non-deterministically. This is shown in Program 2.4, where a tuple space is created and two tuples placed into the tuple space. The `in` primitive will match one of the tuples, but which one is chosen is a non-deterministic choice.

Program 2.4 Example showing multiple tuple competition.

```
simple := proc();
  local ts,x;                 $ Local variables
  ts := NewBag;               $ Create a new tuple space
  lout(ts,[1]);               $ Place a tuple in the tuple space
  lout(ts,[2]);               $ Place a tuple in the tuple space
  x := lin(ts,|[?int]|);      $ Get a tuple out of the tuple space
end proc;
```

These competition characterstics are important to the Linda user, providing natural interaction between processes and tuples. If the user requires more control over the non-determinism then the ability to use explicit coordination of structures in tuple spaces provides a way. For example to control the order in which tuples are retrieved from a tuple space each tuple can be explicitly tagged with a field. Such a method is described in Chapter 3, when describing the stream method for the multiple `rd` problem.

However, sometimes a poor understanding of non-determinism within Linda can led to flawed proposals. The case of the proposal to use *instructional footprinting* with the Linda primtives is a good example.

2.6.1 Instructional footprinting

Landry[LA92, LA93a, LA93b, LA95] examined the application of instructional footprinting to Linda programs in order to reduce the execution times. The basis for the work is the idea that an `in` or `rd` primitive can be split into two sections, the "sending" of the template to the run-time system ($\texttt{in}_{init}$) and the receiving of a tuple from a run-time system ($\texttt{in}_{recv}$). Landry's proposal is that a pre-compiler can automatically split all the `in` and `rd` primitives into their components, and move them apart. Normally, when a Linda primitive is performed the user computation stops, waiting for a reply message. The separation of the request for a tuple and the actual retrieval of the tuple allows user computation to be performed concurrently with the tuple space access, thereby providing a speed increase.

At first sight, the idea seems sound, and the results presented[LA95] indicate that on the particular LAN based kernel used, a significant drop in execution times is achieved in most cases and the execution times never increase. However, Landry failed to account fully for non-determinism and its effect on the Linda programs. He assumed that, using a set of rules, moving computation between the $\texttt{in}_{init}$ and $\texttt{in}_{recv}$ parts of an `in` primitive would not alter the semantics of the program. This is true, the semantics of the program are not altered, but the coordination constructs used in Linda programs often make use of the time computation takes.

In order to demonstrate this consider the *dining philosophers problem*. Landry[LA95] presents a good description of the dining philosopher problem. Program 2.5 and 2.6 are slightly modified code sections taken from [LA95]. Program 2.5 shows the original Linda function for a philosopher, while Program 2.6 shows the so-called optimised version of the same function which the pre-compiler doing the instructional footprinting would produce, with each `in` primitive split into two components. Both pieces of code have been modified to allow the length of time a philosopher thinks before he eats to be specified.

Landry sets all the philosophers to work for the same length of time (0 seconds). Let us consider, for the sake of simplicity, two philosophers, called Phil0 and Phil1, with Phil0 thinking for 20 minutes and Phil1 thinking for 40 minutes. It is assumed that they both eat for the same length of time, 10 minutes. Figure 2.2 shows how the two philosophers spend their time in the original Linda version.

The version that has been optimised using instructional footprinting can produce the same

Program 2.5 Linda code for a philosopher.

```
void Philosopher(int time, int Phil_ID)
{
  think(time);
  in("Room Ticket");
  in("Chopstick",Phil_ID);
  in("Chopstick",(Phil_ID+1) % Num_Phil);
  eat();
  out("Chopstick",Phil_ID);
  out("Chopstick",(Phil_ID+1) % Num_Phil);
  out("Room Ticket");
}
```

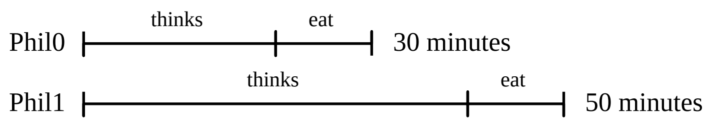


Figure 2.2: How the philosophers spend there time in the Linda version.

results[6] as shown in Figure 2.2. However, it is also possible for the result shown in Figure 2.3 to be produced. In this case Phil0 blocks for 30 minutes, because both philosophers try to grab the room ticket tuple *at the same time*. There is only one room ticket tuple and as the two processes compete for the same tuple, which philosopher gets the room ticket tuple is non-deterministic. If Phil0 gets it then everything will proceed as in the Linda version. If Phil1 gets it, then Phil0 does his thinking and then must wait for Phil1 to release the ticket, so Phil0 is blocked. Phil1 thinks for his 40 minutes then eats and then releases the room ticket tuple. Phil0 now becomes unblocked and is able to eat. In this case the total execution time is 60 minutes, as opposed to 50 minutes representing an increase in the execution time.

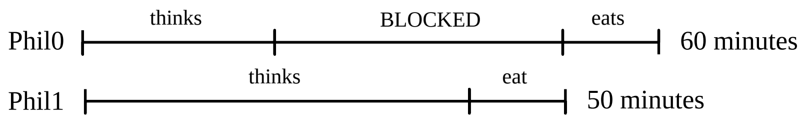


Figure 2.3: How the philosophers may spend their time in the optimised version.

In the Linda version the computation provides a natural way of controlling the access to the dining room. Only when a philosopher is *ready* to eat does he attempt to get the ticket, so once he has the resource (the ticket) he uses it immediately and then releases it. Many Linda programs

[6]The execution time will be slightly less, but because long thinking times and eating times are chosen the effect of reducing the communication time can be effectively ignored.

Program 2.6 “Optimised” Linda code for a philosopher using instructional footprinting.

```
void Philosopher(int time, int Phil_ID)
{
  INinit("Room Ticket");
  INinit("Chopstick",Phil_ID);
  INinit("Chopstick",(Phil_ID+1) % Num_Phil);
  think(time);
  INrecv("Room Ticket");
  INrecv("Chopstick",Phil_ID);
  INrecv("Chopstick",(Phil_ID+1) % Num_Phil);
  eat();
  out("Chopstick",Phil_ID);
  out("Chopstick",(Phil_ID+1) % Num_Phil);

  out("Room Ticket");
}
```

use the fact that computation takes time to do natural load balancing[CG90a, page 114], in these cases the instructional footprinting optimisations can lead to longer execution times, as with this example.

2.7 Summary

An overview of Linda, describing the basic objects of Linda was presented in Section 2.2. A number of extensions to Linda were then described, including multiple tuple spaces and the `collect` primitive, which are adopted in the Linda used within this dissertation. More information about the bulk primitives is presented in Chapter 4. The need for global synchronisations and `out` ordering has been discussed in detail with respect to the `inp` and `rdp` primitives. An overview of the host languages and their Linda embeddings was presented, ending with a description of non-determinsim within Linda. Chapter 5 will present a detailed overview of both closed and open implementations, and the techniques which they use.

In the next chapter a limitation of the functionality of Linda is examined. This limitation forms the justification for a new primitive for Linda, the `copy-collect` primitive, which is a bulk primitive related to the `collect` primitive.

Chapter 3

The multiple `rd` problem

3.1 Introduction

In this chapter, an expressive limitation of the Linda model is identified, which is referred to as the *multiple* `rd` *problem*. A multiple `rd` is defined as an operation where two or more processes are required to concurrently, and non-destructively read one or more tuples from a tuple space which match the same template, where there are at least two tuples that match the template, and at least two of the processes can be satisfied by the same tuple. The problem is that a multiple `rd` cannot be performed efficiently using the current Linda model if two or more processes are concurrently and non-destructively reading from a tuple space using the same template.

As an example, consider a tuple space containing a number of tuples with each containing two fields representing peoples names such as $\langle \text{“Antony”}_{string}, \text{“Rowstron”}_{string} \rangle$. This tuple space is shared among many processes that may require access to the tuples. How would all the tuples representing people whose surname is *Rowstron* be retrieved by a process?

Initially, the answer would appear to be the repeated use of the `rd` primitive. The template $\langle | \square_{string}, \text{“Rowstron”}_{string} | \rangle$ will match a tuple whose surname is *Rowstron*. This will only work if there is a single tuple which matches the template. If all the names of an entire family are in the tuple space, or there are several unrelated people with the same surname stored in the tuple space, the repeated use of a `rd` will not work. The semantics of `rd` mean that if more than one tuple matches a template the tuple returned is chosen non-deterministically. Having discounted the use of the `rd` primitive the answer may appear to be to use the `collect` primitive. The `collect` primitive will match all the matching tuples. However, the primitive is a *destructive* operation therefore the collected tuples are removed so other processes cannot read them. Also, the behaviour of concurrent `collect` primitives is not well defined (see Section 4.2).

There are only two methods that enable many processes to concurrently and non-destructively access a tuple space using the current Linda model. One method is to use a designated tuple as a binary semaphore and the other is to organise the tuples as a stream. Before these two methods are examined and evaluated another example containing the multiple `rd` problem is described.

3.2 Parallel composition of two binary relations

The parallel implementation of the composition of two binary relations is now considered in order to show how the different methods of solving the multiple `rd` problem perform.

3.2.1 Formal definition of the composition of two binary relations

A binary relation is defined as a relation between two sets. A binary relation defines a subset B, of the Cartesian product of the two sets. Therefore, given two sets T and X, the Cartesian product $T \times X$ is defined as:

$$\{(t, x) : (t \in T) \text{ and } (x \in X)\} \tag{3.1}$$

If the ordered pair (s_1, s_2) is a member of the set B, then the binary relation B is said to hold between the two values. This binary relation could for example be "less than", so $s_1 < s_2$. Given two binary relations R and S, their composition $R \circ S$ is defined as:

$$\{(a, d) : ((a, b) \in R) \text{ and } ((c, d) \in S) \mid b = c\} \tag{3.2}$$

3.2.2 The general approach to implementation

This example assumes that the ordered pairs in each set are held in separate tuple spaces, with each tuple representing a single ordered pair. After performing the composition a new tuple space will be created containing the resulting tuples. This is shown in Figure 3.1.

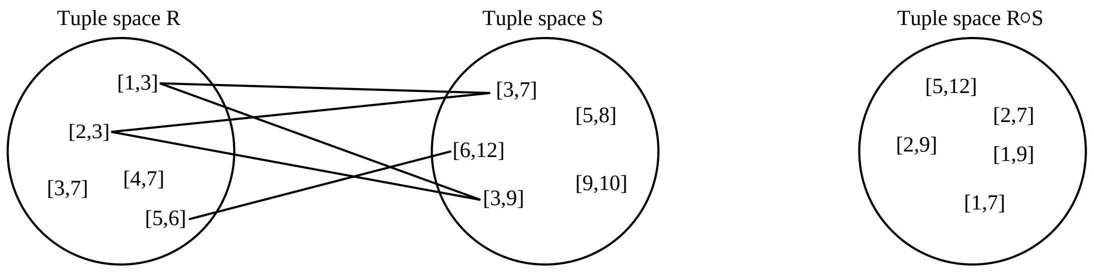


Figure 3.1: Composition of two binary relations represented using three tuple spaces.

Due to the properties of the composition of binary relations it should be simple to implement in parallel, with every pair in tuple space R being compared with each pair in tuple space S concurrently. The results for each pair in tuple space R are independent of the results for any other pair in tuple space R. So a number of processes are used. Each process takes a pair from tuple space R, and checks the chosen pair with every pair in tuple space S. If the second element of the pair from tuple space R is the same as the first element in a pair from tuple space S, a new pair is produced. This new pair contains the first element of the pair from tuple space R and the second element from the pair from tuple space S. The finest grained parallel approach using this method will use a process for every pair in tuple space R. The detecting of the elements in tuple space S which match is an associative matching process which indicates that Linda should be ideal because of the associative matching properties it has.

The multiple `rd` problem is seen within the parallel composition of two binary relations because there are several processes that need to concurrently access the tuple space S non-destructively. The stream and semaphore methods of solving the multiple `rd` problem are now considered using the parallel composition of two binary relations as an example.

3.3 Tuples as semaphores

The first method considered for overcoming the multiple `rd` problem is to use a tuple as a binary semaphore, or *lock tuple*. The lock tuple is a single and unique tuple that allows *user processes* to control access to a tuple space.

The general concept is that a process obtains the lock tuple, then *destructively* removes the matching tuples, using either the `inp` or `collect` primitives[1]. Once all the tuples have been removed they are replaced, and then the lock tuple reinserted. The removal of tuples is acceptable because only a single process can obtain the lock tuple, and therefore access the tuple space at any one time[2] provided the tuple space is returned to the same state as it was when the lock tuple was removed no other process will be aware that the tuples have been removed and replaced.

In the case of the parallel composition of binary relations, the ISETL-Linda code for a worker process is shown in Program 3.1. Each worker process removes a tuple from tuple space R and then tries to remove the lock tuple in tuple space S. There is only one lock tuple in the tuple space S so all but one of the processes will block on the `in` primitive (line `A` in Program 3.1). When a worker process retrieves the lock tuple it has unrestricted access to the tuple space S.

The worker process creates a template using the second field of the tuple removed from tuple space R as the first element of the template. In this example, the template is then used by a `collect` primitive to *move* all[3] the tuples that match the template in tuple space S to a local tuple space. The same operation can be performed using the `inp` primitive.

The worker process then removes each of the tuples from the local tuple space using the `in` primitive. The worker process then places the tuple back into tuple space S. Because of the fine grained nature of the worker processes used in the composition of binary relations, as the worker process returns the tuples to tuple space S it also calculates any results and places them in the result tuple space C. If the computation "associated" with each tuple is more complex then either

[1]If the Linda implementation supports neither of these then the semaphore method cannot be used, and streams *must* be used.

[2]Provided that all the processes accessing the tuple space adhere to the use of the lock tuple.

[3]In this case all is acceptable because the tuple space will be *inactive* if all processes adhere to using the semaphore tuple.

the worker process can place another copy of each tuple in a different tuple space for processing once all the tuples from tuple space S have been replaced, or further processes can be spawned to actually perform the calculations.

Once all the tuples in the local tuple space have been processed and replaced into the tuple space S, the lock tuple is placed back into tuple space S. This means that tuple space S contains all the tuples that were present when the worker process obtained the lock tuple. The tuple which acts as the semaphore can *only* be replaced in the tuple space when the tuple space is in its original state. If the tuple is returned prior to this then the other processes are not guaranteed to find all the tuples that they require.

Program 3.1 A worker process using a tuple as a binary semaphore or lock tuple.

```
comp_worker := func(R,S,C);

  local my_val, my_ts, my_comb, todo;

  my_ts := NewBag;
  my_val := lin(R,|[?int,?int]|);     -- Get the element from R

  dummy := lin(S,|["lock"]|);         -- Get the lock (A)

  todo := lcollect(S,my_ts,|[my_val(2),?int]|);
  while (todo > 0) do            -- Grab matching tuples in S
    todo := todo - 1;
    my_comb := lin(my_ts,|[my_val(2),?int]|); -- Process each one
    lout(C,[my_val(1),my_comb(2)]);   -- Create result tuples
    lout(S,my_comb);                  -- Replace tuple in S
  end while;

  lout(S,["lock"]);                   -- Let the lock tuple go

  return ["TERMINATED"];
end func;
```

3.3.1 Performance

There are two reasons why this is *not* an acceptable solution to the multiple `rd` problem.

- One is that the solution requires the processes that use a tuple space to adhere to using the lock tuple, and there is no guarantee that other processes will adhere to it. Consider the example given in the introduction of this chapter, involving a tuple space containing a set of names. There is no reason why several processes performing different and unrelated tasks may all require access to the name tuples within the tuple space concurrently. Suppose one process does not adhere to the use of a lock tuple, either maliciously or accidently, then all the processes can no longer reliably have access to all the possible tuples.

- The second reason why such an approach is not acceptable is that it creates a sequential bottleneck for the access of the tuple space, as only one process can obtain the lock tuple at any one time. Therefore, in the parallel composition of binary relations example the only speed up achieved is the parallel reading of the tuples from tuple space R. The majority of the time that the program executes only a single worker is active creating a sequential solution because only one process can access the tuples within tuple space S at anyone time.

3.4 Streams

The second approach is to use a *stream*. The basis of this approach is to remove the multiple `rd` problem by having only one tuple match the template being used. This is achieved either by using information which is already in the tuples, or by adding a unique field to each tuple. This means that a unique template can be generated which will match a single tuple in the the tuple space. Any processes which wants to use the tuples within the tuple space must be aware of the fields used within the tuple and, if necessary, how the field is generated. Processes accessing the tuple space use the `rd` primitive to retrieve *every* tuple, and use a local check to see if the tuple is required.

Consider the example of the parallel composition of binary relations, and assuming that the tuple space S contains the five tuples (as shown in Figure 3.1):

$$\langle 3_{integer}, 7_{integer}\rangle, \langle 6_{integer}, 12_{integer}\rangle,$$
$$\langle 3_{integer}, 9_{integer}\rangle, \langle 5_{integer}, 8_{integer}\rangle, \langle 9_{integer}, 10_{integer}\rangle.$$

There is no unique field that allows each tuple to be independently chosen. Therefore a unique field is added to each of the tuples:

$$\langle 1_{integer}, 3_{integer}, 7_{integer}\rangle, \langle 2_{integer}, 6_{integer}, 12_{integer}\rangle,$$
$$\langle 3_{integer}, 3_{integer}, 9_{integer}\rangle, \langle 4_{integer}, 5_{integer}, 8_{integer}\rangle, \langle 5_{integer}, 9_{integer}, 10_{integer}\rangle.$$

After adding the extra first field, each tuple contains a unique field, and the relationship across the tuples between the unique fields is known (an integer counter that is incremented by one for each tuple). This allows a process to access the tuple space using the template $\langle | index_{integer}, \square_{integer}, \square_{integer} | \rangle$ where *index* is a value between one and five in this example. Every worker process takes a tuple from tuple space R, and then reads *every* tuple from tuple space S, using the *index* field to match each tuple in turn. The worker process checks if the returned tuple is actually required and either discards it or uses it accordingly. If the implementation supports the `rdp` primitive then this removes the need to check the tuple *locally*, but all tuples still have to be checked. A template of the form $\langle| index_{integer}, R(2)_{integer}, \square_{integer} |\rangle$ would be used, where *R(2)* is the second element from the tuple retrieved from the tuple space R. The `rdp` primitive would then be used, and would return either the matching tuple or a value to indicate it was not found. Every tuple still has to be checked. The ISETL-Linda code for a worker process using the stream method is shown in Program 3.2.

Program 3.2 A worker process using streams.

```
comp_worker := func(R,S,C,NumTupS);
                    -- NumTupS - No. of tuples in S
  local my_val, my_comb;

  my_val := lin(R,|[?int,?int]|);  -- Get a tuple from R

  while (NumTupS > 0) do           -- Check all tuples in S
    my_comb := lrd(S,|[NumTupS,?int,?int]|);
    NumTupS := NumTupS - 1;
    if (my_comb(2) = my_val(2)) then -- Does the tuple match?
      lout(C,[my_val(1),my_comb(3)]);
    end if;
  end while;

  return ["TERMINATED"];
end func;
```

In this example it is necessary to add an extra field but sometimes a unique field is already present within the tuple. For example, when an image is stored in a tuple space, with each pixel being stored as a tuple of the form:

$$\langle \text{x-coordinate}_{integer}, \text{y-coordinate}_{integer}, \text{pixel value}_{integer} \rangle.$$

A process may want to access all pixels that are of a particular value[4]. Here the obvious template would be $\langle| \square_{integer}, \square_{integer}, \text{pixel value}_{integer} |\rangle$. However, if many processes wish to perform the operation in parallel it will introduce the multiple `rd` problem. Assuming that usually the coordinate system used within the image will be known to the accessing processes, and there is only one pixel value for each coordinate, the coordinate fields within the tuple can be used as the unique fields. The processes can then use a stream approach, reading every coordinate to check if the pixel value is the one required, and discarding if it is not.

[4]See Hough transform, Section 6.3.

3.4.1 Performance

Although, with this method all the worker processes can perform the accessing of a tuple space in parallel, there are two problems that make this approach unacceptable.

- Firstly, it negates the advantages of the tuple matching abilities of Linda. *Every* tuple in the stream structure *must* be read. If there are many tuples in a tuple space and only a few are required, the time cost and communication cost of reading every tuple is considerable. This is compounded if the implementation does not support the `rdp` primitive, because additional checking within the user process of the returned tuple is required, to check if the tuple is one that is required.

- Secondly, every tuple in the tuple space requires a unique field to be added, and all the processes using the tuples must be aware of the unique field and how it is generated. This removes the natural use of a tuple space as the data structure by adding another structure (a stream) to the tuples within the tuple space. In order to achieve this, either the producer must be aware of the need to add this unique field in which case the cost of adding it is minimal, or the tuples are pre-processed to add the unique field before being used.

Even if the producer can add the extra field, and so no pre-processing of the tuples is required, the communication and time costs of checking every tuple explicitly using either the `rd` or `rdp` primitives is unacceptable unless the majority of tuples within a tuple space match the template.

3.5 Experimental results

In order to show the problems of both the binary semaphore and stream methods the execution times of the parallel composition of binary relations using both these methods are considered. The experimental results presented in this section are obtained using ISETL-Linda executing on a transputer based Meiko CS-1 parallel computer using York Kernel I[DWR95] (an overview of the main features of York Kernel I is given in Appendix A). For the experiments the cardinality of the tuple space R is set to five; the cardinality of tuple space S is 50. For every pair (represented as a single tuple) in tuple space R there are four pairs (again, represented as single tuples) in tuple space S that match, therefore the cardinality of the composition tuple space C is 20. The worker processes are altered to enable them to be instructed on how many tuples are processed from tuple space R. Thus, a single worker computes the results for all five pairs in tuple space R, whereas five worker processes each compute the results for a single pair from tuple space R, as in the example code segments Program 3.1 and 3.2. This is used to show that the semaphore method forces sequential access to tuple space S, whilst the stream approach allows parallel access to tuple space S.

The execution time *does not* include the time taken to spawn the worker processes, and does not include the time taken to create the tuple spaces S and R. In the stream approach it is assumed that the producer added the unique field to the tuples as they are created, thus avoiding the need to pre-process the tuples to add the unique field.

The performance for both the methods of solving the multiple `rd` problem are compared against a sequential version of the composition of binary relations. The code for the sequential version is shown in Program 3.3. The sequential version takes each tuple from tuple space R, then uses the `collect` primitive to destructively move to another tuple space every tuple from tuple space S in which the first element of the tuple is the same as the second element of the current tuple chosen from tuple space R. The moved tuples are then destructively read using the `in` primitive from the other tuple space, processed and then placed back into tuple space S. The result tuples are placed in tuple space C. The sequential version uses the tuple spaces to store data structures as do the parallel versions.

Program 3.3 The code for the sequential composition of binary relations.

```
comp_worker := proc(R,S,C,NumTupR);
      - NumTupR is the number of tuples in R
  local my_val, my_ts, my_comb, todo, loop;

  my_ts := |{}|;
  for loop in [1 .. NumTupR] do     -- For all tuples in R
    my_val := lin(R,|[?int,?int]|); -- Get a tuple

    todo := lcollect(S,my_ts,|[my_val(2),?int]|);
    while (todo > 0) do             -- Process the matched tuples
      todo := todo - 1;
      my_comb := lin(my_ts,|[my_val(2),?int]|);
      lout(C,[my_val(1),my_comb(2)]);
      lout(S,my_comb);
    end while;
  end for;
end proc;
```

3.5.1 The binary semaphore method

Figure 3.2 shows the execution times taken for the version using the semaphore method when the number of worker processes are varied from between one and five. Also shown is the time taken for a sequential version of the program. The timings are given in ticks, which are arbitrary units of time (15625 ticks per second).

The sequential version is slightly faster than the parallel version using the semaphore method

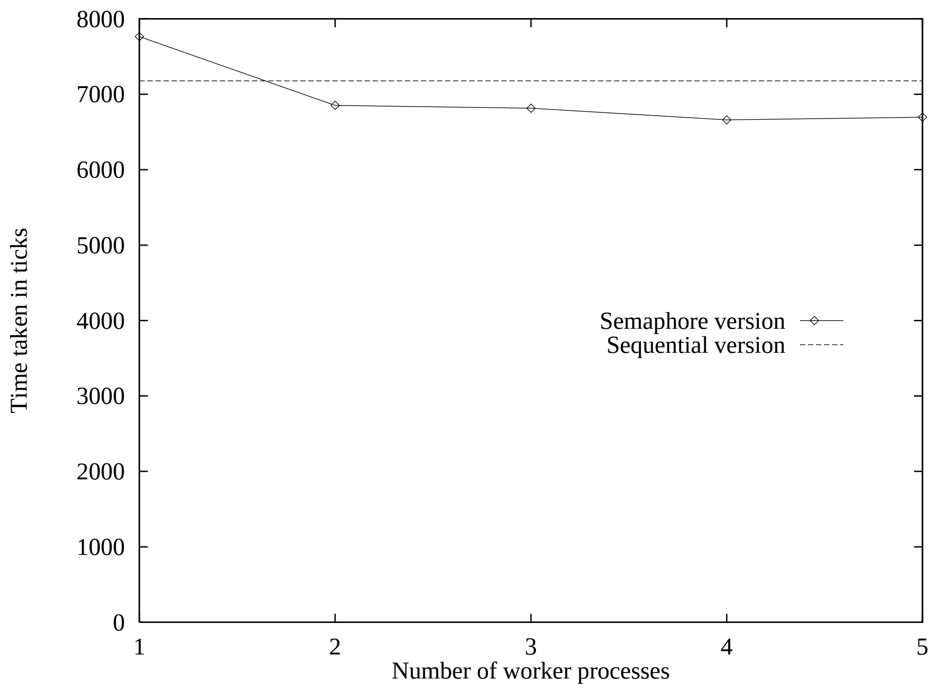


Figure 3.2: Execution time for the parallel composition of binary relations when using the binary semaphore method.

and one worker process. This is because the sequential implementation is similar to the semaphore method except that a lock tuple is not used, as only one process can access the tuple space. This means the difference in the execution times represents the cost of fetching and replacing the lock tuple.

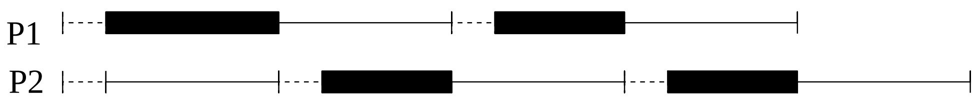
When two worker processes are used the execution time for the semaphore method is slightly less than the execution time for the sequential version. This is because of the parallel access to the tuple space R. The fetching of a tuple from tuple space R is the only work that can be performed concurrently; the access to tuple space S is forced to be sequential.

There is no performance gain by increasing the number of worker processes above two. Ideally, the execution time taken by five worker processes should be one third of the time taken when using two worker processes. When there are two worker processes one will consume three tuples from tuple space R and the other will consume two tuples from tuple space R. Five worker processes will each consume only one tuple from tuple space R. The reason why this does not occur is shown in Figure 3.3 where the solid line represents the time a worker process is accessing tuple space S, and the dotted line represents the time when the worker is accessing tuple space R. The solid thick black lines represent the time when a worker process is blocked awaiting the lock tuple. The length of the time taken by the longest worker is the execution time of the program. Figures 3.3(a) and 3.3(b) show that the time taken by the longest worker process when either two or five worker processes are used is the same. If three or four worker processes are used then the longest worker

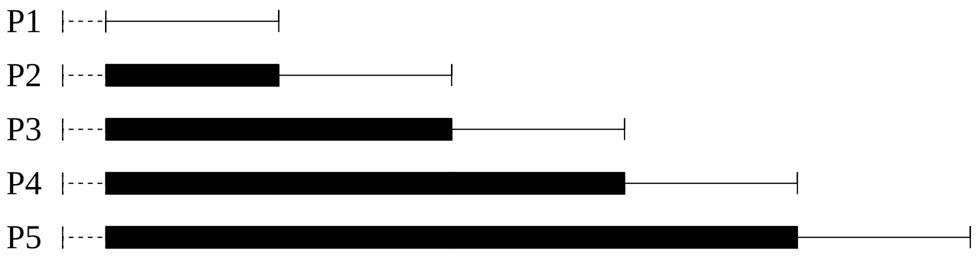
Number of workers	Number of tuples processed per worker	Maximum number of tuples processed by any worker	Execution Time	Time per tuple
1	5	5	49506	9901
2	2,3	3	29633	9877
3	1,2,2	2	20280	10140
4	1,1,1,2	2	20298	10149
5	1,1,1,1,1	1	12079	12079

Table 3.1: Time taken per element in tuple space R as the number of worker processes increase.

process will again take the same time. As the number of worker processes increase there is no performance increase because there is nothing more that can be achieved in parallel.



(a) Execution pattern using two worker processes.



(b) Execution pattern using five worker processes.

Figure 3.3: Execution patterns for two and five worker processes using the semaphore method.

3.5.2 The stream method

Figure 3.4 shows the execution times taken for the version using the stream method when the number of worker processes is varied from between one and five. Again the time taken for the sequential version is also shown and the predicted execution times are also shown. The predicted execution time is calculated on the basis of the time taken for one worker process.

The results show that the execution time is dependent upon the number of worker processes used. The relationship between the execution time and the number of worker processes is shown in Table 3.1. The first column represents the number of worker processes used, the second column shows the number of tuples from tuple space R that each worker process consumes (the total must

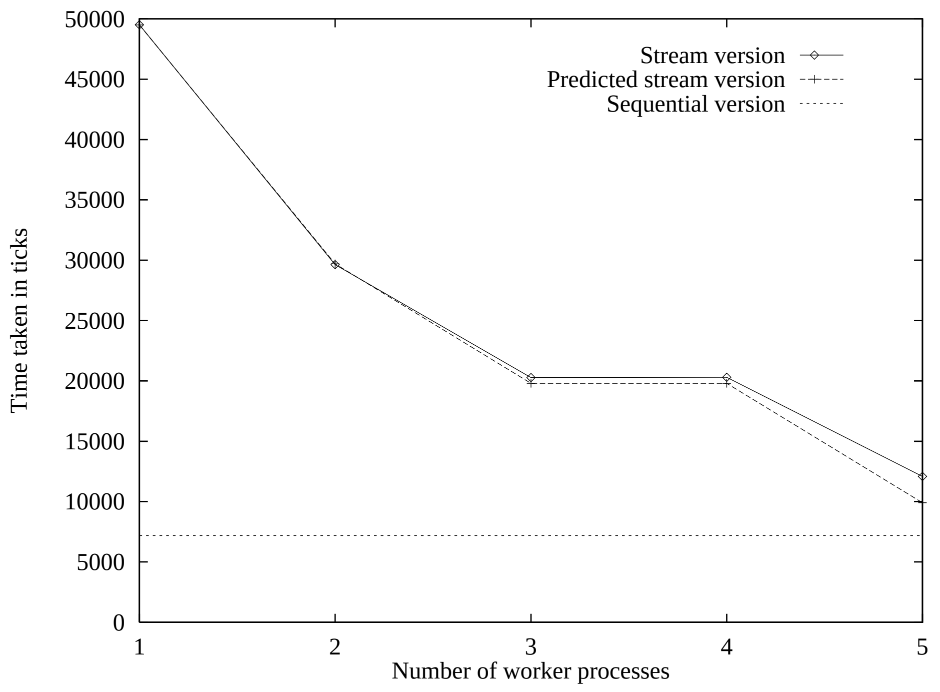


Figure 3.4: Execution time for the parallel composition of binary relations when using the stream approach.

always be five, as there are five tuples in tuple space R). The third column shows the maximum number of tuples from tuple space R a single worker process consumes. The fourth column shows the execution time for the program. The solution is parallel so the time taken depends on the worker process or processes which consume the most tuples from tuple space R. The fifth column shows the time taken for the worker processes to process a single tuple from tuple space R, and is calculated by dividing the execution time (shown in column four) by the number of tuples from tuple space R consumed by the worker process performing the most work (indicated in column three).

The time taken to process a single tuple (column 5) should remain constant as the number of worker processes is increased from one to five. The predicted results shown in Figure 3.4 are based on the execution time using one worker process which takes 9901 ticks per tuple from tuple space R. When five worker processes are used the time taken per tuple from tuple space R increases noticeably. This is because the underlying run-time system being used cannot service the requests fast enough so when there are five worker processes, the run-time system becomes a bottleneck.

Table 3.1 also shows why there is a plateau in the execution times when three and four worker processes are being used. In these cases the maximum number of tuples from tuple space R that a single worker process consumes is two tuples. The execution time of the program does not alter because the execution time is dependent on the time taken by the longest worker process. In both cases the time taken by the longest worker process is the same because they perform the same amount of work.

3.5.3 Experimental conclusions

The experimental results show the dilemma that a programmer faces when needing to perform a multiple `rd` with Linda. The binary semaphore method provides no speed increase as the number of worker processes used increases but is slightly faster than the sequential version when two or more worker processes are used. The stream method shows a speed up as the number of worker processes are increased but it takes a longer time to execute than the sequential version, even when five worker processes are being used.

In Chapter 4 a new primitive is introduced which solves the multiple `rd` problem. However, before considering the new primitive, some further observations are made about the multiple `rd` problem.

3.6 Coarsening the approach

Some experienced Linda programmers suggest that the multiple `rd` problem can *always* be overcome by using a coarser granularity of structures in tuple spaces. In the parallel composition of binary relations to create a coarser grained approach all the pairs stored in tuple space S are placed into a single tuple stored in tuple space S. Now, each of the worker processes removes a tuple from tuple space R and then uses the `rd` primitive to read the single tuple in tuple space S into a local data structure within itself. This local data structure is then used to determine which pairs match with the tuple chosen from tuple space R. A tuple space is a data structure in its own right and it seems wrong to have to use a special local data structure. However, such an approach appears attractive because of the apparent reduction in tuple communication this will entail. The code for the worker process using this coarser approach is shown in Program 3.4.

Figure 3.5 shows the experimental results when using this method, and shows the best case execution time for any other method, which is when a lock tuple is being used. These results show that the adoption of a coarser grained approach has not produced a speed increase over the best of the other methods. It should be noted that for some algorithms a coarser approach will lead to faster execution times. This is dependent upon both the algorithm, the amount of unnecessary "information" communicated and the characteristics of the implementation being used (the processor speed compared to the communication speed).

Since inception Linda has been used for *multiprocessing* (a *single* application consisting of several processes). More recently use has been made of Linda for *multiprogramming*[5] (*several* applications distributed over many processors)[Has94]. When Linda is used for *multiprocessing* it is natural to use closed implementations; the use of tuple spaces can be well defined and controlled, and in such a case the granularity of the program, and data structures can be regulated to gain

[5]It is also possible to refer to *multiprocessing* as *parallel processing* and *multiprogramming* as *distributed computing*.

Program 3.4 A worker process using a coarser data structure.

```
comp_worker := func(R,S,C);

  local my_val, pair_list, todo;

  my_val := lin(R,|[?int,?int]|);    -- Get the tuple from R
  pair_list := lrd(S,|[?tuple]|)(1); -- Get the single tuple

  todo := #pair_list;       -- Traverse the local structure
  while (todo > 0) do
    if (pair_list(todo)(1) = my_val(2)) then
      lout(C,[my_val(1),pair_list(todo)(2)]);
    end if;                  -- Produce the results if needed
    todo := todo - 1;
  end while;

  return ["TERMINATED"];
end func;
```

maximum performance. For example, how is a digitised image stored in a local data structure? The image contains a number of coordinates each with a pixel value associated. In traditional programming such a structure is stored as a two-dimensional array (if the language used supports two dimensional arrays). To retrieve a pixel from the array the pixels coordinates are used as an index into the array.

How would a digitised image be stored in a tuple space? There are several possibilities, each representing a different granularity of data structure. The finest grained representation possible is to use tuples of the form:

$$\langle \text{ x-coordinate }_{integer},\ \text{y-coordinate }_{integer},\ \text{pixel-value}_{integer}\rangle$$

where each tuple represents a single pixel in the image. A medium granularity approach is to use a tuple per row or column of the image:

$$\langle \text{ x-coordinate}_{integer},\ \text{pixel-value } [\text{ x-coordinate }]_{integer-array}\rangle$$

where $pixel-value[x-coordinate]_{integer}$ represents a one dimensional array of all pixels which reside on the column specified by $x-coordinate$. The coarsest data structure is to place the whole image in a single tuple:

$$\langle \text{ image }_{integer-array}\rangle.$$

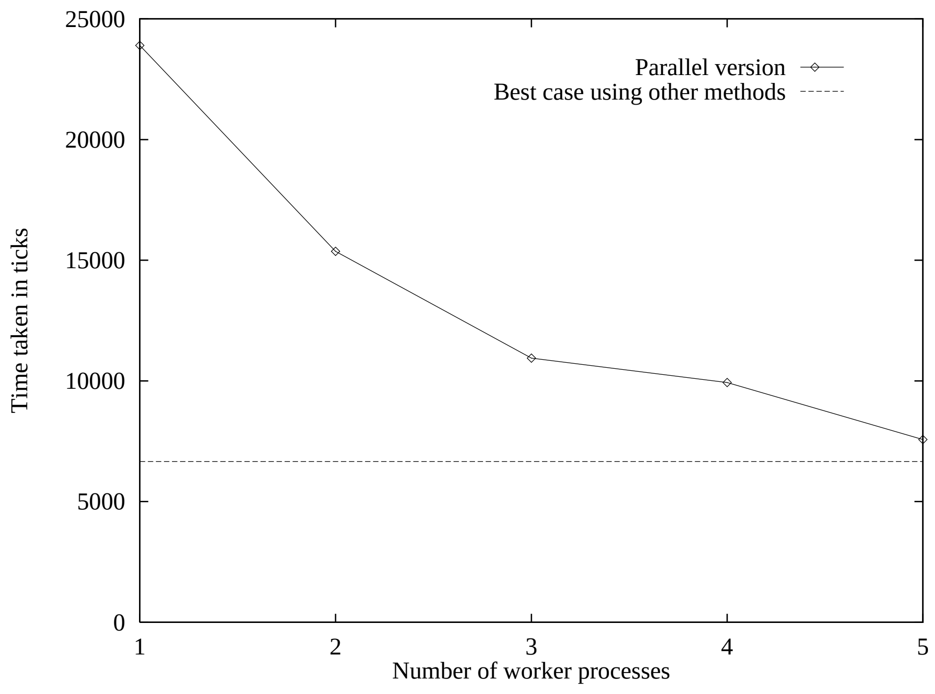


Figure 3.5: The worker process when adopting a coarser approach to the parallel composition of binary relations.

The granularity adopted is important. For example, consider a 1024x1024 image where an integer is represented as four bytes. Table 3.2 shows for each of the above representations the number of tuples required and the *minimum memory* required. The image will always require 4 Megabytes of memory, regardless of the granularity, because there are 1048576 pixels each requiring 4 bytes. The other fields which are also in the tuple occupy memory as well and this is shown in the *overheads* column. Not included in the overhead measurements are the hidden costs of other information that may be stored with each tuple.

Granularity of data structure in tuple space	No. of tuples	Memory required for		Total memory requirements
		image	overheads	
Fine	1048576	4 MBytes	8 MBytes	12 MBytes
Medium	1024	4 MBytes	4 KBytes	4 MBytes
Coarse	1	4 MBytes	0 Bytes	4 MBytes

Table 3.2: Comparison of the number of tuples and minimum memory usage for different granularities of tuple usage.

3.7 Conclusions

The Linda model has been in use for over ten years, so why has the multiple `rd` problem not been identified previously within the Linda model?

When considering data parallelism within the context of the Linda model, some researchers have alluded to the need for some operations that allow the parallel application of a function to all tuples that match a given template[AS91]. Anderson et al. when describing PLinda state that the motivation for such operations is:

> *Since many applications, particularly database ones, are expressed naturally in sets, we expect operations that iterate through sets of tuples to be useful. In vanilla Linda, a* `rd` *may repeatedly return the same tuple, even if several other tuples match.*

There is no discussion of why the `rd` primitive returning the same tuple is a problem, and the current implementations of PLinda do not support such operations.

Currently, Linda programmers either know that the data structure stored in tuple space is such that using the stream approach produces an acceptable performance, or they increase the coarseness of the tuple structure within the tuple space until the level of coarseness removes the multiple `rd` problem, as shown in the previous section with the composition of binary relations. Table 3.2 shows the use of coarser data structures is advantageous especially if memory usage is of primary concern. One of the strengths of Linda is the ability to perform coordination in a natural way, and often fine grained structures are more natural. The binary composition example used in this chapter seems natural using tuples of pairs. The argument as to whether or not the granularity of the data structures stored within tuple spaces overcomes the multiple `rd` problem is largely fruitless when considering *multi-programming*.

When the Linda model is used for *multi-programming*, controlling many aspects of coordination and data structure granularity becomes more complex. The different applications create tuple spaces and share tuple spaces created by other applications, thereby sharing information. Each application controls the granularity of the tuple structure within the tuple spaces it creates. If one application chooses to store an image as a tuple for each pixel then any other application that wishes to use that image tuple space has to adhere to the granularity set by the application which created the tuple space.

This chapter has described the multiple `rd` problem and through the use of an example has shown that the current methods for overcoming the multiple `rd` problem, using a binary semaphore and streams, are not acceptable. Therefore, Linda is unable to perform a multiple `rd` in a viable fashion. In the next chapter a new primitive for Linda is proposed called `copy-collect`. This primitive is used to overcome the multiple `rd` problem.

Chapter 4

Copy-collect: A new primitive for the Linda model

4.1 Introduction

In the last chapter the multiple `rd` problem was described. What makes the multiple `rd` problem frustrating is that it appears natural that several concurrent and non-destructive reads of a number of tuples should be possible. Within the Linda model there is no notion of synchronisation *between* primitives, thus two (or more) Linda primitives can be performed *concurrently*, and the first York Linda kernel (York Kernel I)[DWR95] supports concurrent primitive operations. If two `rd` primitives can be serviced concurrently it should be possible for many processes to perform a multiple `rd` concurrently.

In this chapter a new primitive called `copy-collect` is proposed. The informal semantics of the new primitive are described and it is shown how the primitive is used to overcome the multiple `rd` problem.

4.2 The `copy-collect` primitive

The `copy-collect` primitive is closely related to the `collect` primitive, so first the semantics for that primitive are considered. In Butcher et al.[BWA94] the authors state that the informal semantics of the `collect` primitive are:

> *`int collect(TS destination, <template>)`*
>
> *The (informal) semantics of* `collect` *is that it moves all the tuples which match the* `<template>` *into the tuple space* `destination`*, and returns the number of tuples collected.*
>
> *More formal (and full) semantics for this primitive are currently being investigated. However, for the purposes of this paper we need only one property: given a* ***stable***

> *tuple space (one in which no `ins` or `outs` are occurring), `collect` will remove **all** tuples matching the template. The intricacies of a formal semantics centre on the meaning of 'all tuples' in a non-deterministic active tuple space.*

The authors note that although not used in the paper the primitive could be extendible to allow two tuple spaces to be explicitly stated, a source and a destination tuple space. No more publications have been produced by the authors on further semantics of the `collect` primitive. The difference between the `collect` primitive and the `copy-collect` primitive is that the `copy-collect` primitive *copies* rather than *moves* tuples. However, the semantics given to the `collect` primitive appear rather unclear, particularly the use of the terms *stable* and *active* tuple spaces. Therefore the informal semantics of the `copy-collect` primitive described here will be more comprehensive in order to clarify the ambiguities[1].

n = copy-collect (ts1, ts2, template) This primitive *copies* tuples that match `template` from one specified tuple space (`ts1`) to another specified tuple space (`ts2`). A count of the number of tuples copied (`n`) is returned. Tuple space `ts1` is known as the source tuple space and tuple space `ts2` is known as the destination tuple space.

To determine how many tuples are copied a series of rules are used:

1. If a `copy-collect` primitive and no other Linda primitives are performed using the source tuple space concurrently, then *all* the tuples that match the template will be copied to the destination tuple space.
2. If a `copy-collect` primitive and a `rd` primitive are performed using the same source tuple space concurrently, and one or more tuples exist that can satisfy both templates, then *all* the matching tuples will be copied to the destination tuple space and the `rd` primitive will not block and return a matching tuple. If no matching tuples exist then the `copy-collect` primitive will return zero, and the `rd` primitive will block.
3. If a `copy-collect` primitive and another `copy-collect` primitive are performed using the same source tuple space concurrently, and one or more tuples exist that can satisfy both templates, then *all* the matching tuples will be copied to the destination tuple space for each of the `copy-collect` primitives. If there are no matching tuples then both primitives will return zero.
4. If a `copy-collect` primitive and an `out` primitive are performed concurrently, and the `out` primitive is placing a tuple into the source tuple space that matches the template used in the `copy-collect` primitive, then the result is a non-deterministic choice between copying the inserted tuple or not. All other matching tuples will be copied.

[1]It is proposed that the `collect` primitive uses similar semantics.

5. If a `copy-collect` primitive and an `in` primitive are performed using the source tuple space concurrently, and one or more tuples exist that can satisfy both templates, then the `copy-collect` primitive either copies all the tuples or all the tuples minus the *matched* tuple that the `in` primitive returns (the choice is non-deterministic).
6. If a `copy-collect` primitive and a `collect` primitive are performed concurrently using the same source tuple space, then the number of tuples copied is non-deterministic within the bounds of zero to the maximum number of tuples present that match the template. The number of tuples that the `collect` primitive will move will be the number of tuples present that match the template.

If any primitive occurs concurrently with a `copy-collect` primitive that does not use either a template which matches one or more tuples that the `copy-collect` primitive template matches, or the source tuple space, then there is no interference between them. The exception is when the primitive is either a `collect` primitive or a `copy-collect` primitive performed on the destination tuple space with a template that matches one or more of the tuples being copied. Then each tuple placed into the destination tuple space is non-deterministically copied or moved by the `collect` primitive or `copy-collect` primitive being performed on the destination tuple space. When a value is returned by a `copy-collect` primitive the copied tuples *are* present within the destination tuple space.

The `copy-collect` primitive will never live lock – it will always complete and return a value. Rule 4 states that if an `out` primitive occurs concurrently with a `copy-collect` primitive then the inserted tuple may or may not be included in the copied tuples. Is it possible for one process to perform many primitives concurrently with another process performing a `copy-collect` primitive? Within Linda there is no notion of time associated with a primitive. Therefore, with no loss of generality it can be assumed that all primitives take the same time. The maximum number of `out` primitives that can occur concurrently with a `copy-collect` primitive is the number of user processes minus one. Therefore, the `copy-collect` primitive will complete provided there are a finite number of processes. Pragmatically a `copy-collect` primitive may take longer than a single `out` primitive and therefore, several `out` primitives may occur concurrently with the `copy-collect` primitive. This in itself is not a problem because it is impossible for the process performing the `out` primitives or the process performing the `copy-collect` primitive to know that several `out` primitives from the same processes have occurred, however it is up to the implementor to ensure that the `copy-collect` primitive completes and does not live lock.

To clarify what the rules mean it has been suggested[Woo96] that the primitive order should be considered. This can be achieved by taking a trace of the primitives for a sequential Linda system, where the primitives cannot be serviced concurrently. In such a system an `in` primitive and a `rd` primitive occurring "concurrently" will in reality produce either the trace:

$$[...., in, rd, ...],$$

or

$$[...., rd, in, ...].$$

If the template used in both operations match a single tuple, then the first trace would represent the `in` primitive retrieving the tuple first, and the `rd` primitive blocking. The second trace would represent the `rd` primitive retrieving a copy of the tuple and then the `in` primitive removing the tuple. Both traces are acceptable, and the choice of which trace occurs is non-deterministic. This analogy covers any single tuple primitives (`in`, `out` and `rd`) occurring concurrently with a `copy-collect` primitive. If a `copy-collect` primitive and two `in` primitives occur "concurrently" where the template for each matches several tuples in the tuple space, the possible traces are:

$$[...., in, in, copy\text{-}collect, ...],$$

or

$$[...., in, copy\text{-}collect, in, ...],$$

or

$$[...., copy\text{-}collect, in, in...].$$

Assume that before the first of these primitives occur there are n tuples in the tuple space that match the template used in the `copy-collect` primitive. Then, in the first trace the `copy-collect` primitive will copy $n - 2$ tuples; in the second trace $n - 1$ tuples; and in the third trace n tuples.

The rules described use the ideas of non-determinism currently used within the Linda model (as described in Chapter 2). If a `rd` primitive and an `in` primitive occur concurrently, and they both use a template that can match the same tuple, then they non-deterministically compete for that tuple. The `in` primitive will always be satisfied because the `rd` primitive does not remove the tuple. However the `rd` primitive will either get a copy of the tuple or block. If a `copy-collect` primitive and an `in` primitive occur concurrently they compete for the tuple, if the `in` primitive acquires the tuple first, the `copy-collect` primitive does not copy it. This is described using the traces.

What happens if a `collect` primitive and a `copy-collect` primitive occur concurrently? Rule 6 indicates that the same non-deterministic competition for tuples occurs. Therefore, the `collect` primitive and the `copy-collect` primitive compete for each tuple that matches the template. The sequential traces force the primitives to either get all or none of the tuples because the traces can be either:

$$[...., collect, copy\text{-}collect, ...],$$

or

$$[...., copy\text{-}collect, collect, ...].$$

The choice as to which trace is produced is non-deterministic, but the competition for individual tuples no longer exists. In the first trace the `copy-collect` primitive will not copy any tuples, and in the second trace the `copy-collect` primitive will copy all the matching tuples. The rules appear to embody the true spirit of Linda. The trace semantics for single tuple space primitives provide the same semantics. However, the trace semantics when applied to the bulk tuple primitives interaction (the `collect` and `copy-collect` primitives) provide a subset of the possible results. The primitives could be implemented using such semantics and be valid because the two outcomes described are achievable using the rules given.

Because of the relationship between the `collect` primitive and the `copy-collect` primitive the informal semantics of the `collect` primitive used within this dissertation are now considered similar to the informal semantics of the `copy-collect` primitive, except instead of *copying* the tuples, they are *moved*.

When the primitive returns the count of the number of tuples copied, all copied tuples are present within the destination tuple space. Therefore, the code segment shown in Program 4.1 will always result in tuple spaces `ts1` and `ts2` containing the same number of tuples that match the template $\langle|\square_{integer}|\rangle$. This can be seen as the extension of `out` ordering described in Chapter 2 to cover both the `copy-collect` primitive and the `collect` primitive.

Program 4.1 Demonstration of completion of `copy-collect`.

```
void demo(TS source)
{
TS ts1, ts2;

ts1 = tsc();                      -- Create two tuple spaces
ts2 = tsc();
lcopycollect(source, ts1, ?int); -- copy source -> ts1
lcopycollect(ts1, ts2, ?int);    -- copy ts1 -> ts2
}
```

Finally, on a pragmatic note in Chapter 2 a detailed description of global synchronisation and `out` ordering was given and these are now both considered in the context of the `copy-collect` primitive. For many uses of the `copy-collect` primitive `out` ordering is required since, without `out` ordering it is difficult for one process to indicate to another that a set of tuples exist. For example, consider Program 4.2. Assuming that both functions are passed the same tuple space and no other processes have access to that tuple space, then if `out` ordering is not guaranteed the value of `n` in the function `master` would be non-deterministic between 0 and 100. If `out` ordering is guaranteed the function `master` knows that when the tuple ⟨“COMPLETE”⟩ appears in the tuple space, the other tuples the function `worker` produces are also present in the tuple space. If the function `master` is not aware of how many tuples are produced, the `copy-collect` will gather *all* tuples that were produced.

Program 4.2 `Out` ordering and `copy-collect`.

```
worker := func(ts1);

  local x;

  for x in [1..100] do
    lout(ts1, [x]);
  end for;

  lout(ts1,["COMPLETE"]);

end func;

master := func(ts);

  local n;

  lin(ts,|["COMPLETE"]|);
  n := lcopycollect(ts, my_ts, |[?int]|);

end func;
```

This brings us to the second pragmatic issue, which is whether the `copy-collect` primitive requires a global synchronisation as described in Chapter 2. In Chapter 5 an implementation of the `copy-collect` and `collect` primitives will be presented which does not use any global synchronisation of tuples spaces in implementing the primitive. A global synchronisation could be considered as necessary because, as with the `inp` and `rdp` primitives, different sections of a tuple space could be checked at different times. Therefore, once one section of the tuple space has been checked, a matching tuple is inserted before another section of the tuple space is searched. Under the rules given, if an `out` primitive occurs concurrently with a `copy-collect` primitive then whether this is copied is *non-deterministic*. Therefore, if tuples are missed because they are being inserted after that part of the tuple space has been checked it does not matter. Therefore, a global synchronisation is not required.

4.3 Using `copy-collect` to solve the multiple `rd` problem

The `copy-collect` primitive provides the functionality to overcome the multiple `rd` problem. It allows several processes to concurrently *copy* the tuples they require. Consider again the example where a tuple space is used to store a number of tuples containing peoples' names. All the people with the same surname are required. Using the `copy-collect` primitive it is possible to extract into a separate tuple space all the tuples with the same surname. Hence, to extract all people with the surname *Rowstron,* a `copy-collect` primitive is performed using the template $\langle | \square_{string}, \text{“Rowstron”}_{string} | \rangle$.

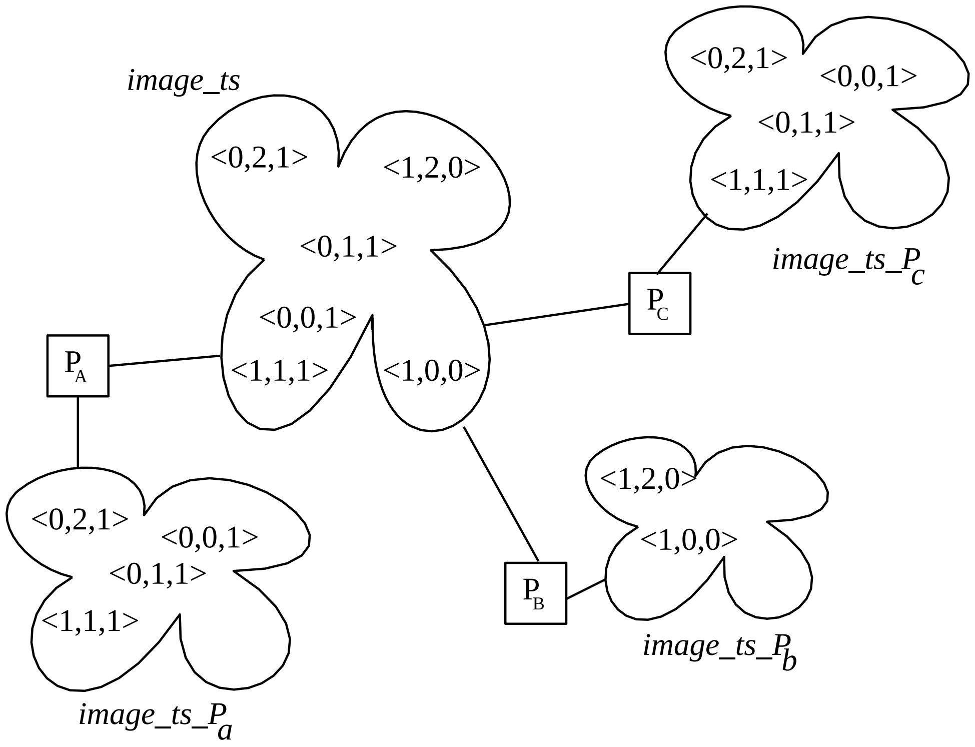


Figure 4.1: Using the `copy-collect` primitive to solve the multiple `rd` problem.

Figure 4.1 shows the use of the `copy-collect` primitive to overcome the multiple `rd` problem. The shared tuple space is called *image_ts* and the processes are called P$_A$, P$_B$ and P$_C$. Each process creates a tuple space (*image_ts_P*$_x$) to which only they have access. They then perform a `copy-collect` primitive using *image_ts* as the source tuple space and *image_ts_P*$_x$ as the destination tuple space. Once the tuples have been copied to the destination tuple space each process can retrieve each tuple in turn using the `in` primitive. Each process knows the number of tuples in the tuple space because the `copy-collect` primitive returns a count of the number copied, and the process can, by destructively removing the tuples, ensure that every tuple is retrieved once and only once without effecting the other processes. Because the tuple space cannot be accessed by any other process the destructive removal of tuples does not affect any other process. The example in Figure 4.1 has two of the processes requiring all the pixels which are set (the third field set to one) and the other process requiring all the pixels which are not set (the third field set to zero).

Program 4.3 The worker process using the `copy_collect` primitive.

```
comp_worker := func(R,S,C);

  local my_val, my_ts, todo, my_comb;

  my_ts := NewBag;

  my_val := lin(R,|[?int,?int]|);  -- Get the tuple from R
  todo := lcopycollect(S,my_ts,|[my_val(2),?int]|);
  while (todo > 0) do              -- Process all matching tuples
    todo := todo - 1;
    my_comb := lin(my_ts,|[my_val(2),?int]|);
    lout(C,[my_val(1),my_comb(2)]);
  end while;

  return ["TERMINATED"];
end func;
```

The parallel composition of two binary relations used as an example in the previous chapter, is now used to show how the `copy-collect` primitive overcomes the multiple `rd` problem in more detail. The worker process using the `copy-collect` method is shown in Program 4.3. The general structure of the approach is the same as in the solutions given in the previous chapter, with each worker process removing a tuple from tuple space R. The worker process then creates a template using the retrieved tuple for use with the `copy-collect` primitive. The second field of the retrieved tuple from tuple space R is used as the first field of the template and the second field of the template is left as a formal of type integer. A `copy-collect` primitive is then performed which copies the tuples from tuple space S to a tuple space which the worker process creates. The count returned by the `copy-collect` primitive is then used to control an iterative loop which destructively reads the tuples from the tuple space and creates the result tuples in tuple space C.

4.4 Experimental results

The experimental results presented in this section, as in the previous chapter, are obtained using ISETL-Linda running on a transputer based Meiko CS-1 parallel computer using the York Kernel I. The `copy-collect` primitive was added to the run-time system by Douglas[DWR95], and the implementation is naive (the approach adopted is described in Chapter 5). As in the previous chapter the worker process was altered to enable the number of tuples from tuple space R to be consumed to be specified.

For the experimental results the cardinality of the binary relation R (stored in tuple space R) is set to five and the cardinality of binary relation S (stored in tuple space S) is set to 50. For every pair in R there are four pairs in S that match, therefore the cardinality of the composition (stored in tuple space C) is 20. This is the same configuration used in the experimental results presented in the previous chapter. Figure 4.2 shows the execution times for the worker processes for computing the composition of two binary relations using `copy-collect`, the best execution time of the other two methods (using a lock tuple with four worker processes), and the expected execution time for the `copy-collect` method is shown.

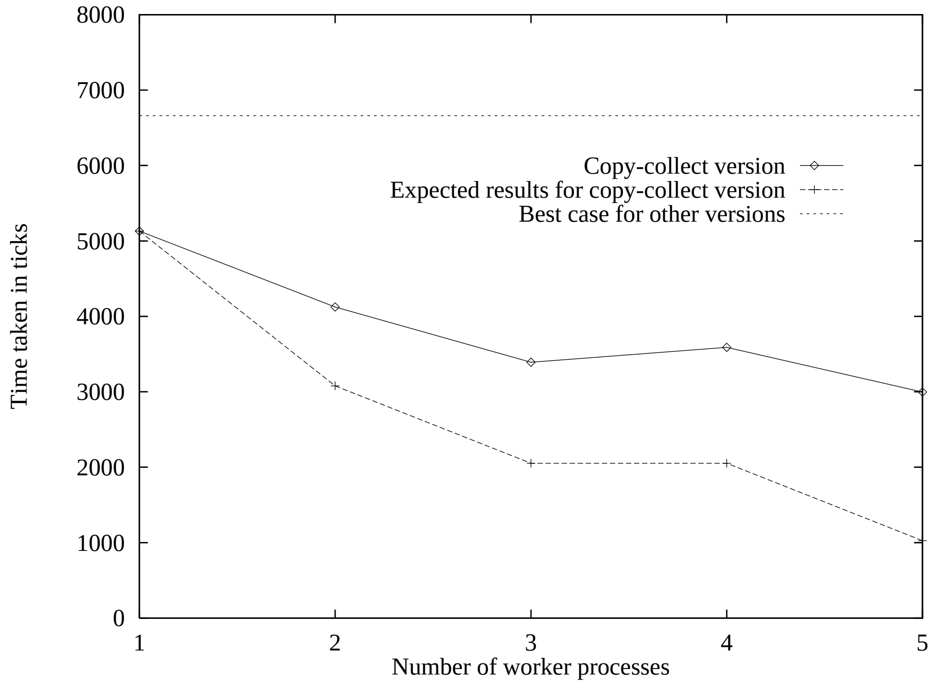


Figure 4.2: Execution time for the parallel composition of binary relations when using the new `copy-collect` primitive.

Figure 4.2 shows some interesting results. Firstly, the execution time using a single process is less than the best time achievable using any number of worker processes for any of the other methods. This is because the *number* of tuple space operations that have to be performed is significantly smaller. This is expanded upon in Section 4.5. The time taken when three and four worker processes are used is similar, for the same reason that in the stream method the time taken for three and four worker processes is similar. The expected results are calculated using the time taken for the single worker process and dividing it by the number of tuples in tuple space R, which is five. As with the stream method, because there is parallel access it is expected that with five worker processes, each processes a single tuple from tuple space R, and therefore, the time each worker takes should be one fifth of the time the single worker process takes. The actual execution times are greater than predicted because the underlying run-time system is "saturating"; in other words the run-time system is receiving more requests than it can process, so becomes a bottleneck. However the performance even with the run-time system saturating is twice as fast as the best time produced by the semaphore or the stream methods, and a speed increase is observed as more worker processes are used.

This shows how the new primitive can be used. The same approach can be used whenever a multiple `rd` is required. The `copy-collect` primitive solves the *multiple* `rd` *problem*. A worker process creates a copy in a tuple space of the tuples that the worker process requires using the `copy_collect` primitive, and then destructively reads them from that tuple space. For example, given a tuple space containing an image with each pixel a separate tuple (`[x_coordinate, y_coordinate, value]`) the command: `copy_collect(image_ts, local_ts, |[?int, ?int, 1]|)` copies all the tuples with a pixel value of one into the local tuple space. Given the tuple space containing tuples representing first names and surnames. Each process would use the `copy-collect` primitive with the template $\langle | \Box_{string}, \text{“Rowstron”}_{\;string} | \rangle$, to copy all the tuples with a surname of "Rowstron" to a separate tuple space, where they can be destructively processed.

So far with all the experimental results the execution time for a specific cardinality of binary relations S and R have been considered. Now the effect of making more tuples in tuple space S match each tuple in tuple space R is considered. For this the cardinality of tuple space R is again fixed at five. Figure 4.3 shows the execution times for five worker processes when the number of tuples in the tuple space S that match *each* tuple in tuple space R is increased from one to 50 (the cardinality of tuple space S is 50).

As the number of tuples in tuple space S that each tuple in tuple space R matches increases there will be an increase in the computation time within each worker process associated with the calculation and placement of the result tuples into tuple space C. Although it might be expected that the stream method should take a constant time because all tuples in tuple space S are always read by every worker process, the actual time increases slightly. This increase is attributable to the extra computation that the worker processes perform. The time taken for the semaphore method increases uniformly with the addition of the extra tuples in tuple space S that match each tuple in tuple space R. The reason why the execution time increases at a greater rate than the other methods, is that the other methods are parallel. So when the number of tuples that match each element in tuple space R increases by one, each of the five worker processes process one more tuple. If the method is parallel then this is performed concurrently. Because the semaphore method is sequential the five tuples are processed sequentially leading to an increase in the execution time which is five times greater than for the parallel methods. The execution time for the `copy-collect` method increases as the number of matching tuples in tuple space S increases. As more tuples match there is extra computation costs associated with each extra tuple processed by the worker processes and

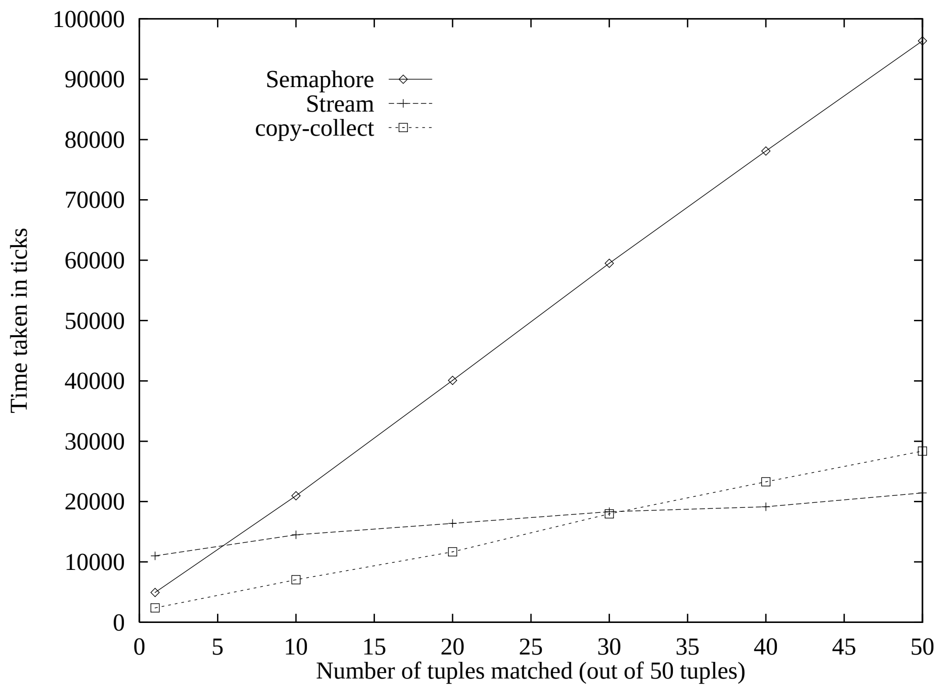


Figure 4.3: Execution time for the parallel composition versus the number of pairs in tuple space S which each pair in tuple space R matches.

there is extra communication because the number of tuple space operations is proportional to the number of tuples in tuple space S that match each tuple in tuple space R. Within Section 4.5 why the stream method performs better than the `copy-collect` when 33 or more tuples out of the 50 are matched is considered.

4.5 Modelling the performance

How the `copy-collect` primitive solves the multiple `rd` problem has been shown by using a number of experiments. To evaluate the three methods (semaphore, streams and the `copy-collect` primitive) in a more general way, a simple model of performance is produced for each of the methods. This allows the performance of each of the methods to be evaluated using arbitrary numbers of processes, tuples and tuples that match a template.

Let there be a tuple space T containing N tuples; a template t with n of the tuples in T matching this template; and P processes needing to perform a multiple `rd` concurrently. Initially, assume that $P = 1$. How many and which Linda primitives are required in order for the process to read all the tuples n and leave tuple space T in its original state?

Stream method If there are N tuples in T then each tuple will be read once using a `rd` primitive.

Therefore, in the stream method the number of Linda primitives required is:

$$\text{No. of Linda primitives} = N \times \texttt{rd}. \tag{4.1}$$

Binary semaphore method There are two approaches to implementing the semaphore method, one uses the `collect` primitive and the other uses the `inp` primitive. For the `collect` approach the number of Linda primitives required is: the number of primitives required for obtaining and replacing of the lock tuple (a single `in` and `out`); the primitive to move the matching tuples to another tuple space (a `collect`); the number of primitives to remove the tuples from that tuple space (n `in`); and the number of primitives to replace them in T (n `out`). Therefore, the number of Linda primitives required is:

$$\text{No. of Linda primitives} = (n+1) \times \texttt{out} + (n+1) \times \texttt{in} + \texttt{collect}. \tag{4.2}$$

For the `inp` approach the number of Linda primitives required is: the number of primitives required for the semaphore access (an `in` and an `out`); the number of primitives to move the matching tuples to another tuple space (n `inp` + n `out` + `inp` which fails); the number of primitives to remove the tuples from that tuple space (n `in`); and the number of primitives to replace them in T (n `out`). Therefore, the number of Linda primitives required is:

$$\text{No. of Linda primitives} = (2n+1) \times \texttt{out} + (n+1) \times \texttt{in} + (n+1) \times \texttt{inp}. \tag{4.3}$$

copy-collect method The number of Linda primitives required is: the primitive to copy the tuples to a tuple space (a `copy-collect`); and the number of primitives to remove the tuples from the tuple space (n `in`). Therefore, the number of Linda primitives required is:

$$\text{No. of Linda primitives} = \texttt{copy-collect} + n \times \texttt{in} \tag{4.4}$$

When $P > 1$ the number of primitives required *in total* will also rise proportionally with the number of processes. Each process has to perform the same number of Linda primitives to access the same tuples in the tuple space. Therefore, each of the equations given above must be multiplied by the number of processes performing the operation (assuming that all the processes wish to access the same number of tuples). The absolute primitive count is defined as the number of primitives a set of processes need in order to perform a multiple `rd`.

Assuming 100 tuples in a tuple space ($N = 100$), Figure 4.4 shows the absolute primitive count for the stream method, Figure 4.5 shows the absolute primitive count for the semaphore method, and Figure 4.6 shows the absolute primitive count for the `copy-collect` method. Figure 4.7 shows the absolute primitive counts for both the semaphore and stream methods on the same graph. In all the figures the label A stands for the absolute primitive count, n and P are as defined above,

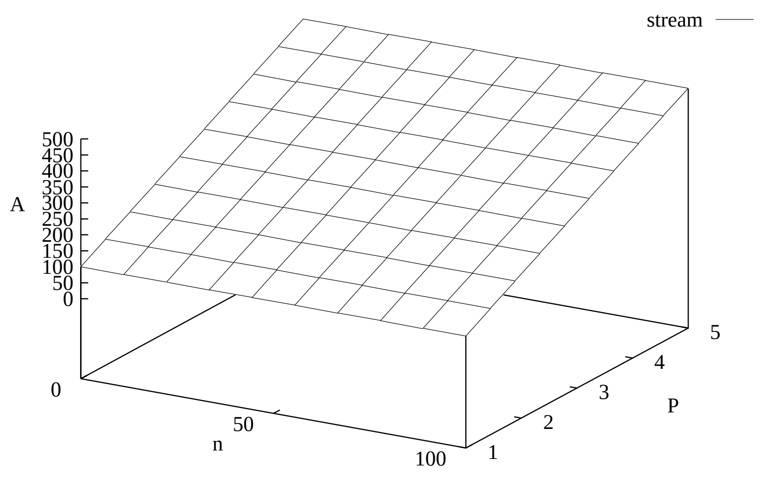


Figure 4.4: Absolute primitive count for the stream method for the multiple `rd` problem.

where the range of n is 0 to N and the range of P is 1 to 5, representing 1 to 5 processes performing the multiple `rd` in parallel. Analysis shows that the semaphore method requires more primitives than the stream method once 49% of the tuples are required (ie. n is 49% of N, in this instance $n = 49$).

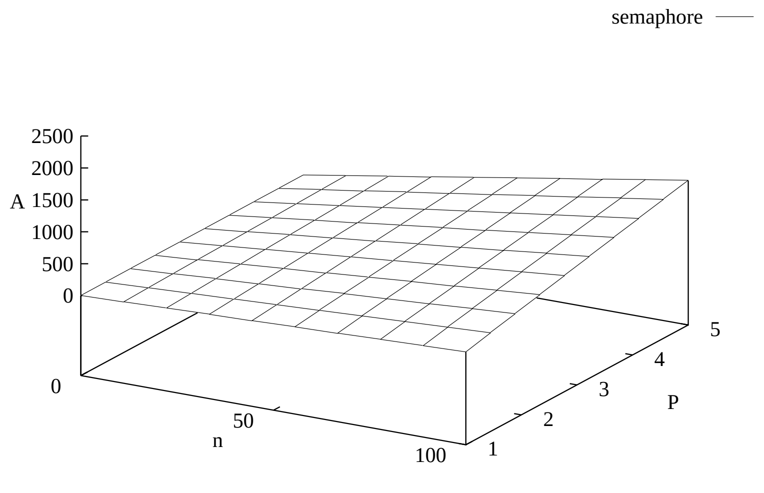


Figure 4.5: Absolute primitive count for the semaphore method for the multiple `rd` problem.

Figure 4.6 shows the absolute primitive count for the `copy-collect` method. Analysis shows that the `copy-collect` method will always have the lowest absolute primitive count

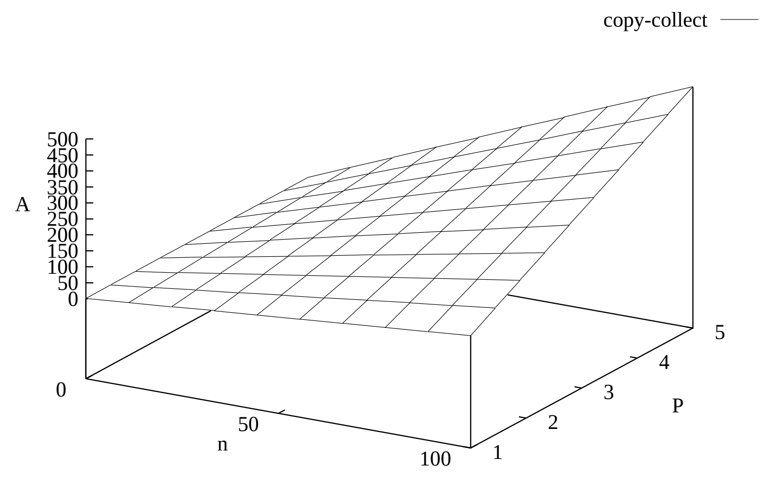


Figure 4.6: Absolute primitive count for the `copy-collect` method for the multiple `rd` problem.

except when $n = N - 1$ when the stream method will have a lower absolute primitive count.

Does this imply that the `copy-collect` method is the best? The use of the absolute primitive count is not the best measure, because Linda makes no assumptions about primitives being serviced sequentially. In an ideal Linda system if two processes perform a `rd` concurrently these would be serviced concurrently. Therefore, as well as considering the absolute primitive counts, a count of the number of primitives that *cannot* be performed concurrently should be considered. The primitive counts for both the stream and `copy-collect` versions are independent of the number of processes performing the multiple `rd`. The stream method uses only `rd` primitives and so they can all be serviced concurrently with other `rd` primitives. The `copy-collect` method also uses primitives that can be serviced concurrently. The primitive count for the semaphore method will rise proportionally with the number of processes as each will perform an `in` primitive on the lock tuple, and then $P - 1$ will block. In the case of the semaphore method when using the `collect` primitive the number of primitives that cannot be performed concurrently is:

$$\text{No. of Linda primitives} = ((n + 1) \times \texttt{out} + (n + 1) \times \texttt{in} + \texttt{collect}) \times P. \qquad (4.5)$$

When using the `inp` primitive the number of primitives that cannot be performed concurrently is:

$$\text{No. of Linda primitives} = ((2n + 1) \times \texttt{out} + (n + 1) \times \texttt{in} + (n + 1) \times \texttt{inp}) \times P. \qquad (4.6)$$

Figures 4.8 and 4.9 show how the count of the number of Linda primitives that cannot be performed concurrently (L) varies as the value of P and n are varied. In both cases the number

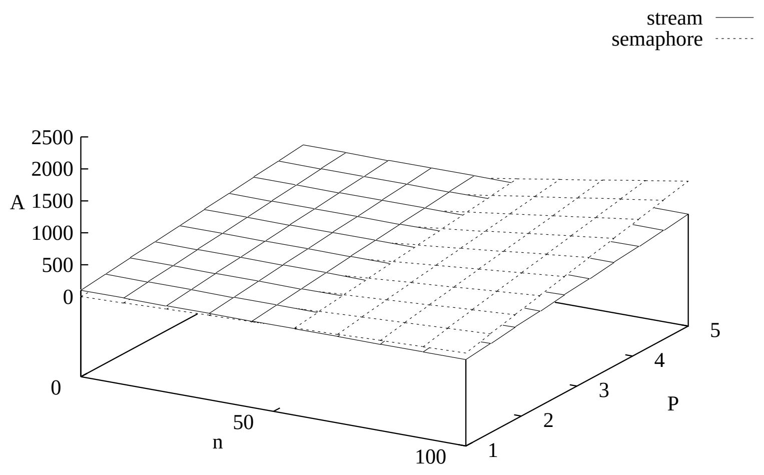


Figure 4.7: Comparison of absolute primitive counts for the stream and semaphore methods.

of tuples in T, (denoted as N) is fixed at 100. Figure 4.8 shows how the stream and semaphore methods perform, and Figure 4.9 shows how the stream and semaphore (`collect` approach) methods perform. Figure 4.8 has a Z axis representing the number of processes performing the multiple `rd`

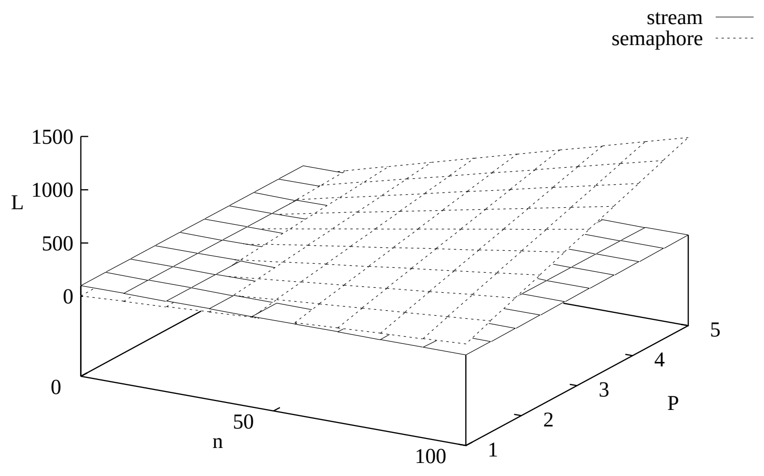


Figure 4.8: Comparison of the non-concurrent primitive counts for the stream and semaphore methods to the multiple `rd` problem.

because the number of primitives that cannot be performed concurrently in the semaphore method depends on the number of processes. Because both the stream and `copy-collect` methods are independent of the number of processes performing the multiple `rd`, this information is not required on the graph (represented by the Z axis in Figure 4.8).

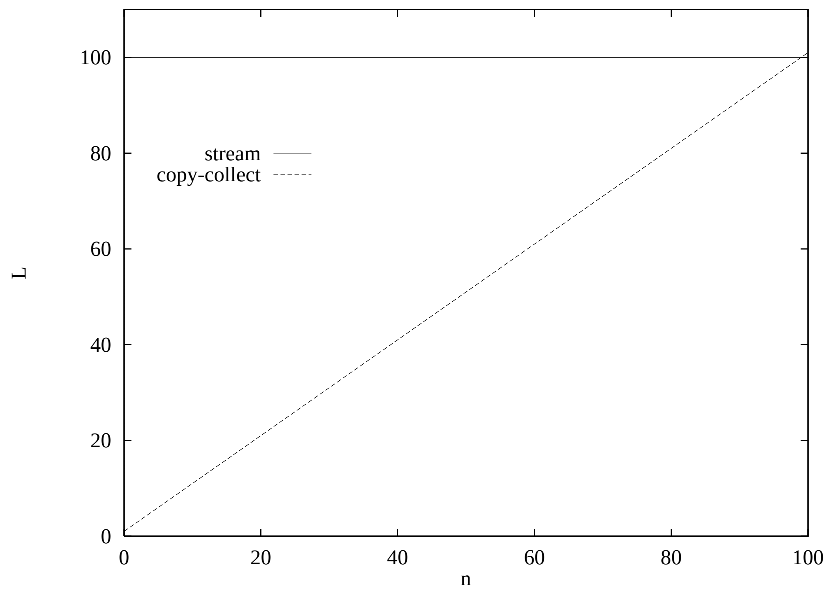


Figure 4.9: Comparison of the non-concurrent primitive counts for `copy-collect` and stream methods to the multiple `rd` problem.

Simple analysis indicates that when one or more processes wish to perform a multiple `rd`, the `copy-collect` primitive method utilises less primitives if either the absolute primitive count or the number of primitives that cannot be performed concurrently is considered, except when all the tuples in a tuple space match. In this instance the stream method is better regardless of which count is considered. The use of primitive counts assumes that all the primitives take the same length of time. Pragmatically, this is not the case, but they provide an indication of the performance. Given a specific Linda implementation the performance of the methods can be compared using both the primitive counts. To show this the Meiko CS-1 ISETL-Linda implementation is now considered. (See Appendix A for the characteristics of the York Kernel I which the ISETL-Linda implementation uses).

For the implementation on the Meiko CS-1 the time cost of performing all the Linda operations is described in Table 4.1. These timings are calculated using tuple spaces containing 100 tuples and 10000 tuples. As the number of tuples within a single tuple space increase so do the costs associated with matching the tuples. Therefore, the time that an `in` primitive and a `rd` primitive take is dependent on the template used and the number of tuples in the tuple space. In the worst case all the tuples are matched, and in the best case the first tuple found matches the template. When 100 tuples are used, the best case and worst case for an `in` primitive yield similar results. When there are 10000 tuples the difference between the execution times is significant.

Primitive	Time in ticks		
	100 tuples	10000 tuples	
	Average	Best case	Worst case
`out`	71	74	
`in`	130	124	1192
`rd`	128	121	1183
`collect`	458	17675	
`copy-collect`	726	44290	

Table 4.1: The time taken to perform the Linda operations using ISETL-Linda on a Meiko CS-1.

The primitive count models described in conjunction with the primitive timings given in Table 4.1 for 100 tuples are used to estimate the performance of the different methods of the parallel composition of binary relations on the Meiko CS-1. The timings for 100 tuples are used because the number of tuples in a single tuple space on which primitives are performed does not rise above 100 in the parallel composition of binary relations. The absolute primitive count represents an upper bound of the performance (worst case) and the primitive count for the number of primitives that cannot be performed concurrently represents a lower bound for the performance (best case). This is because the absolute primitive count represents the sequential servicing of the primitives, whilst the primitive count for the number of primitives that cannot be performed concurrently represents all the primitives being serviced in parallel. A distributed implementation will service some primitives in parallel but others will be serviced sequentially.

Using the models described above and *including* the cost of performing a single `out` primitive for every tuple to tuple space C when a match is found, it is possible to predict the time taken *for the synchronisation* within the parallel composition implementations. The expected time spent performing synchronisations in the best case is shown in Figure 4.10 and in the worst case is shown in Figure 4.11. These are created assuming the same program characteristics as used to obtain the results in Figure 4.3, namely 5 worker processes, with tuple space R having a cardinality of 5, tuple space S having a cardinality of 50, and with n representing the number of tuples each tuple in tuple space R matches in tuple space S.

If these expected communication timings are compared with the experimental results used in Figure 4.3 there are a number of points that should be noted. The actual timings for the semaphore approach are greater than the estimated times. This is because the actual timings *include* the time taken for the computation as well as the coordination. The calculated timings represent only the coordination time. The actual execution times for both the `copy-collect` method and the stream method lie in between the worst and best case times. As both of these methods perform the computation in parallel it is expected that the effect of computation on the actual results is less than for the semaphore approach, where the computation is not performed concurrently. In the best case, estimated coordination times of the stream method become the fastest when approximately

43 tuples from tuple space S match each tuple from tuple space R. In the worst case estimated coordination times of the stream method are always slower. In the achieved results the stream method becomes the fastest method when approximately 33 tuples match. This would indicate that the run-time system is performing most of the tuple operations concurrently, considering the computation that the actual timings include.

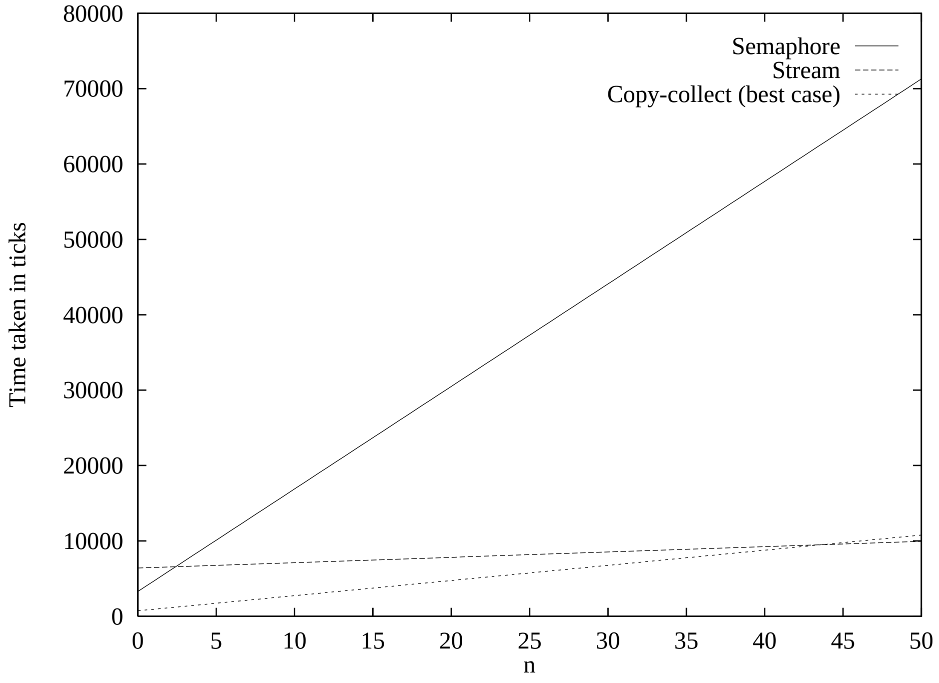


Figure 4.10: Best case expected communication overheads for the three different methods to the multiple `rd` problem.

The results and the primitive counts represent a fair way of comparing the best case and worse case performance of the three methods for overcoming the multiple `rd` problem in general, and for a particular implementation. The actual performance depends largely on the implementation being used. In most implementations the cost of performing an `in` primitive and a `rd` primitive should be comparable. The cost of performing the `collect` and `copy-collect` primitives *should* be comparable to the cost of performing an `in` primitive that blocks plus the time overheads to either copy the tuples or attach them to a different tuple space. Unless the `copy-collect` primitive is implemented badly, the performance, in general, should be better than the semaphore method and the stream method.

In the next chapter the implementation of the `copy-collect` primitive is considered, and a more efficient implementation approach is suggested, which makes the `copy-collect` primitive far more efficient. Before considering the implementation of the `copy-collect` primitive, the primitive is compared to other proposed primitives, and other uses of the `copy-collect` primitive are considered.

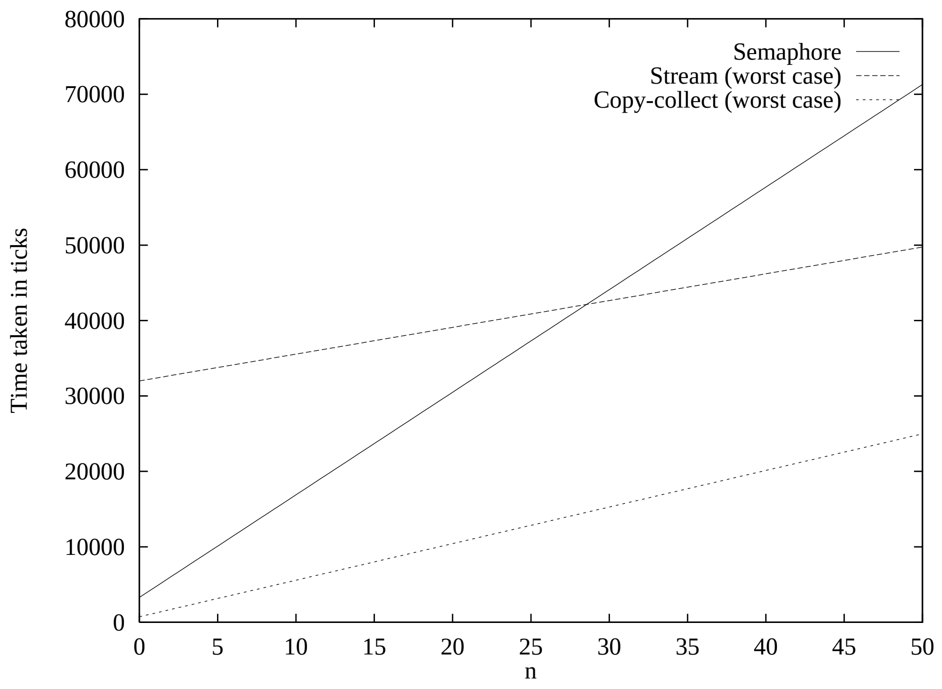


Figure 4.11: Worst case expected communication overheads for the three different methods to the multiple `rd` problem.

4.6 Comparison with other similar proposals

In Chapter 2 a number of proposed primitives were described, including a `copy_contents` primitive, a bounded multiple `rd` primitive and a `rd()all` primitive. These primitives can all potentially be used to overcome the multiple `rd` problem, and each of these primitives are now considered in detailed. A description of their function is given in Section 2.3.2.

copy_contents [NS93, NS94] This primitive is the closest to the `copy-collect` primitive. The `copy_contents` primitive copies *all* tuples in a tuple space and does not return a count of the number of tuples copied. This primitive has two disadvantages when compared to the `copy-collect` primitive.

- The lack of "global information".

  The `copy-collect` primitive returns a count of the number of tuples copied. This information can be used to control how the tuples are processed. The information provides an indication to the process that performed the operation of the number of tuples, and this process can then control how the tuples are consumed. For example, if there are many tuples then several processes may be created to perform the processing, and if there are only a few tuples then no other processes may need to be spawned.

Because both the `collect` and `copy-collect` primitives return counts, they can replace the primitives of `inp` and `rdp` respectively. The `copy_contents` and `move_contents` primitives do not replace the `inp` and `rdp` primitives. Indeed, to overcome the multiple `rd` problem using the `copy_contents` primitive an `inp` primitive would still have to be supported. This is because the processes performing the multiple `rd` *need* to know how many tuples should be consumed. For example, how would all the people with the surname "Rowstron" be retrieved from a tuple space containing tuples representing people's names (as used in Chapter 3)? A process would duplicate the entire tuple space using the `copy_contents` primitive, then *how many* `in` primitives would the process perform to retrieve the correct tuples (containing the surname "Rowstron")? If too many were performed the user process will block forever on an `in` primitive, and if too few were performed some of the people whose surname is "Rowstron" would be missed. Extra information could be placed into the tuple space, such as a tuple containing a counter and the name "Rowstron", however, such an approach can be seen to rapidly become unsatisfactory. Therefore, the `inp` primitive is required, to be used to destructively remove the tuples and return a value indicating when no more tuples are available.

- The lack of selectivity of which tuples are duplicated.

  The duplication of *all* tuples within a tuple space is unnecessary in order to overcome the multiple `rd` problem. If entire tuple spaces are duplicated it leads to large tuple spaces, which need to be searched, manipulated, and stored. In experience with using the `copy-collect` primitive an entire tuple space is duplicated rarely, normally selective subsets of tuples are required from a tuple space. Where an entire tuple space was required, there was a template that matched all the tuples in the tuple space. Obviously, there is no guarantee that tuples which match different templates will need to be duplicated, but multiple `copy-collect` primitives could be used if required.

The `copy-collect` primitive provides the ability to duplicate *only* the tuples needed, and provides enough information, in the form of a count of the number of duplicated tuples, to make the post-processing of the duplicated tuples easy. Therefore, the `copy_contents` primitive is not considered to have sufficient flexibility to overcome the multiple `rd` problem in a satisfactory manner.

bounded multiple `rd` [Kie96] This primitive is again similar to the `copy-collect` primitive. Kielmann[Kie96] has proposed a Linda system called Objective Linda for use in open systems. When describing the background information to justify the primitive which Objective Linda supports Kielmann[Kie96] states:

> *"Another approach is presented in [BWA94] and introduces a collect operation which atomically returns all tuples matching a given template in a certain tuple*

> *space. This approach allows to select multiple objects to be consumed, but in the case of a RM-ODP trader, unrestrictedly returning the complete list of service providers might still be too much (eg. with respect to memory size) or at least too inefficient.*
>
> *There is also a demand for an in operation which atomically removes several objects* [tuples] *from an object space* [tuple space]."

There are a number of observations that should be made. The `collect` primitive returns *only* a count of the number of tuples moved *not* the tuples themselves, and the `copy-collect` primitive returns *only* a count of the number of tuples copied. The underlying run-time system controls the placement of tuples, and these *need not* either individually or as a collective, be moved to the processor on which the process that performs the `collect` primitive resides.

Kielmann then continues to propose the bounded primitives as described in Chapter 2. The bound multiple `rd` primitive copies a number of tuples (objects) from a tuple space (object space) to a local data structure, called a multi-set. The maximum and minimum number of tuples to be copied can be specified, as can a timeout for how long the primitive should block. Kielmann[Kie96] states:

> *"Here,a Multi-set is a simple container type with the operations put and get and the predicate nbr_items denoting the number of items stored inside."*

The bounded `rd` primitive can be used to overcome the multiple `rd` problem. However, there are a number of observations that make such a primitive less attractive than the `copy-collect` primitive:

- Firstly, when being used to overcome the multiple `rd` problem the primitive would not be used with bounds, all tuples that match the template are required. Therefore, the statement that: "complete list of service providers might still be too much (eg. with respect to memory size) or at least too inefficient", applies to the bounded `rd` primitive. The management of the storage of tuples with the `copy-collect` primitive is left to the underlying run-time system.
- Secondly, the primitive appears unnatural, particularly in its use of a multi-set. A multi-set is a tuple space. Therefore, it would be more logical to say that the bounded `rd` primitive returns a tuple space, with the copied tuples present within that tuple space. Then, a simple abstraction is to specify the tuple space into which the copied tuples are placed, which is *very* similar to the original proposal for the `collect` primitive[BWA94], where only the destination tuple space is specified (the source is assumed to be the global tuple space).

As the distinction is made between tuple spaces and the returned multi-set, a number of operations which can be performed on multi-sets have to be provided, and Kielmann proposed the operations: *put*, *get* and *nbr_items*. The use of the operation *nbr_items* allows the number of copied tuples to be calculated for use within the process to ensure all the tuples are found when being removed from the multi-set. The *put* and *get* operations would map onto the Linda `in` (or `inp`) and `out` primitives.

The use of a separate data structure, the multi-set, is not necessary. The same operations can be performed upon a tuple space as a multi-set, except for the counting of the number of tuples within a tuple space. The `copy-collect` primitive provides this information. Hence the `copy-collect` primitive appears more natural and a closer fit with Linda than a bounded `rd` primitive. *If* there is a need for a bounded primitive that copies a minimum or maximum number of tuples then the `copy-collect` primitive could potentially be extended to support this.

rd(template)all(function) [AS91] This primitive is another primitive which is similar to the `copy-collect` primitive[2]. This primitive is interesting because computation is combined with a primitive. The function is applied to each of the tuples that is matched by the template provided. The description given is an overview of the primitive rather than a detailed description. The primitives are stated not to be atomic, and therefore the set of tuples that are being matched can potentially change. Anderson[AS91] states that a "snapshot" semantics were not used because of the implementation difficulties such semantics pose. There are a number of questions about the semantics of the `rd()all` primitive and its relations:

- Is the primitive guaranteed to terminate? If a process is constantly inserting tuples does this primitive terminate?
- Does the `rd()all` primitive automatically terminate once all matching tuples are found, given that an `rdp()all` primitive is also proposed?
- What happens to any tuples that the function provided within the primitive produces? If they match the template used in the primitive are they matched?
- The function within the primitive appears to be able to side-effect tuple spaces. Do the functions run concurrently? If not this allows deadlock conditions to be introduced trivially, by placing dependencies between the functions used in the primitive.
- Where is the function given in the primitive executed?
- How is the interaction between several `all` primitives managed?

The semantics for such a primitive would be complex. The combining of computation and communication appears to be against the natural philosophy of Linda. A parallel program

[2]The idea of `rd`-loops is proposed by Leichter[Lei89]. This similar to the `rd()all` primitive considered here.

consists of coordination and computation. The coordination is provided by a coordination language, and the computation provided by a programming language. The primitive does not necessarily allow the same level of control as the `copy-collect` primitive. For example, because the `copy-collect` primitive returns the number of tuples copied it is possible for the program to control the number of processes that are used to consume the tuples. Therefore, if there are a small number of tuples one process can be used, and if there are many tuples more processes can be used. The `rd()all` primitive does not provide the same flexibility as the `copy-collect` primitive.

The `rd()all` primitive can be emulated using the `copy-collect` primitive, and this is shown in Program 4.4. This particular implementation ensures that the `rd()all` primitive terminates, tuples produced by the function in the primitive are not matched by the primitive itself, and the function can deadlock if the function relies on tuples produced by other instantiations of the same function.

Program 4.4 Emulating a `rd()all` primitive using the `copy-collect` primitive.

```
rd_all := proc(ts, template, f);
          -- ts is the tuple space, template is the template,
          -- and f is the function

  local my_ts, my_tuple;

  my_ts := NewBag;
  todo := lcopycollect(ts,my_ts,template); -- Get the tuples

  while (todo > 0) do                    -- Process the tuples
    todo := todo - 1;
    my_tuple := lin(my_ts,template);  -- Get a tuple
    f(my_tuple);                         -- Apply the function
  end while;
end proc;
```

Having considered other proposed primitives that could be used to solve the multiple `rd` problem, it is possible to conclude that the `copy-collect` primitive is the most suited because it supports two unique features: it allows the user to specify a template for tuple matching and therefore is selective in the tuples it duplicates; and it returns only a count of the number of tuples copied, not the tuples themselves. A number of other issues are now considered.

Hasselbring[Has94] discusses the addition of primitives, specifically to “extend generative communication with data parallelism”, which is seen as the motivating factor behind the proposal of the `rd()all` (and its relatives). Hasselbring[Has94, page 82] concludes that:

Because of these problems we do not consider to extend generative communication in PROSET *with data-parallel operations. Such an extension would cause the serious problems both for the definition of the semantics and for the implementation.*

However, Carriero[CG90a] states there are three paradigms for coordinating in coordination languages (particularly with reference to Linda); *result parallelism*, *agenda parallelism* and *specialist parallelism*. When discussing *agenda parallelism*[CG90a, page 19] Carriero states[CG90a, page 20] that:

"Data parallelism is a restricted kind of agenda parallelism."

As agenda parallelism is supported within Linda it is logical to conclude the data parallelism is supported, and indeed in Carriero et al.[CG92] the ability of Linda to express data parallelism is discussed in depth. The need for new primitives, like the `copy-collect` primitive, is not driven by the need to introduce new coordination paradigms to Linda, but by the need to be able to perform certain operations satisfactorily. In the case of the `copy-collect` primitive this is a multiple `rd` operation.

Finally, a brief comparison between the use of the `copy-collect` primitive and the use of first class tuple spaces to overcome the multiple `rd` problem is considered. In Section 2.3.1 a description of multiple tuples was presented. If tuple spaces are first class, as in MTS-Linda[Jen93] and Bauhaus Linda[CGZ95] then can they be used to overcome the multiple `rd` problem? It is possible to produce a copy of an entire tuple space, by simply using the `rd` primitive to take a copy of it. This solution is similar to merging the bounded `rd` and the `copy_contents` primitives. All the tuples in a tuple space can be copied into a local data structure within a single process. The tuples which are then required can be retrieved from that data structure and processed. However, as with the bounded `rd` primitive, what happens if the local memory is not large enough to store the tuple space? The `copy-collect` primitive does not return the tuples so they remain stored within the run-time system, which can control their placement. Also, operations that can be performed on a tuple space stored within a process would have to be used, to at least retrieve the tuples from the data structure. Given the implementational difficulties (see Section 5.5) that having first class tuple spaces, combined with the unanswered questions that remain about how tuple spaces can be manipulated, and that the copying of a tuple space involves the retrieval of an entire tuple space, the `copy-collect` primitive is still required if tuple spaces are first class objects.

4.7 New coordination constructs

In this chapter a new primitive was presented, `copy-collect`. It has been shown how this primitive solves the multiple `rd` problem. In this section another use of the `copy-collect` primitive is considered, when the `copy-collect` primitive is used for the detection of completion of worker processes. Many Linda programs are written in a master worker style of parallelism (or agenda parallelism)[CG90a]. This means that there are a number of worker processes working concurrently with a single master process overseeing the worker processes. With this style of Linda programming usually, when a set of worker processes are all blocked, they have reached a completion state. Once this has been identified by the master process a *poison pill* is passed to the worker processes. Program 4.5 shows the basic structure of a worker process that uses a poison pill. The worker process keeps removing tuples which contain "information" to be processed. This is repeated until the worker process reads a tuple which has a value that it recognises as a "poison pill". Once the poison pill has been identified the worker process terminates, knowing that all the required work has been performed by the set of worker processes. To ensure that the other worker processes terminate as well, the process replaces the poison pill tuple before terminating. The use of poison pills has been well documented[CG89a, Lei89, CG90a].

Program 4.5 Example of the use of poison pills in a worker process.

```
int worker(void)
{
  int task_id;

  in("work",?task_id);
  while (task_id != POISON_PILL)
  {
    process_task(task_id);
    in("work",?task_id);
  }
  out("work",POISON_PILL);
  return 0;
}
```

How does the master process which produces the poison pill know when to produce it? If the poison pill is produced before all the tasks have been consumed by the worker processes, they may terminate before completing all the work. In the simplest case the consumer process consuming the results produced by the worker processes knows how many results are expected. Once the consumer process has consumed the required number of results it produces the poison pill tuple and thus initiates the termination of the worker processes. The occasions when the consumer process knows the number of results required is limited and the more usual approach is to use a tuple as a global counter. The worker process using a tuple as a global counter is shown in Program 4.6 where the tuple $\langle \text{“count”}_{string}, 0_{integer} \rangle$ is the tuple which is used as the global counter. The procedure `start` spawns a number of worker processes which consume tuples, and a function `producer`. The function `producer` creates a hundred tuples with a string and an integer in each tuple. These tuples are consumed by the spawned worker processes (the function `consumer`).

Program 4.6 Example of the use of counters and poison pills.

```
int producer(void)
{
  int x, value;

  for (x = 0; x < 100; x++)
  {
    in(uts, "count",?value);
    out(uts, "count",value+1);
    out(uts, "work",x);
  }
  return 0;
}
```

```
int consumer(void)
{
  int task_id, value;

  while (true)
  {
    in(uts, "work",?task_id);
    if (task_id == POISON_PILL)
      break;
    process_task(task_id);
    in(uts, "count",?value);
    out(uts, "count",value-1);
  }
  out(uts, "work",POISON_PILL);
  return 0;
}
```

```
void start(void)
{
  int x;

  out(uts,"counter",0);
  for (x = 0; x < WORKERS; x++)
    eval(uts, "consumer",consumer());

  eval(uts, "producer", producer());
  in(uts, "producer",0);
  in(uts, "count",0);
  out(uts, "work",POISON_PILL);
}
```

The main process waits until the producer has finished, and then waits for the counter tuple to reach zero. Each time the `producer` function places a “work” tuple in the tuple space the global tuple counter is incremented by one, and every time a worker process consumes a “work” tuple it decrements the global tuple counter by one. If a worker process creates more work tuples then it increments for each tuple it produces. The function `start` *blocks* waiting for the `producer` function to finish. Once the `producer` function has finished the `start` function then again *blocks* waiting for the global tuple counter to have a zero value. When the global counter is zero all the `consumer` functions will be blocked, and all the "work" tuples will have been processed. When the `start` function becomes unblocked it places the poison pill tuple in the tuple space so initiating the termination of all the `consumer` functions.

The producer and consumer functions execute concurrently. The global counter tuple can become a bottleneck, because all the worker processes and the producer process compete for the tuple. When updating the global counter, the tuple containing it has to be removed from the tuple space, the counter updated, and then a new tuple created in the tuple space with the new global counter value. This means that the access of the global counter is sequential. If the program is coarse grained this is not such a problem as few processes may need to concurrently update the counter. However, as the granularity of the program is decreased, or the number of processes being used increases, the regularity with which multiple processes want to update the global counter concurrently increases. Therefore, this single tuple acts as a bottleneck for the entire system. An alternative approach using the `copy-collect` primitive is shown in Program 4.7.

In Program 4.7 there is a counter for *every* process which wishes to access the global counter. The worker processes consuming the work tuples no longer decrement a single global counter every time a work tuple is consumed but rather decrement *any* one of the counters. When the producer creates a work tuple it increments *any* one of the counters. A worker process can create more work tuples, by simply producing them and updating the counter as appropriate. When all the counters summed have a value of zero, and the producer process has either terminated or indicated that it has stopped producing work tuples, then all the work tuples that have been produced by the worker processes and the producer process have been processed, and the worker processes are blocked waiting for more work tuples.

How can a check be made to see if the counter tuples summed are zero? This is achievable using the `copy-collect` primitive to "grab" a copy of the counter tuples using the template $\langle|\text{“count”}_{string}, \square_{integer}|\rangle$. If there are less counters copied than exist then this means that at least one process is currently updating a counter and therefore at least one of the worker processes is still working and a completion state has not been reached. If all the counter tuples are copied then all the counters are summed to check if the total is zero. If the total is zero, then the worker processes have completed and the poison pill can be inserted into the tuple space. In order to perform the check a tuple space is created into which the counter tuples are copied. The repeated creation of a tuple space in the checking loop will result in a large number of unreachable tuple spaces being created. Unreachable in the sense that they contain tuples but the tuple space handle is out of scope and not stored in a tuple space. Current work by Menezes et al.[Men96] shows that garbage collection of such tuple spaces is achievable.

Program 4.7 Example of the usage of the `copy-collect` primitive and poison pills.

```
int producer(void)
{
  int x, val;

  for (x = 0; x < 100; x++)
  {
    out(uts, "work", x);
    in(uts, "count",?val);
    out(uts, "count", val+1);
  }
  return 0;
}
```

```
int consumer(void)
{
  int task_id, value;

  while (true) {
    in(uts, "work", ?task_id);
    if (task_id == POISON_PILL)
      break;
    process_task(task_id);
    in(uts, "count", ?value);
    out(uts, "count", value-1);
  }
  out(uts, "work",POISON_PILL);
  return 0;
}
```

```
void start(void)
{
  int count, mts, finished = false;

  for (count = 0; count < WORKERS; count++) {
    eval(uts, "consumer",consumer());
    out(uts,"count",0);
  }
  eval(uts, "producer", producer());
  out(uts,"count",0);
  in(uts, "producer", 0);
  while (finished == false)
  {
    mts = tsc();
    if (lcopycollect(uts, mts, "count", ?int) == WORKERS + 1)
      if (sum_tuples(mts) == 0)
        finished = true;
  }
  out(uts, "work", POISON_PILL);
}
```

By creating one counter tuple for each process which accesses the counter, the counters will never act as a bottleneck. Any process which wishes to access a counter tuple will be able to do so. In the example given all the counter tuples appear the same. However, it is possible to have each process maintain its own tuple in the tuple space which represents its current state. For example, each process could have its own counter, which would be used to determine not only when all the processes have completed but also the number of work tuples a worker process had consumed and produced. The `copy-collect` primitive has been used in this way in a parallel algorithm for *stable assignment*[WR95][3]. In the program for the stable assignment problem completion of the assignment occurs when all the processes are blocked. The state of each process is stored in a tuple in a tuple space. The `copy-collect` primitive is used as described above to detect completion by "grabbing" the state tuples for all the processes. Once they have been "grabbed" they are checked to see if there is one tuple for every process and if the process's state contained in the tuple is correct for completion to occur.

Which of the two methods, the use of a global counter or the polling of distributed state using the `copy-collect` primitive, is better is a subjective question? The use of the `copy-collect` primitive follows more in the asynchronous nature of the Linda model, but the continuous polling may cause extra load on the run-time system. Alternatively, the use of a global counter requires a global tuple shared by all the processes which can become a bottleneck.

4.8 Conclusion

The proposal of new primitives for Linda is common place. Section 2.3.2 outlined some of the more sensible suggestions. The addition of new primitives should not be motivated by the need to make something easier for the run-time system implementor, or to make something which is implicit (using compile-time analysis), explicit. The `copy-collect` primitive solves a real problem with Linda and the expressive power of the primitive has been shown through the use of experimental results and predicted results.

What primitives are required if multiple tuple spaces are adopted within the Linda model? The answer to this is that the `collect` primitive and the `copy-collect` primitive are required. The justification for the `collect` primitive is given in Butcher at al.[BWA94]. The justification for the `copy-collect` primitive is the multiple `rd` problem as specified in Chapter 3. Are any other primitives needed when multiple tuple spaces are added to the Linda model? So far there appears to be no requirement for more general primitives, such as entire tuple space copies or moves. These more general primitives could potentially be created using a number of `collect` or `copy-collect` primitives if they were required.

This leads to the second question presented in Chapter 1. How can the bulk primitives of `collect` and `copy-collect` be implemented efficiently within an open Linda implementa-

[3]This was originally known as the *stable marriage problem*, but in these politically correct times people prefer *stable assignment problem*.

tion? So far, the implementation of the `copy-collect` primitive has not been addressed in any depth. In the next chapter the implementation of the `copy-collect` primitive is considered. By combining implicit information about multiple tuple spaces and knowledge about the use of the `copy-collect` (and `collect`) primitives a novel Linda run-time system is proposed.

Chapter 5

The implementation of bulk primitives

5.1 Introduction

Having presented the new `copy-collect` primitive in Chapter 4, in this chapter the efficient implementation of both the `collect` and `copy-collect` primitives is considered. These bulk primitives require multiple tuple spaces, and creating an efficient implementation of multiple tuple spaces provides a foundation for a fast and efficient implementation of the bulk primitives. The original inspiration came from observing the use of the bulk primitives in programs using the York Kernel I with ISETL-Linda on the Meiko CS-1.

In this chapter a new kernel is described called the York Kernel II, which has been fully implemented. The performance of the York Kernel II is shown in Chapter 6, and Chapter 7 gives a detailed proposal for the extension of the ideas presented in this chapter to create a truly hierarchical kernel. Before the techniques used in the York Kernel II are described, a detailed review of implementations is given in the next section and a "naive" approach to implementing bulk primitives is described.

5.2 Review of implementations

It has already been alluded to that implementations can be classified as either open or closed implementations. All Linda implementations require some run-time system which is referred to as the *kernel*. In some implementations this is a set of library routines which are linked in at compile time, in other implementations it is a single separate process, and in some other implementations it is a set of distributed processes. If the kernel is distributed then the different processes will be referred to as *kernel processes*. Different implementations and implementors refer to these kernel processes by different names, for example *tuple space manager, TS-manager* and *TSM* have all been used. The term *kernel process*, will be used regardless of what the authors originally christened their processes.

A *closed implementation* is considered an implementation which requires information about all processes which are to communicate via tuple spaces to be available when the kernel starts. The implication of using a closed implementation is that processes cannot leave and join at will, because information about the processes that are to communicate is required when the kernel starts. Most closed implementations require either the object code or source code for all processes to be available at link or compile time. Such closed implementations have the advantage of being able to use compile-time analysis, and therefore, normally consist of two sections; a pre-compiler (or compiler) and a kernel (either distributed or non-distributed). The compilers perform some form of compile time analysis to enable better control and management of tuples. Most early implementations of Linda were closed implementations and were produced by researchers at Yale University[Car87, BCG89, Lei89, Zen90], where Linda was originally created.

An *open implementation* is defined as an implementation where the processes communicating through tuple spaces need no information about the processes with which they share tuples and vice-versa. Also the kernel requires no prior knowledge about processes when it starts. This means that processes (programs) can leave and join at will, because no information about processes (programs) is required when the *kernel* starts. The communicating processes can be written independently, and even in different programming languages. Open implementations consist of a kernel and sometimes a pre-compiler or compiler. The role of the pre-compiler is normally to provide a more natural syntax for the Linda primitives embedded in the host language. Because not all the processes are available to the pre-compiler less analysis of tuples and tuple usage can be performed. There have been several open implementations, including [DWR95, RDW95, SCM93, Pin91, Ams95, FGY95, Ban96, Tol95a].

5.2.1 Open implementations versus closed implementations

Most of the recent work has been performed on the development of open implementations. The performance achievable by open implementations is currently below that of closed implementations which use compile time analysis because of the performance increases that compile time analysis can provide. Closed implementations have the ability to alter the kernel's fundamental characteristics based on knowledge of how a program uses tuples. This should lead to a reduction in the number of messages being sent between the user processes and the kernel, and as the communication costs are significant, there should be an improvement in performance. The best closed implementation using compile-time analysis is the SCA C-Linda, a commercial system based on the implementations produced at Yale University.

However, the drawback with closed implementations is that they are restrictive. Most of the closed implementations are used for *multiprocessing*[Has94] (a single application utilising several processes) rather than *multiprogramming*[Has94] (many applications possibly utilising several processes). With multiprocessing it is easier to control the whole system, a group of programmers create the application and they are able to design the coordination patterns of the application. There is little use of spatial and temporal separation, because all the processes are available at compile time. Whereas multiprogramming uses more of the general features of Linda, such as the ability to support processes which are spatially and temporally separated[Ban96, Kie96, CG90b]. Applications need to be able to join and leave the kernel at will. Also open implementations support the concept of persistence of tuple spaces. One application can place information in a tuple space and then another application can use that information at any time in the future. Persistent tuple spaces have been likened to files within a file space[Gel89]. Information can be placed within a tuple space as it would be placed into a file, and then retrieved at a later date.

5.2.2 Closed implementation techniques

Carriero[Car87] implemented the first Linda system for both shared memory parallel computers (Encore Multimax and Sequent Balance) and a distributed memory parallel computer (S/Net). The Encore Multimax and Sequent Balance implementations relied on the use of compile-time analysis. The compile-time analyse involved the examination of the tuples and templates to enable efficient data structures to be constructed for storing the tuples. The analysis specifically examined field types and actuals present within the templates and tuples. Once this information is known the fields which need to be matched at run-time can be calculated, and redundant fields can be removed. The shared memory implementation placed the data structure in which the tuples are stored in the shared memory. The distributed memory implementation on the S/Net did not use any compile time analysis and simply replicated a simple data structure for storing tuples within each processor module of the computer, and used broadcasts to all nodes to ensure that the data structures were kept synchronised.

The same compile time analysis techniques used in the Encore Multimax and Sequent Balance implementations are used by Bjornson et al.[BCG89, Bjo92] and Zenith[Zen90]. However, both these implementations examined how the kernel could be implemented for distributed memory parallel machines where the replication of all tuples on all the nodes is unacceptable. In these implementations the kernel is distributed over several processors within the parallel machine and the tuples stored on one of the many kernel processes. The tuples are distributed across the kernel processes using a hashing function. For a given tuple the hashing function identifies a unique kernel process for that tuple. For a given template the hashing function identifies the kernel process on which a matching tuple would reside. The kernel architecture used within these implementations provides the basic architecture that has been widely used in most kernels since then.

The next major advancement of compile time analysis was again introduced by Carriero[CG91a, CG90b, CG91b]. Instead of just analysing the tuples and templates to generate efficient data structures, and detect which fields need to be matched at run-time, the compiler actually performed “partial evaluation” of the Linda primitives. The basic approach is to recognise how tuples are being used and then implement a suitable approach to deal with the coordination patterns. For example, if there are a number of processes performing `in("semaphore")` followed by `out("semaphore")` the compile-time system can recognise this coordination pattern. Once recognised the kernel at run-time can create a counter to act as that particular tuple. Whenever an `in("semaphore")` is performed the kernel simply decrements the global counter if it is greater than zero otherwise the primitive blocks. Whenever an `out("semaphore")` is performed the kernel increments the counter. The compile time analysis recognises a particular coordination pattern (in this case a tuple being used as a semaphore) and instructs the kernel to use a more efficient mechanism to control that tuple. The kernel is also able to ensure that the mechanism is placed in a kernel process close to the user processes using the semaphore. The compile time analysis is also capable of recognising when tuples are being used as global counters, and instead of removing the tuple, updating it and then replacing it, the operation is implemented as a counter stored within the kernel[1]. The analysis also improves the placement of tuples, for example with the ability to detect that tuples can only be consumed by a particular user process implies that the tuples can be sent directly to that user process. Also, if a tuple, once produced, is only non-destructively read then it can be broadcast to all user processes that could potentially access it.

The commercial version of C-Linda, SCA C-Linda, is based on the work of the Yale University researchers[Car87, BCG89, Lei89, Zen90]. More optimisations have probably been developed for compile time analysis by SCA but have not been published due to commercial considerations.

Work outside Yale University on closed implementations has also been performed. This has mainly concentrated on the development of “hierarchical” kernels[dHM91, CdHMW92, CW92, MP93]. The underlying idea is that by grouping processes which “share” tuples a more efficient implementation can be made.

Matos et al.[MP93] have created an implementation based on the use of multiple tuple spaces called Linda-Polylith. The multiple tuple space model adopted is a hierarchical one. The compile time analysis can be considered to produce a tree, where the nodes represent a tuple space and the leaves represent user processes. If a process is to access a tuple space then the process must be a descendent of the node which is the tuple space. The root node is the global tuple space, so all processes can access it. The problem with such an approach is the fixed nature of the communication allowed. A tuple space handle cannot be passed to other processes because they are fixed at compile time. However, the concept of a hierarchical kernel is an interesting one, and is used in Chapter 7. A more in depth description of Linda-Polylith is given in Section 5.5.1.

Clayton et al.[CW92, dHM91] described their kernel as a hierarchical kernel, however, it appears to use a flat structure of their kernel processes. They use compile time analysis to group tuples in a similar manner as Carriero[Car87] to allow distribution across a number of kernel processes. They also use compile time analysis to create a static placement mechanism for spawned processes[dHM91, CdHMW92]. This relies on a machine description; compile time information about when processes are spawned; and heuristic rules to decide statically (at compile time) where the processes should be placed.

[1]See the description of the `update` primitive in Chapter 2.

Within this section a brief overview of the role of compile time analysis within closed implementations has been presented. The kernel implementation which is described in this chapter is an open implementation. Most of the compile time analysis techniques described here are not suitable for open implementations because they require information which is not available. There are a few of the compile time analysis techniques that can be used, for example the conversion of an `in`/`out` pair (as described in Section 2.3.2) into a single operation. However, within the context of this dissertation the primary interest is with the implementation of the `copy-collect` primitive (and the bulk primitives) rather than other optimisations that are already well known and documented.

There are many similarities between the kernels used in closed implementations and those used in open implementations. In many ways the kernels currently used for open implementations are cut down versions of the kernels used in closed implementations. In the next section an overview of current techniques which are used in kernels (appropriate for open implementations) is presented.

5.3 Kernel implementation techniques

The basic role of a kernel in an open implementation is to manage tuples. It "receives" messages containing instructions (which normally map onto the Linda primitives), it processes these messages, and returns, if appropriate, a tuple or reply message. All kernels have a number of basic characteristics which are:

- Tuple distribution, which is how the tuples are going to be distributed *across* a number of kernel processes,
- Tuple format, which is the format of the tuples,
- Tuple storage, which is how the tuples are stored within a single kernel process, and
- Eval implementation, which is how the `eval` primitive is implemented.

The characteristics are not disjointed and making decisions about using one approach for one characteristic can often limit the choices for another characteristic. The tuple distribution mechanisms and the implementation of `eval` are now considered in more detail.

5.3.1 Tuple distribution

How are sets of tuples distributed across the kernel (as opposed to within kernel processes)? There are four approaches used within current kernels to controlling the distribution of tuples across a kernel[ACGK88, CSS94, Cam96]. These four approaches are:

Centralised This is where the kernel is a single process. All tuple space operations are sent to this single process, and all the tuples are stored in it.

The advantages of such an approach is that the kernel is simple and all the tuples are kept together which means it is easy to take a “snapshot” of the current state of the tuple space. This property has made the centralised approach popular in many implementations supporting fault tolerant tuple spaces, such as PLinda[JS94, Jeo96] and Paradise[Ass96].

The disadvantage of having a centralised kernel is that the single kernel process becomes a bottleneck. As more processes try to perform tuple space operations concurrently the kernel simply cannot service them fast enough. If either a small number of user processes are to be used, or the number of tuple space accesses that a set of user processes are to perform is low then a centralised approach provides acceptable performance. This type of approach is used in Parlin[SCM93], TsLib[SVS94], PLinda[JS94, Jeo96], Paradise[Ass96] and Glenda[SAB94].

Uniform distribution This is where the kernel is distributed (there is more than one kernel process) and the tuples are distributed evenly over the kernel processes. This is often achieved by every user process having two sets of kernel process identifiers called an *in-set* and an *out-set*. Whenever a tuple is placed into a tuple space the tuple is broadcast to all the kernel processes in the *out-set*. Whenever a tuple is required from a tuple space the request is sent to all the kernel processes in the *in-set*. If the tuple is retrieved using an `in` primitive then all the kernel processes in the *in-set* have to synchronise to update the tuple spaces to ensure that two user processes cannot retrieve the same tuple. If there are t tuple space servers the cardinality of the *out-set* can vary from one to t (and the cardinality of *in-set* will vary from t to one). All the *out-sets* present within the user process must include a member from each of the *in-sets* in all of the user processes and vice-versa. Carriero’s S/Net implementation[Car87] uses this approach with an *in-set* being a local kernel process (one that resides on the same processor as the user process) and the *out-set* being all kernel processes. This is because the S/Net provided a cheap broadcast function. If this kind of approach is required then it is more common to adopt an approach known as *intermediate uniform distribution*.

Intermediate uniform distribution This is a particular case of uniform distribution. If there are t nodes then the cardinality of both the *in-set* and the *out-set* are $\sqrt{t}$. This is shown in Figure 5.1.

This variant of uniform distribution has been proved the most optimal uniform distribution[ACGK88] in terms of the number of nodes involved in an `in` primitive and an `out` primitive. This particular approach is adopted in the Linda machine[ACGK88, KACG87, KACG88], where the bus that joins the different Linda nodes provides the arbitration necessary to ensure that the tuple spaces remain consistent when several `in` primitives are performed by different processes concurrently. A number of other implementations have used the same approach, X-Linda[Faa91] (for transputer

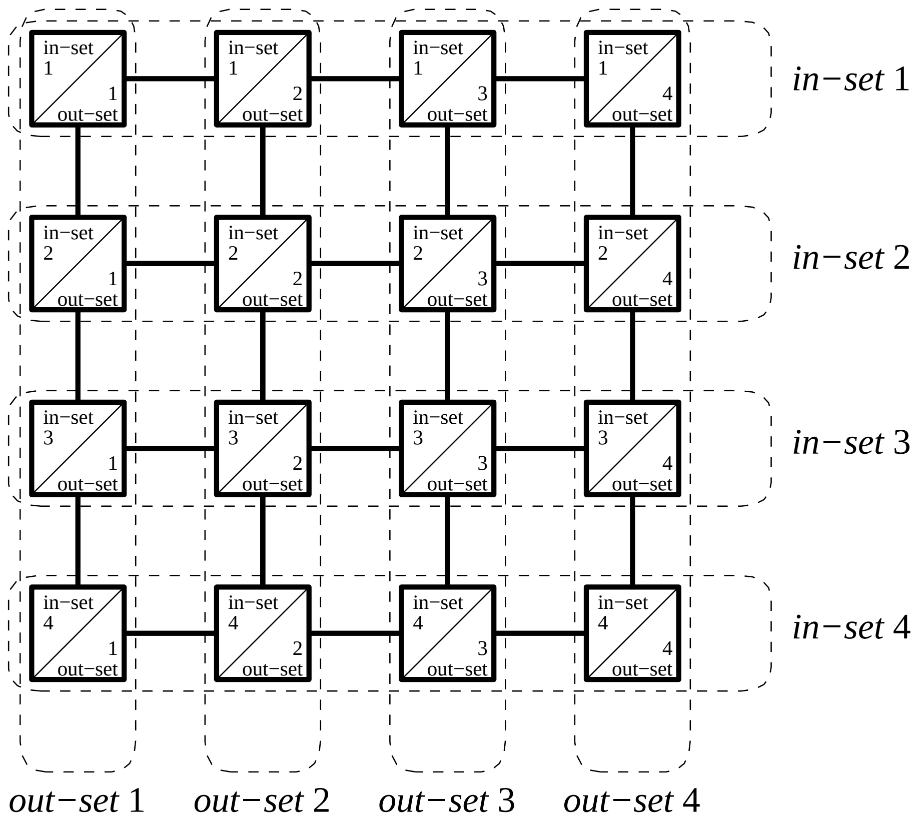


Figure 5.1: Intermediate uniform distribution using 16 kernel processes.

meshes) and the Bag-machine implementation[Tol95b] (network of workstations). In these cases the communication costs of synchronising the duplicated tuples is too great[Faa91] to make the method efficient without specialist hardware support (as in the Linda machine). Whenever a tuple is retrieved by a user process from a kernel process, the kernel processes which are members of the *in-set* used by the user process have to determine which can provide a suitable tuple. If many kernel processes can provide a suitable tuple then one has to be chosen. Once the kernel process has been chosen it then has to inform the kernel processes that are in the same *out-set* that the tuple is being removed. Without the support of special buses such an approach requires a large amount of communication to control all the arbitration that is needed[Faa91]. Tolksdorf[Tol96] has created a kernel that can dynamically change over time allowing the number of kernel processes to be both increased and decreased, where the distribution strategy is based on intermediate uniform distribution.

This leads to the final general type of tuple distribution, Distributed hashing.

Distributed hashing Distributed hashing is another distribution mechanism for use in distributed kernels, and the kernel process which stores a particular tuple is chosen by using the properties of the tuple or template being used. In order to do this a hashing function is used which when applied to a tuple or template provides the kernel process on which either the tuple should reside, in the case of a tuple, or where a matching tuple would reside in the case of a template. Hashing is discussed in detail by Bjornson[Bjo92], and is used as the basis for most open implementations, including several previous implementations at the University of York[DWR95, RDW95]. The aim is to develop a hashing function that has two properties. Firstly, this should provide a unique mapping between every tuple and a template which matches it to a single kernel process and secondly it should provide a good distribution of the tuples over the kernel processes. This has the advantage that the kernel process given by the hashing algorithm will contain a matching tuple, if there is one in the tuple space, which removes the problems of searching multiple kernel processes. However, pragmatically this has only been achieved effectively in closed implementations. For open implementations no general purpose hashing algorithms have been created because of the limited amount of information potentially provided within a template and the lack of compile-time analysis of all tuples and templates used within a system. Therefore, in open systems, hashing functions are chosen that enable every *tuple* to be hashed to a unique kernel process and a template hashed to a set of kernel processes. In the best case the cardinality of this set will be one because the information the hashing algorithm uses for a tuple is present in the template. The request for the tuple is then either broadcast to all the kernel processes produced by the hashing algorithm for a particular template, or to a particular kernel process. If there is a broadcast to more than one kernel process then some form of arbitration has to be performed by the user process (transparent to the Linda programmer) as there may be more than one tuple returned. If the request is sent to a single kernel process and that kernel process cannot find the tuple it will then broadcast the message or pass it to another kernel process. The original kernel process then deals with the arbitration. An interesting point is that the kernel created by Bjornson[Bjo92] provides dynamic analysis of tuple accesses. Therefore, if a particular process is consuming tuples of a particular kind, then the hashing functions in the user processes are dynamically altered (by messages from the kernel) to send the tuples to the kernel process that is local to the user process consuming the tuples. This technique is called *bucket switching*.

The choice of the distribution approach used depends largely on the requirements of the system. Most current implementations use the distributed hashing approach, because it is more efficient and it does not require the synchronisation of kernel processes in *out-sets* whenever a tuple is destructively retrieved. The kernel developed in this chapter uses an approach based on the distributed hashing method, which is discussed later in this chapter.

5.3.2 The `eval` primitive

In Chapter 2 the `eval` primitive was described. To reiterate; the `eval` primitive creates an *active* tuple, which is a tuple with one or more of the fields a function which are evaluated concurrently.

Whenever the evaluation of a particular field is completed the value produced is placed within the active tuple. Once all the fields have been evaluated the active tuple becomes a passive tuple.

Linda 1 did not contain an `eval` primitive, but the first implementations of Linda supporting what is now called Linda 2, contain an `eval` primitive and introduce the concept of *active* tuples. The need for active tuples appears unclear in the early implementations as they cannot be matched or manipulated by any user process.

Implementation strategies vary from not providing any sort of `eval` primitive, through mapping the `eval` primitive onto the basic spawning characteristics of the system being used (for example, Glenda[SAB94], PLinda[JS94], eLinda[WC95] and York Kernel I (PVM version)[RDW95]) to providing a mechanism that literally places an "active tuple" within the tuple space that can be manipulated by other processes, for example MTS-Linda[NS94] (although not fully implemented).

The creation of a tuple in the tuple space when all the functions have been evaluated appears desirable. It provides a simple and effective mechanism to allow other processes to detect when a set of processes have terminated. Allowing active tuples to be retrieved raises many questions. How are active tuples matched, and in particular how is a field which is a process matched? How does a process know if it is getting a process or a value? What happens to the matched process and what does it mean to perform a `rd` primitive matching an active tuple?

Nielsen et al.[NS94] discusses these issues, proposing that matched processes are bound to variables, and the addition of a `touch` primitive which forces the functions to be evaluated before the tuple can be retrieved. The example given in Nielsen et al.[NS94, page 27] implies that the `touch` primitive is given a template that matches the tuple. The ability to manipulate active tuples raises many questions about how this should be managed and implemented, which have not yet been sufficiently answered.

There has been some research into how the `eval` primitive can be implemented to provide a passive tuple upon completion, without supporting the manipulation by user processes as active tuples[HKCG91, RA95]. Both approaches are similar and require compile time analysis which makes them useful for closed implementations, but not for open implementations. Both approaches use the concept of "`eval` servers" which receive instructions to execute particular functions. When a user process performs an `eval` primitive the description of the function or functions to be executed are sent to the `eval` servers. When each function has been evaluated the `eval` server updates a shared structure (which is a special tuple), inserting the resulting value. When all the functions are evaluated and the values inserted the tuple becomes a passive tuple. All the communication is achieved through the use of tuple spaces (including the passing of arguments to the functions). In both approaches care has to be taken to ensure that "spurious" deadlocks do not occur if there are fewer `eval` servers than spawned processes, because the `eval` servers can only sequentially evaluate a single function. Spurious deadlocks can be produced if there is a synchronisation between two process which are considered to be executing concurrently but are in fact not. Consider the two functions in Program 5.1. Assuming another process has created a tuple space and then uses the `eval` primitive to spawn these two process concurrently. These two process need to be executed concurrently, therefore, if there was only one `eval` server, one function would be picked and executed, but it would block *until* the other process was executed. If the `eval` server is only able to service one function at a time, then the `eval` server will be deadlocked and the other process will never execute.

Program 5.1 An example of a "spurious" deadlock.

```
process_one := func(ts1);
  lout(ts1, ["HELLO"]);
  lin(ts1, |["REPLY"]|);
  return 0;
end func;

process_two := func(ts);
  lin(ts, ["HELLO"]);
  lout(ts, |["REPLY"]|);
  return 0;
end func;
```

An alternative adopted by Clayton et al.[CdHMW92] in a transputer implementation of Linda involves the development of a static heuristic approach to the placement of processes. The approach is only suitable for a closed implementation and is restrictive, assuming certain types of characteristics about the use of Linda programs, and is not suitable for use in open implementations.

In the ISETL-Linda implementation[DWR95, DRW95] at the University of York, the system supports an `eval` primitive which produces a passive tuple but can only contain one function to be evaluated. This is achieved without using compile time support as the language is interpreted, but by using "ISETL engines" which receive both the function and a partial tuple. Once the function has been evaluated by the ISETL engine it updates the partial tuple by filling in the returned value from the function, and then places the (complete) tuple into the appropriate tuple space. The ISETL engines use threads and can therefore evaluate more than one function concurrently which ensures that spurious deadlocks, due to functions not executing concurrently, cannot occur[2].

The kernel which is described in this chapter provides a simple `eval` primitive which allows a single process to be spawned. There is no concept of an active tuple, and a tuple is not created in the tuple space when the function evaluation is completed. If a tuple is required the function must create it before completing. It is not possible to pass any initial parameters to a spawned process.

[2]Provided that the number of threads being executed on each ISETL Engine is within the bounds of the hardware being used.

If initial parameters are required they must be passed through a shared tuple space. The York Kernel II provides a universal tuple space to which all processes have access. It would probably be possible to extend the work of Hupfer at al.[HKCG91] to provide more general `eval` servers that could be used within open implementations, however this has not been considered within this dissertation.

This section has presented a brief overview of some of the main concepts and characteristics of open implementations. In the next section the addition of explicit information to Linda programs is considered as a mechanism for improving the performance of Linda programs.

5.4 Adding explicit information to Linda programs

The addition of explicit information to Linda programs can often improve the performance of the kernel. The explicit information can take many forms including special primitives and "hints" (or *pragmas*). The special primitives are treated in a more efficient manner than the equivalent using Linda primitives. Examples of such primitives were outlined in Chapter 2 (the `add` and `update` primitives). Many programming languages already make use of pragmas which are either language specific or compiler specific but do not normally alter the semantics of the program in which they are used. There are a number of examples, including the use of `register` in C. This instructs the compiler, that in the programmer's opinion, a variable should be stored in a register. If the compiler chooses to ignore this the program's semantics will not change. Ada also uses compiler hints in the form of *pragmas*[3], which control such things as whether functions should be inlined and what type of optimisation should be used. Within the context of Linda, pragmas have mainly been used to help control tuple spaces, and in particular how tuples flow through tuple spaces, and how the tuple space is used.

Controlling the flow and order of tuples in and out of tuple spaces[Row95] can be used to remove some of the costs of the extra synchronisation associated with forcing a specific order on the tuples within a tuple space. Within the context of Linda the order in which tuples are removed from a tuple space has no relationship with the order in which they are inserted. Pragmatically, in most implementations, there is a relationship between the order in which tuples are inserted into a tuple space and the order in which they are removed, which is deterministic. This is acceptable in implementations because the order in which the tuples are returned is a member of the set of all possible orderings. If the implementation was fully non-deterministic then it could always produce the same ordering. A dilemma that often faces a programmer is that a small performance advantage can be gained by ordering tuple retrieval, but the cost of managing the ordering in terms of extra coordination outweighs the benefits. Therefore, if a tuple space could be tagged as a *last in first out* queue, or a *first in first out* queue the benefits of tuple ordering are achieved without the extra synchronisation costs. If the implementation cannot support particular ordering then the program

[3]Some of the Ada pragmas alter the semantics of a program.

will still work (as long as the program has been written to be independent of the ordering), but without the performance gain of having ordered the tuples.

Eilean[CSS94] uses programmer hints which are used to aid in the distribution of tuples. The hints take the form of library calls indicating how certain tuples are used within a program, typing the tuples as being one of the following classes: producer-consumer, result, write-many, read-most and general read/write. Once classified the kernel treats each type of tuple differently, allowing more efficient placement and retrieval of tuples. A more specific approach has been suggested by Wilson[Wil91] where configuration files are created to allow the kernel to be configured and then the programs explicitly state where individual tuples should be placed. Such an approach may lead to increased performance but degrades Linda into little more than a system providing asynchronous buffered communication channels between processes, similar to many message passing systems such as PVM[SDGM94].

In the description of the implementation of MTS-Linda[NS94] it is suggested that tuple spaces should be explicitly tagged to indicate their use. Therefore, a tuple space could be tagged as a persistent tuple space, a tuple space to be replicated, a tuple space that compile time analysis should be performed on, a local tuple space, etc.. The kernel then treats the tuple space appropriately. MTS-Linda is considered in depth in Section 5.5.2.

The addition of explicit information to Linda programs moves the onus of producing efficient kernels from the kernel developer to the Linda programmer. This means that the Linda programmer has to understand and appreciate how the underlying implementation works in order to be able to write efficient programs. The aim should be to make writing programs easier for Linda programmers rather than more complex. This has led to a search for an alternative to the use of pragmas and new primitives which has resulted in a programming tool called the Linda Program Builder[ACG94] being developed. The Linda Program Builder is an interactive tool which supports the design and development of Linda programs. The user is able to design programs by choosing code templates which generate the code for different coordination patterns and constructs. Because the Linda Program Builder is aware of which code templates were used to generate a sequence of Linda primitives, it knows more information about how the tuples are being used. This extra information has been used in conjunction with the Yale C-Linda compiler[CG90b] to enable the compiler to further optimise the programs. Therefore, although a standard Linda program is produced by the Linda Program Builder it is able to add compiler hints. Unfortunately many of the optimisations used within the Linda Program Builder are suitable for closed implementations rather than for open implementations. All communicating processes are developed using the Linda Program Builder, and consistency in the use of particular tuples can be checked and enforced. All the compiled processes know how a particular tuple or tuples are stored or how a coordination construct is implemented.

5.5 Implementing multiple tuple spaces

Throughout this dissertation references are made to tuple space handles and names. In most implementations the name of a tuple space acts as its handle as well. The name of a tuple space is a (unique) tag that allows a tuple space to be identified. In open implementations the tuple space name must be unique because when a tuple space is created there is no knowledge about which processes will eventually be able to access the tuple space. If two tuple spaces have the same names then it would be impossible to distinguish between them. In this dissertation tuple space names such as "TS1" are used, however in reality they are usually a combination of several pieces of information which when combined create unique names.

The simplest way of adding multiple tuple spaces is to allow the programmer to pick the tuple space names as in Glenda[SAB94]. However, this means that several tuple spaces can unintentionally be called the same name, particularly in open implementations, and individual processes cannot create tuple spaces that other processes cannot access. So, it is normal for either the kernel or the user processes to create the tuple space names. In MTS-Linda[NS93] the kernel chooses names by providing a global name generator which ensures that every tuple space name is unique. Such an approach requires communication between the user processes and the kernel whenever a tuple space is created. Therefore, it is more common to allow the user processes to create the tuple space names[DWR95]. In a LAN implementation a user process is likely to use: the computer's name (IP address), process identifier, local counter within the user process, and potentially the date and time. The advantage of producing the tuple space names locally is that there is no communication required with the kernel, but the tuple space names are usually longer. Any user process which is aware of the name can then potentially write tuples to the tuple space.

The kernel has to manage the access of the tuple spaces which the tuple space names represent. The simplest approach is to treat the tuple space name as an extra field within tuples and templates. Therefore, the first field of every tuple and template has to be an actual of the type string[SAB94]. The tuple $\langle 10_{integer}, 1.0_{float}\rangle$ stored in a tuple space with the handle "4000.13.1" would be stored as the tuple $\langle$"4000.13.1"$_{string}, 10_{integer}, 1.0_{float}\rangle$. The syntax of the primitives embedded into the host languages disguises this so the programmer is unaware of this. When an `in` primitive is performed the template has the tuple space name added to it as its first field. This will always be an actual rather than a formal within the template. The kernel then treats the tuples and templates as it would do if multiple tuple spaces did not exist.

By not considering the tuple space name as part of the tuple a number of implementations have used tuple space names in a more efficient manner. Two implementations of particular interest are Linda-Polylith[MP93] and MTS-Linda[NS93], which are now described in greater detail.

5.5.1 Linda-Polylith

Linda-Polylith[MP93] is interesting because it uses classification of tuple spaces to achieve better performance, and utilises a hierarchical kernel. It is a closed implementation which uses compile time analysis to analyse the use of tuple spaces, and produce a tree like structure of tuple spaces. The position of the tuple spaces within the structure is static, and is fixed at compile time. The leaves of the tree are considered to be user processes, and each node of the tree is considered as a store for a *single* tuple space.

Program 5.2 Linda-Polylith program example.

```
tuplespace global              void main()
{                              {
  tuplespace local1              int i, num;
  {                              rd(0, "NUMBER", ?num)
    init_process main();         for (i=0; i< num; i++)
    init_tuple("NUMBER",5);      {
  }                                out(0,"fork",i);
  tuplespace local2                eval(0,philosopher(i))
  {                                if (i>0)
    init_process main();              out(0,"ticket");
    init_tuple("NUMBER",7);      }
  }                            }
}
```

Matos et al.[MP93] give the example shown in Program 5.2. The program consists of a *tuple space description* which names the tuple spaces; defines any tuples that should be placed inside the tuple space when it is created; and also specifies any initial processes that are executing “under” a tuple space. Therefore, in this example, there is a global tuple space with two other tuple spaces beneath it. The function `main` is spawned into each of these two lower tuple spaces. The function `main` is an initialisation function for a dining philosophers program. It creates the tuples which represent the room tickets and the forks, and spawns the philosopher processes. The number used as the first parameter in the Linda primitives in the function `main` are used to indicate how far up the tree the tuple space to be used resides. In this case the value 0 means the first level, which will be either tuple space `local1` or tuple space `local2` depending on the tuple space to which `main` is attached. If the value 1 is used, regardless of which local tuple space the function `main` is attached to, the tuple space `global` will be used. Hence, the two `main` functions could communicate with each other through the tuple space `global` by specifying 1 as the first parameter within the Linda primitives.

There appears to be a restriction that a spawned process must share the same parent tuple space as the process which creates the spawned process. The Linda-Polylith system has a number of other restrictions in order to aid compile time analysis, which are of no interest here.

Linda-Polylith uses the addition of *explicit* information to aid in the placement of tuple spaces. A tuple space is managed by a single process, which can be considered as a node in a tree. Processes cannot dynamically create tuple spaces, and handles for tuple spaces cannot be passed between processes. The explicit information required is complex and substantial, and requires the programmer to envisage the structure of the kernel in order to pass tuples through it. Such an explicit approach could not be used within open implementations as the structure of the tuple spaces is not known either at compile time, or when the kernel starts. The number of tuple spaces, and the processes which use them will change dynamically over time in a kernel for an open implementation.

In closed implementations[CG90b, CG91a] it is possible to analyse the tuples and the templates, and to create a similar structure to the one Linda-Polylith creates. Instead of expecting the programmer to perform the partitioning and organising of the tuple spaces, the global tuple space is partitioned automatically, and distributed over several processes. This compile-time analysis implicitly gathers much of the information which is provided in a Linda-Polylith program explicitly.

Linda-Polylith is interesting because it uses information about tuple spaces. However, the very explicit nature of the approach taken means that it is not only complex for a programmer but also restricted to closed implementations. An interesting concept is the storing of tuple spaces within a kernel with their position dictating who can access them. Given that the implementation is a closed implementation it is likely that messages for higher levels in the tree will be sent directly to the relevant kernel process, rather than passing them up the tree. This then makes the kernel, from an implementors point of view, flat. In Chapter 7 the concepts used in the York Kernel II are extended to create a truly hierarchical kernel, which is based on the same concept of a tree, although instead of each node storing a single tuple space, the nodes store sets of tuple spaces, and the tuple spaces migrate up the tree.

5.5.2 MTS-Linda

MTS-Linda[NS93, NS94] is an implementation based on the work of Jensen[Jen93]. MTS-Linda allows the user to tag a tuple space as a particular type of tuple space. The implementation supports only local tuple spaces and shared tuple spaces, but Nielsen et al.[NS94] suggest other types of tuple spaces could be used, such as replicated tuple spaces.

MTS-Linda uses both a hierarchical and a flat tuple space structure for tuple spaces. The hierarchical structure is created because processes execute inside a tuple space, and then the tuple spaces they create exist “within the process”. The tuple space in which the process resides can be considered as the parent tuple space of the process. Tuple spaces that are created within another process can be *copied* to the parent tuple space, but *not moved* to the parent tuple space. The tuple spaces used to create the hierarchical tuple spaces are called local tuple spaces. A local tuple space can only be accessed by the process that created it, and any other processes that have the tuple space as their parent tuple space. Therefore, it is common to have several processes accessing a “local” tuple space.

Shared tuple spaces are tuple spaces which are created so any process can access them provided that the processes has the tuple space handle. The tuple space handles can be passed using tuples in other tuple spaces.

MTS-Linda uses explicit tagging of tuple spaces to provide more information about the tuple spaces. However, MTS-Linda treats the underlying implementation of the tuple spaces as the same, regardless of whether they are shared tuple spaces or local tuple spaces. The fact that a tuple space is known to be a local tuple space, and subsequently can be accessed by a subset of all the processes provides information that could be used within the implementation to manage tuple spaces and control where spawned processes are executed. Indeed, the hierarchical kernel described in Chapter 7 provides an ideal match for such a hierarchy of tuple spaces. The suitability of hierarchical tuple spaces as embodied in local tuple spaces appears unclear, because of the apparent need to provide a flat tuple space structure as well.

5.6 A naive approach to implementing the bulk primitives

Having considered a brief overview of current implementation techniques, a naive implementation of the `copy-collect` primitive is considered, with respect to open implementations. A better implementation, based on using the classification of tuple spaces is presented later in this chapter and used in the York Kernel II. Because of the close relationship between the `collect` and `copy-collect` primitives the implementation of both these primitives is considered in this section.

In open implementations which use distributed hashing there are two approaches to implementing the bulk primitives. The first approach is for where the kernel process chosen to store a tuple is based solely on the fields within the tuple and is not influenced by the tuple space name. Thus a tuple will always reside on the same kernel process regardless of the tuple space to which it belongs. As neither a `copy-collect` nor a `collect` primitive alter the fields of a tuple, the tuples will remain resident on the same kernel process after a `collect` or `copy-collect` primitive has been performed. The implementation requires all kernel processes on which a matching tuple could reside to be contacted. Each kernel process checks the tuples it stores to find any matching tuples. Each matching tuple is either re-tagged as belonging to the destination tuple space, if a `collect` primitive is performed, or duplicated[4] and then re-tagged if a `copy-collect` primitive is performed. Each contacted kernel process returns a count of the number of tuples copied or moved, and all these are summed at the user process and returned as the result of the `collect` or `copy-collect` primitive.

If the kernel process on which a tuple resides is dependent on the tuple space name (and poten-

[4]The physical duplication of tuples *may* not be necessary under some tuple storage schemes.

tially the tuple) then both the `collect` and `copy-collect` primitives may require the movement of tuples from one kernel process to another. The general bulk movement of tuples around a kernel is an expensive process which occupies communication bandwidth and processing power. This generally leads to poor performance of the kernel. However, in the right circumstances making the `collect` and `copy-collect` primitives bulk move tuples can lead to a performance increase. This will be discussed in detail later in this chapter, and is utilised in the York Kernel II.

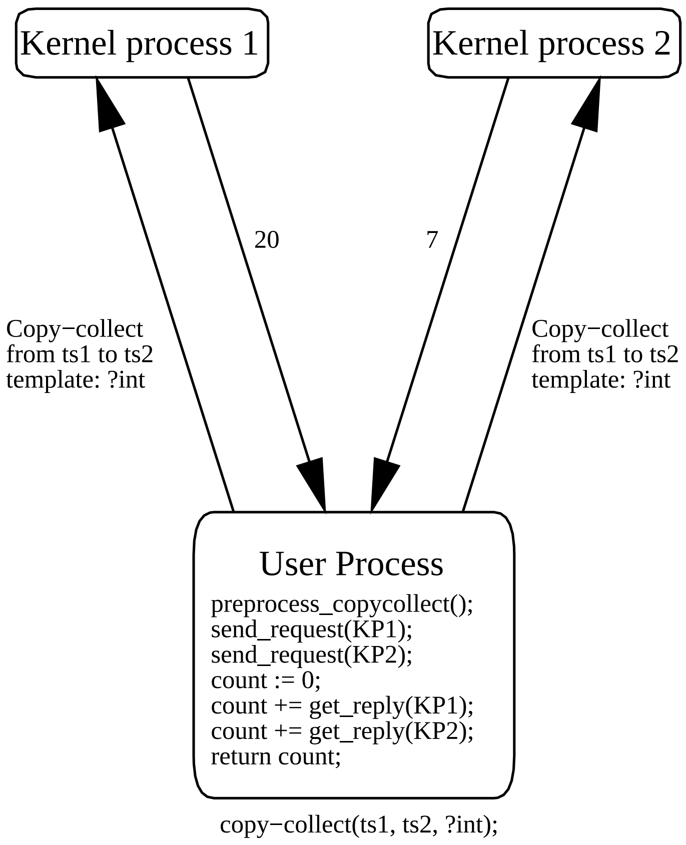


Figure 5.2: A naive approach to implementing the `copy-collect` primitive.

The first approach described is used in the first York Kernel I[DWR95, RW96a]. Even though the tuples that match the template can (and should be) distributed over several kernel processes, the approach does not require any global synchronisation between the different kernel processes managing the tuples. Figure 5.2 shows a distributed kernel with two kernel processes, and a single user process. The programmer uses a `copy-collect` primitive to move all tuples containing a single integer from tuple space `ts1` to tuple space `ts2`. The `copy-collect` primitive is implemented as a library routine which encodes the tuple, and dispatches the command to the appropriate kernel processes. In this case both kernel processes can potentially contain tuples of single integers. The user process then blocks waiting to receive replies from all the kernel processes contacted. Each kernel process contacted returns a counter indicating the number of tuples duplicated. When all the contacted kernel processes have returned their counters, they are summed by the user process's `copy-collect` library routine to produce the number of tuples actually duplicated by the entire kernel. The individual kernel processes do not communicate with each other, and no tuples are moved within the kernel.

The duplication of tuples may not necessarily involve a physical replication of the tuples within the physical memory of the kernel processes. It should be feasible to provide a data structure which allows for a tuple to be present in many tuple spaces but stored in the physical memory only once. Therefore, logically the tuple appears in several tuple spaces, but in reality it is physically only stored once. This has not been adopted in the York Kernel II.

The naive way of implementing the `copy-collect` and `collect` primitives when the kernel uses distributed hashing has been described. Their implementation when a centralised kernel is used is easier, as the command is sent to the single kernel process, which performs the operation and then returns a count of the number of tuples either copied or moved. When either uniform distribution or intermediate uniform distribution is used the cost of performing either a `collect` or a `copy-collect` primitive is expensive. This is because the duplication of tuples occurs in the kernel processes in the *out-set*. When tuples are either moved or duplicated all the kernel processes in the *in-set* must be updated. Every kernel process in an *in-set* has the potential to have tuples that match a given template, and every *in-set* and *out-set* must have at least one kernel process in common, so in the worst case *every* kernel process has to be updated.

Having considered the implementation, especially in the context of distributed hashing which is the most popular method for open implementation, it is interesting to consider how, by altering the semantics of the `copy-collect` and `collect` primitives the need for *global synchronisation* is introduced into a distributed kernel.

How the `copy-collect` and `collect` primitives interleave with other primitives is important. The rules as given in Chapter 4 state that if *two* `collect` primitives[5] occur concurrently using the same template and source tuple space, then the number of tuples moved from the source tuple space is the number that match the template, but the number moved to each of the destination tuple spaces is between zero and the maximum number of tuples available that match the template. The sum of the results of the two `collect` primitives will be the number of tuples removed from the source tuple space.

In Chapter 4 a suggestion to use traces to help define the semantics of the `copy-collect` primitive was discussed. The effect of using the traces is to make all the primitives atomic. In order to show the consequences to the implementation of making the bulk primitive atomic let us consider the case of two `collect` primitives being performed concurrently. Each uses the same source tuple space, different destination tuple spaces, and the same template. If the primitives were atomic then the result would be that one tuple space would be empty and the other tuple space would have all the matching tuples in it.

[5]Assuming that the semantics of the `collect` primitive are similar to the `copy-collect` primitive.

If the bulk primitives were made atomic then this would require a *global synchronisation* between all the kernel processes that are involved in the operation to be performed. Consider the example shown in Figure 5.2. If there are two user processes the first one performing a `collect(ts1, ts2, ?int)` and the second performing a `collect(ts1, ts3, ?int)` concurrently there will be two messages arriving at each of the kernel processes. The potential message ordering for each of the two kernel processes is shown in Table 5.1 (where the abbreviation `col` is used to mean a message containing a `collect` primitive request). This shows that the messages can arrive in any order at each of the kernel processes. In Case 1 tuple space `ts2` will contain all 27 tuples (assuming the same tuple distribution as in Figure 5.2). In Case 4 tuple space `ts3` will contain all 27 tuples. These are the only cases which fulfill the atomic semantics. In Case 2 tuple space `ts2` will contain 20 tuples and tuple space `ts3` will contain 7 tuples, and vice-versa in Case 3. Therefore, if atomic semantics are used the two kernel processes have to synchronise in order to ensure they perform the `collect` commands in the *same order*. Whereas, without the atomic semantics Cases 1 and 3 are quite acceptable and *no* global synchronisation is required.

Case	Kernel process 1		Kernel process 2	
1	col(ts1, ts2, ?int)	col(ts1, ts3, ?int)	col(ts1, ts2, ?int)	col(ts1, ts3, ?int)
2	col(ts1, ts2, ?int)	col(ts1, ts3, ?int)	col(ts1, ts3, ?int)	col(ts1, ts2, ?int)
3	col(ts1, ts3, ?int)	col(ts1, ts2, ?int)	col(ts1, ts2, ?int)	col(ts1, ts3, ?int)
4	col(ts1, ts3, ?int)	col(ts1, ts2, ?int)	col(ts1, ts3, ?int)	col(ts1, ts2, ?int)

Table 5.1: Table showing how the `collect` primitive messages arrive at the two kernel processes.

The implementation difficulties of a particular set of semantics is not necessarily a valid reason to avoid using them. But, the semantics chosen in Chapter 2 not only appear natural within Linda where there is competition for tuples already between the different primitives, but also remove the necessity for a distributed kernel to perform a global synchronisation whenever a `collect` or `copy-collect` primitive is used. The *global* synchronisation of the kernel (or even just a subset of all the kernel processes which can contain the tuples) is an expensive operation, and should be avoided if possible.

5.7 Classification of tuple spaces

A more intelligent approach to implementing both the `collect` and `copy-collect` primitives is based on the premiss that they both provide information about where a tuple (or set of tuples) is likely to be used. The York Kernel II is able to use such information to provide an increase in performance over using the methods described so far. This is achieved by using a simple classification of tuple spaces, which the kernel is able to perform *without* the use of any pragmas or other explicit information added to a Linda program. A Linda program written for C-Linda using the

York Kernel I could use the York Kernel II without being altered, and providing the `collect` or `copy-collect` primitives are used a performance increase should be observed.

The fundamental difference between the York Kernel II and others is its ability to dynamically alter where tuple spaces are being stored and consequently move tuples around en-mass. This is achieved by *dynamically* watching where tuple space handles are, and subsequently tagging every tuple space in the system accordingly. Tuple spaces are tagged as either a *local tuple spaces* (LTS) or a *remote tuple spaces* (RTS) relative to a user process. The definitions are:

Local tuple space For a given user process a tuple space is said to be a *local tuple space* (LTS) if that process created the tuple space and a tuple containing the handle of that tuple space has neither been placed in a remote tuple space nor been passed as an argument to a spawned process.

Remote tuple space For a given user process a tuple space is said to be a *remote tuple space* (RTS) if the process did not create it or if the process did create it and a tuple containing the tuple space handle has either been placed in another RTS or passed as an argument to a spawned process.

There are three important points about the classification of tuple spaces:

- Firstly, not all user processes know about all tuple spaces. The assumption is that the classification of a tuple space relative to a user process must be achieved using only knowledge about the tuple space handles that are currently in scope in the user process and their history relative to the user process. Therefore, there is no need for "global" repositories of tuple space classifications.

- Secondly, if a tuple space is classified as a RTS relative to a user process then that tuple space will not be classified as a LTS relative to any other user process. This can be justified by considering how tuple space handles are passed between user processes. If process P_1 creates a tuple space T_1, the only way process P_2 can also know about T_1 is if P_2 has retrieved a tuple from a RTS (for example from `UTS`) that contains the tuple space handle of T_1 or if process P_1 spawns process P_2 and passes the tuple space handle as an argument. If the tuple space handle is passed through a tuple space then T_1 must be classified as a RTS relative to process P_1 because process P_1 has placed a tuple in a RTS containing the tuple space handle for T_1, and T_1 must be classified as a RTS relative to process P_2 because process P_2 has not created tuple space T_1. If process P_1 passes the tuple space handle of T_1 to process P_2 as an argument of the process then tuple space T_1 will be classified as a RTS relative to process P_1 because process P_1 has passed tuple space T_1 as an argument to a spawned process, and T_1 will be classified as a RTS relative to process P_2 because process P_2 has not created tuple space T_1. Therefore, it should be impossible for a tuple space to be classifies as a RTS relative to one user process whilst classifying it as a LTS relative to another user process. The assumption is made that each tuple space name is unique, so it is not possible for two user processes to create the same tuple space.

- Thirdly, the tuple spaces are classified by the *kernel*, *not* the programmer. From a programmers point of view there is no difference between a tuple space classified as a RTS and a LTS, they are both simply tuple spaces which are created and used in exactly the same way.

The underlying idea is that an LTS should be stored "as close as possible" to the user process which knows it as a LTS. A RTS should be stored in such a way that the cost of retrieving tuples from an RTS is minimised over all the user processes that can access it.

The classification of tuple spaces produces a better way of implementing the `collect` and `copy-collect` primitives. These bulk primitives are used to move or copy tuples from one tuple space to another. Tuple spaces are stored in the kernel using locality information. The implementations of the `collect` and `copy-collect` primitives implicitly harness this information about tuple spaces because the primitives move or copy tuples between tuple spaces. The aim is that the access costs of a moved or copied tuple should be less than that cost would have been if the tuples had not been moved or copied.

The dynamic classification of tuple spaces is the main technique behind the kernel implementation described in this chapter. The classification described here and used in the kernel is simple yet effective as the results in Chapter 6 show. In Chapter 7 the concepts of tuple classification are extended to produce a more graduated classification scheme. In the next section the general structure of the kernel is considered.

5.8 The York Kernel II

In this section a two layer hierarchical kernel, known as the York Kernel II, is described which supports the classification of tuple spaces. The kernel described here is an open implementation, supporting persistence and allowing processes to join and leave freely. The kernel requires no information provided by either special compilers or pre-processors and does not make any assumptions about the host languages being used. The classification of a tuple space is achieved by *dynamically* using only information gathered since the user process joined the Linda kernel. A process is said to have joined the kernel when the first Linda primitive is performed.

The architecture of the kernel is modelled around the concept of local and remote tuple spaces, creating a two layer hierarchy. An outline of the kernel architecture is shown in Figure 5.3. The kernel has two distinct sections, the *tuple space server* and a number of *local tuple space managers*.

Tuple Space Server A Tuple Space Server (TSS) is a dedicated *system* that exists to store and manage RTSs. The TSS can be a single process (a dedicated server) or it can be a set of processes. If the TSS is distributed then the distribution of tuples will be performed in a similar way to traditional implementations. In the York Kernel II the TSS is distributed. The

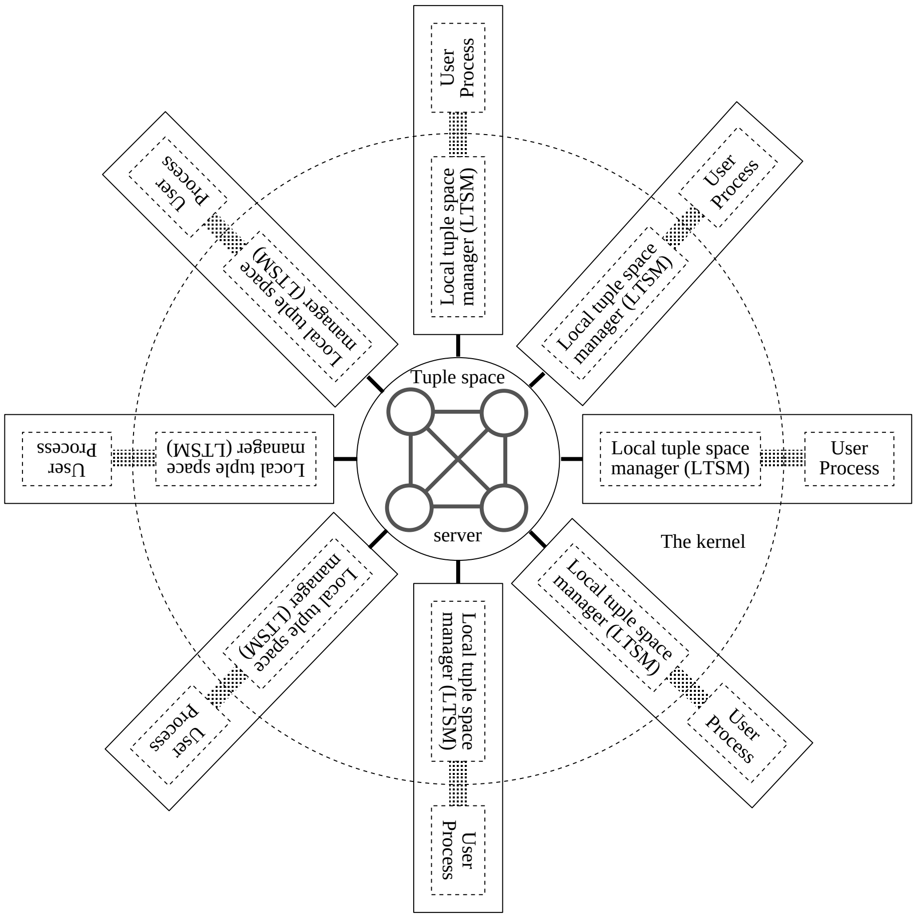


Figure 5.3: The York Kernel II architecture.

processes that create the TSS are referred to as TSS processes. Each of the TSS processes can *communicate* with other TSS processes and with all local tuple space managers.

Local Tuple Space Manager A Local Tuple Space Manger (LTSM) is attached to each user process. Each LTSM is distinct from the TSS but is aware of it, or if the TSS is distributed the LTSM is aware of all the TSS processes. However, the TSS is not explicitly aware of the LTSMs. This allows LTSMs to join and leave the kernel without affecting the kernel. Unlike the TSS the LTSM does not service remote requests, but produces the requests and accepts replies from requests *it* made. The LTSM also initiates all the movement of tuple spaces and packets of tuples within the kernel. The LTSMs are used to store LTSs and if a tuple space is a LTS then it is stored on a single LTSM. The LTSMs do not communicate with each other and do not share information in any way. Each LTSM is able to calculate *dynamically* the classification for a tuple space it is presented with, and is implemented as a set of library routines which the user process calls.

A LTS can never reside on the TSS and a RTS can never reside in a LTSM. Tuples that belong to a LTS, in general, cannot reside on a TSS and tuples that belong to a RTS cannot reside in the LTSM. The only exception to this is when the transfer of tuples occurs, when a tuple space changes from being a LTS to a RTS or a number of tuples are moved from a LTS to a RTS then tuples belonging to a LTS may briefly reside on a TSS and vice-versa. This will be discussed in detail later in Section 5.16.

When a user process performs a Linda primitive it calls a library routine in the LTSM. The LTSM decides, using the tuple space handle, whether it can satisfy the request locally, or if a message has to be sent to the TSS. With primitives that deal with single tuples (`out`, `in`, `rd` primitives) this is simple because there is only a single tuple space being used, and the tuple space either resides locally or on the TSS.

However, with the primitives that deal with more than one tuple space (`collect` and `copy-collect` primitives) the LTSM has to decide where the operations should be performed, which is dependent upon the classification of the source and destination tuple spaces. If the tuple spaces are not both classified as the same (LTS or RTS) then the tuples being copied or moved will move from a LTSM to the TSS or vice-versa. Whenever a set of tuples is to be moved from the TSS to a LTSM or vice-versa, the LTSM initiates the movement. Table 5.2 shows where the operations are performed with respect to the classification of both the source and destination tuple spaces.

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

Table 5.2: Table showing where the `collect` and `copy-collect` primitives are performed based on the classification of their source and destination tuple spaces.

The York Kernel II has been implemented on top of PVM[SDGM94] (Parallel Virtual Machine), which provides a mechanism to control the creation and subsequent communication between "processes". PVM uses a message passing paradigm, and provides an interface to TCP/IP communication between workstations on a LAN. PVM has been used so that the kernel can easily be ported, and used with a LAN of heterogenous workstations. In the following sections a more detailed account of some of the general methods used in the kernel, and how the bulk movement of tuples is used is presented.

5.9 Tuple Distribution within the kernel

The tuple distribution strategy within the kernel uses a two stage hashing method which is performed within the LTSM. The first stage uses the tuple space handle and produces a result of either LTSM or TSS. If the result is LTSM then the tuple is inserted into the local tuple storage data structure within the LTSM performing the hashing. If the result is TSS then a second hashing stage is used which is based on the technique of distributed hashing. Normally, when using this approach the hashing algorithm would produce a single TSS process for a tuple, and a non-empty set of TSS processes for a template (which could contain just one TSS process). However, in this implementation, only field type information is used, and the hashing algorithm always returns a set of TSS processes regardless of whether a tuple or template is used. The LTSM then picks one of the TSS processes represented in the set at random, and dispatches the tuple to it.

Within the TSS there is the possibility that individual tuples move from one TSS process to another TSS process because of the way that tuples are found when an `in` primitive is performed, and this is discussed in Section 5.13. There is no bulk movement of tuples from one TSS process to another TSS process.

5.10 Tuples and tuple storage within the kernel

The tuples are encoded as a sequence of bytes at run-time within the language interface of the LTSM (see Section 5.11). The first byte represents the number of fields present within the tuple, the second byte is used to indicate if there are any tuple space handles present within the tuple and then each tuple field is represented in the encoded tuple. Each tuple field is composed of a byte representing the type of the field, which is unique for each type, followed by two bytes containing the number of bytes needed to store the field value, and then the actual field value in as many bytes as required. A template is created in a similar manner, except whenever a formal is used in the template, the field length for that field is set to zero, and the tuple field value is omitted.

Using such an encoding for tuples and templates means that all the information required to perform the match in the kernel is present in the tuple and template. The matching algorithm is completely independent of the types used. Therefore, other language interfaces can be created, and as new types are needed, because the types that the language supports are more varied than the current types being used, there is no need to alter any other host language embeddings. If a particular language does not support a data type supported by other languages then that language will not be able to consume tuples which contain fields of the unknown type. Some care has to be taken to ensure the same type in two different languages[6] has the same type identifier, so the two languages can communicate values of that type through tuples. Although the matching algorithm is independent of the types used, the kernel is not independent of the types used, because it must

[6]or, indeed, in two embeddings of Linda in different implementations of the *same* language.

be able to "track" tuple space handles in tuples. The only type identifier the kernel is aware of is the type identifier for tuple space handles, which is fixed. Section 5.15 details how tuple space handles are tracked. If a pre-compiler was used then the tuples and templates could be partially encoded at compile time.

A graphical representation of the tuple storage data structure that is used in both the LTSM and the TSS processes is shown in Figure 5.4. The "tuple space" is represented by a record which contains the tuple space name, and a number of pointers to lists of tuples. Each pointer represents a list of tuples of a particular number of fields, with the final pointer pointing to a list of all tuples that do not belong to any of the other lists. It should be noted that there is a number associated with each list of tuples. This number represents the number of tuples *missing* from that list. This is used in order to maintain the `out` ordering when bulk tuple space operations are performed, tuple spaces moved or individual tuples migrated. This is explained in detail in Section 5.16.

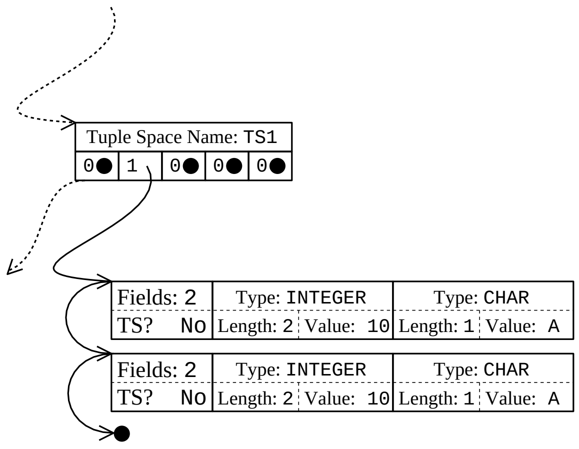


Figure 5.4: The tuple storage data structure used within TSS processes and the LTSM.

5.11 The Local Tuple Space Manager

The LTSM is the part of the kernel that is "included" in user processes. This means that the LTSM needs to be flexible enough to be included in different host languages. In order to do this two interfaces to the LTSM are used; called the *language interface module* and the *LTSM interface module*. The *language interface module* contains all the routines that the user processes call, and contains all the routines for the management of encoding and decoding of tuples to and from a form that the host language is capable of using. If tuples are first class objects in the language, as in ISETL, then the Linda primitives may return tuples. If they are not first class objects then the values within the encoded tuple have to be transfered to other variables, as in C-Linda.

The *LTSM interface module* provides routines for each of the Linda primitives. All the routines expect two strings, representing a tuple space name and the encoded tuple or template as appropriate. These representations should be independent of the host language used, as the same tuple specified in two different host languages will be encoded into the same string. Therefore, to embed the Linda primitives using the York Kernel II into a new host language, only a new language interface module has to be developed. A C language interface model has been developed.

One of the problems facing an implementor of a distributed *heterogeneous* system is representation of data types[ZG96]. This can vary from the ordering used to store bytes (eg. big endian and little endian) to the internal representations of "complex" types in different languages or compilers. For example, one language or compiler may use a different character to terminate a string compared with another language or compiler. The advantage of the kernel not depending on the types and their representations is that the kernel is not effected by the representation chosen by the language or compiler. It treats tuples and templates as a string of bytes. The only byte-ordering in tuples or templates of importance is the bytes which represent the length of the field (which are assumed to be stored as big-endian[7]). The structuring of fields is controlled by the language interface module. Whenever a new type identifier is added its representation is specified by the language interface implementor who adds it, and then future language interface modules must respect this. If a technique such as "receiver makes right"[ZG96] is adopted this could be incorporated within the language interface module.

The LTSM is also able, under some circumstances, to detect when a process deadlocks. Both the `in` and `rd` primitives block when there are no tuples available which satisfy the template used. When there is *never* a tuple that will satisfy the request, the user process executing the `in` or `rd` primitive will block forever. In closed systems using compile time analysis it is often possible to detect some of the primitives that will block forever and produce appropriate warning messages. If a user process is blocked, waiting for a tuple in a LTS, the LTSM can detect this. The LTSM knows that no other process can place tuples into a LTS, so the LTSM can terminate the user process, and produce an appropriate message. Current research[Men96] is examining ways of detecting deadlocks and more generally how to perform garbage collection within distributed open implementations.

5.12 Implementing the `out` primitive

When an `out` primitive is performed the LTSM checks the tuple space, into which the tuple is being inserted, to see if it is a LTS. If the tuple space is a LTS then the tuple is placed into the local tuple storage data structure. If the tuple space is not a LTS then a set of possible TSS processes are calculated using the second stage hashing. One of the calculated TSS processes is chosen at random and the tuple is dispatched to that TSS process. In order to ensure that `out` ordering is maintained the TSS process issues an acknowledgement which the LTSM must receive *before* the next tuple is inserted into a tuple space (or before a `collect` or `copy-collect` primitive is

[7]The most significant byte in the lowest numeric byte address of the two.

performed). The management of the acknowledgement messages is considered in Section 5.17.

5.13 Implementing the `in` and `rd` primitives

When either an `in` or `rd` primitive is performed the LTSM checks the tuple space, from which a tuple is required, to see if it is a LTS. If it is then the local tuple storage data structure is checked. If the tuple space is a RTS then the tuple will reside on a TSS process. The second stage hashing is used and provides a set of possible TSS processes on which a tuple that matches the template *could* reside. A matching tuple may reside on either all of these TSS processes, on some, or on none of them. Therefore, if a matching tuple exists it needs to be found.

There are several ways in which this could be achieved. The first is a broadcast from the LTSM to all the possible TSS processes. The LTSM would then act as the arbitrator, potentially receiving a number of tuples and then picking one, and returning the rest to the appropriate TSS processes. An alternative approach is to pick one of the TSS processes at random and then allow that one to arbitrate, which is the method adopted in the York Kernel II described in this chapter because it requires less communication and control.

5.14 Implementing the bulk primitives

When either a `collect` or `copy-collect` primitive is performed the LTSM checks the source and destination tuple spaces being used. If they are both LTSs then the LTSM performs the operation locally. If the source tuple space is a LTS and the destination tuple space is a RTS then the duplication occurs locally and the copied or moved tuples are dispatched to the TSS processes, in packets of *multiple* tuples.

If the source tuple space is a RTS then the operation will be performed on the TSS. The TSS processes which could contain matching tuples are asked to perform the operation by the LTSM. If the destination tuple space is a LTS then each of the contacted TSS processes creates a tuple space. The TSS processes perform the operation placing the tuples in the destination tuple space they have created, and returns a count of the number of tuples placed in the tuple space to the LTSM. Once the counts have been received then the bulk movement of the tuples is initiated by the LTSM from the TSS processes to itself. This is achieved by the LTSM requesting from each TSS process the destination tuple space. Each TSS process packs the tuples from the destination tuple space into packets, then removes the tuple space and sends the tuples to the LTSM which issued the request. If both the source and destination tuple spaces are RTSs, then the same operations are performed as when the destination tuple space is a LTS, except the tuples are not moved to the LTSM.

It appears more efficient if the TSS automatically packs and dispatches the tuples to a LTSM if the destination tuple space is a LTS. This was originally tried but was found to provide poor performance. The LTSM *has* to receive all the counts before the primitive can complete. The time taken to unpack the tuples is significant. Therefore, to receive all the packs of tuples from the TSS processes, unpack the tuples, and insert them into the tuple data structure takes a relatively long time, compared with unpacking a number of single integer messages and summing them. An optimisation was applied which is described in Section 5.17, which required each TSS process to send two messages, one containing the count and the other containing the tuples. However, the packet containing the tuples often arrived before all the count messages had arrived, which significantly reduced the effectiveness of the optimisation. This was overcome by requesting the tuples to be sent once all the counts had been received.

5.15 Tracking tuple space handles

In the descriptions of how the primitives are implemented, it is stated that the LTSM checks the tuple space handle to see if it is a LTS or a RTS. How does a LTSM know whether a tuple space is a LTS or a RTS? The LTSM tuple space classification is achieved by each LTSM monitoring two events; the creation of tuple spaces and the movement of tuple space handles in tuples going *out* of the LTSM to the TSS.

From the definition of an LTS, given earlier, a tuple space can *only* be a LTS if the handle for the tuple space has neither been placed in a RTS nor passed as an argument to a process, and the user process to which the LTSM is attached created the tuple space. The classification is performed by checking the local tuple storage data structure to see if an entry for the tuple space exists, and if it does the tuple space is a LTS, otherwise it is a RTS.

Whenever a tuple space is created it has to be a LTS. Therefore, when the procedure which initialises a tuple space (`tsc` primitive routine) in the LTSM interface module is called, it also creates an empty entry in the local tuple space data structure. In order to detect when a tuple space handle is leaving the LTSM checking is performed at two points. Firstly, when a tuple is being encoded (within the language interface module) and secondly when a bulk movement of tuples occurs. The LTSM interface module provides a routine which, when given a destination tuple space handle and a tuple space handle that appears in a tuple, checks to see if the destination tuple space is a RTS, and so converts the tuple space represented in the tuple to a RTS, if it is a LTS.

Tuple encoding Whenever an `out` primitive is performed the routine for the `out` primitive in the language interface module encodes the tuple. As each field is encoded it checks to see if the field is a tuple space handle, and if so the routine in the LTSM interface module is called with the destination tuple space handle and the tuple space handle in the field.

This ensures that if a tuple is inserted into a RTS, all the tuple spaces that are represented within the tuple become RTSs.

Bulk movement Whenever a tuple space or a set of tuples is being moved from the LTSM to the TSS each tuple has to be checked to see if it contains a tuple space handle. If any of the tuples contain handles to a LTS these tuple spaces need to be transfered to the TSS, and they then become RTSs. Figure 5.5 shows why this is necessary. The tuple space `UTS` is a RTS, and both the tuple space `TS0` and the tuple space `TS1` are LTSs. The tuple space handle for tuple space `TS1` is embedded within a tuple contained in tuple space `TS0`. A tuple is placed in tuple space `UTS` which contains the handle for tuple space `TS0`. At this point both the tuple space `TS0` *and* the tuple space `TS1` become RTSs. This is because the handle for the tuple space `TS1` now resides in a RTS, so the tuple space handle for `TS0` is present in a tuple in a RTS, so it must become a RTS.

When a tuple is inserted into a packet of tuples the tuple is checked. If a tuple has any tuple space handles within it, they are treated as though the tuple was being encoded. The LTSM interface module routine is called, and initiates any tuple space movement that is necessary.

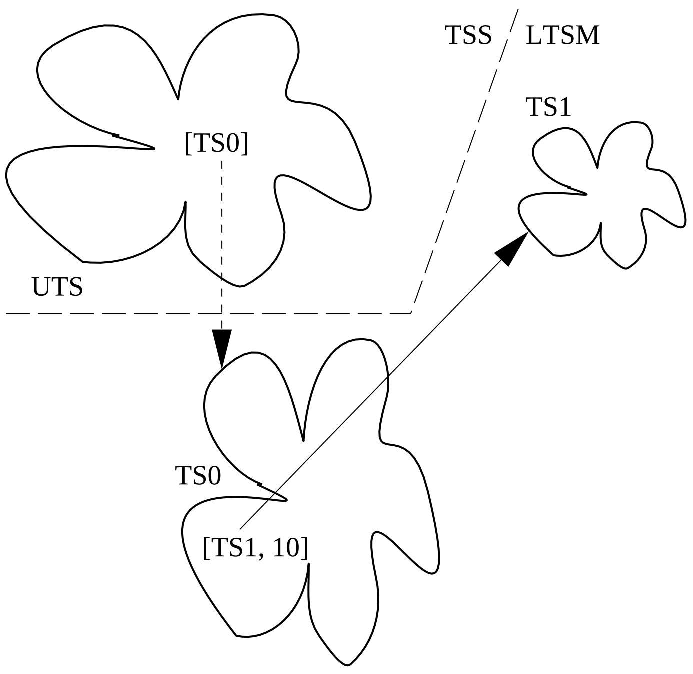


Figure 5.5: Tuple space handles embedded within tuple spaces.

Initial experiments showed that there are significant costs involved in checking each tuple to see if it contains tuple space handles when the bulk movement of tuples occurs. This, in conjunction with the fact that in most cases no tuples contained tuple space handles, led to the addition of a flag within the encoded tuple structure to indicate if one or more tuple space handles are present within the encoded tuple (see Section 5.10). Checking this flag means that the tuple can be quickly and efficiently checked for tuple space handles. If it is set then each field in the tuple can be checked.

In the original definitions of a LTS and a RTS indicated that if a tuple space handle was passed as an argument to a spawned process then the tuple space would be classified a RTS. The York

Kernel II does not support the passing of arguments to a process[8], therefore does not need to check for tuple space handles within the arguments. However, if the passing of parameters was supported, they would need to be checked for tuple space handles. The `eval` servers as proposed by Hupfer et al.[HKCG91] provide the passing of arguments to a spawned function via tuple spaces, and therefore the checking used in the encoding of the tuple would find the tuple space handle if it as passed as an argument, and convert the tuple space to a RTS if necessary. If tuple spaces were not used, then in the same way that the `out` primitive routine in the LTSM interface module checks the tuple being created, the `eval` primitive routine would have to check the arguments being passed. Because the tuple space handle is passed to the spawned process, the process does not create the tuple space, so it should not be present in the tuple storage data structure, and therefore the spawned process considers it a RTS.

The scheme adopted makes the introduction of "special" global tuple spaces easy. For example, the universal tuple space (`UTS`) is predefined in the Linda header files. The tuple space is not created by the user process so the LTSM does not insert it into its local tuple storage data structure and hence automatically treats it as a RTS. Because the TSS processes do not need a tuple space initialised within their tuple storage data structure, the TSS processes do not need to be informed of global tuple spaces. The tuple space handle name is manually chosen, and the tuple space name must *not* be one that the LTSM can generate. Pragmatically, this is simple to ensure due to the format of LTSM generated tuple space names.

5.16 The bulk movement of tuples

The need for `out` ordering has been reiterated in several places within this dissertation. How is the `out` ordering of tuples affected by the bulk movement of tuples? When tuples are in transit they are neither in the TSS nor a LTSM and this has serious implications if not managed properly. Consider the program fragment shown in Program 5.3, where it is assumed that tuple space `ts2` is a LTS for `process_one`, `ts3` is a LTS for `process_two`, and `ts1` is a RTS which is known to only `process_one` and `process_two`. It is also assumed that both functions are executing concurrently. The functions `process_one` and `process_two` synchronise using the tuple $\langle$"DONE"$_{string}\rangle$, and if `out` ordering is preserved the expected outcome is that the variable `n` in both of the functions will be the same value.

Because the function `process_one` performs a `collect` primitive from a LTS to a RTS, the LTSM attached to `process_one` will perform the operation and then dispatch the moved tuples to the TSS. If the `out` primitive "overtakes" the tuples whilst they are moving to the TSS processes, there is the chance that `process_two` becomes "unblocked" and then performs the `collect` primitive *before* the tuples have arrived at the TSS. The same problem exists if tuples are moved from RTSs to LTSs, and when whole tuple spaces are moved from a LTSM to the TSS.

[8]The passing of arguments is achieved by passing them in a tuple through a shared tuple space, such as `UTS`.

Program 5.3 Example of `out` ordering and the bulk movement of tuples.

```
int process_one(void)
{
....
n = collect(ts2, ts1, ?int, ?int);
out(ts1, "DONE");
......
}

int process_two(void)
{
......
in(ts1, "DONE");
n = collect(ts1, ts3, ?int, ?int);
......
}
```

Two mechanisms are used to ensure that the problem does not occur. When tuples are being transfered from the TSS to a LTSM a counter is used to indicate if tuples are expected. In Section 5.10 the data structure used to store tuples was discussed, and each list of tuples has a number associated with it, which is a counter used to indicate the number of tuples *missing* from the structure. Therefore, in Figure 5.4 one tuple containing two fields is missing. If a `collect` or `copy-collect` primitive is performed with a template with two fields, then the LTSM process will perform the operation on the tuples that reside in the tuple space, and *wait* for the remainder to arrive. If an `in` primitive or a `rd` primitive is performed and a tuple that has already arrived matches the template then that tuple is retrieved, and the user process continues. Tuples are only moved from a TSS to a LTSM after a `collect` or `copy-collect` primitive, which provides the number of tuples copied or moved, and is the same number of tuples that the LTSM expects to receive, and is used to set the counter.

The second mechanism is used when tuples are transfered from a LTSM to a TSS, either after a bulk primitive or a tuple space movement. The `out` ordering is guaranteed in the same manner as the `out` primitive, by using acknowledgement messages. Each packet that is dispatched to the TSS requires an acknowledgement before any operation that inserts tuples is performed on any tuple space.

5.17 Optimising the York Kernel II

There are two particularly interesting optimisations that are used to increase the performance of the York Kernel II. These are the optimisation of the `out` primitive and the tuple insertion in a LTSM.

5.17.1 `out` optimisation

In Chapter 2 `out` ordering was described. In order to achieve this each `out` primitive to a RTS requires an acknowledgement before the next tuple insertion operation to a RTS (or bulk movement to the TSS) is performed. A naive approach is to translate an `out` primitive to a RTS into a simple send message followed by a wait for an acknowledgement message from the TSS process. However, such an approach leads to `out` primitives taking nearly as long to execute as an `in` primitive.

In the York Kernel II the `out` primitive to a RTS sends the message, sets a flag to indicate that an acknowledgement message is expected, and then returns to the user's program. When an `out` primitive (or `collect` or `copy-collect` primitive) is performed all the packing of the message is completed so the message is ready for sending to a particular TSS process, before the acknowledgement flag is checked. If an acknowledgement message is required and has not arrived the system waits until one arrives. Even if there are two `out` primitives performed one after another there is an improvement in performance because the second `out` primitive performs all its preparation of the message before checking the acknowledgement flag. The bulk movement of tuples from the LTSM to the TSS also requires an acknowledgement, and the same flag (and hence optimisations) are used for these acknowledgements.

5.17.2 LTSM tuple insertion optimisation

Whenever multiple tuples are moved around the system, they are moved in packets containing many tuples. When either a LTSM or a TSS receives a packet containing many tuples the obvious approach is to unpack them and insert them into the tuple storage data structure.

The LTSM does not do this, it attempts to lazily unpack the packet of tuples as and when the tuples are needed. It is often the case that the template used with a `copy-collect` or `collect` primitive is used with subsequent primitives. Hence, the first tuple unpacked from the packet will match the template used in the primitive. Therefore, instead of always inserting the packet when a tuple is requested the LTSM checks to see if a packet of tuples is available for a particular tuple space. If so, the LTSM takes a tuple from the packet and matches it with the template. If they match, then no further tuples are unpacked and the user's program continues. If the tuple does not match then it is inserted in the tuple storage data structure and the next tuple in the packet is checked. If another Linda primitive is used which requires a message to be returned from a TSS process, then all the tuples in the packet are inserted in the tuple storage data structure.

This approach is never slower than inserting the tuples directly into the tuple storage data structure when the packet of tuples arrives, and as will be seen in Chapter 6, can provide a significant speed increase.

5.18 Why classify tuple spaces?

Why does the ability to make a simple classification of the tuple spaces lead to an increase in the performance of the kernel? The ability to classify a tuple space as a LTS or RTS provides no speed increase in a parallel program using primitives for simple synchronisation with “single” tuples, as the bulk movement of tuples does not occur. The advantages become apparent as data structures are stored as tuples in tuple spaces where the bulk movement of tuples is used implicitly. Individual tuples are still used to provide control in the programs, such as to indicate that worker processes have completed, but collections of tuples are required for processing.

The bulk movement of tuples may initially appear an expensive operation, but there are two attributes which make the bulk movement of tuples and tuple spaces advantageous: the control of packet size and less communication.

Control over packet size

The first advantage is the ability to control the packet size. When a set of tuples are being moved from a TSS to a LTSM or vice-versa it is possible to control how many tuples are packed into a single packet (or indeed how many bytes are packed into a single packet). With many communication mechanisms the time taken to send a packet is not linear with respect to the packet size. When considering an Ethernet based LAN the sending of small packets across it is a more expensive operation than sending larger packets. Therefore, the ability to bulk move tuples is cheaper than moving a set of tuples one at a time.

In order to show the advantages of controlling the packet size, the characteristics of the Ethernet used at the University of York are considered. The characteristics measured are derived using test programs written in PVM[SDGM94], which the York Kernel II uses for communication. The measurements were produced by passing messages between two Silicon Graphics Indy workstations, connected using a 10 Megabit per second non-dedicated Ethernet.

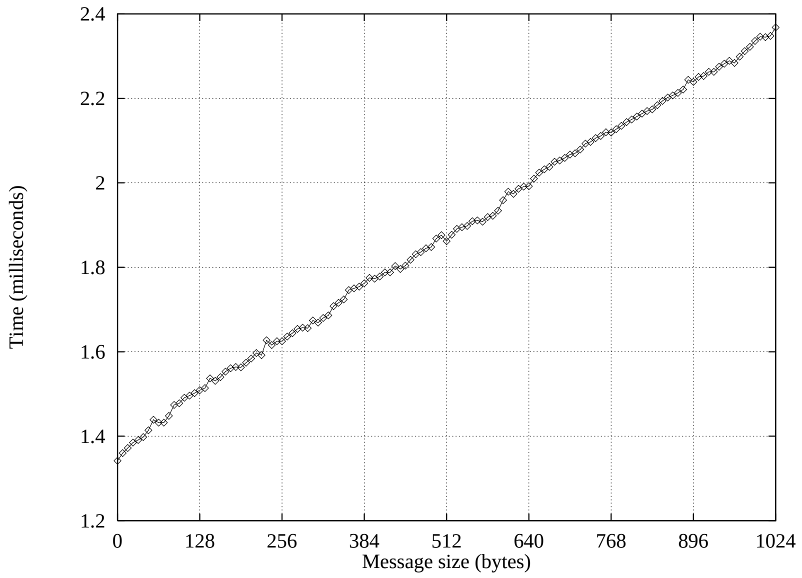


Figure 5.6: Latency for messages up to 1024 bytes in size using PVM.

Figure 5.6 shows the time taken to send messages of between 0 bytes and 1024 bytes, and Figure 5.7 shows the time taken to send a message of between 128 bytes and 100 Kilobytes in size between the two workstations. Figure 5.8 shows the bandwidth in megabytes per second that is achievable for messages of sizes between 0 bytes and 1024 bytes, and Figure 5.9 shows the bandwidth for messages of between 128 bytes and 100 Kilobytes in size. These charts clearly show that as the message size increases the bandwidth increases, so more data can be transfered per second across the network as the message size increases. A packet size of about 30 Kilobytes provides the best bandwidth.

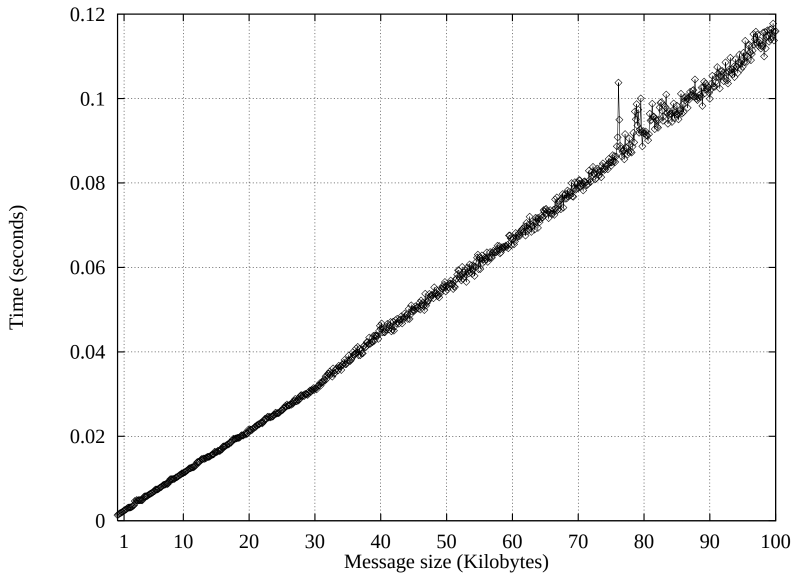


Figure 5.7: Latency for messages up to 100 Kilobytes in size using PVM.

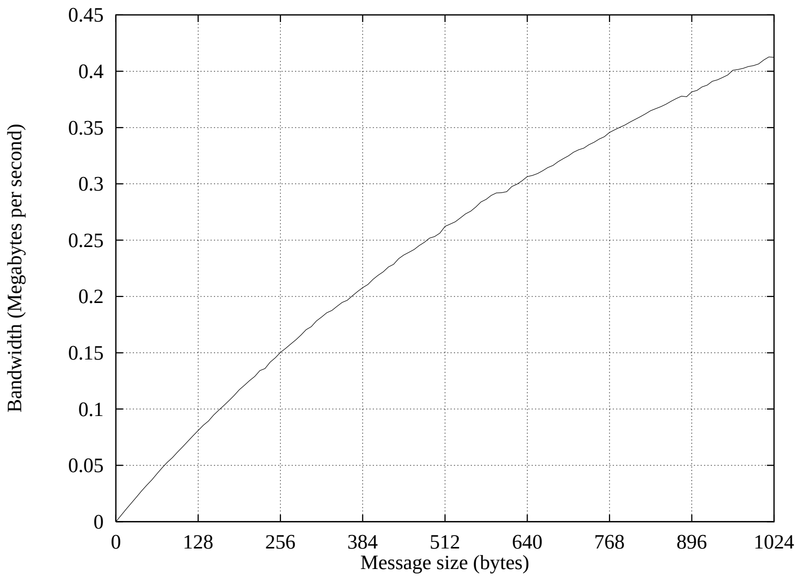


Figure 5.8: Bandwidth versus messages size up to 1024 bytes.

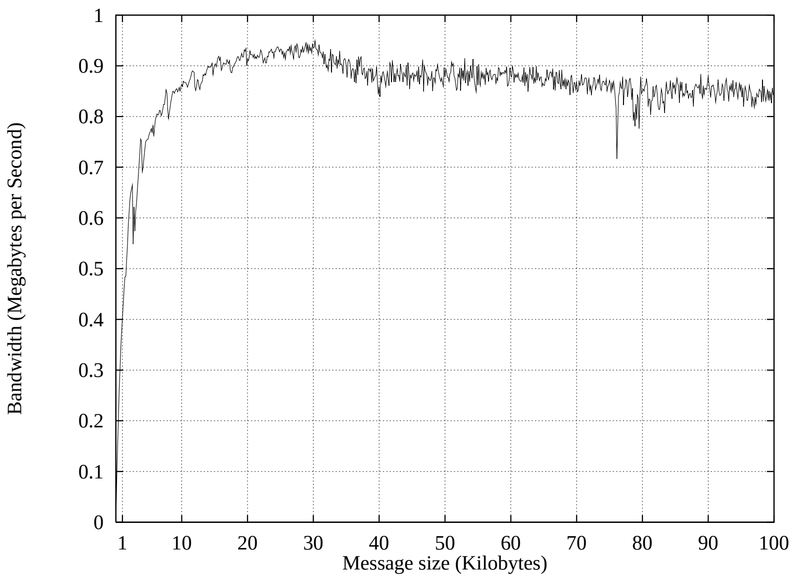


Figure 5.9: Bandwidth versus messages size up to 100 Kilobytes.

Figure 5.10 shows the time taken for a single byte to be transfered given a particular message size between 1 byte and 1024 bytes, and Figure 5.11 shows time taken for a single byte to be transfered given a particular message size between 128 bytes and 100 Kilobytes in size. These graphs show that the cost of sending a byte is reduced significantly as more bytes are packed into a single message.

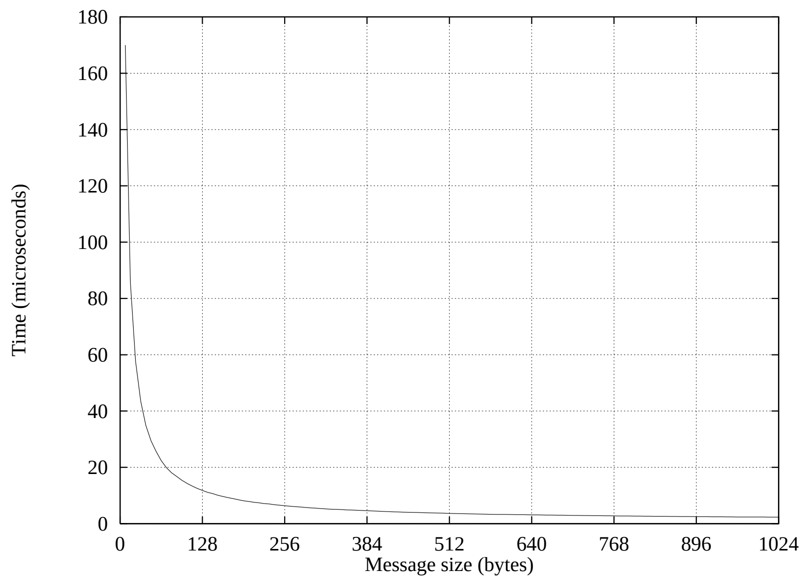


Figure 5.10: The time taken to send a single byte for messages size up to 1024 bytes.

In many Linda programs the size of individual tuples can be small, containing just a few fields occupying less than 50 bytes[9]. The actual message dispatched through the network will require certain other information, such as the destination tuple space or the source tuple space. Even with this information the entire message will be less than 100 bytes. Therefore the cost of sending these little tuples is very large compared with the cost packing them into packets and then sending them. Therefore, the ability to pack several tuples (or indeed several hundred tuples) into a single message allows the message sizes to be increased achieving better communication performance.

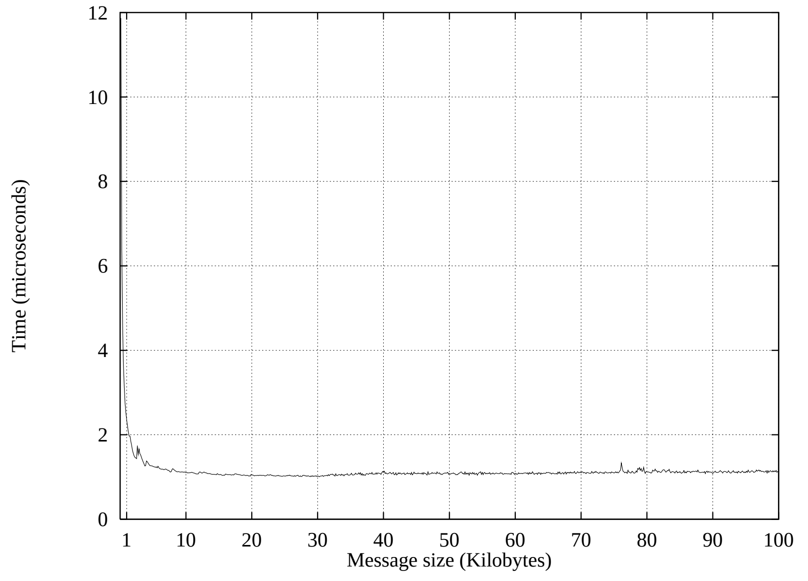


Figure 5.11: The time taken to send a single byte for messages size up to 100 Kilobytes.

[9]A tuple containing seven integers would require approximately 50 bytes.

The advantage gained is dependent on the network used. If the network characteristics are such that the cost of sending a single byte is virtually independent of the packet size, the bulk movement of tuples is still favorable because less communication is required.

Less communication

The second advantage is that less communication is required. If tuples are moved to where they are to be used the amount of communication necessary is reduced. Every `in` primitive performed on a RTS requires *at least* two messages. The first message sends the template to the TSS process and then a second message is required to return the tuple to the calling user process. There may be further communication required between the TSS processes to find a matching tuple. Therefore, if N tuples are required from a RTS, there will be at least $2 \times N$ messages between the user process and the TSS. If the tuples are moved to a LTSM (say, using the `copy-collect` primitive), and then read from a LTS, the number of messages required will be 4 times the number of TSS processes on which the tuples may reside. The first message sent to each TSS process will be the request to perform the operation, then the second message is from each TSS process returning a count of the number of tuples duplicated. Then the LTSM will send a message requesting the tuples and finally the TSS processes send a message containing all the tuples. When the actual `in` primitives are performed there will be *no* communication with any other process, as the tuples are stored in the LTSM. The `in` primitives call a function that will search a local data structure stored within the LTSM which is part of the process. Therefore, given T TSS processes, the N tuples would require only $4 \times T$ messages. This could potentially be reduced to $2 \times T$ messages if the message containing the count of the tuples copied or moved, and the message containing the tuples is compressed into one message, saving a message, and there would be no need for a request message for the tuple space.

A similar reduction in communication is observed if a tuple is inserted into a LTS rather than a RTS. The insertion of a tuple in a RTS requires two messages, the message to the TSS process and the acknowledgement message. For N tuples to be inserted $2 \times N$ messages are required to insert them in a LTS. If the tuples are inserted in a LTS, at most $2 \times T$ messages are required, the message with tuples in it and an acknowledgement message from each TSS process sent a packet. An empty packet is never sent, so if only one tuple is inserted into a LTS which then becomes a RTS, there will be only 2 messages: a packet containing one tuple and an acknowledgement message from the TSS process receiving the packet. The reduction in the number of messages is shown in Table 5.3.

In systems where the cost of sending a byte in a message is independent of the message size there will still be an improvement in performance because less communication between processes is needed, and the total number of bytes in all the messages for a particular operation when they are bulked moved will be less than if they were moved as individual tuples[10]. Only if a communication system has characteristics which make the cost of sending a byte in a packet cheaper than accessing

[10]Provided the tuples are used.

LTSM	N `in` primitives	N `out` primitives
Enabled	$4 \times T$ (inc. `copy-collect` primitive)	$2 \times T$ (worst case) 2 (best case)
Disabled	$2 \times N$	$2 \times N$

Table 5.3: Table summarising the reduction in communication when using the LTSM (where T is the number of TSS processes).

local memory will the bulk movement of tuples not provide an improvement in performance.

5.19 Conclusions

In this chapter the novel techniques used within the York Kernel II have been described. The kernel uses an efficient method of deciding where tuple spaces should be stored, based on implicit information gathered at run time, requiring no explicit information to be added to a Linda program. The ability to classify tuple spaces allows the effective implementation of the bulk primitives of `collect` and `copy-collect`.

One issue not addressed so far is what happens if a processor does not have sufficient physical memory to store an entire LTS? For example, when the number of tuples being transfered from a RTS to a LTS makes the LTS too large to be stored by the LTSM. The York Kernel II assumes this cannot happen, and as the workstations being used support virtual memory this has not proved a problem. However, if a device with less memory was attached to the kernel it could be a problem. A LTS can be seen as a special case of a RTS. Any tuple space that is a LTS *could* be converted to a RTS, which is potentially distributed over many processes, even if only one process can access it. Therefore, if a tuple is being inserted into a LTS using an `out` primitive and there is insufficient memory, the LTS can be made into a RTS, and then the tuple inserted as though it was a RTS. The `collect` and `copy-collect` primitive implementations could be altered to return the size of the copied or moved tuples as well as the number copied or moved tuples. The LTSM can therefore decide if it has enough space to store a LTS, or whether the LTS should be transfered to a RTS.

York Kernel II has been built on top of PVM[SDGM94]. In the next chapter the performance of the York Kernel II is evaluated. In Chapter 7 the shortcomings of the current York Kernel II are described and the technique of classifying tuple spaces is extended to overcome some of the shortcomings of the York Kernel II.

Chapter 6

Performance of the York Kernel II

6.1 Introduction

This chapter evaluates the performance gains that are achieved by the kernel outlined in the previous chapter. A number of simple experiments are used to show the performance advantages that the bulk movement of tuples can achieve by comparing the performance of a number of common coordination patterns using the York Kernel II with the LTSM enabled and disabled. When the LTSM is disabled the kernel degenerates into a traditional implementation with all the tuple spaces being stored on the TSS and no bulk movement of tuples, similar to the York Kernel I[DWR95] and other such implementations.

The performance of a "real world" example, the Hough transform, with the LTSM enabled and disabled is also shown. The Hough transform is a common image processing algorithm. Using this example the performance of the York Kernel II is also compared with the performance of SCA C-Linda, a commercial *closed* implementation which uses compile-time analysis.

6.2 Experiments

All the experiments are conducted using a number of Silicon Graphics Indy workstations using a 10 Megabit per second non-dedicated Ethernet connection. The York kernel II is initialised so that all the tuples are distributed across all the workstations used. Unless otherwise stated, all experiments used tuples containing a single integer so matching the template $\langle| \square_{integer} |\rangle$. Also the test processes are the only processes using the kernel and all the execution times stated in this chapter are given in seconds and represent "wall clock" timings.

The Ethernet used is non-dedicated and subsequently other users effect the time it takes to send messages over it so the experimental results were all gathered during the early hours of the morning when the Ethernet load should be minimal. Six experiments are used to show the performance of the York Kernel II. The C-Linda source code for the experiments is given in Appendix B. These six experiments are:

Experiment one This experiment shows that when a tuple space is a RTS the use of the LTSM does *not* effect the execution times. In order to show this 1000 tuples are placed in the RTS `UTS` using the `out` primitive, and then all 1000 are retrieved from the `UTS` in any order using the `in` primitive.

Experiment two This experiment shows that when a tuple space is a LTS the use of the LTSM provides faster access to the tuples within the tuple space. In order to show this 1000 tuples are placed in a LTS using the `out` primitive and then all 1000 tuples are retrieved from the LTS in any order using the `in` primitive. This experiment is similar to experiment one, except that the tuple space is a LTS rather than a RTS.

Experiments one and two are designed to show that the LTSM does not provide an overhead for the access of RTS, and that the speed of LTS access is improved by using the LTSM. The next four experiments show that the bulk movement of tuples (and tuple spaces) between the LTSM and the TSS is effective.

Experiment three This experiment shows that the movement of entire *tuple spaces* between the LTSM and the TSS provides better performance than not moving them. In order to show this 1000 tuples are placed into a LTS using the `out` primitive. A tuple containing the handle of this LTS is then placed into a RTS (`UTS`) using the `out` primitive. The movement of the entire tuple space occurs when the tuple space changes from being classified as a LTS to a RTS. Subsequently, all 1000 tuples are retrieved from the tuple space (which has become a RTS), into which they were placed, using the `in` primitive.

Experiment four This experiment shows that the movement of multiple *tuples* from a LTSM to the TSS provides better performance than not moving them. In order to show this 1000 tuples are placed into a LTS using the `out` primitive, and are then copied using the `copy-collect` primitive from this tuple space to a RTS (`UTS`). Then all 1000 tuples are retrieved from the RTS, into which they were copied, using the `in` primitive.

Experiment five This experiment shows that the movement of multiple *tuples* from the TSS to a LTSM provides better performance than not moving them. In order to show this 1000 tuples are placed into a RTS (`UTS`) using the `out` primitive, these are then copied using the `copy-collect` primitive from this RTS to a LTS. All 1000 tuples are then retrieved from the LTS, into which they were copied, using the `in` primitive.

Experiments three, four and five are designed to show that the bulk movement of tuples and tuple spaces provides a performance increase. The final experiment is designed to show the performance achievable when multiple block movements of tuples are performed one after another.

Experiment six This experiment shows that bulk movement of tuples can be performed one after another, moving multiple tuples from a LTSM to the TSS and back to the LTSM. In order to show this 1000 tuples are placed into a LTS using the `out` primitive, and they are then moved using the `collect` primitive from this tuple space to a RTS (`UTS`). The tuples are then copied using the `copy-collect` primitive back from the RTS to a LTS and finally all 1000 tuples are retrieved from the LTS into which they were copied, using the `in` primitive.

In all the experiment descriptions whether a tuple space is a RTS or a LTS is explicitly stated. There is no distinction as far as the programmer is concerned. Each tuple space is simply a tuple space (see source code in Appendix B).

6.2.1 Experimental results

Experiment one

Table 6.1 shows the execution times taken to perform the two operations: the insertion of the tuples, and the removal of the tuples. The experiment is performed with both the LTSM enabled and disabled using a number of different workstation configurations. In each case the number of workstations used indicates the number of machines over which the tuples stored in a RTS are distributed. Figure 6.1 shows a graphical summary of these results.

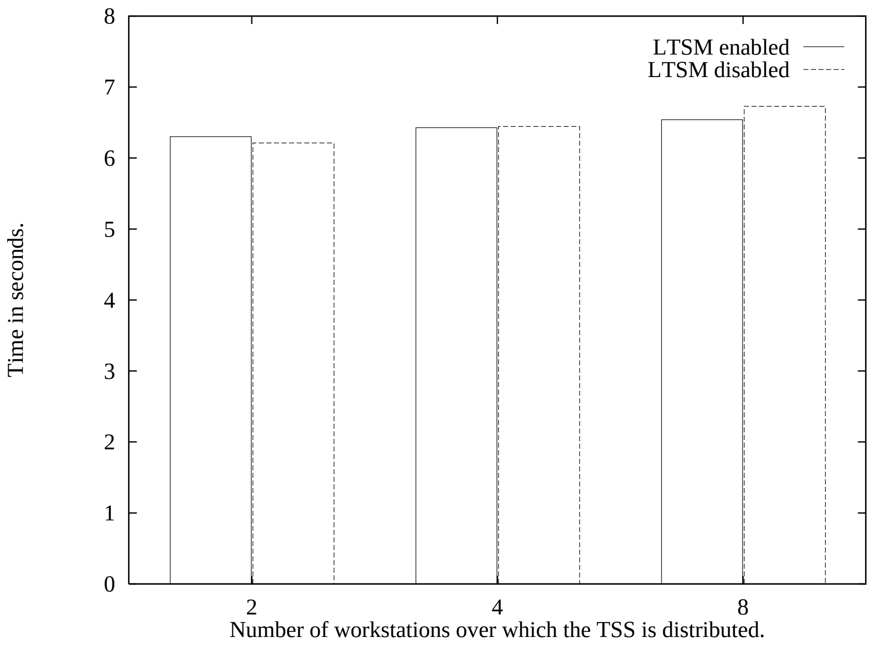


Figure 6.1: Summary of the results of experiment one.

LTSM	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
	Two workstations		Four workstations		Eight workstations	
1000 `out`	2.970	3.026	3.062	3.049	3.191	3.013
1000 `in`	3.243	3.275	3.384	3.379	3.538	3.526
Total	6.213	6.301	6.446	6.428	6.729	6.539

Table 6.1: Experiment 1 - Accessing a RTS with the LTSM enabled and disabled.

Experiment two

Table 6.2 shows the execution times taken to perform the two operations: the insertion of the tuples, and the removal of the tuples. The experiment is performed with both the LTSM enabled and disabled using a number of different workstation configurations. Figure 6.2 shows a graphical summary of these results.

LTSM	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
	Two workstations		Four workstations		Eight workstations	
1000 `out`	2.975	0.018	2.908	0.019	2.975	0.019
1000 `in`	3.270	0.019	3.380	0.018	3.513	0.019
Total	6.245	0.037	6.288	0.037	6.488	0.038

Table 6.2: Experiment 2 - Accessing a LTS with the LTSM enabled and disabled.

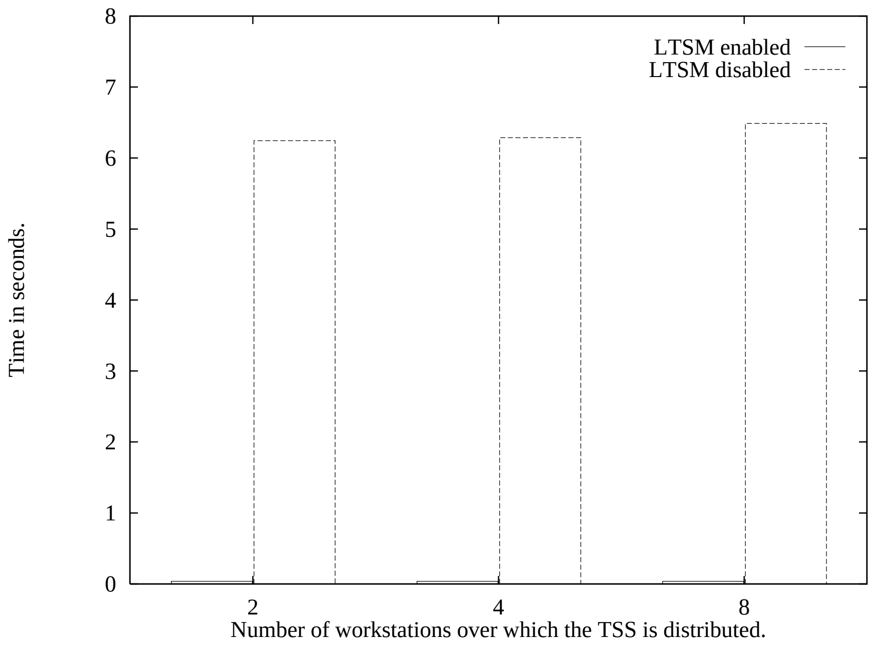


Figure 6.2: Summary of the results of experiment two.

Experiment three

Table 6.3 shows the execution times taken to perform the three operations: the insertion of the tuples, the placing of the tuple into `UTS`, and the removal of the tuples. Figure 6.3 shows a graphical summary of these results.

LTSM	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
	Two workstations		Four workstations		Eight workstations	
1000 `out`	2.889	0.019	2.816	0.018	2.940	0.019
1 `out`	0.003	0.044	0.003	0.056	0.003	0.081
1000 `in`	3.265	3.912	3.400	3.639	3.524	3.827
Total	6.157	3.975	6.219	3.713	6.467	3.927

Table 6.3: Experiment 3 - Changing a tuple spaces classification from a LTS to a RTS.

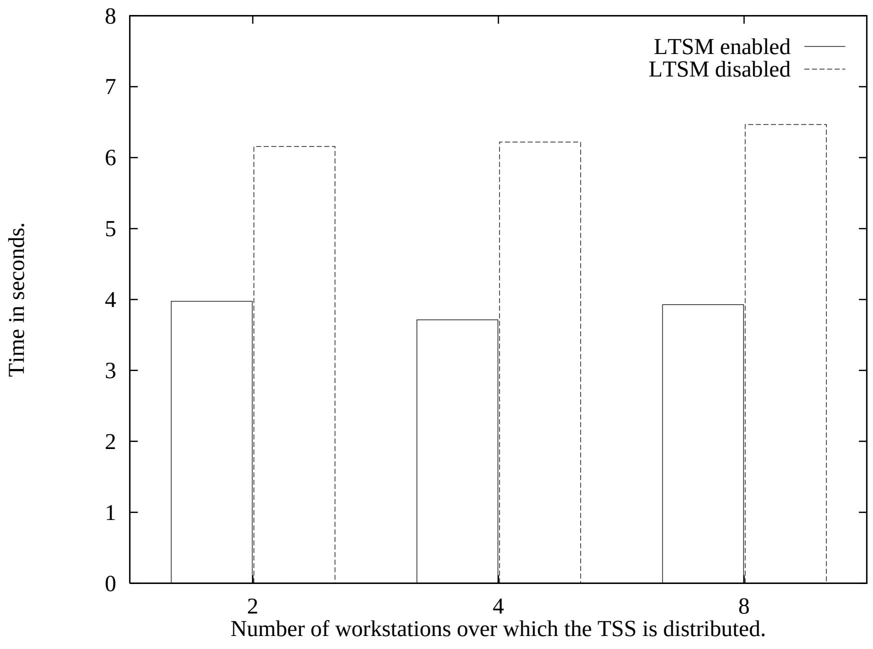


Figure 6.3: Summary of the results of experiment three.

Experiment four

Table 6.4 shows the execution times taken to perform the three operations: the insertion of the tuples, their duplication, and the retrieval of the copied tuples. Figure 6.4 shows a graphical summary of these results.

LTSM	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
	Two workstations		Four workstations		Eight workstations	
1000 `out`	2.797	0.018	2.818	0.019	2.919	0.018
`copy-collect`	0.015	0.054	0.011	0.064	0.010	0.093
1000 `in`	3.248	3.921	3.286	3.752	3.382	3.755
Total	6.060	3.993	6.115	3.835	6.311	3.866

Table 6.4: Experiment 4 - Moving tuples from a LTS to a RTS using the `copy-collect` primitive.

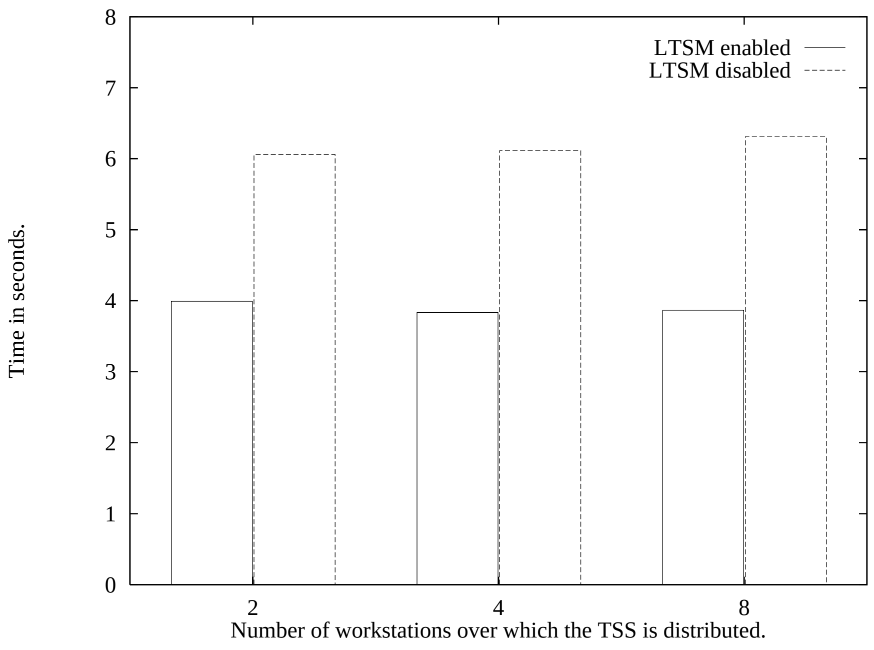


Figure 6.4: Summary of the results of experiment four.

Experiment five

Table 6.5 shows the execution times taken to perform the three operations: the insertion of the tuples, their duplication, and then retrieval from the LTS. Figure 6.5 shows a graphical summary of these results.

LTSM	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
	Two workstations		Four workstations		Eight workstations	
1000 `out`	2.870	2.924	2.864	2.885	2.911	2.902
`copy-collect`	0.014	0.014	0.011	0.012	0.010	0.013
1000 `in`	3.297	0.068	3.407	0.067	3.538	0.058
Total	6.181	3.006	6.271	2.964	6.459	2.973

Table 6.5: Experiment 5 - Moving tuples from a RTS to a LTS using the `copy-collect` primitive.

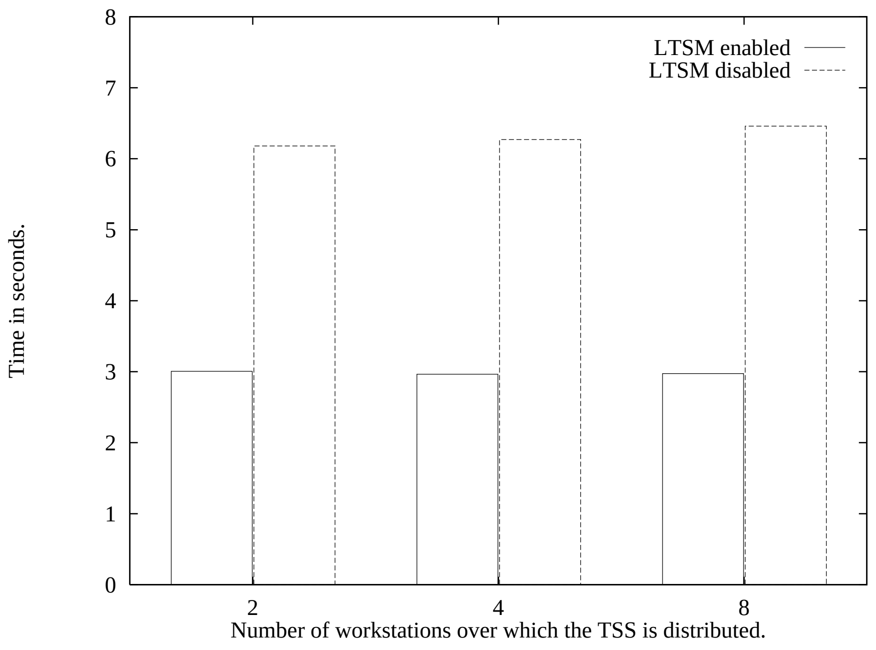


Figure 6.5: Summary of the results of experiment five.

Experiment six

Table 6.6 shows the execution times taken to perform the four operations: the insertion of the tuples, their movement, duplication and their retrieval. Figure 6.6 shows a graphical summary of these results.

LTSM	Disabled	Enabled	Disabled	Enabled	Disabled	Enabled
	Two workstations		Four workstations		Eight workstations	
1000 `out`	2.967	0.019	2.884	0.018	2.952	0.019
`collect`	0.003	0.052	0.003	0.070	0.003	0.085
`copy-collect`	0.007	0.705	0.007	0.361	0.008	0.285
1000 `in`	3.345	0.063	3.441	0.057	3.576	0.058
Total	6.322	0.839	6.335	0.506	6.539	0.447

Table 6.6: Experiment 6 - Moving tuples from a LTS to a RTS and then back to a LTS.

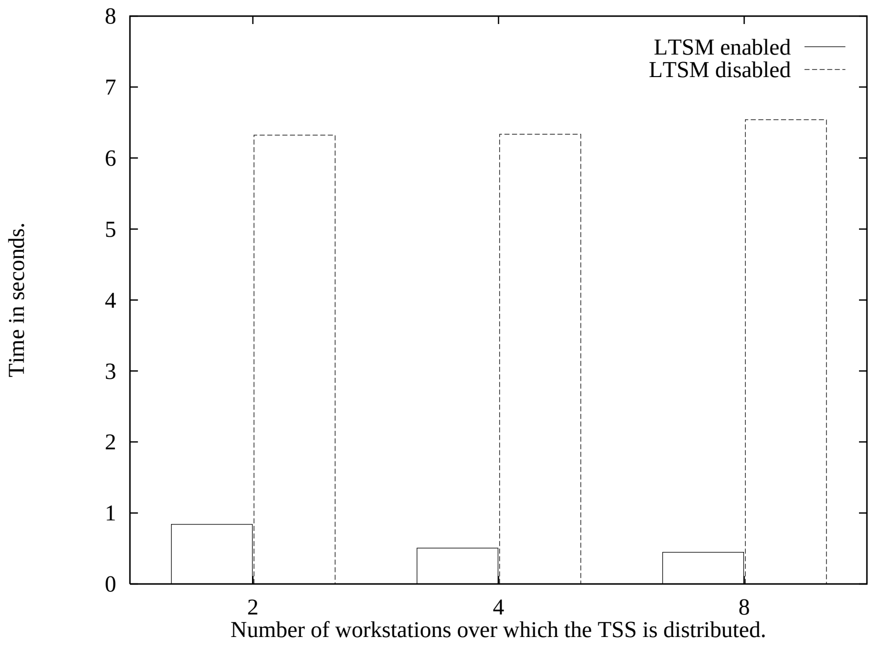


Figure 6.6: Summary of the results of experiment six.

6.2.2 Experimental conclusions

Experiments one to six - combined results

Figure 6.6 shows a graphical summary of all the results for experiments one to six.

Experiment one

The results show that when the LTSM is enabled it has no effect on the time taken to perform operations on a RTS. The time to perform the 1000 `out` primitives is almost constant regardless of the number of workstations over which the RTS is distributed, but the time taken to perform the 1000 `in` primitives increases as the number of workstations over which the TSS and consequently

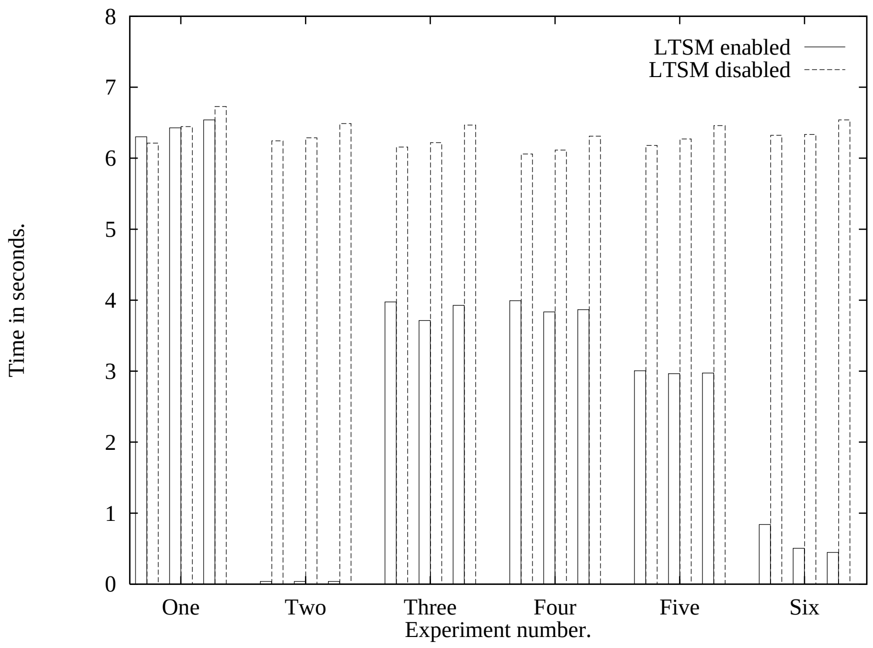


Figure 6.7: Summary of the results of all six experiments.

the RTSs are distributed is increased. This is as expected because every time an `out` primitive is performed it selects a single TSS process and sends a "message" to it. Whether there are two TSS processes or eight TSS processes the time taken to send the message is the same, but the time taken for an `in` primitive is dependent on the number of TSS processes on which a matching tuple can reside. This is because when a tuple is not available then other TSS processes, that could potentially contain a matching tuple, are contacted to see if they have a matching tuple. The more potential TSS processes that could store a matching tuple the higher the communication costs required to find a tuple. For more details on how an `in` primitive works see Section 5.13.

Experiment two

The results when the LTSM is disabled are similar to the results obtained in experiment one. This is as expected because when the LTSM is disabled all the tuples are stored on the TSS, and there is no distinction between a LTS and a RTS in terms of accessing them, so the execution times should be the same as in experiment one. Again, as the number of workstations increases the time to perform the 1000 `out` primitives remains constant and the time taken to perform the 1000 `in` primitives increases slightly. When the LTSM is enabled the execution time for both the 1000 `out` primitives and the 1000 `in` primitives is independent of the number of workstations the TSS is distributed over and is many times faster than when the LTSM is disabled. This is because the operations are local to the process and require no communication with any other process, either on the same workstation or on other workstations. The speedup provided by the LTSM is approximately 165 times than when the LTSM is disabled.

Experiments one and two show both the speedup achievable by using the LTSM for accessing tuples stored in LTSs and that the LTSM does not reduce the access times for tuples stored on the TSS in RTSs. The fast insertion and retrieval of tuples from a LTS stored in the LTSM shown does not show how such an approach improves the performance for communicating processes. The results for the next four experiments show that the bulk movement of tuples (and tuple spaces) between the LTSM and the TSS is effective, which is required if processes are to communicate.

Experiment three

All the results when the LTSM is disabled are similar to the results for experiments one and two when the LTSM is disabled. This is because the single extra `out` primitive, which places the tuple containing the handle for the tuple space in UTS, takes little time (less than 0.003 seconds), and therefore the operation is comparable to performing 1000 `out` primitives to a tuple space stored on the TSS, and then retrieved from the TSS. When the LTSM is disabled there will be *no* bulk movement of tuples from the LTSM, and there is no bulk movement of tuples between different TSS processes.

When the LTSM is enabled the total execution time of the test program is approximately a third faster than when the LTSM is disabled. This is because the tuples are placed into a LTS which is stored in the LTSM, and then the entire tuple space is bulk moved on to the TSS when the handle for the LTS is placed in a RTS (`out(uts, ts)` is performed). The times taken to execute this `out` primitive and the subsequent 1000 `in` primitives show that the single `out` primitive takes longer than when the LTSM is disabled, and the time increases as the number of workstations that the TSS is distributed over increases. This is because the time taken for the single `out` primitive is the time taken to send the `out` message to the appropriate TSS process (as when the LTSM is disabled) *and* to pack the all the tuples in the tuple space and dispatch them to the appropriate TSS processes.

The time the `out` primitive takes is dependent on the number of TSS processes because the more TSS processes there are, the more messages need to be prepared and dispatched, and the time cost of creating the messages and initialising the sending of more messages takes longer. The time taken to unpack the tuples within the TSS processes does not effect the time of the `out` primitive because the LTSM does not “pause” until all the acknowledgements from the TSS processes are received (see Section 5.3 for more details of the acknowledgements required). The extra time taken in performing the 1000 `in` primitives is attributable to the *first* `in` primitive performed, as the `in` primitive cannot be performed until all the acknowledgements from the TSS processes despatched packets of tuples have been received. A TSS process dispatches an acknowledgment *before* placing the tuples in its tuple storage data structure. Once the acknowledgements are received by the LTSM it is able to send the template for the first `in` primitive to an appropriate TSS to perform the tuple retrieval. This request for a tuple is only serviced by the TSS process when it has finished inserting the tuples into its data structure.

In theory a trade off could be used here, the more TSS processes the smaller the number of tuples dispatched to each of them, and the larger the communication costs because, as the Ethernet performance charts (in Chapter 5) show, as the package size decreases the effective bandwidth available decreases. But, the fewer tuples in a packet the less unpacking is required by the TSS process receiving the packet, and the TSS processes will be able to service the next request sooner. Pragmatically it is difficult to determine exactly what time costs are attributed to each operation at such a level. It is plausible that in this experiment the best performance is achieved when the tuple space is transferred to the four TSS processes. In this case a packet size of about 2.2 Kilobytes[1] is sent to each of the TSS processes containing 250 tuples. The time taken to dispatch and unpack the tuples is less than sending a single packet of 9 Kilobytes and unpacking 1000 tuples or eight packets of 1.1 Kilobytes containing just 125 tuples.

The important point is that the results show that the bulk movement of a tuple space does not impede the performance of the kernel, and indeed provides a speed up of at least 1.5 times over using only the TSS to store all the tuples.

Experiment four

Regardless of whether the LTSM is enabled or disabled the execution times in each case are similar to the execution times for the same case in experiment three. As with the last experiment when the LTSM is disabled there is no bulk movement of tuples between the LTSM and TSS processes, and between TSS processes.

The similarities between these results and those of experiment three are expected as fundamentally the same operations are being performed, except that instead of the 1000 tuples being moved as a "tuple space" they are moved as a 1000 tuples. As with experiment three the time taken to perform the operation that initiates tuple movement (the `copy-collect` primitive) and the 1000 `in` primitives is larger where the LTSM is enabled than when it is not. The reason for this is the same as in experiment three when the `out` primitive and the first `in` primitive took longer. However, the time taken to perform the `copy-collect` primitive is always slightly longer than the time taken to perform the `out` in experiment three. The difference is accounted for by the time cost of performing the match between the template and every possible tuple in the tuple space. When a LTS handle is placed in RTS the entire LTS is moved and there is no matching of tuples. When a `collect` or `copy-collect` primitive is used matching of each potential tuple has to be performed.

Experiment five

The execution times when the LTSM is disabled are comparable to those in experiment four. This is as expected because when the LTSM is disabled the program used in this experiment and

[1]A single tuple containing just an integer will be coded into a structure about 9 bytes long. Therefore, 1000 tuples at 9 bytes is 9000 bytes divided evenly between the four TSS processes being used.

experiment four are the same because regardless of whether a tuple space is a RTS or a LTS, it is stored on the TSS.

When the LTSM is enabled the placing of the tuples into a RTS takes the same time as when the LTSM is disabled. When considering the movement of tuples from a LTSM to the TSS there are two identifiable operations incurring extra time overheads due to the bulk movement. These are the instructions that initiate the movement of the tuples and the first `in` primitive performed on the moved tuples. The time cost of performing the `copy-collect` primitive appears significantly less than the time cost of performing the `out` or `copy-collect` primitives in experiments three and four. The reason for this is that duplication of the tuples is performed in the TSS processes concurrently and the count returned. The user process can then continue after requesting that the tuples be sent (see Section 5.17 for more details) whilst the tuples are being packaged and transmitted to the LTSM by the TSS processes.

The 1000 `in` primitives take only approximately 0.04 seconds longer than the equivalent operations in experiment one. When the tuples are moved from a LTSM to the TSS (experiment three and four) the difference between the LTSM enabled and disabled times to perform the 1000 `in` primitives is approximately 0.4 seconds. This would imply that a bulk movement of tuples takes approximately 0.4 seconds. If the time taken to perform the 1000 `in` primitives when the LTSM is enabled in this experiment is compared with the same operation in experiment two, the difference is only approximately 0.04 seconds. This implies that overheads of bulk moving tuples between the TSS and LTSM is only 0.04 seconds. Why does the bulk movement of tuples from the TSS to the LTSM appear to be have lower overheads for the same number of tuples, particularly considering the time to perform the `copy-collect` primitive is less than to perform the equivalent tuple movement initiating primitive in experiment three and four? There are two factors that justify this, firstly the LTSM does not wait until all the tuple packets from the TSS processes are received, as the first packet received has a matching tuple. Secondly the first tuple removed from the first packet matches the template. Therefore the tuples are never inserted into the tuple storage data structure (as described in Section 5.17). The time the 1000 `in` primitives take drops when the number of TSS processes increases because the more TSS processes, the less tuples are stored on each one, so the time taken for the TSS processes to pack the tuples is less. Thus the first packet of tuples will arrive at the LTSM sooner, so a match can be found sooner.

This experiment is of particular interest as it contains the coordination pattern that is performed when solving the multiple `rd` problem using the `copy-collect` primitive. The use of the `copy-collect` primitive to get a copy of all the required tuples followed by the consumption of all of them using the `in` primitive with the same template as used in the `copy-collect` primitive.

These experiments show the basic performance of the York Kernel II when multiple tuples are moved. The LTSM has been shown to improve the performance of the kernel. The sixth experiment shows the performance achievable when multiple block movements are performed one after another.

Experiment six

When the LTSM is disabled the results are very similar to the other experimental results involving the placement and removal of 1000 tuples from a tuple space which is stored on the TSS. The difference between the time taken to perform the `collect` primitive and to perform the `copy-collect` primitive shows time overheads associated with duplicating the tuples rather than just altering the tuple space to which they belong. When tuples are duplicated memory has to be assigned and the tuple physically duplicated (see Section 5.6).

When the LTSM is enabled the execution times display a number of interesting characteristics. The time taken to perform the `copy-collect` primitive is significantly longer than in experiment five where a `copy-collect` primitive is performed involving the bulk movement of tuples from the TSS to a LTSM. This is because the `copy-collect` primitive requires a message from each of the TSS processes which could contain tuples that match the template (in this case all of them). Therefore, the time taken by the `copy-collect` primitive represents the time taken for the tuples to reach the TSS process (initiated by the `collect` primitive), to be unpacked, duplicated and the count of tuples duplicated and return the count to the LTSM. As the number of TSS processes which store the tuples increases, the `copy-collect` primitive takes less time, because as the number of the TSS processes increases, the number of tuples that each receives decreases. The time spent by the TSS processes inserting them into the tuple data structures, matching them and duplicating them also decreases. As with experiment five once transfer to the LTSM has been initiated by the `copy-collect` primitive the 1000 `in` primitives take a comparable time to the same operation in experiment five, which would be expected.

Concluding remarks

The experimental results show the performance of the York Kernel II with respect to its ability to move multiple tuples in a single operation. The experiments are specifically designed *not* to test the template and tuple matching, or the underlying tuple storage data structures. Although the kernel is designed to be efficient, the more efficient ways of performing these operations using pre-compiler support have not been considered. The kernel is designed to show how the bulk movement of tuples can be used to improve performance. The experimental results show that the use of the LTSM does not degrade the performance of the kernel when using RTSs and increases the performance of the kernel when using LTSs. The experiments show sets of operations which, from experience, appear common in Linda programs, which use multiple tuple spaces and the `collect` and `copy-collect` primitives.

The experiments compare the performance of the kernel when the LTSM is enabled and disabled, but do not consider the performance against other implementations. Even when the LTSM is disabled the performance is better than the York Kernel I[DWR95, RDW95]. Although many of the underlying implementation decisions were based on methods used in the York Kernel I: York Kernel I uses a two stage hashing algorithm for the placement of tuples rather than a one stage hashing algorithm. The format of the tuples used in the York Kernel II are far simpler[2] than the format used in York Kernel I, so the matching of templates against tuples is faster and a better tuple storage data structure is used in the York Kernel II. York Kernel II also gains performance by using PVM efficiently, but the LAN version of the York Kernel I is a port[RDW95] of an existing implementation developed for a Transputer based system, and so it does not use PVM to the full. An example of the performance of the York Kernel I is given in Table 6.7 which shows the results for experiment one. The results should be compared with the results for experiment one for the York Kernel II (Table 6.1). The time taken to perform the 1000 `out` primitives is significantly less than for the York Kernel II, because the PVM port of the York Kernel I does not fully support `out` ordering[3] and so does not use acknowledgements for `out` primitives as does the York Kernel II implementation. This means that only a single message is required for every `out` primitive rather than the two messages required in the York Kernel II.

	Using a TSS distributed over two workstations
1000 `out`	1.654
1000 `in`	8.391
Total	10.045

Table 6.7: The performance of York Kernel I when placing tuples in a RTS or a LTS.

Comparison between York Kernel II and other implementations that are open systems is difficult for several reasons. There are few kernels publicly available and those kernels that are not publicly available quote results using different workstations and networks. This makes the comparison of results impossible. The public versions available are PLinda[Jeo96, JS94] and Glenda[SAB94], but both implementations use a centralised TSS rather than a distributed TSS as used by the York Kernel II, so any comparison is biased in favour of the York Kernel II. PLinda is designed to show fault tolerant techniques which degrades performance further by requiring it to regularly take snapshots of the tuple spaces. The performance the York Kernel II is better than Glenda and should be better than PLinda, even with the LTSM disabled because they use centralised TSSs. In order to show this the execution timings for Glenda to place a 1000 tuples into a RTS using the `out` primitive and then retrieve them using the `in` primitive (experiment one) is given in Table 6.8.

The results show the performance of the single TSS process both on the same workstation that the test program is being executed, and on a different workstation. Table 6.8 shows that the performance of the Glenda performing the `out` primitives exceeds that of the York Kernel II. This

[2]York Kernel I allowed embedded data structures within tuples, therefore tuples can contain sets, tuples, bags, etc; and matching can be performed on any field either within the tuple or the data structure within the tuple.

[3]The Mieko CS-1 provided synchronous communication, whereas PVM provides asynchronous communication. Out ordering is guaranteed if synchronous communication channels are provided.

	TSS resident on	
	same workstation	different workstation
1000 `out`	1.541	1.084
1000 `in`	6.088	12.763
Total	7.629	13.847

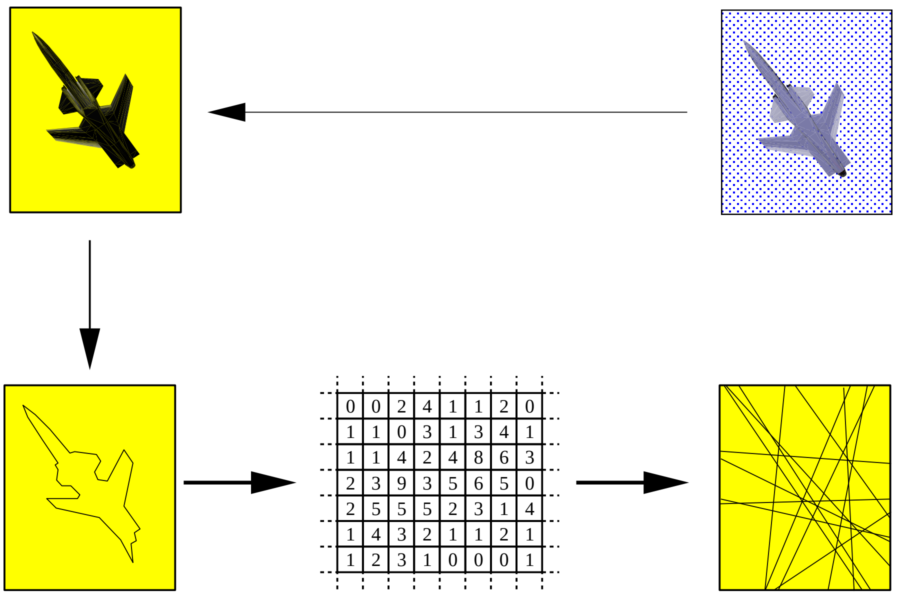
Table 6.8: The performance of Glenda when placing tuples in a RTS or a LTS.

is because Glenda uses a centralised TSS, and PVM which guarantees message ordering between two processes. Therefore, an acknowledgment message for each `out` primitive is not required, because the message for an `out` primitive is guaranteed to arrive at the centralised TSS *before* the next message sent from a process. Hence, as long as the centralised TSS process services the messages in the order they arrive at the TSS `out` ordering is guaranteed. The time taken to perform the 1000 `in` primitives is greater than the time taken using the York Kernel II because the tuple storage mechanism and the matching of tuples with templates is inefficient. Glenda is also built on top of PVM and, as with the York Kernel I, it is not used in an efficient manner.

Owing to the lack of suitable open implementations, in the next section, the performance of the York Kernel II is compared with the performance of SCA C-Linda. This is a commercial C-Linda produced using the techniques developed at Yale. It is probably the best implementation currently available, primarily because it is a closed implementation and uses compile time analysis to gain performance. SCA C-Linda does not support the `collect` and `copy-collect` primitives, and does not have multiple tuple spaces. So to compare the performance of the two systems a "real world" case study is used. This is a commonly used method from the image processing field, the Hough transform.

6.3 The Hough transform

The Hough transform[DH72] is an image processing algorithm, and is referred to as an intermediate-level vision operation[WMM$^+$92]. The Hough transform is used to detect straight lines (and other shapes such as circles) in binary images. Figure 6.8 shows its role in a general image processing system. The initial image (top-right) is a grey-scale image of an aeroplane in flight. The grey scale image is then processed, using a thresholding algorithm[SSWC88], to produce a binary image from the grey-scale one. The thresholding process attempts to distinguish the sky and the aeroplane, by making the aeroplane pixels set and the sky pixels unset. This is represented as the next image in Figure 6.8. The next stage is to detect the outline of the plane. This can be achieved using a number of techniques, including 4/8-connectedness[DF86]. These stages are all described as low-level image processing. If the aeroplane is to be detected as an aeroplane, some sort of object matching (a high-level operation) has to be performed. The image is transformed to the Hough space (or Parameter space), and this is used to determine straight lines in the image. The last two images in Figure 6.8 show these stages. This final stage is further processed and used as the input to an object matching process, which will identify the image as an aeroplane (hopefully).



0	0	2	4	1	1	2	0
1	1	0	3	1	3	4	1
1	1	4	2	4	8	6	3
2	3	9	3	5	6	5	0
2	5	5	5	2	3	1	4
1	4	3	2	1	1	2	1
1	2	3	1	0	0	0	1

Figure 6.8: Processing an aeroplane image.

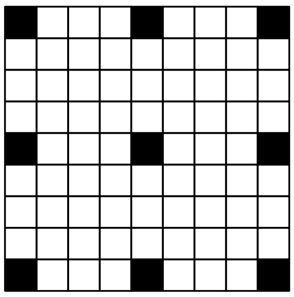
The work described here focuses on the basic Hough transform for detecting straight lines and a more formal overview of it is now given. The Hough transform maps a binary image pixel, (x, y), in the *coordinate space* (the binary image) to a sinusoidal curve in the *parameter space* (or *Hough space*). This sinusoidal curve is 'plotted' as a set of coordinates in the parameter space. The version of the Hough transform which is used is to detect straight lines in images is described by Equation 6.1, where (ρ, θ) pairs represent solutions of the equation given a specific (x, y).

$$x \cos\theta + y \sin\theta = \rho \tag{6.1}$$

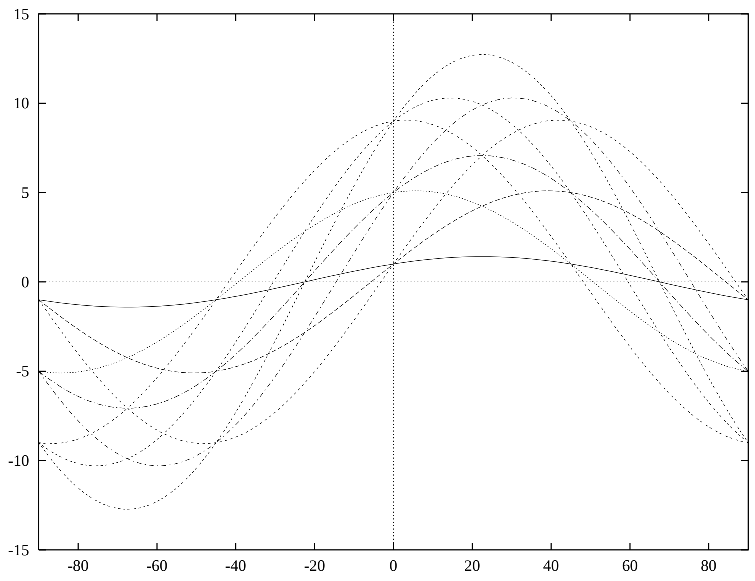
Therefore, for each pixel (x, y) in the coordinate space the set of (ρ, θ) pairs define a sinusoidal curve in the parameter space. The range of θ is $\pm 90^\circ$ and the range of ρ is the range D to $-D$ where D is defined in Equation 6.2. The granularity of the parameter space is controlled by controlling the quantisation of θ, and throughout all the experiments a granularity of $\pm 1^\circ$ is assumed. Therefore for every pixel in the coordinate space the Equation 6.1 produces 180 (ρ, θ) pairs.

$$D = \max(x, y, \frac{\sqrt{2}}{2}x + \frac{\sqrt{2}}{2}y) \tag{6.2}$$

Given two points in coordinate space, the equation of the line joining them is determined by the point of intersection of their corresponding curves in parameter space. The (ρ, θ) value at the intersection is substituted into Equation 6.1 creating a function that describes a straight line in the coordinate space. In order to detect the straight lines in an image, the parameter space is examined for intersections, and the number of curves that intersect at a particular point is equal to the number of image pixels in the coordinate space lying on that line in the coordinate space. The Hough transform therefore consists of two stages: the transformation from coordinate space to parameter space; and the subsequent processing of the parameter space. Only the transformation stage will be considered. Figure 6.9(b) shows the resulting parameter space for the simple image shown in Figure 6.9(a). A more detailed description of the Hough transform can be found in [GW87].



(a) Coordinate space



(b) Parameter space

Figure 6.9: A simple image and its Parameter space after the Hough transform has been applied.

6.4 Parallel decomposition of the Hough transform.

Much work has already been done on the parallel decomposition (using a data parallelism style) of the Hough transform[WMM$^+$92, YA85]. The method adopted here is the data parallel approach proposed by Yalamanchili et al.[YA85]. There are two primary ways of implementing a data parallel Hough transform. The first is to divide the coordinate space into segments then have a worker process for each segment. The worker takes each pixel in the segment it is responsible for and calculates the entire curve it produces in the parameter space. The second is to divide the parameter space into segments and then have a worker process for each of those segments. Every pixel within the coordinate space is processed by every worker, but only the part of the curve which bisects the segment of the parameter space which the worker process is responsible for, is calculated.

Segmenting the coordinate space creates two problems. The first problem is load balancing. The pixels within an image are not always distributed evenly over all the segments and in most cases they are unlikely to be. The execution time of each worker process is proportional to the number of pixels within the segment of the image for which it is responsible. Therefore, there is the possibility that some workers will have no work to perform, whilst others will have significantly more than the average. The second problem is that all the workers want to update the parameter space. During the execution of the Hough transform many of the positions within the parameter space are updated by many different workers, causing contention problems. The implementation within a Linda context produces no particular difficulties owing to the nature of the blocking primitives, but the load balancing problems are significant within a Linda context.

Segmenting the parameter space means that each worker process only updates a small segment of the parameter space, to which it will have exclusive access, but every process has to read every pixel in the coordinate space in order to create its segment of the parameter space. Figure 6.10 shows the parallel decomposition of the problem within a Linda context. The image tuple space contains the image, in the form of triples, $\langle x - coordinate_{integer}, y - coordinate_{integer}, pixel - value_{integer}\rangle$. The image is a binary image, so the $pixel$-$value$ will always be either one or zero. The image could be stored using just pairs, $\langle x - coordinate_{integer}, y - coordinate_{integer}\rangle$ with the assumption that if the tuple is present the pixel is set and if it is not set then the pixel is not present[RW95]. It is assumed here that the all the pixels are represented in the tuple space. In general this approach is better because the producer of the image tuple space may not know which processes are going to use it. Some processes may want the pixels with zero values, which is costly to determine if only the tuples which represent set tuples are present. In Figure 6.10 there is a second tuple space which is used to store the parameter space and four worker processes. Each worker is required to *read* all the image tuples with a *pixel-value* of one, and update the quarter segment of the parameter space for which it is responsible. The *reading* of the tuples by all four processes produces the *multiple* `rd` *problem*. Therefore, each process takes a copy of the tuples representing the tuples which are set and then destructively removes them from the copy it has taken and updates its segment of the parameter space accordingly.

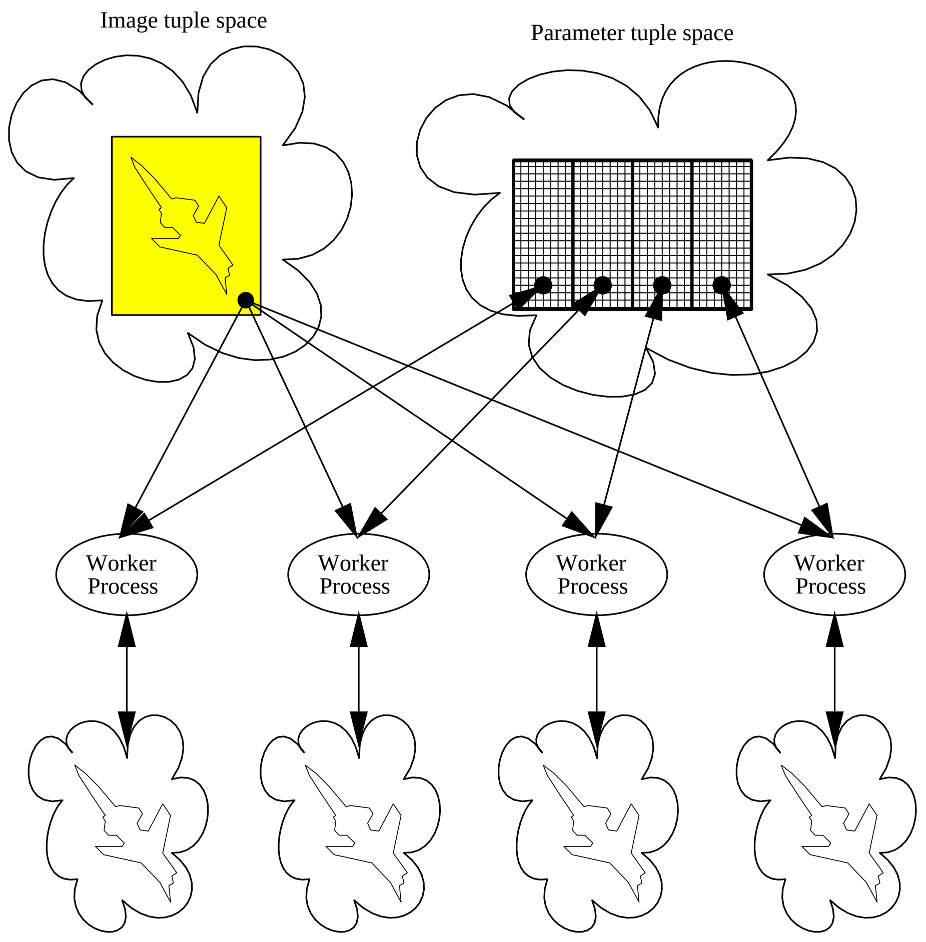


Figure 6.10: The parallel decomposition of the Hough transform

In the implementation, the parameter space quarters are created using local data structures within the worker process, and then transfered to the parameter tuple space after all the image pixels have been processed. Processing of the parameter tuple space cannot start before a segment has been fully completed so this is an appropriate approach. Within the parameter tuple space the parameter space is stored as a sparse data structure. Tuples within the parameter space are triples of the form $\langle \rho_{integer}, \theta_{integer}, counter_{integer} \rangle$. If the *counter* is to be zero for a particular (ρ, θ) pair then the tuple is not present. The parameter space is processed in such a way to detect "peaks", as they represent the lines with the most pixels on. Therefore, points with a zero count are not normally required in further processing.

The SCA C-Linda implementation has been implemented slightly differently, because it does not support the `collect` and `copy-collect` primitives or multiple tuple spaces. The different tuples spaces are emulated by tagging the tuples with a string to represent the tuple space to which they belong. The SCA C-Linda uses the stream approach to overcome the *multiple* `rd` *problem*.

6.4.1 Experimental results

The experimental results show two things. One is the effect of using the bulk movement of tuples for performance gains. The other is to reinforce the *multiple* `rd` *problem*, and how the `copy-collect` primitive overcomes it.

The first set of results show the effect of using bulk movement of tuples. In this case every pixel in the image is set to one. In Chapter 4 the time cost (in terms of primitives) of using the stream method and the `copy-collect` method for overcoming the multiple `rd` problem are compared. This showed that the number of primitives required by the stream method is $N \times \texttt{rd}$ (Equation 4.1). The number of primitives required for the `copy-collect` method is $\texttt{copy-collect} + n \times \texttt{in}$ (Equation 4.4). When all the tuples in the tuple space are required (which is when all the pixels are set), $n = N$. Therefore, the stream approach should be slightly more efficient, requiring one less primitive. The other question is the cost of performing an `in` primitive and a `rd` primitive similar? The difference is that the amount of tuple template matching is less in the `copy-collect` method because the `in` primitive matches all the tuples copied by the `copy-collect` primitive. Therefore, every tuple retrieved requires only one tuple template matching operation, but whenever a `rd` primitive is used, potentially many tuples have to be matched against a template before a suitable match is found. In the SCA C-Linda very efficient hashing algorithms are used (created at compile time) which means that the number of tuples actually checked when a `rd` primitive is performed should be very small. Combining this with the ability to place tuples based on the compile time analysis should make the cost of performing the `rd` primitive as efficient (if not more efficient) than the York Kernel II performing an `in` primitive on a tuple space stored on the TSS. By comparing the time taken when all the pixels are set provides an acceptable way of comparing the performance of the York Kernel II and the SCA C-Linda.

Table 6.9 shows the execution times in seconds for the Hough transform using four Silicon Graphics Indy workstations connected by a 10 Megabit per second non-dedicated Ethernet when all the pixels are set in a 256x256 binary image. The execution times represent the time to create the image tuple space; perform the Hough transformation of it; and produce a tuple space containing the parameter space. The time cost of spawning the worker processes is not included because of the difficulties in determining this for the SCA C-Linda[4]. For all the parallel versions four workers are used, one placed on each of the four Silicon Graphics Indy workstations. For the York Kernel II the TSS is distributed over the four workstations as well. The table shows the time taken for the SCA C-Linda, York Kernel II with the LTSM enabled and disabled, and two sequential versions using York Kernel II.

The sequential version using setup one uses a single machine for the kernel and the program. The sequential version using setup two has the kernel distributed over four workstations and the program running on one of those. The sequential version is identical to the parallel version except no workers are spawned, the function that is spawned in the parallel versions is instead just called

[4]In the case of the York Kernel II the time to spawn four processes is under 5 seconds.

in the sequential versions. The execution times of the sequential versions are very similar. With a single process accessing the tuple space, the speed gains of having the tuple space distributed over four workstations is lost in the communication overhead of sending packets around the system.

	SCA	York Kernel II with LTSM		Sequential version	
	C-Linda	disabled	enabled	Setup 1	Setup 2
256x256 image 100% pixels set (65536 pixels)	523.20	670.48	56.32	68.70	72.02

Table 6.9: Comparing the performance of the bulk movement of tuples with traditional techniques.

The difference in execution times between the SCA C-Linda and LTSM disabled kernel represents the performance improvement SCA C-Linda achieves through using compile time analysis (about 28% speed increase). Also both the SCA C-Linda and the LTSM disabled kernel take considerably longer to execute than the sequential versions. When the LTSM is enabled the performance of the kernel improves significantly, providing over a 900% speed improvement over the best of the other parallel versions. It also produces an execution time that is less than both the sequential versions. Table 6.10 shows the speed up of the LTSM enabled version over the other versions.

	York Kernel II with LTSM enabled speedup against			
	SCA C-Linda	LTSM disabled	Sequential	
			Setup 1	Setup 2
256x256 image 100% pixels set (65536 pixels)	9.3	11.9	1.2	1.3

Table 6.10: Speedup over other implementations when using the York Kernel II with the LTSM enabled for the parallel Hough transform when all pixels in an image are set.

The maximum performance increase achievable for this parallel algorithm, as more processors are added, is linear speedup, therefore the speedup of the four worker parallel version over the sequential versions, in the best case, should have been four times. There are several reasons why this does not occur. Primarily, all the worker processes are performing the same operations on the tuple space at the same time. All the primitives want to copy the matching tuples at approximately the same time. Each of the TSS processes receives several `copy-collect` messages almost simultaneously, and as it can only service one at a time a bottleneck occurs for a small period. Secondly, the costs of communication increases as the number of processes increase. Thirdly, the algorithm is fine grained and not ideally suited to a network of workstations. The computation costs are small compared with the communication costs. What is significant is the speedup of the York Kernel II with LTSM enabled over it with the LTSM disabled and the SCA C-Linda implementations.

Further experimental results

As already stated the parallel implementation suffers from the multiple `rd` problem. In Chapters 3 and 4 the parallel composition of binary relations was used to show the performance differences between the different approaches to overcome the multiple `rd` problem. The Hough transform provides another opportunity to show the effectiveness of both the `copy-collect` primitive as a means of overcoming the multiple `rd` problem and the LTSM.

	SCA	York Kernel II with LTSM		Sequential version	
	C-Linda	disabled	enabled	Setup 1	Setup 2
256x256 image 100% pixels set (65536 pixels)	523.20	670.48	56.32	68.70	72.02
256x256 image 75% pixels set (49152 pixels)	523.37	559.03	42.61	53.31	57.66
256x256 image 50% pixels set (32768 pixels)	510.92	447.14	26.39	37.88	41.06
256x256 image 25% pixels set (16384 pixels)	514.47	337.91	15.19	22.67	25.53
256x256 image 0% pixels set (0 pixels)	520.59	182.15	7.46	5.18	6.85

Table 6.11: The advantages of using the `copy-collect` method for the multiple `rd` problem.

Table 6.11 shows the results when different numbers of pixels are set within the image. For all the versions except the SCA C-Linda the execution time is linked to the number of pixels set to one in the image, regardless of whether the LTSM is enabled or disabled. Even when the LTSM is disabled using the `copy-collect` method provides better performance than using the SCA C-Linda (and the stream method) when about 60% of the tuples represent a set pixel in the test image. The actual speedups of the LTSM enabled kernel over the other versions are shown in Figure 6.13.

The results show that the performance of the SCA C-Linda version is practically independent of the number of pixels set. This appears logical because this version uses the stream method to overcome the multiple `rd` problem. Subsequently, regardless of how many image tuples actually represent a set pixel, all the pixel tuples are read. However, the amount of computation and the number of tuples placed into the parameter space *is* dependent on the number of pixels set. For the test images used Table 6.12 shows the number of tuples that would be placed into the parameter tuple space. This implies that the execution time should decrease as the number of set pixels within the image decreases.

Pixels set	Number of tuples in image	Number of non-zero elements in the parameter space
100%	65536	58484
75%	49152	51210
50%	32768	43881
25%	16384	36558
0%	0	0

Table 6.12: Number of non-zero elements in Parameter space for test images used.

	York Kernel II with LTSM enabled speedup against			
	SCA C-Linda	LTSM disabled	Sequential 1	Sequential 2
256x256 image 100% pixels set (65536 pixels)	9.3	11.9	1.2	1.3
256x256 image 75% pixels set (49152 pixels)	12.3	13.1	1.3	1.4
256x256 image 50% pixels set (32768 pixels)	19.4	16.9	1.4	1.6
256x256 image 25% pixels set (16384 pixels)	33.9	22.2	1.5	1.7
256x256 image 0% pixels set (0 pixels)	69.8	24.4	0.7	0.9

Table 6.13: Speedup of the parallel Hough transform.

The reason why the execution time for the SCA C-Linda version is not less (it would be expected to be approximately 490 seconds) when there are no pixels set in the test image could be due to the optimisations that the pre-compiler performs. For example, when a program creates

1000 tuples containing a single integer, and then consumes the tuples in any order (using a template $\langle|\Box_{integer}|\rangle$) the program takes 22.93 seconds to execute. If the program is altered to remove the tuples in the same order in which they are inserted the program takes 3.89 seconds to execute. As the order in which they are retrieved in the altered program is a valid ordering for the tuples to be returned in the first program it would be expected that the execution time of the first program should be no longer than the execution time for the second program.

The execution times for all the other versions show a dependency on the number of tuples within the image tuple space. This is to be expected as all the other versions use the `copy-collect` method to overcome the multiple `rd` problem, so the number of tuples read and processed depends on the number of tuples representing set pixels in the image.

The Hough transform is by its nature fine grained. The implementation strategies adopted have tried to mirror this. Therefore, there is a significant amount of communication compared to computation. The Hough transform is chosen to show the effectiveness of the LTSM in such situations. The fact that there is little speed increase over the sequential version when four worker processes are being used, although not desirable, indicates that the algorithm with such a small image is not suited to distribution over networks of workstations. What the results do show is that without using the LTSM there is no point in performing the Hough transform in parallel. It should be noted that all of the sequential execution timings have been obtained with the LTSM enabled. The results for the sequential version with the kernel distributed over four workstations and the LTSM disabled is given in Table 6.14.

256x256 image 100% pixels set (65536 pixels)	256x256 image 75% pixels set (49152 pixels)	256x256 image 50% pixels set (32768 pixels)	256x256 image 25% pixels set (16384 pixels)	256x256 image 0% pixels set (0 pixels)
625.94	537.42	455.03	363.89	191.53

Table 6.14: Execution timings for the sequential versions using setup two with the LTSM disabled.

6.4.2 Conclusions

The experimental results show that the performance of the York Kernel II increases when the LTSM is being used. The performance of the York Kernel II has been compared with the commercial SCA C-Linda which uses compile time analysis allowing it to efficiently place and store tuples. Normally, the performance of open implementations and closed implementations are not compared because of the advantages provided by using compile time analysis within closed implementations. By showing the performance gains, when using an image with all pixels set, it is possible to fairly compare the performance of the York Kernel II with the SCA C-Linda. When the LTSM is disabled the expected results of an open implementation are produced, but when the LTSM is enabled the results show that an open implementation using the techniques used in the York Kernel II is many times faster than the closed implementation.

The multiple `rd` problem effects "real world" problems and the results show how the `copy-collect` primitive can be used to efficiently solve the problem.

In many parallel algorithms and programs the use of the `collect` and `copy-collect` primitives are inappropriate. In these cases the compile time analysis that SCA C-Linda uses will normally provide better performance. The aim is to show that *appropriate* use of the `copy-collect` and `collect` primitives can improve performance and that improvement can be enhanced by using the implementational techniques for the bulk movement of tuples as used in the York Kernel II.

6.5 Conclusion

In Chapter 5 a description of how a two layer kernel can be created that uses locality information provided by multiple tuple spaces to create efficient implementation of the bulk primitives of `collect` and `copy-collect`. In this chapter the performance of the York Kernel II, a kernel implementation which uses the concepts described in Chapter 5, has been shown. A number of common Linda coordination operations have been used to show the speed increase that efficient implementation of the bulk primitives can provide.

The performance of the York Kernel II, an open implementation, has been compared to the performance of the SCA C-Linda closed implementation. In order to be able to compare the two effectively the Hough Transform (an image processing algorithm) was used.

The results in this chapter support the claim that the proposed method is more efficient than a naive approach, and adds further experimental results to support the inclusion of the `copy-collect` primitive within Linda to overcome the multiple `rd` problem.

The performance gains observed are only possible if multiple tuple spaces are used within a program, and the bulk primitives are used. Algorithms that use Linda to simply pass messages (or data structures) between processes will not observe any speedup using the York Kernel II. However, the speeds achievable mean that in many cases data structures that might be passed as a single tuple can be stored in tuple spaces as a collection of tuples, providing a more natural programming style.

Chapter 7

Generalising tuple space classification

7.1 Introduction

The bulk movement of tuples has been shown to be effective in producing a performance increase in the York Kernel II. However the York Kernel II, described in Chapter 5, has three shortcomings which are: eagerness in the movement of tuple spaces; the inability to move RTSs to LTSs; and the kernel does not support a large number of workstations.

Initially, these shortcomings are considered in more detail, and then a more graduated approach to the classification of tuple spaces is presented which builds on the ideas in Chapter 5. This more generalised classification of tuple spaces overcomes the problems of eagerness of tuple movement and scalability. It does not address the problem of RTSs becoming LTSs.

A kernel supporting the approach outlined in this chapter has not been implemented. Before such a kernel can be realised there are many other problems that need to be overcome. However a simple simulator has been created, which is described at the end of this chapter.

7.2 Eagerness in tuple space movement

The movement of a tuple space occurs the moment that another process can potentially access that tuple space. Consider the example procedure shown in Program 7.1. Using the York Kernel II the tuple space `ts1` will be changed from being a LTS to become a RTS when the handle for `ts1` is placed in `UTS` (a RTS). This means that all the `in` primitives will be performed on a RTS, regardless of whether any other processes can actually access `ts1`.

The example given in Program 7.1 is rather contrived, but the example in Program 7.2 shows the same problem in a more practical way using the master/worker style of parallelism. The two functions `producer1` and `producer2` both create a tuple space, and place 100 tuples into that tuple space. However, `producer1` places the handle of the tuple space into `UTS` *after* the tuples have been placed into the tuple space, and `producer2` places the tuple space handle into `UTS` *before* the tuples have been placed in the tuple space. The function `consumer` removes a

Program 7.1 An example showing the eagerness of the York Kernel II.

```
void demo_eager(void)
{
  int j;
  TS ts1;

  ts1 = tsc();

  for(j = 0; j < 100; j++)
    out(ts1, j);

  out(uts, ts1);   /* TS1 moved */

  for (j = 0; j < 100; j+)
    in(ts1, j);
}
```

tuple containing a tuple space handle from `UTS` and removes a 100 tuples from that tuple space. Assume that in a system, the `consumer` function and either one or other of the producer functions are executing concurrently. The end result will be the same regardless of which of the two producer functions is used, but which will provide the best performance? The answer is it depends on the amount of computation performed in both the producer and the consumer. If the consumer takes longer to process an individual tuple than it takes the producer to create it, then making the tuple space a RTS before the producer starts should yield the best performance. If the producer takes very little time then the conversion of the tuple space to a RTS once the tuples have been inserted may provide better performance. This is because every `out` primitive performed on a RTS requires two messages to be sent.

This extra communication affects the time taken to perform each `in` primitive because it reduces the bandwidth of the Ethernet and causes the TSS processes to perform more work (processing the `out` primitives) thus potentially having to queue the consumer‘s `in` primitives. This places an onus on the program writer to decide which will give the best performance. A lazier kernel would result in the tuple space being moved only when it was required. The movement of entire tuple spaces could be relaxed so that the movement occurs either when a tuple containing the tuple space handle is actually *removed* from a RTS by another process, or when another process actually accesses the tuple space for the first time.

Program 7.2 A second example showing the eagerness of the York Kernel II.

```
int producer1(void)
{
  int j;
  TS ts1;

  ts1 = tsc();
  for(j = 0; j < 100; j++)
    out(ts1, j);
  out(uts, ts1);   /* TS1 LTS -> RTS */
  return 1;
}

int producer2(void)
{
  int j;
  TS ts1;

  ts1 = tsc();
  out(uts, ts1);   /* TS1 LTS -> RTS */
  for(j = 0; j < 100; j++)
    out(ts1, j);
  return 1;
}

int consumer(void)
{
  int j;
  TS ts1;

  in(uts, ?ts1);
  for (j = 0; j < 100; j+)
    in(ts1, j);
  return 1;
}
```

There are three reasons why a lazier approach to the movement of tuples is not adopted in the York Kernel II. The first reason is the increased communication costs that would be imposed. As the TSS is distributed every TSS process would have to be told that the tuple space resides on a particular LTSM, and if they receive an operation to be performed on the tuple space, then the tuple space must be moved from the LTSM to the TSS. The communication costs of ensuring that all tuple spaces are aware of where the LTSM is, followed by the communication necessary to inform the TSS process that the tuple space has now moved, would be very high. Secondly, adopting a lazier approach to the movement of tuple spaces would require the LTSM to "interrupt" the user process in order to service the request for the tuple space to be moved. Thirdly, the LTSM is linked into the user process. When the user process terminates the LTSM terminates, and subsequently all the tuples stored within are lost. If a lazier approach is adopted then tuples which belong to a RTS can reside on a LTSM. Therefore, if tuples are residing on a LTSM which belong to a RTS, and the user process terminates, the tuples stored in the LTSM are lost. These three factors make the lazier movement of tuple spaces unattractive for use in the York Kernel II.

7.3 Reclassification of a RTS to LTSs

A looser definition of a LTS would be: a tuple space is a LTS only when one process has the tuple space handle in scope and the tuple space handle is not currently in any tuples present within a RTS. The current definition is: a tuple space is a LTS if it is created by the process and the tuple space handle has not been placed in a tuple in a RTS. When the current definition is used, there are only a few situations where a tuple space is classified as a RTS, when it could be classified as a LTS under the looser definition. Consider the program shown in Program 7.3. Function `test` calls function `one` which creates a tuple space (referred to as T_1), and places its handle in a tuple in `UTS`. When the tuple space is created it is a LTS. When the tuple is placed into `UTS` the created tuple space becomes a RTS. The function `test` then evaluates function `two` concurrently. Function `two` gets a tuple from `UTS` which contains a tuple space handle. Assuming that the tuple containing the handle for T_1 is the only tuple in `UTS` that matches the template used by the `in` primitive within function `two`, this will be returned. T_1 can now *only* be used by function `two` and could therefore be stored "as close as possible" to function `two`, and considered a LTS.

Another situation where a tuple space is classified as a RTS, and when it could be classified as a LTS under the looser definition, is when spawned processes die. If a program uses the master/worker style of parallelism a master process may create a tuple space, and create a number of worker processes that use the tuple space. When the tuple space is created it is a LTS. When the worker processes start using it the tuple space has to be a RTS. When all the worker processes die, the tuple space, under the looser definition, may become a LTS again.

Within the context of the York Kernel II, the communication costs associated with using a definition of a LTS that allows a RTS to become a LTS is too high for the few occasions when this

Program 7.3 Example showing when a RTS could be a LTS.

```
void one(void)
{
  TS ts_handle;

  ts_handle = tsc();   /* Create a tuple space */
  out(uts, ts_handle); /* Place the handle in a tuple space */
}

char *two(void)
{
  TS ts_handle;

  in(uts, ?ts_handle); /* Read a tuple space handle from UTS */
  return "TERMINATED";
}

void test(void)
{
  one();
  eval(two());
}
```

occurs. It would be necessary to create a graph of all the user processes which have the handle to each tuple space, and whenever a tuple space handle goes out of scope within a process the graph would need updating. The graph would also have to contain all the information about which tuple spaces have tuples with the tuple space handles in them.

Early work by Menezes et al.[Men96] suggests that the maintenance of such a graph may have other uses, such as in the garbage collection of unaccessible tuple spaces. If these graphs were found to be necessary to provide other facilities within a kernel, then their use to enable tuple spaces to move from being RTSs to LTSs would be acceptable, and the looser definition of a LTS could be used.

7.4 Scalability

The York Kernel II, like many other Linda implementations, has been developed for use with a Local Area Network (LAN) and a relatively small number of workstations (approximately thirty workstations). Linda is ideally suited for use in distributed computing because it supports asynchronous processes that communicate despite being temporally and spatially separated. Within the foreseeable future there will be kernels developed for use with Wide Area Networks (WAN). These kernels are going to be required to support thousands or even millions of workstations and computing devices. The technique of bulk moving tuples within a kernel, as used in the York Kernel II, is scalable provided the classification of tuple spaces is made less rigid.

In the next section the generalisation of the classification of tuple spaces is considered to enable multi-layer or N-layer hierarchical kernels which can potentially support many geographically distributed workstations to be created, although before this is achieved there are many other problems that need to be answered!

7.5 N-layer hierarchical kernel

In the York Kernel II a tuple space is either a RTS or a LTS. A LTS is stored in a LTSM and a RTS is stored on the TSS. Therefore the classification of a tuple space can be represented by the layer on which it resides. In the N-layer hierarchical kernel a tuple space will not be classified by either user processes or individual processes within the kernel, but by which layer within a hierarchy it is stored.

The N-layer hierarchical kernel can be considered as a tree of TSSs, as shown in Figure 7.1. Each node of the tree represents a TSS, and is referred to as a TSS node. User processes are connected to the leaf nodes of the tree. An arc between nodes represents a communications link, which could be: a socket between process on the same computer; a socket over a LAN; a dedicated virtual communication channel in a parallel computer; a socket over a WAN; a radio link to an orbiting satellite; etc.. A user process can communicate with only one of the TSSs, and the communications link between the TSS and the user process can take one of many forms, but the most likely are either an interface to a set of library routines; a socket between processes on the same computer; a socket over a LAN or a dedicated virtual communication channel in a parallel computer. Figure 7.1 shows an example of a five layer hierarchical kernel where each of the TSS nodes has been given a letter to allow easy identification for the descriptions that follow later in the chapter. A user process "chooses" a leaf TSS node to be its contact with the kernel.

A TSS is defined in Chapter 5 as: "A Tuple Space Server (TSS) is a dedicated *system* that exists to store and manage RTSs. The TSS could be a single process (a dedicated server) or it can be a set of processes". In this chapter the definition of a TSS is changed slightly to: A Tuple Space Server (TSS) is a dedicated *system* that exists to store and manage tuple spaces. A TSS is considered as a single process, but it could potentially be a set of processes.

The work in this chapter assumes that each TSS is a single process. Distribution of tuples throughout a kernel occurs because different tuple spaces are stored on different TSSs. Using the analogy used by Douglas[DWR95], if tuple spaces are considered as layers of a cake then in the distributed TSS as used in Chapter 5 each TSS process has a slice of the cake. In the hierarchical

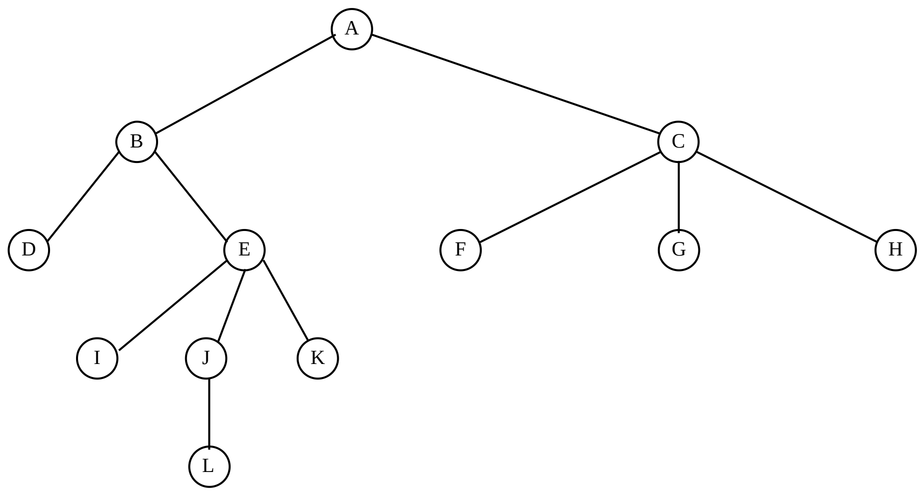


Figure 7.1: An example five layer hierarchical kernel.

kernel described in this chapter each TSS node has a number of layers of the cake, but not the whole cake.

In this kernel the TSS nodes not only manage the tuple spaces, but decide when tuple spaces should be moved up the tree, and control the retrieval of tuples from tuple spaces higher up the tree. Each TSS node can only communicate with its parents and children. The TSS node does not need to know if it is communicating with another TSS node or a user process. Each TSS node knows nothing about the depth or breadth of the tree.

The nearer the root TSS node of the tree that a tuple space is stored, the more global the tuple space. The `UTS` will be stored on the root node because all processes can access that tuple space. A user process can only access a tuple space if the TSS node it is attached to is a descendent of the TSS node on which the tuple space is stored. When a user process creates a new tuple space it is stored on the TSS node to which the user process is connected. As more processes become able to access the tuple space, it moves up the tree.

As already discussed, in the York Kernel II the movement of tuple spaces is eager. The moment a tuple space becomes a RTS it is moved on to the TSS. Such an approach *cannot* be adopted in the N-layer hierarchical kernel because such an approach would lead to all tuple spaces either moving to the root TSS node of the tree or remaining on the TSS node on which they were created. This is because in order for two unrelated processes to swap a tuple space handle it must be passed through a tuple space. Consider two user processes using the N-layer hierarchical kernel. When the two user processes commence the only tuple space they have in common is `UTS` which must reside on the root of the tree as it is accessible by all processes. If the processes wish to share a tuple space that one of the processes has created, a tuple must be passed to the other process through the tuple space `UTS`. As soon as a tuple containing a tuple space handle is placed into `UTS` the tuple space will migrate to the same layer as the tuple space in which the tuple was placed, which is the root TSS node. The migration to the root TSS node occurs because the tuple containing the tuple space handle is placed in a tuple space which all user processes can access. Therefore, the tuple space must move to a level at which all user processes can access it. This is unsatisfactory because all tuple spaces either reside on the leaf TSS nodes or on the root TSS node creating a two layer hierarchical kernel, similar to the York Kernel II.

To overcome this the N-layer hierarchical kernel *has* to be less eager about moving tuple spaces. Although this leads to an increase in communication the movement of tuple spaces can still be achieved dynamically and implicitly. *No* extra information is required from the programmer and the same program used with the York Kernel II will be able to use the N-layer hierarchical kernel without alterations.

7.6 The TSS node structure

Each TSS node is only aware of its parent node and child nodes, and can only communicate with these TSS nodes. The TSS nodes receive messages and service them. There are ten possible messages that a TSS node can receive. Each message type is now described and the pseudo-code for the operations a TSS node must perform when the message is received is given. It is assumed that: the communication system preserves the order of messages sent from one TSS node to another TSS node; that a single TSS node can only service one message at a time; and it services the messages from a particular source (either a user process or another TSS node) in the order they are received. Before the messages are considered the nomenclature is clarified.

The kernel uses *tags* to help it control the placement of tuples and the flow of messages within it. There are two types of tags, a message tag and a tuple space tag. There is, at most, one message tag associated with a message. They are used to provide information about either the destination of a message, or the path that the message has followed through the TSS node tree. A tuple space tag is attached to a tuple space handle, and is used to store information about where a tuple space resides within the kernel. Usually, the tag will indicate where the tuple space resides. However, as the tuple spaces move up the tree the tag becomes out of date. In order to overcome this each TSS node maintains a table of all the tuple spaces that pass through it. This ensures that if a tuple space has moved above the TSS node in the tree, and the TSS node contains a tuple space which contains a tuple which has a field that refers to that tuple space, the tuple space tag can be detected as incorrect. The possible messages that a TSS node can receive are: out message; in message; rd message; reply message; request message; packet message; collect message; copy-collect message; packet down message; c-reply message and a create message. These are now described in detail.

out message

This message represents an `out` primitive. The message takes the form;

$$[out_{identifier}, destination_{tuple\ space}, tuple]$$

where $out_{identifier}$ is a field which allows the TSS node to recognise that the message is an out message, $destination_{tuple\ space}$ is the tuple space into which the tuple is to be placed, and $tuple$ is the tuple. The pseudo-code for an out message is shown in Figure 7.2.

```
if is_local(destination_{tuple space}) then
  insert(tuple, destination_{tuple space})
else
  for all tuple spaces handles in tuple do
    ts = get_next_ts(tuple)
    if is_local(ts) then
      create_ts_tag(ts)
    end if
    if exists_ts_tag(ts) then
      add_TSS_identifier_to_tag_tail(ts)
    end if
  end for
  pass_message(parent_TSS)
end if
```

Figure 7.2: The pseudo-code for managing an out message within a TSS node in the N-layer hierarchical kernel.

The aim of the pseudo code is to give a high level overview of how a TSS node processes the message. The functionality of the functions used in the pseudo-code are:

- *is_local* checks to see if the specified tuple space is stored locally.
- *insert* inserts a tuple in the specified tuple space (assuming it is stored locally).
- *get_next_ts* finds the next tuple space handle in the tuple.
- *create_ts_tag* creates an empty tag for the specified tuple space handle. If a tag is already associated with the tuple space it is cleared.
- *exists_ts_tag* checks to see if there is an initialised tuple space tag for the specified tuple space handle.
- *add_TSS_identifier_to_tag_tail* adds the current TSS node name (that the connected TSS nodes know) to the tail of the specified tuple space.
- *pass_message* sends the updated message to a specified connected TSS node.

When the TSS receives an out message it attempts to see if it can service it locally. If it cannot, it checks to see if any tuple space handles are present and if so, updates them appropriately. When the tuple is inserted into the destination tuple space the first element of any tuple space tag is the TSS node on which the tuple space was last seen. If the tuple space tag does not exist then the tuple space must reside higher up the tree.

in message

This message represents an `in` primitive. The message takes the form:

$$[in_{identifier}, source_{tuple\ space}, template] : message\ tag$$

where $in_{identifier}$ is a field which allows the TSS node to recognise that the message is an in message, $source_{tuple\ space}$ is the tuple space from which a tuple is to be fetched, and $template$ is the template to be used for matching the tuple. The message has a message tag. The pseudo-code for an in message is shown in Figure 7.3.

```
if is_local(source_tuple space) then
  tuple = find_match(source_tuple space, template)
  for all tuple space handles in tuple do
    ts = get_next_ts(tuple)
    if not is_local(ts) and not in_pass_table(ts) and exists_ts_tag(ts)then
      if get_ts_tag_tail(ts) <> get_msg_tag_tail() then
        create_ts_local(ts)
        mark_ts_tuples_pending(ts)
        send_request(get_ts_tag_tail(ts), ts, remove_ts_tag_tail(ts))
        clear_ts_tag(ts)
      else
        ts = remove_ts_tail_tag(ts)
      end if
    end if
  end for
  send_reply(get_msg_tag_tail(), tuple, remove_msg_tag_tail())
else
  add_TSS_identifier_msg_tag()
  pass_message(parent_TSS)
end if
```

Figure 7.3: The pseudo-code for managing an in (and rd) message within a TSS node in the N-layer hierarchical kernel.

The functionality of the new functions used in the pseudo-code are:

- *find_match* searches the specified tuple space for a tuple that matches the template. If a tuple is not available the function will queue the request until a matching tuple is inserted within the tuple space.
- *in_pass_table* checks the local table of tuple spaces that have passed through the TSS node to see if the specified tuple space has passed through.
- *get_ts_tag_tail* returns the tail element of the specified tuple space tag.
- *get_msg_tail_tag* returns the tail element of the message tag.
- *create_local_ts* creates the specified tuple space locally within the TSS node.

- *mark_ts_tuples_pending* marks the specified tuple space to indicate that there is a packet of tuples expected.
- *send_request* produces a request message with the destination TSS node as the first value (it must be either the parent or a child of the TSS node), the tuple space which is required as the second field and the message tag to be attached to the message as the third field.
- *remove_ts_tag_tail* removes the tail element from a tuple space handle tag.
- *clear_ts_tag* removes the tag associated with the specified tuple space handle.
- *send_reply* sends a reply message to the child TSS node specified, with the tuple and message tag also specified.
- *remove_msg_tag_tail* removes the tail element from a message tag.
- *add_TSS_identifier_msg_tag* adds the TSS node identifier to the tail of the message tag.

When an in message is received by the TSS node it checks to see if the tuple space resides locally. If it does then it finds a tuple that matches the template. The matched tuple is then checked for tuple space handles. If a tuples space handle is found, and if it has no tag or is listed in the passed through table, then the tuple space must reside higher up in the tree. If the tuple space resides higher up the tree or locally on the TSS node then there is no need to move the tuple space. If a tuple space handle with a tag which has not moved up the tree is found, then if the TSS node that the matched tuple is to be sent to next, and the tail element of the tuple space tag are the same then there is no need to move the tuple space to this TSS node. Otherwise the tuple space must be moved to this TSS node. In order to achieve this the required tuple space is created locally, and marked as missing tuples. A request is then dispatched for that tuple space to be moved, using the tuple space tag as the message tag. The result tuple is then dispatched to the original sourcing TSS node, using the message tag of the in message as the message tag of the reply message, so that the reply message retraces the path of the in message through the kernel.

If the source tuple space specified within the in message does not reside locally, the TSS node identifier is added to the message tag and the message is dispatched to the parent of the TSS node. The name of the TSS node is added to the message tag, so that when the source tuple space is found the matching tuple can be returned back down the kernel.

rd message

This message represents a `rd` primitive. The message takes the form:

$$[rd_{identifier}, source_{tuple\ space}, template] : message\ tag$$

where $rd_{identifier}$ is a field which allows the TSS node to recognise that the message is a rd message, $source_{tuple\ space}$ is the tuple space from which a tuple is to be fetched, and $template$ is the template to be used for matching the tuple. The message has a message tag. The pseudo-code for the rd message is the same as for an in message, except that the *find_match* function does not remove the matching tuple.

reply message

This message is used for passing a result tuple down the tree when a tuple is returned from an in or rd message. The message takes the form:

$$[reply_{identifier}, tuple] : message\ tag$$

where $reply_{identifier}$ is a field which allows the TSS node to recognise that the message is a reply message, $tuple$ is the tuple which is being returned as a result of an in or rd message. The message has a tag which represents the path down the tree that the message is to take. The pseudo-code for a reply message is shown in Figure 7.4.

```
for all tuple space handles in tuple do
  ts = get_next_ts(tuple)
  if not is_local(ts) and not in_pass_table(ts) and exists_ts_tag(ts) then
    if get_ts_tag_tail(ts) <> get_msg_tag_tail() then
      create_ts_local(ts)
      mark_ts_tuples_pending(ts)
      send_request(get_ts_tag_tail(ts), ts, remove_ts_tag_tail(ts))
      clear_ts_tag(ts)
     else
       ts = remove_ts_tail_tag(ts)
     end if
   end if
end for
if exists_msg_tag() and (get_msg_tag_tail() <> my_user_process) then
  dest = get_msg_tag_tail()
  remove_msg_tail_tag()
  pass_message(dest)
else
  pass(tuple, user_process)
end if
```

Figure 7.4: The pseudo-code for managing a reply message within a TSS node in the N-layer hierarchical kernel.

The functionality of the new functions used in the pseudo-code are:

- *pass* sends a tuple to a user process.
- *exists_msg_tag* checks to see if there is an initiated message tag associated with the message.

When the TSS node receives the reply message, it checks all the tuple space handles in the tuple to see if any of them need to be moved up the tree in a similar fashion to the checks performed when the TSS node receives an in message and finds a tuple that matches. Once any necessary tuple space movements have been organised, the TSS node then checks to see if it is the recipient. This can occur if either the message tag is empty, implying it must have sourced it, or if the tail element is a recognised user process that it manages. If this is the case the result tuple is dispatched to the user process. If the reply message is not for one of the TSS nodes user processes, then it sends the reply message to the next TSS node, and removes that TSS node from the message tag which is passed with the message.

request message

This message is a request for a tuple space and is used when a tuple space is to be moved. The destination TSS node produces the request and it is sent down the tree. The message takes the form:

$$[request_{identifier}, source_{tuple\ space}] : message\ tag$$

where $request_{identifier}$ is a field which allows the TSS node to recognise that the message is a request message and $source_{tuple\ space}$ is the tuple space which is required to be moved up the tree. The message has a tag which represents the path down the tree that the message is to take to reach the TSS node which has the tuple space. The message may not reach that TSS node if a TSS node the message travels through first, contains the tuple space. The pseudo-code for a request message is shown in Figure 7.5.

```
if is_local($source_{tuple\ space}$) then
  **tuples** = close_tuple_space($source_{tuple\ space}$)
  for all **tuple** in **tuples**
    for all tuple spaces handles in **tuple** do
    **ts** = get_next_ts(**tuple**)
    if is_local(**ts**) then
      create_ts_tag(**ts**)
    end if
    if in_pass_table(**ts**) then
      clear_ts_tag(**ts**)
    end if
    if exists_ts_tag(**ts**) then
      add_TSS_identifier_to_tag_tail(**ts**)
    end if
  end for
  packet_message(**tuples**)
else
  if exists_msg_tag() then
    **dest** = get_msg_tag_tail()
    remove_msg_tail_tag()
    pass_message(**dest**)
  end if
end if
```

Figure 7.5: The pseudo-code for managing a request message within a TSS node in the N-layer hierarchical kernel.

The functionality of the new functions used in the pseudo-code are:

- *packet_message* produces a packet message containing all the specified tuples which is dispatched to the parent TSS node.
- *close_tuple_space* returns all the tuples that are in the specified tuple space and any state associated with the tuple space, and removes the specified tuple space from the local TSS node.

When the TSS node receives the request message it checks to see if the requested tuple space resides locally. If it does then the tuple space is packed into a packet message, and all the tuples are checked for tuple space handles. If any exist they are updated appropriately. If there are any tuples pending from a bulk tuple movement then this information (state) is transfered with the tuple space. If the tuple space does not reside locally then the message is passed to another TSS node by removing the next TSS node's name from the message tag and passing the request message to that TSS node.

packet message

This message is a packet of many tuples used for moving multiple tuples *up* the tree structure. The message takes the form:

$$[packet_{identifier}, destination_{tuple\ space}, tuples]$$

where $packet_{identifier}$ is a field which allows the TSS node to recognise that the message is a packet message, $destination_{tuple\ space}$ is the tuple space into which the tuples are to be placed, and $tuples$ are the tuples that are being moved up the tree. The pseudo-code for a packet message is shown in Figure 7.6.

```
if is_local($destination_{tuple\ space}$) then
  insert_tuples($tuples$, $destination_{tuple\ space}$)
  if is_tuple_space then
    reset_ts_tuples_pending($destination_{tuple\ space}$)
  end if
else
  for all tuple in tuples do
    for all tuple space handles in tuple do
      ts = get_next_ts(tuple)
      if is_local(ts) then
        create_ts_tag(ts)
      end if
      if exists_ts_tag(ts) then
        add_TSS_identifier_to_tag_tail(ts)
      end if
    end for
  end for
  add_ts_pass_table($destination_{tuple\ space}$)
  pass_message(parent_TSS)
end if
```

Figure 7.6: The pseudo-code for managing a packet message within a TSS in the N-layer hierarchical kernel.

The functionality of the new functions used in the pseudo-code are:

- *insert_tuples* inserts a number of tuples into the specified tuple space.
- *is_tuple_space* checks to see if the tuples represent an entire tuple space movement or just a number of tuples.
- *reset_ts_tuples_pending* alters a tuple space to indicate that the tuples pending have arrived.
- *add_ts_pass_table* adds the specified tuple space to the pass through table.

When the TSS node receives the packet message it checks to see if the destination tuple space is local. If it is local, the tuples are inserted. If the tuples represent a tuple space then the states of the tuple space is altered to indicate that a reply from a request message has arrived. A tuple space can potentially be waiting the arrival of more than one packet of tuples marked as a tuple space (from different request messages). If the destination tuple space is not local all the tuples are checked and any tuple space handles detected are updated accordingly. The destination tuple space is placed in the pass-through table because the tuple space has passed through the TSS node. This ensures that any references to the tuple space location within any tuple space tags stored on the TSS node can be detected as being no longer valid.

collect message

This message represents a `collect` primitive. The message takes the form:

$$[collect_{identifier}, source_{tuple\ space}, destination_{tuple\ space}, template] : message\ tag$$

where $collect_{identifier}$ is a field which allows the TSS node to recognise that the message is a collect message, $source_{tuple\ space}$ is the source tuple space used in the `collect` primitive, $destination_{tuple\ space}$ is the destination tuple space into which the tuples are to be placed and $template$ is the template to be used to match the tuples. The message has a message tag to allow a count of the number of tuples moved to be returned to the user process. The pseudo-code for a collect message is shown in Figure 7.7.

The functionality of the new functions used in the pseudo-code are:

- *find_matching* finds all tuples within a tuple space that match the specified template.
- *send_packet_down* causes a packet down message to be dispatched to the destination TSS node containing a number of tuples for the specified tuple space, and the packet down message is given the specified message tag.
- *send_creply* dispatches a c-reply message to the specified TSS node containing the value specified by the second field and the c-reply message is given the message tag as specified by the third field.

```
if is_local(source_tuple space) then
  tuples = find_matching(source_tuple space, template)
  if is_local(destination_tuple space) then
    insert_tuples(tuples, destination_tuple space)
  else
    if not exists_ts_tag(destination_tuple space) then
      for all tuple in tuples
        for all tuple spaces handles in tuple do
          ts = get_next_ts(tuple)
          if is_local(ts) then
            create_ts_tag(ts)
          end if
          if in_pass_table(ts) then
            clear_ts_tag(ts)
          end if
          if exists_ts_tag(ts) then
            add_TSS_identifier_to_tag_tail(ts)
          end if
        end for
      for all
      packet_message(destination_tuple space, tuples)
    else
      for all tuple in tuples do
        for all tuple space handles in tuple do
          ts = get_next_ts(tuple)
          if not is_local(ts) and not in_pass_table(ts) and exists_ts_tag(ts) then
            if get_ts_tag_tail(ts) <> get_msg_tag_tail() then
              create_ts_local(ts)
              mark_ts_tuples_pending(ts)
              send_request(get_ts_tag_tail(ts), ts, remove_ts_tag_tail(ts))
              clear_ts_tag(ts)
            else
              ts = remove_ts_tail_tag(ts)
            end if
          end if
        end for
      end for
      dest = get_ts_tag_tail(destination_tuple space)
      msg_tag = remove_ts_tag_tail(destination_tuple space)
      send_packet_down(dest, destination_tuple space, tuples, msg_tag)
    end if
  end if
  send_creply(get_msg_tag_tail(), cardinality(tuples), remove_msg_tag_tail())
else
  if is_local(destination_tuple space) then
    create_ts_tag(destination_tuple space)
  end if
  if exists_ts_tag(destination_tuple space) then
    add_TSS_identifier_to_tag_tail(destination_tuple space)
  end if
  add_TSS_identifier_to_msg_tag()
  pass_message(parent_TSS)
end if
```

Figure 7.7: The pseudo-code for managing a collect message within a TSS node in the N-layer hierarchical kernel.

When the TSS node receives a collect message it checks to see if the source tuple space is local. If the source tuple space is local it retrieves all the tuples that match the specified template from the source tuple space. If the destination tuple space is local then the tuples are inserted into it, and a c-reply message is dispatched to the calling process. If the destination tuple space does not reside locally then all the matched tuples are checked for tuple space handles, and updated appropriately depending on whether the destination tuple space is above or below the current TSS node in the tree. The position of the destination tuple space can be determined by checking if the destination tuple space has a tuple space tag associated with it. If so the tuple space resides below the current TSS node, if not the tuple space resides above the current TSS node. If the destination tuple space resides below the current TSS node the tuples are dispatched in a packet down message. If the destination tuple space resides above the TSS node the tuples are dispatched in a packet message, which is marked to show that this is *not* an entire tuple space being moved (the destination tuple space remains on this TSS node).

copy-collect message

This message represents a `copy-collect` primitive. The message takes the form:

$$[copy\text{-}collect_{identifier}, source_{tuple\ space}, destination_{tuple\ space}, template] : message\ tag$$

where $copy\text{-}collect_{identifier}$ is a field which allows the TSS node to recognise that the message is a collect message, $source_{tuple\ space}$ is the source tuple space used in the `copy-collect` primitive, $destination_{tuple\ space}$ is the destination tuple space into which the tuples are to be placed, and $template$ is the template to be used to match the tuples. The message has a message tag to allow the counter of the number of tuples copied to be returned to the user process. The pseudo-code for the copy-collect message is the same as for a collect message, except that the *find_matching* function does not remove the matching tuples from the source tuple space.

packet down message

This message is a packet of many tuples and is used for moving multiple tuples *down* the tree structure. The message takes the form:

$$[packet\text{-}down_{identifier}, destination_{tuple\ space}, tuples]$$

where $packet\text{-}down_{identifier}$ is a field which allows the TSS node to recognise that the message is a packet down message, $destination_{tuple\ space}$ is the tuple space used into which the tuples are to be placed and $tuples$ are the tuples. The pseudo-code for a packet down message is shown in Figure 7.8.

When a TSS node receives a packet down message it checks to see if the destination tuple space resides locally. If it does the TSS node inserts the tuples. If the tuple space does not reside locally all the tuples are checked for tuple space handles. Each of the tuples is treated as though it was a matched tuple in a reply message. Therefore, if the packet down message is to be sent to a child

```
if is_local($destination_{tuple\ space}$) then
  insert_tuples($tuples, destination_{tuple\ space}$)
else
  for all **tuple** in *tuples*
    for all tuple space handles in **tuple** do
      **ts** = get_next_ts(*tuple*)
      if not is_local(**ts**) and not in_pass_table(**ts**) and exists_ts_tag(**ts**) then
        if get_ts_tag_tail(**ts**) <> get_msg_tag_tail() then
          create_ts_local(**ts**)
          mark_ts_tuples_pending(**ts**)
          send_request(get_ts_tag_tail(**ts**), **ts**, remove_ts_tag_tail(**ts**))
          clear_ts_tag(**ts**)
         else
          **ts** = remove_ts_tail_tag(**ts**)
         end if
       end if
    end for
  end for
  if exists_msg_tag() then
    **dest** = get_msg_tag_tail()
    remove_msg_tail_tag()
    pass_message(**dest**)
  end if
end if
```

Figure 7.8: The pseudo-code for managing a packet down message within a TSS node in the N-layer hierarchical kernel.

of the TSS node, and there are tuple space handles for tuple spaces stored on, or as descendants of, a different child TSS node, the tuple space is moved to this TSS node. Once any necessary movement of tuple spaces has been initiated the packet down message is passed to the next TSS node as specified by the message tag.

c-reply message

This message is used for returning the count of the number of tuples either copied or moved by a `copy-collect` or `collect` primitive. The message takes the form:

$$[c\text{-}reply_{identifier}, count] : message\ tag$$

where $c\text{-}reply_{identifier}$ is a field which allows the TSS node to recognise that the message is a collect or copy-collect reply message, $count$ is a count of the number of tuples that were copied or moved. The message has a tag which represents the path down the tree that the message is to take to reach the user process which performed the `collect` or `copy-collect` primitive. The pseudo-code for a reply message is shown in Figure 7.9.

When the TSS node receives a c-reply message it checks to see if the message was intended for one of its user processes which occurs when either the message tag is empty or the tail element of the tag represents a user process attached to the TSS node. If the message is for a user process

```
if exists_msg_tag() and (get_msg_tag_tail() <> my_user_process) then
  dest = get_msg_tag_tail()
  remove_msg_tail_tag()
  pass_message(dest)
else
  pass(integer, user_process)
end if
```

Figure 7.9: The pseudo-code for managing a c-reply message within a TSS node in the N-layer hierarchical kernel.

attached to the TSS node, then the result for the `collect` or `copy-collect` primitive is passed to the user process. If the TSS node does not recognise the end of the message tag as a user process and the message tag is not empty, the TSS node passes the message the TSS node specified as the tail element of the message tag, and removes it from the tail of the message tag.

create message

This message is used to create a tuple space. The tuple space will always be created on the TSS node to which user process is attached. The message takes the form:

$$[create_{identifier}]$$

where $create_{identifier}$ is a field which allows the TSS node to recognise that the message is a create message. The TSS node creates a unique tuple space name (see Section 5.5 for details of how unique tuple space names could be created), and returns the handle to the user process.

These are the only messages that a TSS node can receive or produce.

7.7 Demonstration of the N-layer hierarchical kernel

In order to show how the concept of a N-layer hierarchical kernel works a simple example is used. The example processes are shown in Program 7.4. It is assumed that the kernel is configured as a five layer hierarchy as in Figure 7.1. The user process `test1` which is attached to TSS node $\mathcal{L}$, creates a tuple space and then places a tuple containing the handle of that tuple space into `UTS`. The other user process `test2` which is attached to TSS node $\mathcal{D}$, then retrieves a tuple containing a tuple space handle. The Figures 7.10 to 7.13 show the distinct stages that will occur and the messages that flow around the kernel.

Figure 7.10 shows the first operation of the process `test1` which is the creation of a tuple space. The tuple space `UTS` already exists as a shared universal tuple space which implies that it resides on the root node of the tree. The user process communicates with its host TSS node $\mathcal{L}$ to create a tuple space. The node TSS $\mathcal{L}$ creates a tuple space which has a unique name for the entire kernel. For the sake of clarity a token name of `TS1` is used for this tuple space in this example. The tuple space is stored on TSS node $\mathcal{L}$.

Program 7.4 Example processes for showing the inner workings of the N-layer hierarchical kernel.

```
test := func();

  local ts_result;

  ts_result := tsc();
  out(uts,[ts_result]);

  return 0;
end func;

test2 := func();

  in(uts,|[?ts]|);

  return 0;
end func;
```

Figure 7.11 shows the next operation, an `out` primitive which places a tuple containing the tuple space handle for `TS1` into tuple space `UTS`. TSS node $\mathcal{L}$ receives the out message. This TSS node checks to see if the destination tuple space (`UTS`) resides locally. As `UTS` does not reside locally, the TSS node prepares to pass the message to its parent. But before doing so it checks the tuple to see whether there are any tuple space handles present within it. For each tuple space handle the TSS node checks if the tuple space resides locally and if the tuple space does, the TSS node creates a tuple space tag containing the TSS nodes name (`.L`) and associates the tag with the tuple space handle. If a tuple space tag existed previously it is cleared. TSS node $\mathcal{L}$ then sends the out message to its parent, TSS node $\mathcal{J}$. TSS node $\mathcal{J}$ performs the same checks, as though it had received the out message directly from a user process. When checking the tuple for tuple space handles the handle for tuple space `TS1` is found. As the tuple space does not reside locally, TSS node $\mathcal{J}$ checks for a tuple space tag associated with the tuple space handle, and if it exists it adds to the tail of the tag its TSS node name. TSS node $\mathcal{J}$ then sends the message to TSS node $\mathcal{E}$, its parent. The message continues up the tree, with each TSS node processing it, and finally the message arrives at TSS node $\mathcal{A}$, which recognises the tuple space `UTS` as a tuple space which resides locally. The tuple is then inserted in the local tuple storage data structure. The tuple will at this point contain the tuple space handle `TS1` with an associated tag of `.L.J.E.B`. This indicates the tuple spaces position from the TSS node on which the tuple is stored.

The second user process then performs the `in` primitive which is shown in Figure 7.12. TSS node $\mathcal{D}$ receives an in message requesting a tuple from the tuple space `UTS`. It checks to see if

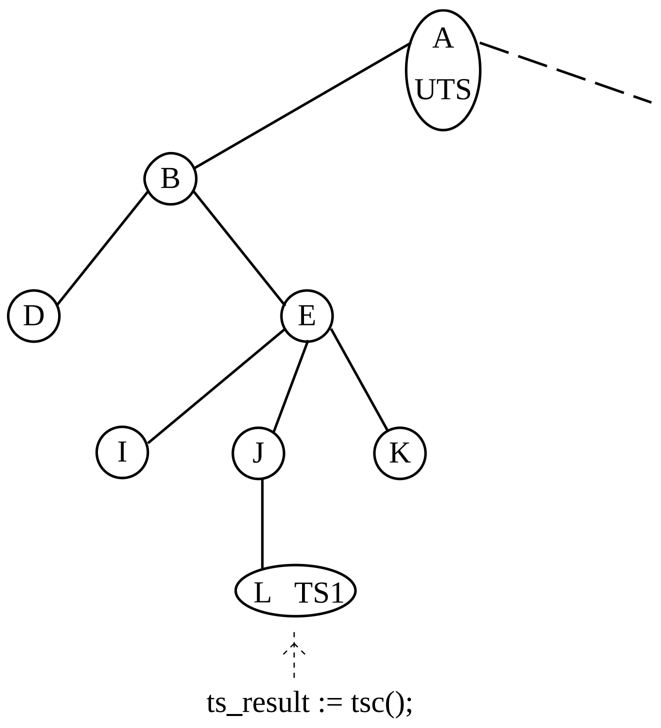


Figure 7.10: Creating the tuple space.

the tuple space `UTS` resides locally. As it does not and a reply is needed (an `in` primitive returns a tuple) it adds a message tag to the message. The message tag is of the form `.P1.D` where `P1` represents the identifier for the user process that TSS node $\mathcal{D}$ uses. The addition of the `P1` to the message tag is optional. For example, if the TSS node is linked into the user process then there is no need to have this field (the `.P1`), as only one process can ever access the TSS node. Whenever a result message arrives at the TSS node it has to be for the user process which is linked into the TSS node.

As with the out message, the in message is passed up the tree until a TSS node recognises the tuple space. This will again be TSS node $\mathcal{A}$ which recognises that the tuple space `UTS` resides locally and performs the search of `UTS` which finds the tuple that has just been inserted.

Figure 7.13 shows the second stage of the in message, where the matched tuple is returned to the blocked user process. TSS node $\mathcal{A}$ examines the tag attached to the in message, which will be `.P1.D.B`. The tail of the tag indicates the child TSS node to which the tuple should be sent. However, before the tuple is dispatched it is checked to see if it contains any tuple space handles. For each tuple space handle in the matched tuple, TSS node $\mathcal{A}$ examines the tuple space tag attached to the tuple space handle. In this example the tuple space tag attached to tuple space `TS1` is `.L.J.E.B`.

As there is a tuple space handle with a tag in the tuple, TSS node $\mathcal{A}$ checks to see if the tuple space resides locally and if so sets the tag to empty. It also checks its table of tuple spaces that

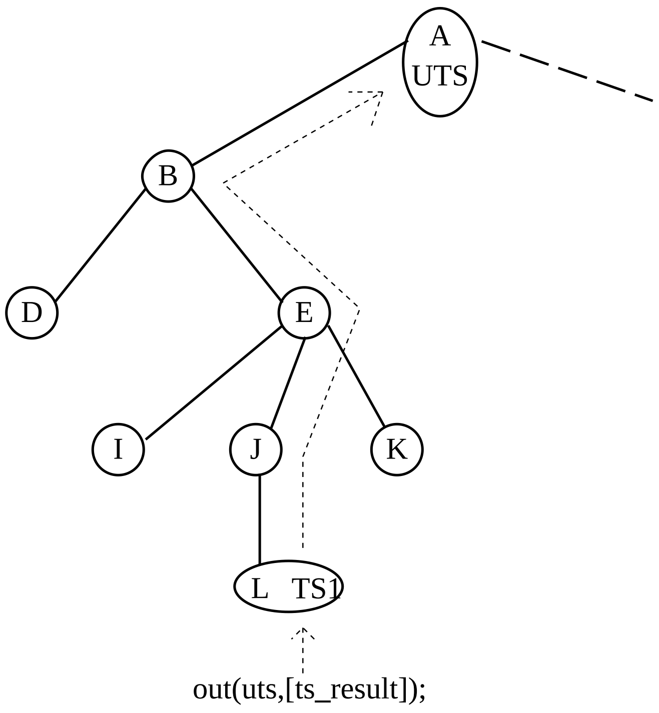


Figure 7.11: Performing the `out` primitive.

have passed through, and if the tuple space name resides in this table the tag is also reset to empty. This is because the tuple space may have moved above the TSS node *since* the tuple was inserted. If the tuple space has moved above the current TSS node it *must* have passed through this current TSS node. In this example it neither resides locally nor has passed through so the tag is valid. The TSS node $\mathcal{A}$ checks the tail elements of message tag (`.P1.D.B`) and the tuple space handle tag (`.L.J.E.B`). As the tail element of both tags are the same (`B`), TSS node $\mathcal{A}$ removes both the tail elements from both tags and sends a reply message to TSS node $\mathcal{B}$.

TSS node $\mathcal{B}$ receives the reply message (denoted by *a* in Figure 7.13). TSS node $\mathcal{B}$ then performs the same checks for tuple space handles as TSS node $\mathcal{A}$. It finds the tuple space handle and checks that it is neither stored locally nor is present in the pass through table. After not finding it, TSS node $\mathcal{B}$ compares the message tag with the tuple space handle tag. The message tag is `P1.D` and the tag associated with the tuple space handle is `.L.J.E`. TSS node $\mathcal{B}$ checks the tail elements of the tags, which do differ indicating that a tuple space has to be moved. TSS node $\mathcal{B}$ immediately creates a tuple space locally with the same name as the tuple space in the tuple (`TS1`) which is marked to indicate that a transfer of tuples into it is expected resulting from a request message. TSS node $\mathcal{B}$ then creates a request message for a tuple space to be moved. This request message contains the tuple space name that is required (`TS1`) and the request message is given the message tag that was the tag associated with the tuple space (`.L.J`) with the tail element removed. The tail element of the tuple space tag identifies to which child TSS node the message should be

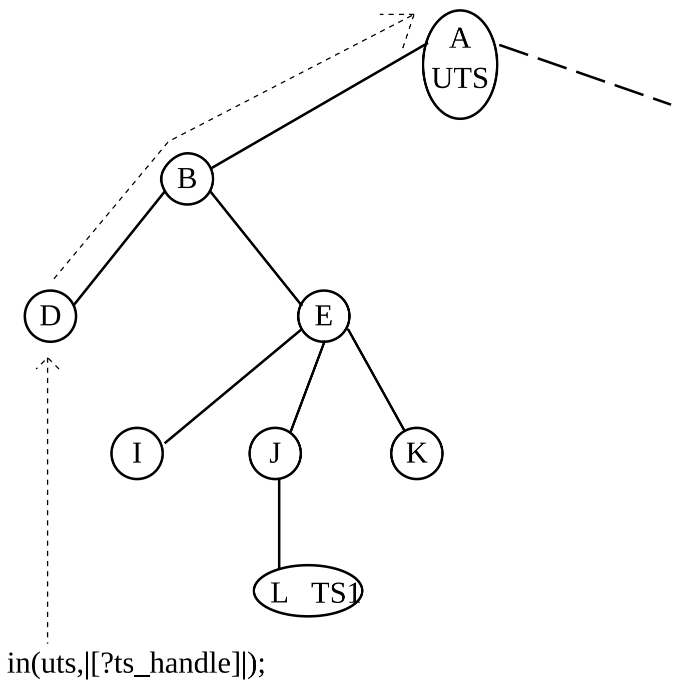


Figure 7.12: The first stage of performing the `in` primitive.

sent (TSS node $\mathcal{E}$). This is represented as message *b* in Figure 7.13. TSS node $\mathcal{E}$ receives the message, checks to see if the tuple space has been moved locally, and as it does not find the tuple space, forwards the message to the TSS node specified by the last element of the message tag, and removes the tail element from the message tag. TSS node $\mathcal{L}$ receives the message, and finds the tuple space locally. TSS node $\mathcal{L}$ then dispatches the entire tuple space to its parent. The parent checks to see if the tuple spaces exist locally, and if not sends the message to its parent. *Each* tuple moved must be checked for tuple space handles and these updated accordingly. The message for the movement of the tuples is represented by *c* in Figure 7.13.

Concurrently with messages *b* and *c* TSS node $\mathcal{B}$ sends the reply message for the user process to the next TSS node name in the message tag (`D`) from the reply message denoted by a, with the tail element of the message tag removed. The tuple space handle tag for the tuple space `TS1` is removed from within the tuple being sent by TSS node $\mathcal{B}$. TSS node $\mathcal{D}$ receives the reply message. The only element left in the message tag is the name of the user process which requires the result tuple (`P1`). TSS node $\mathcal{D}$ detects this and passes the tuple to the user process. When the user process receives the result tuple it becomes unblocked and continues executing. The tuple space movement can occur concurrently with the user process computation and other tuple space accesses.

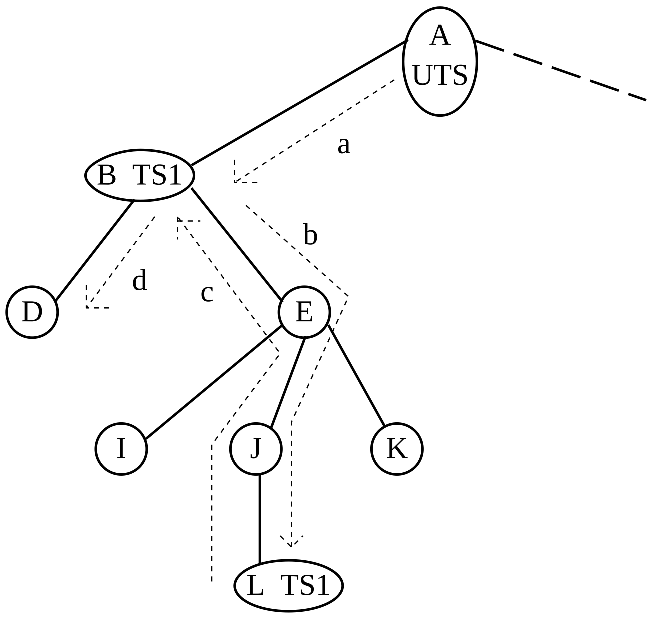


Figure 7.13: The second stage of performing the `in` primitive.

7.8 Conclusions

In this chapter the extension of the classification of tuple spaces to allow the development of hierarchical kernels which provide support for many workstations which are geographically diverse has been described. Before such a kernel could be implemented there are many other questions which need to be answered (see Chapter 8). In this chapter a detailed proposal of how tuple space movement would be used in such a kernel has been given. The classification of a tuple space within the hierarchical kernel is based on its position within the hierarchical kernel rather than on how an individual process perceives it. In the York Kernel II the LTSM was able to classify a tuple space as either a LTS or RTS. In this kernel a TSS node only knows if a tuple space resides locally or not. If it does not reside locally it must reside on a TSS node higher up the kernel.

In order to test the concepts a simple simulator has been created (written in ISETL[BDL89]). The simulator allows a tree structure for the kernel to be defined, and for the operations to be performed. It does not support the insertion and removal of tuples, but allows the insertion and removal of tuple space handles into and from tuple spaces. Messages can be inserted into TSS node message queues, simulating user processes sending messages to the TSS nodes. These messages can be observed moving through the hierarchical kernel, and tuple spaces are moved appropriately. The aim of the simulator was to allow the basic concepts of how and when tuple space movement occurs to be examined and checked.

Chapter 8

Conclusions and Future research

8.1 Introduction

In Chapter 1 two issues were raised:

- The sufficiency or otherwise of the original set of Linda primitives (given multiple tuple spaces), and
- how can the bulk primitives of `collect` and `copy-collect` be implemented efficiently within an *open* Linda implementation?

The aim of this dissertation has been to address these issues. The answer to the first is that there is the need for two extra primitives to be added to Linda. The first of these is the `collect` primitive which was proposed and justified by Butcher et al.[BWA94] and was introduced when the issue was first posed, the second primitive required is the `copy-collect` primitive which is proposed and justified within this dissertation in Chapters 3 and 4.

The second issue is answered by the development of a Linda run-time system using a novel technique to track and dynamically move tuples and tuple spaces. Because of the close relationship between the `collect` and `copy-collect` primitives and multiple tuple spaces the technique has allowed for the efficient implementation of these bulk primitives. The technique is described in detail in Chapter 5. Chapter 6 presented the experimental results obtained using a Linda run-time system (York Kernel II) which uses the techniques. The performance of the York Kernel II was shown to be better than the performance both of an open implementation which does not use the technique and of a closed implementation. In both cases a “real world” example was used.

Chapter 7 considered the shortcomings of the current implementation and a proposal for a more graduated approach to classifying tuple spaces by their position within a hierarchical kernel was presented. The proposed kernel supports many more geographically diverse workstations (and processes) than the York Kernel II. In the proposed kernel the cost of accessing a tuple space is linked to its position within the hierarchical kernel. The proposed kernel should provide similar performance for LAN based computing compared with the York Kernel II with the LTSM enabled, and support WAN based computing which is something the York Kernel II does not. A direct comparison of the proposed hierarchical kernel and the York Kernel II is unfair due to the different intended uses of the kernels, one is to provide support for LAN based computing (York Kernel II) and the other (hierarchical kernel) is to provide support for WAN based computing.

The work described within this dissertation has shown that the current Linda model is unable to perform a particular operation, called a multiple `rd`, efficiently. The addition of the `copy-collect` primitive has been shown through the use of experimental results to overcome the multiple `rd` problem.

The ability of the implementation to track tuple spaces and tuples, and then move them around in single operations, has led to a large performance improvement over traditional implementation techniques. The idea of moving tuples within a kernel is largely considered a bad approach by the Linda community. However, by moving the multiple tuples around the system in single operations, in a sensible and logical fashion the bulk movement of tuples has been shown to be advantageous. This is achieved by ensuring that, for most of the time, tuples are only moved within the kernel when the cost of retrieving those tuples is reduced for at least one user process. The intelligent movement of tuples can and will have to be used in future open kernels to ensure performance of the kernels are to be comparable with the performance of closed implementations.

The demand for geographically distributed computing, or Internet computing, is driving the need for systems that can handle thousands of workstations, and share information between them. The hierarchical kernel proposed in the Chapter 7 is potentially able to support many more workstations than traditional kernels. Although not implemented a detailed proposal was presented.

8.2 Future research

The specific research problems that follow on from this dissertation are:

- A formal semantics for the `copy-collect` primitive.

  There are no widely accepted formal semantics of Linda, and consequently there are no formal semantics for the `copy-collect` primitive. There is a need for a formal framework and a deeper understanding of the interaction of `copy-collect` primitive with other primitives.

- The control of tuple space handle passing.

  Throughout this dissertation the assumption has been that tuple space handles are passed through tuple spaces. Is this a fair assumption? If a process wants to pass a tuple space handle to another process how can this be performed reliably? In open systems the passing of tuple space handles through a global tuple space (such as `UTS`) is perhaps unacceptable. Further work is required to answer these questions. Following on from this should it be possible to control what the other processes do to that tuple space? Should it be possible to make tuple spaces read only, write only, etc.?

- A better `eval` mechanism.

  In the York Kernel II the `eval` primitive is mapped on to the spawn function of PVM. A better `eval` mechanism could be developed, which uses the concepts of `eval` servers[HKCG91, RA95], but for open implementations rather than closed implementations.

- The N-layer hierarchical kernel poses many questions which require future research.

  - TSS node distribution.

    Can the TSS nodes in the hierarchical tree be distributed as is the tuple space server in the York Kernel II, and indeed is it desirable? The reason why distributed kernels are currently used is because the kernels become bottlenecks, they are unable to process all the messages that they receive fast enough. The hierarchical kernel presents a new way of distributing tuples, based on the tuple space in which they reside. If the TSS nodes run on dedicated hardware and have high bandwidth network connections will they be able to service all the requests without being distributed themselves?

  - Caching and migration

    If distributed TSS nodes are used within the hierarchical kernel then the movement of tuples within the processes that combined make a single TSS node may provide increased performance. How to cache and migrate tuples efficiently within a TSS node has not been investigated as it is pointless within the context of a LAN implementation, but within the context of a WAN implementation it will have to be considered, if distributed TSS nodes are used.

  - Fault tolerance

    A hierarchical kernel with several thousand workstations would require some sort of fault tolerance, at least for the kernel. Current work on fault tolerance and Linda[Jeo96, JS94] provides an insight into how this may be achieved, but many questions still remain unanswered.

8.2.1 A "Linda" for distributed computing?

A more fundamental questioning of Linda and its suitability for "open" systems is perhaps needed, based on experiences gained from profiling the kernel and observations made whilst gathering the experimental results. Linda can be considered as having two parts; a tuple space model and a set of access primitives. Many of the desirable features of Linda are associated within the tuple space model. The spatial and temporal separation of processes and the asynchronous nature of Linda is provided by the tuple space model rather than the access primitives to the tuple spaces. Are the Linda primitives suitable for open implementations for a network of workstations?

To date Linda has primarily been used for small scale distributed processing. A small number of workstations connected by a local area network. The characteristics of Linda are that processes have asynchronous communication and are both spatially and temporally separated so this should make it ideal for geographically distributed processing or multiprogramming.

In Chapter 1 it is stated that:

> *The only way that two processes can communicate is via a tuple space, and therefore Linda provides asynchronous communications between processes.*

This is correct as between user processes there is asynchronous communication. A process places a tuple in a tuple space and continues with the user's program. Another process then reads this tuple, and the two user processes have communicated asynchronously. However, the access to the tuple space is not asynchronous (except for an `out` primitive if `out` ordering is not supported).

The descriptions of the Linda primitives state that the only primitives that block are the `in` and `rd` primitives *if* a tuple is *not* available. This implies that a process which uses the Linda primitives should only block if either an `in` or a `rd` primitive is performed and the required tuple is not available. Pragmatically, in any practical Linda system, a process will always "wait" even if the required tuples are available because of the overheads associated with finding the matching tuple and, as far as the process is concerned, *waiting* and *blocking* are indistinguishable. Therefore, most of the Linda primitives "block" unnecessarily. While the user processes are blocked they cannot perform computation, so the current Linda primitives provide synchronous tuple space access. To overcome this, Linda should be split into two distinct sections, the Linda primitives and the tuple space model. The tuple space model represents the concept of shared tuple spaces containing tuples. The primitives which are used to interface to the tuple space model should be tailored to requirements of the user and the environment. Therefore, for distributed computing, where the communication times can be large and subsequently the times that primitives spend "waiting" can be large, a new set of access primitives should be defined, such as the BONITA primitives[RW97], which embody the idea of asynchronous tuple space access.

8.3 Contributions

8.3.1 Multiple `rd` operation

A multiple `rd` operation is where more than one process wishes to non-destructively access a subset of all the tuples in a tuple space which match the same template. In Chapter 3 an experimental study of the multiple `rd` operation was presented. The use of the "stream" approach and the use of semaphores (lock tuples) were shown to overcome the problem, although not in a satisfactory manner. The semaphore approach was a sequential solution, and the stream approach required the checking of all tuples in a tuples space, regardless of whether they were actually required.

8.3.2 Proposal for the adoption of the `copy-collect` primitive

In Chapter 4 a new primitive was proposed, which overcomes the multiple `rd` problem. Experimental results, using a naive implementation of the `copy-collect` primitive were given which show that the `copy-collect` primitive can be used to provide an efficient solution to the multiple `rd` problem. Some simple performance models of the different approaches to overcoming the multiple `rd` problem were presented, in order to show that in general, the performance of the method using the `copy-collect` primitive will be better.

8.3.3 A novel kernel for Linda

The York Kernel II is presented which uses dynamic movement of tuples and tuple spaces to improve performance. The kernel uses *implicit* information provided within Linda programs, rather than expecting the user to provide explicit information. Chapter 6 shows that the performance of the kernel, when using the multiple tuple spaces, and the `collect` and `copy-collect` primitives to be better than a traditional implementation. A "real life" program was used to show the performance gain over a commercial closed C-Linda implementation.

8.3.4 Detailed description of a hierarchical kernel

In Chapter 7 the shortcomings of the York Kernel II and a detailed description of the structure of a generalised hierarchical kernel which overcomes these problems was presented. The kernel should support greater numbers of workstations than current implementations.

8.4 Closing remarks

The work in this dissertation represents the foundation for future work on distributed run-time systems providing shared tuple spaces for inter-process communication and coordination that support large numbers of workstations and computing devices.

Appendix A

Overview of the Linda implementations

Feature	York Kernel I	York Kernel II	SCA C-Linda
Primitives			
`in`	⋆	⋆	⋆
`out`	⋆	⋆	⋆
`rd`	⋆	⋆	⋆
`inp`	⋆		⋆
`rdp`	⋆		⋆
`collect`	⋆	⋆	
`copy-collect`	⋆	⋆	
Platform	Meiko / LAN	LAN	LAN
Multiple tuple spaces	⋆	⋆	
Open implementation	⋆	⋆	
Closed implementation			⋆
Compile-time analysis			⋆
Distributed tuple storage	⋆	⋆	⋆
Bulk movement of tuples		⋆	
Bulk movement of tuple spaces		⋆	
`out` ordering		⋆	

Table A.1: A comparison of the three Linda implementations used within this dissertation.

Multiple tuple spaces

The original Linda model included only one tuple space, referred to as the Global Tuple Space (`GTS`). More recent implementations allow multiple tuple spaces. More information can be found in Section 2.3.1.

Open implementation

Implementations which are open allow processes and programs to join and leave the system at will. Open implementations embody the ideas of persistence, where once a tuple space is created it will remain within the system. They also allow the notions of temporal and spacial separation present within Linda. A process can communicate with another process (program) that has not yet been written before the first process terminates. More information can be found in Chapter 5.

Closed implementation

A closed implementation does not allow processes to join and leave freely. In its weakest form information about all the processes that wish to communicate must be present when the system starts. However, normally with closed implementations compile time analysis (see next section) is used, therefore all the source code for the processes which wish to communicate must be available at compile/link time. More information can be found in Chapter 5.

Compile time analysis

Compile time analysis is used by some closed implementations to gain information that allows more efficient run-time support. In the simplest case this may be transforming recognisable coordination structures into more efficient ones. Within the context of closed implementations compile-time analysis would be used to calculate the most efficient tuple distribution strategies. In many cases the use of tuples between different processes, can be reduced to passing a message directly between the two processes, rather than using tuple spaces. More information can be found in Chapters 5.

Distributed tuple storage

In a run-time system using distributed tuple storage the tuples are not stored on a single server, but distributed over a number of processors (or workstations). This is done because a single server can become a bottleneck. More information can be found in Chapter 5.

Bulk movement of tuples and Bulk movement of tuples spaces

This is where the implementation is able to dynamically and intelligently move blocks of tuples and tuple spaces in order to achieve better performance. If this approach is not adopted tuples remain in the same physical position within the run-time system from insertion by a user process until they are requested by another user process. More information can be found in Chapters 5, 6 and 7.

Appendix B

Source code for experimental results

This appendix contains the C-Linda source code for each of the experiments one to six presented in Chapter 6.

B.1 Experiment one

Program B.1 Program used for experiment one.

```
#include <linda.h>

int main(int argc, char *argv[])
{
  int lp, tmp;

  start_timer();                /* Initialise the timer */
  for (lp = 0; lp < 1000; lp++)
    out(UTS, lp);               /* Place 1000 tuples into UTS */
  timer_split("Done outs.");    /* Note current time */
  for (lp = 0; lp < 1000; lp++)
    in(UTS, ?tmp);              /* Retrieve 1000 from UTS */
  timer_split("Finished.");     /* Note current time */
  print_times();                /* Print timings */
  return 0;
}
```

B.2 Experiment two

Program B.2 Program used for experiment two.

```
#include <linda.h>

int main(int argc, char *argv[])
{
  int lp, tmp;
  TS ts;

  start_timer();
  ts = tsc();                    /* Create a LTS */
  for (lp = 0; lp < 1000; lp++)
    out(ts, lp);                 /* Place 1000 tuples into LTS */
  timer_split("Done outs.");
  for (lp = 0; lp < 1000; lp++)
    in(ts, ?tmp);                /* Retrieve 1000 from LTS */
  timer_split("Finished.");
  print_times();
  return 0;
}
```

B.3 Experiment three

Program B.3 Program used for experiment three.

```
#include <linda.h>

int main(int argc, char *argv[])
{
  int lp, tmp;
  TS ts;

  start_timer();
  ts = tsc();                  /* Create a LTS */
  for (lp = 0; lp < 1000; lp++)
    out(ts, lp);
  timer_split("Done outs.");
  out(UTS, ts);                /* LTS becomes a RTS */
  timer_split("Done out.");
  for (lp = 0; lp < 1000; lp++)
    in(ts, ?tmp);
  timer_split("Finished.");
  print_times();
  return 0;
}
```

B.4 Experiment four

Program B.4 Program used for experiment four.

```
#include <linda.h>

int main(int argc, char *argv[])
{
  int lp, tmp;
  char *ts;

  start_timer();
  ts = tsc();
  for (lp = 0; lp < 1000; lp++)
    out(ts, lp);
  timer_split("Done outs.");
  copycollect(ts, UTS, ?int);         /* LTS -> RTS */
  timer_split("Done copy-collect.");
  for (lp = 0; lp < 1000; lp++)
    in(UTS, ?tmp);
  timer_split("Finished.");
  print_times();
  return 0;
}
```

B.5 Experiment five

Program B.5 Program used for experiment five.

```
#include <linda.h>

int main(int argc, char *argv[])
{
  int lp, tmp;
  char *ts;

  start_timer();
  ts = tsc();
  for (lp = 0; lp < 1000; lp++)
    out(UTS, lp);
  timer_split("Done outs.");
  copycollect(UTS, ts, ?tmp);         /* RTS -> LTS */
  timer_split("Done copy-collect.");
  for (lp = 0; lp < 1000; lp++)
    in(ts, ?tmp);
  timer_split("Finished.");
  print_times();
  return 0;
}
```

B.6 Experiment six

Program B.6 Program used for experiment six.

```
#include <linda.h>

int main(int argc, char *argv[])
{
  int lp, tmp;
  char *ts;

  start_timer();
  ts = tsc();
  for (lp = 0; lp < 1000; lp++)
    out(ts, lp);
  timer_split("Done outs.");
  collect(ts, UTS, ?tmp);            /* LTS -> RTS */
  timer_split("Done collect.");
  copycollect(UTS, ts, ?tmp);        /* RTS -> LTS */
  timer_split("Done copy-collect.");
  for (lp = 0; lp < 1000; lp++)
    in(ts, ?tmp);
  timer_split("Finished.");
  print_times();
  return 0;
}
```

Bibliography

[ACG94] S. Ahmed, N. Carriero, and D. Gelernter. A program building tool for parallel applications. In *DIMACS Workshop on Specifications of Parallel Algorithms*. Princeton University, USA, 1994.

[ACGK88] S. Ahuja, N. Carriero, D. Gelernter, and V. Krishnaswamy. Matching language and hardware for parallel computation in the Linda machine. *IEEE Transactions on Computers*, 37(8):921–929, 1988.

[Ams95] D. Amselem. A window on shared virtual environments. *Presence-teleoperators and virtual environments*, 4(2):140–145, 1995.

[AS91] B. Anderson and D. Shasha. Persistent Linda: Linda + Transactions + Query Processing. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

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

[Ass96] Scientific Computing Associates. *Paradise: User's guide and reference manual*. Scientific Computing Associates, 1996.

[Ban96] M. Banville. Sonia: an adaption of Linda for coordination of activities in organisations. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 57–74. Springer-Velag, 1996.

[BCG89] R. Bjornson, N. Carriero, and D. Gelernter. The implementation and performance of Hypercube Linda. Technical Report YALEU/DCS/RR-690, Yale University, 1989.

[BDL89] Nancy Baxter, Ed Dubinsky, and Gary Levin. *Learning discrete mathematics with ISETL*. Springer Verlag, 1989.

[BJ96] P. Brooke and J. Jacob. Expressing Linda in CSP. Technical Report PLASMA 21, Department of Computer Science, University of York, 1996.

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

[BKS91] R. Bjornson, C. Klob, and A. Sherman. Ray tracing with Network Linda. *SIAM News*, 24(1), 1991.

[BS92] J. Black and C. Su. Networked parallel seismic computing. In *24th Annual Offshore Technology Conference*, pages 169–176, 1992.

[But90] P. Butcher. A behavioural semantics for Linda-2. Technical Report YCS 137, Department of Computer Science, University of York, 1990.

[BW91] K. De Bosschere and L. Wulteputte. Mutli-Prolog: Implementation on an 88000 shared memory multiprocessor. Technical Report DG 91-19, University of Ghent, Belgium, 1991.

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

[Cag93] L. Cagan. Investment analysis on networked workstations. *High Performance Computing Review*, 1993.

[Cam96] D. Campbell. On the implementation of an asymmetric hyperspace in linear memory: implementing tuple spaces. Technical Report YCS 274, Department of Computer Science, University of York, 1996.

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

[Car95] N. Carriero. Private communication, 1995.

[CCH91] C.J. Callsen, I. Cheng, and P.L Hagen. Optimizing Linda. Master's thesis, University of Aalborg, Denmark, 1991.

[CCZ93] L. Cagan, N. Carriero, and S. Zenios. A computer network approach to pricing mortgage-backed securities. *Financial Analyst's Journal*, pages 55–62, 1993.

[CdHMW92] P. Clayton, F. de Heer-Menlah, and E. Wentworth. Placing processes in a Transputer-based Linda programming environment. Technical report, Rhodes University, 1992.

[CFG93] N. Carriero, E. Freeman, and D. Gelernter. Adaptive parallelism on multiprocessors: Preliminary experience with Piranha on the CM-5. Technical Report 969, Yale University, 1993.

[CFGK94] N. Carriero, E. Freeman, D. Gelernter, and D. Kaminsky. Adaptive parallelism and Piranha. Technical Report 954, Yale University, 1994.

[CG89a] N. Carriero and D. Gelernter. How to write parallel programs: A guide to the perplexed. *ACM Computing Surveys*, 21(3):323–357, 1989.

[CG89b] N. Carriero and D. Gelernter. Linda in context. *Communications of the ACM*, 32(4):444–458, 1989.

[CG90a] N. Carriero and D. Gelernter. *How to write parallel programs: A first course*. MIT Press, 1990.

[CG90b] N. Carriero and D. Gelernter. Tuple analysis and partial evaluation strategies in the Linda precompiler. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Research Monographs in Parallel and Distributed Computing, pages 114–125. MIT Press, 1990.

[CG91a] N. Carriero and D. Gelernter. A foundation for advanced compile-time analysis of Linda programs. In *Languages and Compilers for Parallel Computing*, volume 589 of *Lecture Notes in Computer Science*, pages 389–404. Springer-Verlang, 1991.

[CG91b] N. Carriero and D. Gelernter. New optimization strategies for the Linda precompiler. In G. Wilson, editor, *Linda-like systems and their implementation*, Edinburugh Parallel Computing Centre, pages 74–82. Technical Report 91-13, 1991.

[CG92] N. Carriero and D. Gelernter. Data parallelism and Linda. In *Languages and Compilers for Parallel Computing*, volume 757 of *Lecture Notes in Computer Science*, pages 145–159. Springer-Verlang, 1992.

[CG93] N. Carriero and D. Gelernter. Linda and message passing: What have we learned? Technical Report YALEU/DCS/RR-984, Yale University, 1993.

[CGKW93] N. Carriero, D. Gelernter, D. Kaminsky, and J. Westbrook. Adaptive parallelism with Piranha. Technical Report YALEU/DCS/RR-954, Yale University, 1993.

[CGZ95] N. Carriero, D. Gelernter, and L. Zuck. Bauhaus Linda. In Paolo Ciancarini, Oscar Nierstrasz, and Akinori Yonezawa, editors, *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 66–76. Springer-Velag, 1995.

[Cia91] P. Ciancarini. PoliS: a programming model for multiple tuple spaces. In *Proceedings of the sixth International Workshop on Sofware Specification and Design*, pages 44–51, 1991.

[CJY95] P. Ciancarini, K.K. Jensen, and D. Yankelvich. On the operational semantics of a coordination language. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 77–106. Springer-Velag, 1995.

[COW96] D. Campbell, H. Osbourne, and A. Wood. A survey of Linda semantics and design issues. Technical Report YCS 277, Department of Computer Science, University of York, 1996.

[CSS94] J. Carreria, L. Silva, and J. Silva. On the design of Eilean: A Linda-like library for MPI. Technical report, Universidade de Coimbra, 1994.

[CW92] P. Clayton and E. Wentworth. Placing processes in a transputer-based Linda programming environment. Technical report, Rhodes University, 1992.

[DF86] M.J.B. Duff and T.J. Fountain. *Cellular Logic Image Processing*, pages 20–22. Academic Press, 1986.

[DH72] R.O. Duda and P.E. Hart. Use of the Hough Tranformation to detect lines and curves in pictures. *Communications of the ACM*, 15(1):11–15, 1972.

[dHM91] F. de Heer-Menlah. Analyzing communication flow and process placement in Linda programs on transputers. Technical report, Rhodes University, 1991.

[DRRW96] A. Douglas, N. Röjemo, C. Runciman, and A. Wood. Astro-gofer: Parallel functional programming with co-ordinating processes. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 686–693. Springer-Verlang, 1996.

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

[DWR95] A. Douglas, A. Wood, and A. Rowstron. Linda implementation revisited. In P. Nixon, editor, *Transputer and occam developments*, Transputer and occam Engineering Series, pages 125–138. IOS Press, 1995.

[Faa91] C. Faasen. Intermediate uniformly distributed tuple space on transputer meshes. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.

[FGK$^+$91] M. Factor, D. Gelernter, C. Kolb, P. Miller, and D. Sittig. Real-time performance, parallelism and program visualisation in medical monitoring. *IEEE Computer*, 1991.

[FGY95] M. Feng, Y. Gao, and C. Yuen. Implementing Linda tuple space on a distributed system. *International Journal of High Speed Computing*, 7(1):125–144, 1995.

[GC92] D. Gelernter and N. Carriero. Coordination languages and their significance. *Communications of the ACM*, 35(2):97–107, 1992.

[Gel85] D. Gelernter. Generative communication in Linda. *ACM Transactions on Programming Languages and Systems*, 7(1):80–112, 1985.

[Gel89] D. Gelernter. Multiple tuple spaces in Linda. In E. Odijk, M. Rem, and J.-C. Syre, editors, *PARLE '89: Parallel Architectures and Languages Europe. Volume II: Parallel Languages*, volume 366 of *Lecture Notes in Computer Science*, pages 20–27. Springer-Verlang, 1989.

[GJK93] D. Gelernter, M.R. Jourdenais, and D. Kaminsky. Piranha scheduling: Strategies and their implementation. Technical Report 983, Yale University, 1993.

[GK92] D. Gelernter and D. Kaminsky. Supercomputing out of recycled garbage: Preliminary experience with Piranha. In *Proceedings of the Sixth ACM International Conference on Supercomputing*, pages 19–23, July 1992.

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

[Has94] W. Hasselbring. *Prototyping Parallel Algorithms in a set-orientated language*. PhD thesis, University of Essen, 1994.

[HKCG91] S. Hupfer, D. Kaminsky, N. Carriero, and D. Gelernter. Coordination applications of Linda. In J.P. Banâtre and D. Le Métayer, editors, *Research Directions in High-Level Parallel Programming Languages*, volume 574 of *Lecture Notes in Computer Science Volume*, pages 187–194. Springer-Verlang, 1991.

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

[Jag91] S. Jagannathan. Optimizing analysis for first-class tuple spaces. In A. Nicolau, D. Gelernter, T. Gross, and D. Padua, editors, *Advances in Languages and Compilers for Parallel Computing*, pages 44–70. MIT Press, 1991.

[Jel90] R. Jellinghaus. Eiffel Linda: An object-orientated Linda dialect. *ACM SIGPLAN Notices*, 25(12):70–84, 1990.

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

[Jeo96] K. Jeong. *Fault-tolerant parallel processing combining Linda, checkpointing and transactions*. PhD thesis, New York University, 1996.

[JS94] K. Jeong and D. Shasha. Persistent Linda 2: a transaction/checkpointing approach to fault-tolerant Linda. In *Proceedings of the 13th Symposium on Fault-Tolerant Distributed Systems*, 1994.

[KACG87] V. Krishnaswamy, S. Ahuja, N. Carriero, and D. Gelernter. The Linda machine. In *Proceedings 1987 Princeton Workshop on Algorithm Architecture and Technology Issues for Models of Concurrent Computations*, pages 697–717, 1987.

[KACG88] V. Krishnaswamy, S. Ahuja, N. Carriero, and D. Gelernter. Architecture of a Linda co-processor. In *Proceedings 15th Annual International Symposium on Computer Archiatecture*, pages 240–249, 1988.

[Kie96] T. Kielmann. Designing a coordination model for open systems. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 267–284. Springer-Velag, 1996.

[LA92] K. Landry and J. Arthur. Instructional footprinting: A basis for exploiting concurrenty through instructional decompostion and code motion: A research prospectus. Technical Report TR 92-33, Virginia Polytechnic Institute and State University, 1992.

[LA93a] K. Landry and J. Arthur. Achieving asynchronous speedup while preserving synchronous semantics: An implementation of instructional footprinting in Linda. Technical Report TR 93-33, Virginia Polytechnic Institute and State University, 1993.

[LA93b] K. Landry and J. Arthur. Instructional footprinting and semantic preservation in Linda. Technical Report TR 93-22, Virginia Polytechnic Institute and State University, 1993.

[LA95] K. Landry and J. Arthur. Instructional footprinting and semantic preservation in Linda. *Concurrency: Practice and Experience*, 7(3):191–207, 1995.

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

[Men96] R. Menezes. Garbage collection in multiple tuple space Linda. 1st year DPhil report, Department of Computer Science, University of York, 1996.

[MM91] F. Musgrave and B. Mandelbrot. The art of the fractal landscape. *IBM Journal of Research and Development*, 1991.

[MP93] G. Matos and J. Purtilo. Reconfiguration of hierarchical tuple-spaces: Experiments with Linda-Polylith. Technical Report CSD TR 3153, University of Maryland, 1993.

[Nar89] J. Narem Jr. An informal operational semantics of C-Linda V2.3.5. Technical Report YALEU/DCS/TR-839, Yale University, 1989.

[NB92] I. Nelken and R. Bjornson. Fast lady. *RISK*, 5(4), 1992.

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

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

[Pin91] J. Pinakis. A distributed typeserver and protocol for a Linda tuple space. Technical report, University of Western Australia, 1991.

[RA95] P. Robinson and J. Arthur. Distributed process creation within a shared data space framework. *Software: Practice and Experiance*, 25(2):175–191, 1995.

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

[RDW96] A. Rowstron, A. Douglas, and A. Wood. Copy-collect: A new primitive for the Linda model. Technical Report YCS 268, Department of Computer Science, University of York, 1996.

[Row95] A. Rowstron. Fined grained parallellism using Linda: Beating synchronisation costs. Unpublished internal research note, 1995.

[RW95] A. Rowstron and A. Wood. Implementing mathematical morphology in ISETL-Linda. In *IEE 5th International Conference on image processing and its applications*, pages 847–851, 1995.

[RW96a] A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of workstations. In L. Bougé, P. Fraigniaud, A. Mignotte, and Y. Robert, editors, *Euro-Par'96*, volume 1123 of *Lecture Notes in Computer Science*, pages 510–513. Springer-Verlang, 1996.

[RW96b] A. Rowstron and A. Wood. An efficient distributed tuple space implementation for networks of workstations. Technical Report YCS 270, Department of Computer Science, University of York, 1996.

[RW96c] A. Rowstron and A. Wood. Solving the Linda multiple `rd` problem. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 357–367. Springer-Velag, 1996.

[RW97] A. Rowstron and A. Wood. BONITA: A set of tuple space primitives for distributed coordination. In Hesham El-Rewini and Yale N. Patt, editors, *Proceedings of the 30th Hawaii International Conference on System Sciences,* volume 1, pages 379–388. IEEE Computer Society Press, January 1997.

[SAB94] R. Seyfarth, S. Arumugham, and J. Bickham. Glenda users guide. Technical report, University of Southern Mississippi, 1994.

[SCM93] J. Silva, J. Carreria, and F. Moreria. ParLin: from a centralized tuple space to adaptive hashing. Technical report, Universidade de Coimbra, 1993.

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

[SLSC92] R. Sues, Y. Lua, M. Smith, and L. Cagan. PFRAM's parallel solution can determine fatigue life. *High Performance Computing Review*, 1992.

[SSWC88] P.K. Sahoo, S. Saltani, A.K.C Wong, and Y.C. Chen. A survey of thresholding techniques. *Computer Vision, Graphics and Image Processing*, 41:233–260, 1988.

[SVS94] L. Silva, B. Veer, and J. Silva. The Helios tuple space library. In *Proceedings 2nd Euromicro workshop on Parallel and Distributed Processing*, pages 325–333, 1994.

[Tol95a] R. Tolksdorf. *Coordination in open distributed systems*. PhD thesis, Berlin University, 1995.

[Tol95b] R. Tolksdorf. A machine for uncoupled coordination and its concurrent behaviour. In *Object-Based Models and Languages for Concurrent Systems*, volume 924 of *Lecture Notes in Computer Science*, pages 176–193. Springer-Velag, 1995.

[Tol96] R. Tolksdorf. Coordinating services in open distributed systems with Laura. In Paolo Ciancarini and Chris Hankin, editors, *Coordination Languages and Models, Proceedings of Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 386–402. Springer-Velag, 1996.

[WC95] G. Wells and A. Chalmers. An extended Linda system using PVM. In *PVM Users' Group Meeting, Pittsburgh*, 1995.

[Wil91] G. Wilson. Improving the performance of generative communication systems by using application-specific mapping functions. In *Linda-like systems and their implementation*, volume Technical report 91-13, pages 129–142. EEPC, 1991.

[WMM$^+$92] A.M. Wallace, G.J. Michaelson, P. McAndrew, K.G. Waugh, and W.J. Austin. Dynamic control and prototyping of parallel algorithms for intermediate- and high-level vision. *Computer*, pages 43–45, 1992.

[Woo96] A. Wood. Private communication, 1996.

[WR95] A. Wood and A. Rowstron. Deadlock and algorithm design: Stable marriages in Linda. *Unpublished*, 1995.

[YA85] S. Yalamnchili and J. Aggarwal. A system organization for parallel image processing. *Pattern Recognition*, 18:17–29, 1985.

[YFY96] C. Yuen, M. Feng, and J. Yee. BaLinda suite of languages and implementations. *Journal of Software Systems*, 32:251–267, 1996.

[Zen90] S. Zenith. Linda coordination language; subsystem kernel architecture (on transputers). Technical Report YALEU/DCS/RR-794, Yale University, 1990.

[ZG96] H. Zhou and A. Geist. "Receiver Makes Right" data conversion in PVM. Technical report, Oak Ridge National Laboratory, 1996.