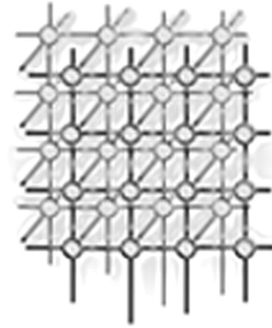
CONCURRENCY—PRACTICE AND EXPERIENCE
Concurrency: Pract. Exper. 2003; **00**:1–7   *Prepared using* cpeauth.cls *[Version: 2001/03/05 v2.01]*



Linda implementations in Java for concurrent systems

G. C. Wells[1,*,§], A. G. Chalmers[2] and P. G. Clayton[1]

[1] *Department of Computer Science, Rhodes University, Grahamstown, 6140, South Africa*
[2] *Department of Computer Science, University of Bristol, Merchant Venturers Building, Bristol BS8 1UB, U.K.*

SUMMARY

This paper surveys a number of the implementations of Linda that are available in Java. It provides some discussion of their strengths and weaknesses, and presents the results from benchmarking experiments using a network of commodity workstations. Some extensions to the original Linda programming model are also presented and discussed, together with examples of their application to parallel processing problems.

KEY WORDS: Linda; Java; associative matching

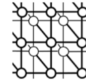
INTRODUCTION

The Linda[†] coordination language for concurrent programming was first proposed by David Gelernter at Yale University in the mid-1980's[1]. Despite considerable initial interest in this approach, and the establishment of a company to produce and support commercial implementations of Linda, the Linda model waned in popularity in the 1990's. However, in recent years there has been a resurgence in interest, particularly with regard to Java[‡] implementations of Linda. This paper surveys the current state of Linda research and development, focussing specifically on Java implementations of Linda. Performance measurements for the more popular, commercial implementations are presented to allow a comparison of these.

Furthermore, our own work has pointed to some problems with the associative matching technique used for data retrieval in Linda. These problems have potentially serious performance

*Correspondence to: Department of Computer Science, Rhodes University, Grahamstown, 6140, South Africa
§E-mail: G.Wells@ru.ac.za
†Linda is a registered trademark of Scientific Computing Associates.
‡Java is a registered trademark of Sun Microsystems, Inc.

Received 29 January 2003
Revised 17 June 2003
Copyright © 2003 John Wiley & Sons, Ltd.



implications, which are discussed. The survey of Linda systems in the first section of the paper concentrates on a number of systems that have introduced alternative matching mechanisms. Our proposed solution to the problems inherent in simple associative matching has been implemented in a system called *eLinda*. The eLinda system is described briefly, together with an example illustrating its application to the problem of visual language parsing.

The Linda programming model

Linda is a coordination language for parallel and distributed processing, providing a communication mechanism based on a logically shared memory space called *tuple space*. On a shared memory multi-processor the tuple space may actually be shared, but on distributed memory systems (such as a network of workstations) it is usually distributed among the processing nodes. We will focus on distributed memory systems in this paper. Whatever the implementation strategy that is employed, the tuple space is structured as a bag of *tuples*. An example of a tuple with three fields is `("point", 12, 67)`, where 12 and 67 are the x and y coordinates of the point represented by this tuple.

As a coordination language, Linda is designed to be integrated with a sequential programming language (called the *host language*—Java in the case of the implementations discussed in this paper). Linda effectively provides an application programmer with a small set of operations that may be used to place tuples into tuple space (`out`) and to retrieve tuples from tuple space (`in` which removes the tuple, and `rd` which returns a copy of the tuple, leaving the tuple in tuple space). The two "input" operations also have predicate forms (`inp` and `rdp`) which do not block if the required tuple is not present, but return immediately with an indication of failure. The specification of the tuple to be retrieved makes use of an associative matching technique whereby a subset of the fields in the tuple have their values specified and these are used to locate a matching tuple in the tuple space. For example, if a point such as that in the example above was required then the following operation would retrieve it: `in("point", ?x, ?y)`. The tuple specification here, `("point", ?x, ?y)`, is referred to as an *anti-tuple*. Any tuple with the same number and type of fields and with the string `"point"` in the first position would match this request. When a successful match is made the variables `x` and `y` are assigned the values of the corresponding fields of the matching tuple.

The original Linda model also provides for dynamic process creation by means of the `eval` operation. This is not an essential part of the paradigm, and it has been shown that `eval` may be implemented in terms of the other operations with some support from a preprocessor[2]. The `eval` operation will not be considered any further in this paper.

On distributed memory systems there are many possible strategies for implementing the shared tuple space[1, 3, 4]. These issues are transparent to the programmers and users of the system, who perceive the tuple space as a single, shared resource. A common approach is to make use of a centralised server, but this may become a bottleneck. Two other common implementation strategies are hashing systems where the contents of tuples are used to allocate them to particular processors, and partitioned systems where tuples with a common structure are allocated to a specific processor. Another approach is to have the tuple space fully distributed, such that any tuple may reside on any processing node.

Copyright © 2003 John Wiley & Sons, Ltd.

The small set of operations, the associative retrieval mechanism and the shared tuple space all combine to provide a very useful simplicity and flexibility for constructing concurrent applications. On the other hand, a criticism of Linda has been that it is, at worst, inefficient, and, at best, subject to unpredictable performance[5], as the simplicity of the model hides the underlying complexity of the data sharing and communication required.

Further details of the Linda model of concurrent programming may be found in [6].

JAVA IMPLEMENTATIONS OF LINDA

As mentioned in the introduction, recent years have seen a resurgence in interest in the Linda approach to concurrent programming, especially in Java. One of the first steps in this direction was the development of JavaSpaces by Sun Microsystems, with input from the original Yale Linda team. This was produced by Sun as a component of the Jini project, intended to simplify the networking of heterogeneous systems[7]. As with many of the aspects of Java technology developed by Sun, JavaSpaces was intended as a model, or an architecture, that could be implemented by third parties. Consequently, the JavaSpaces product released by Sun is intended simply as a reference implementation. This philosophy has recently been supported by GigaSpaces Technologies Ltd. who have implemented the JavaSpaces specfication in their GigaSpaces[§] product. IBM have also developed a Java-based Linda system, called TSpaces, in their alphaWorks research division. This contains a number of extensions to the original Linda programming model.

In addition to these systems developed with the backing of commercial organisations, there have been a number of academic research projects that have implemented Linda-like systems in Java. The following sections outline the unique features of these commercial and research systems.

TSpaces

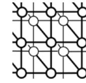
In IBM's words TSpaces is intended as "the common platform on which we build links to all system and application services"[8]. Within this grand vision they identify "Tier 0 devices" (i.e. systems smaller than traditional desktop or laptop machines, such as PDA's, embedded processors, etc.) as a particular area of interest[9].

The implementation of TSpaces is simple and elegant—all that is required is that a single server process be running on the network. The server makes use of a textual configuration file, and provides a useful web interface for monitoring and configuration purposes. Applications wishing to make use of the TSpaces service need only know the network hostname of the computer running the server.

TSpaces provides a large number of operations over and above the basic Linda operations (which have slightly different names). There is a `delete` operation that will simply delete

[§]GigaSpaces is a registered trademark of GigaSpaces Technologies Ltd.

Copyright © 2003 John Wiley & Sons, Ltd.



a matching tuple from the tuple space without returning it to the application. There are also operations for the input and output of multiple tuples: `scan`, `countN`, `consumingScan`, `deleteAll`, `multiWrite` and `multiUpdate`. There are a number of operations that specify tuples by means of a "tuple ID" rather than the usual associative matching mechanisms: `update`, `readTupleById` and `deleteTupleById`. There is also the `rhonda` operator, which performs an atomic synchronisation and data exchange operation between two processes.

Lastly, TSpaces provides an "event registration" mechanism. This allows a process to request notification when a certain tuple is written to the tuple space or deleted from it. While the original Linda model does not provide any form of event notification, it is not difficult to emulate efficiently for writes, using threads and the standard `rd` operation. Event notification on deletions is possible to emulate by polling with `rdp`, but this is not efficient.

TSpaces transports tuples across the network using the standard Java object serialisation mechanisms and TCP/IP sockets. Tuples are simply objects that consist of a number of `Field` objects (or `FieldPS` objects which preserialise to a byte array to allow the server to work with unknown classes). Wildcard or formal values for anti-tuples are specified by `Field` objects containing a class type (e.g. `String.class`), rather than a data value. (TSpaces thus restricts tuples to containing objects, not primitive values, for matching purposes). Matching is performed using the standard `equals` method, and, in some cases, the `compareTo` method, specified by the `Comparable` interface. Matching can be done using so-called "indexed tuples". In this case the fields may be named, ranges of values may be included in the matching process, and AND and OR operations may be specified. These features may all be used in combination. It is also possible to perform matching on XML¶ data contained in tuples.

Tuples may have an expiration time set, and there is support for transactions. Furthermore, access control is provided for tuple spaces. This is based on user names, passwords and groups, and provides a level of control similar to that of the UNIX file access control mechanisms. TSpaces also provides persistence for the tuple spaces.

TSpaces does not provide a preprocessor, as originally envisaged by the original Yale researchers. The current implementation of TSpaces makes use of a centralised server model, which may become a performance bottleneck.

Adding new commands

New commands can be added to the TSpaces system relatively easily. This ability, together with the rich set of operations supported and complex matching criteria described above, provides a facility similar to the extended features of eLinda. This section presents a brief overview of the mechanisms for adding new commands to TSpaces.

The implementation of TSpaces makes use of a number of layers of software. At the lowest level the tuples themselves are stored in a form of database. This may be an actual database product (such as IBM's DB2), or simply some form of data structure in the computer's main memory. Above this is the tuple management layer, which handles the retrieval of tuples from

¶Extensible Markup Language, a specification for structured documents produced by the World Wide Web Consortium[10].

Copyright © 2003 John Wiley & Sons, Ltd.

the tuple space database. Above this layer (and accessed through a well-defined API) is the operator management level. This is comprised of a number of “factory” objects arranged in a list. The factories are responsible for creating “tuple handlers” for each command that is passed to the tuple space. If a factory does not recognise a particular command then it is passed down to the next factory in the list.

Users with appropriate permission levels can add new factories and handlers to the system dynamically, providing a great deal of flexibility. However, this is a complex process from a programmer’s perspective, as has also been noted by Foster *et al*[11]. Some of the complexity could perhaps be handled by providing classes with methods to automate the installation and initialisation of the new tuple handlers, but this would have to be done by the writers of the new factories and command handlers.

JavaSpaces

JavaSpaces[12] is a complex product and relies heavily on a number of other technologies from Sun. As stated in the introduction to this section, it forms part of the Jini system, and so makes extensive use of the Jini API[7]. Network support is provided by the Java RMI (Remote Method Invocation) protocol[13]. Furthermore, distribution of classes to clients is handled by the standard Internet hypertext protocol (HTTP). This means that before a JavaSpaces application can be started the following set of services must be running:

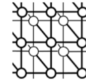
- a web (HTTP) server (a minimal one is provided with the Jini/JavaSpaces release)
- an RMI activation server (part of the standard RMI software bundled with Java)
- a Jini lookup service (alternatively the RMI registry service can be used, but this is discouraged as support for this option may be discontinued by Sun in the future)
- a Jini transaction manager
- a JavaSpaces server

Most of these services (and any application programs) also require extensive setting of command line parameters, further adding to the overall complexity of using JavaSpaces. Applications are also required to run a security manager, whether security checking is required or not. A typical command line required to run a JavaSpaces application is as follows:

```
java -Djava.security.policy=D:\JavaProgs\policy.all
  -Doutrigger.spacename=JavaSpaces
  -Dcom.sun.jini.lookup.groups=public
  -Djava.rmi.server.codebase=http://host/space-examples-dl.jar
  -cp D:\JavaProgs\space-examples.jar;D:\JavaProgs\classes
  sun.applet.AppletViewer worker.html
```

JavaSpaces supports the basic Yale Linda operations (the names differ from the original names used by the Yale group, but essentially the same functionality is provided). Tuples can be created by the programmer from any classes that implement the Jini `Entry` interface. Only the public fields of these classes that refer to objects are considered for matching purposes (i.e. private fields are ignored, as are fields of the primitive data types).

Copyright © 2003 John Wiley & Sons, Ltd.



Tuples are transmitted across the network using a non-standard form of serialisation, whereby only public fields of classes are serialised and multiple references to the same object cause multiple copies to be serialised. Matching of tuples with anti-tuples (called *templates* in JavaSpaces) is done using byte-level comparisons of the data, not the conventional `equals` method. Matching can make use of object-oriented polymorphism for matching sub-types of a class.

The Linda programming model is extended in JavaSpaces to provide support for commercial applications in two ways: transaction-handling and "leases" (similar to the tuple expiration mechanism in TSpaces). There is also an event registration mechanism, similar to the event notification facility in TSpaces, but only for writes. As noted previously, this is not difficult to emulate using threads and the `rd` operation.

A centralised tuple storage approach is used and this may become a performance bottleneck in large systems. Like TSpaces, JavaSpaces does not provide a preprocessor.

GigaSpaces

GigaSpaces[14] is a relatively recent system developed as a commercial implementation of the JavaSpaces specification. As such it is compliant with the Sun specifications, while adding a number of new features. These include operations on multiple tuples, updating, deleting and counting tuples, and iterating over a set of tuples matching an anti-tuple. There are also distributed implementations of the Java Collections `List`, `Set` and `Map` interfaces, and a message queuing mechanism.

Considerable attention has been paid to the efficient implementation of GigaSpaces. This includes the provision of facilities such as buffered writes, and indexing of tuples (with or without intervention from the application developer).

There is also support for non-Java clients to access GigaSpaces through the use of the SOAP protocol over HTTP. Lastly there is support for web servers to make use of GigaSpaces to share session information (potentially even between separate web servers).

Comparison of TSpaces and JavaSpaces with the Yale Linda model

Table I summarises the differences between the original Yale Linda model, TSpaces and JavaSpaces (GigaSpaces is very similar to JavaSpaces). In general it can be seen that TSpaces and JavaSpaces provide considerably more functionality than the original Yale Linda model, with the exception of the `eval` operation (it should be noted that the original Linda model is just that, whereas the other two systems are object-oriented implementations of that model, with extensions). Notable extensions in JavaSpaces and TSpaces are transaction support and leases/expiration, which are important considerations for commercial applications.

TSpaces is unique in providing an extensible form of matching through the ability to add new commands. We will return to this subject in a later section.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

Table I. Comparison of the features of TSpaces, JavaSpaces and Yale Linda

Feature	**TSpaces**	**JavaSpaces & GigaSpaces**	**Yale Linda**
"Rich typing" (tuples as classes)	Yes	Yes	No
Objects (with methods)	Yes	Yes	No
Matching subtypes	Yes	Yes	No
Fields *must* be objects	Yes	Yes	No
More than one tuple space	Yes	Yes	No
Leases/expiration	Yes	Yes	No
Transaction support	Yes	Yes	No
Event notification			
Writes	Yes	Yes	No
Deletions	Yes	No	No
Extensible matching	Yes	No	No
Has `eval`	No	No	Yes

Research projects

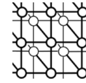
This section provides a brief overview of a number of academic research projects that have developed implementations of Linda in Java with novel or extensible matching mechanisms. There are, of course, many more implementations of the Linda model in Java and in other programming languages—several of these are discussed in [15].

XMLSpaces

XMLSpaces is designed to support the use of XML data in tuples[16]. It is based on TSpaces, and considerably extends the XML support already provided by TSpaces. The `Field` class used by TSpaces for the fields in tuples is subclassed to create a class called `XMLDocField`. This new class overrides the matching method used by TSpaces to provide matching on the basis of the XML content of the field. The matching is performed by a method of the anti-tuple that can be provided by the application programmer. This results in a great deal of flexibility for XML matching operations. A number of matching operations are currently supported, including the use of XML query languages, such as XQL[17] and XPath[18].

XMLSpaces further extends TSpaces by supporting a distributed tuple space model, rather than the centralised model used by TSpaces. The distributed tuple space support is provided in a flexible and tailorable way, allowing different methods for the distribution of tuples to be used, and selected dynamically when an application starts. A subset of the basic TSpaces operations is augmented with distributed versions that take into acccount the distribution of the tuple space (e.g. `distributedWrite` and `distributedWaitToTake`).

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls



CO³PS

CO³PS stands for "Computation, Coordination and Composition with Petri net Specifications"[19, 20]. This builds on the usual coordination model of a concurrent system, where the responsibilities for computation are handled by the host language, and for coordination by the coordination language, as exemplified by Linda. Petri nets are used in CO³PS for the specification of the computational aspect of a system.

The coordination model used in CO³PS is based closely on that of Objective Linda[21, 22]. As such it shares the approach of Objective Linda in allowing the method for the matching of tuples to be overridden. The main application of this in CO³PS is to support the introduction of *non-functional requirements*. The developers of CO³PS distinguish two phases of application design:

1. The *logical phase*, which concentrates on the programming logic—the functional requirements of the system.
2. The *non-functional phase*, in which issues such as efficiency, load-balancing, security, etc. are taken into account.

This approach might be summarised as: "first get it working, then get it working well". In order to support this technique, they make use of a *reflective architecture*, i.e. an architecture that permits the designer to reflect on the behaviour of the system, and to adapt it, without affecting the interaction with clients. The developers of CO³PS go to great lengths to explain that this should be done without impacting on the semantics of the coordination operations.

A further unique feature of CO³PS is the introduction of *composition* as a third aspect of the behaviour of a concurrent system, orthogonal to computation and coordination. Composition refers to the view of an application as a configuration composed of a collection of agents. These agents may recompose themselves into new configurations as needed during the execution of the application.

Java-Linda

Java-Linda is a student project at Yale University[23]. It provides a subset of the features of the original Linda system (for example, there is only partial support for the `eval` operation, no preprocessor, etc.). TCP/IP is used for communication in Java-Linda, with a simple, centralised server.

As there are no extensions to the original Linda model present in Java-Linda, it is of little interest, except for the novel way in which it implements the standard associative matching mechanism. Any object can be used as a tuple in Java-Linda. This would seem to pose some difficulties for the Java-Linda system that needs to perform matching operations on these objects. The solution to this problem that has been adopted is that the Java-Linda system interrogates the structure of the objects in order to extract the fields and perform matching. This has been done using the serialised version of the objects, which includes a considerable amount of information about the structure of the serialised object. This metadata is parsed to extract the information about the structure of the object and this is then used to guide the

Copyright © 2003 John Wiley & Sons, Ltd.

matching process. The matching itself is done on the bytes in the serialised form of the object. This process is described as "tedious and time-consuming"[23], but is convenient and elegant for application programmers. This should be contrasted with JavaSpaces, which appears to do very simple byte-level matching on objects.

Mobile Coordination

The work on mobile coordination performed at the University of Cambridge is very similar in many respects to I-Tuples, a C implementation of Linda[11]. I-Tuples is intended to provide for efficient updates of tuples (e.g. incrementing a counter stored as a tuple). However, the motivation behind the unique features of the mobile coordination project are very different[24]. Like I-Tuples, the idea of mobile coordination is to move the processing of tuple space operations from the application processing node to the server. However, in the mobile coordination project this was done with a view to enhancing fault tolerance, rather than improving performance. In fact, the mobile coordination mechanism is presented as an alternative to the transaction mechanisms found in the current commercial implementations of Linda. Despite this, the performance results reported for the Java implementation of mobile coordination show that in many cases it can lead to improved performance.

ELINDA

The eLinda system is modelled closely on the standard Linda model, supporting the five basic Linda tuple space operations. The current implementation of eLinda has been developed in Java to allow experimentation with the concepts that it embodies.

The eLinda system has three extensions to the original Linda programming model:

- A "broadcast" output operation.
- Support for distributed multimedia applications.
- The Programmable Matching Engine, providing a very flexible matching mechanism.

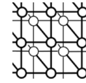
In this paper we will discuss only the last of these extensions. Full details of the other features can be found in [15].

The implementation of eLinda

Three different implementations of eLinda have been developed in order to explore different approaches for handling the tuple space and the attendant communication requirements. The names given to these implementations and their key features are as follows:

eLinda1 Fully distributed tuple space, where any tuple may reside on any processing node. This poses particular problems for matching in that many processors may be required to participate in a matching operation. This was developed as the initial version and much of the focus of the work was on this approach.

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls



eLinda2 Centralised tuple space, where all the tuples are stored on a single "server" node. This is the approach used by JavaSpaces, TSpaces and GigaSpaces.

eLinda3 Centralised tuple space, as in eLinda2, but with "broadcast" tuples cached on each processing node.

Unless otherwise noted, references in this paper to "eLinda" refer either to features common to all the implementations, or to eLinda1.

The Programmable Matching Engine

The Programmable Matching Engine (or *PME*) allows the use of more flexible criteria for locating tuples using the associative matching mechanism. This is useful in situations where a global view of the tuples in tuple space is required. For example, consider a scenario where a tuple is required that has a numeric field, the value of which is the *closest* to some specified value (i.e. not necessarily *equal* to the given value).

Such queries can be expressed using the standard Linda associative matching methods, but will generally be quite inefficient. In the example above, the application would have to retrieve all of the eligible tuples using `in`, work through them to locate the one with the required "closest value" and then return the tuples to tuple space. If the tuple space is located on a remote processing node or distributed over many processing nodes, this will result in a large volume of network traffic. Furthermore, the parallelism of the system is reduced as the application holds all of the tuples for the period required to determine which one is required, possibly holding back other processes.

In situations such as these, where the tuple space is distributed (as in eLinda1), searching for a tuple may involve accessing the sections held on all the processors in parallel. This problem is handled efficiently in eLinda1 by distributing the matching engine so that network traffic is minimised, and moving the necessary computation out of the application and into a special form of matcher. For example, in searching for the "closest" tuple, each section of the tuple space would be searched locally for the tuple with the closest value and that returned to the matcher running in the originating process, which would then select the closest of all the replies received. This process is completely transparent to the application, which simply inputs a tuple, using a specialised matcher. From the application programmer's perspective this could be expressed simply as:

```
in.closest("point", ?x, ?=y)
```

The notation that is used is to follow the Linda input operation with the name of the matcher to be used (`closest` here). The field (or fields) to be used by the matcher is denoted by `?=`. In this case, the operation specified is to retrieve the tuple with the value closest to the current value of the variable `y`. In practice, the matcher is specified as an object that implements a specific Java interface. This is passed to the eLinda system, together with the anti-tuple specification.

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

In addition to the simple usage illustrated by the above example, matchers may also perform *aggregated operations* where a tuple is returned that summarises or aggregates information from a number of tuples. For example, a matcher might calculate the total of numeric fields in some subset of the tuples in tuple space. It is also possible to write matchers that return multiple tuples, providing a facility similar to that provided by the TSpaces `scan` operation.

The facilities provided by the Programmable Matching Engine are compared with a number of other extended Linda systems in [25]. These include those mentioned previously in this paper, and others, such as the work of the coordination research group at the University of York[26, 27, 28], Objective Linda[22, 21], and ELLIS[29].

Limitations of the Programmable Matching Engine

There are some limitations to the kinds of matching operations that are supported by the PME. Notably, some matching operations may still require a complete global view of the tuple space (one simple example of this is where a tuple is required that has the *median* value of some field). In such situations the facilities offered by the PME may not be ideal, as all the tuples need to be gathered together in order to find the result. However, it is important to note that such problems are handled no less efficiently than if the application were to handle them directly, using a conventional Linda system.

Furthermore, the PME approach minimises the network traffic in such cases. This is due to the fact that the distributed matchers would return their local tuples in a single network transfer, rather than one-by-one as would be the case in a conventional Linda system. The potential to reduce the network traffic involved may be even greater as the distributed matchers could return an array containing only the values of the field needed to determine the median value. Once the median was determined then that tuple could be fetched from the processor that held it. If the total size of the tuples in relation to the size of the field required for the determination of the median is large, then this will greatly reduce the volume of network traffic.

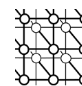
Lastly, the use of the PME will generally simplify application development. This is particularly true where a pre-written matcher is available. It is envisaged that any commercial implementation of this concept would be provided with a library of common matching operations, written in such a way as to provide a useful set of generic matching facilities. More specialised matchers would have to be written as part of the development of the application for which they were required. Such matchers could then be added to the library of existing matchers for future use. It is also possible that writing specialised matchers could become a service provided by an entity separate from the application development team.

Implementing new matchers

Writing matchers, while not a trivial operation, is not overly complex. The programming interface required of a matcher is simply two methods to be provided by a Java object. To support the writing of matchers there is a programming interface to the tuple space system that allows direct access to the tuples in a carefully controlled manner. These facilities allow new matchers to apply the standard associative matching algorithm in various ways, and to

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



interact directly with the eLinda system (e.g. retrieving tuples from tuple space, replacing unwanted tuples, deleting local and remote tuples, broadcasting requests to other processors and subsequently retrieving the results of such requests, etc.). Of course, this interaction allows the programmer to access the tuples in tuple space at a lower level of abstraction than usual, and care needs to be taken to preserve the semantics of the Linda tuple retrieval operations.

While lines of code are a notoriously poor indication of complexity, they can give an approximate indication of the difficulty of writing a customised matcher. For example, a matcher to find the total of numeric tuple fields, is written in 175 lines of extensively commented Java code. Some more reliable complexity metrics are presented below (in Table II).

Applications of the Programmable Matching Engine

The examples of new matchers given above have been in the domain of numeric applications. These are convenient for the purposes of the discussion as they are simple, easily explained and easily understood. However, it would be incorrect to believe that the PME was only useful for numeric problems—it is just as applicable to textual or other problems. Some examples that emphasise the generic nature of the PME are:

- A string matcher could match string fields using some alphabetic measure of "closeness", or even approximate homophonic matching.
- A spatial matcher could compare two fields, taken to be x and y coordinates to locate a tuple corresponding to a point in some two-dimensional space (or, equivalently, in three or more dimensions).
- A matcher could be written to locate tuples with fields corresponding to a date or time in some range of temporal values.
- A matcher could make use of "fuzzy logic" to locate a tuple with some associated degree of certainty of its suitability.
- A matcher could be written to select a tuple at random from some subset of the available tuples[‖].
- A matcher could be written to extract XML data from a tuple and perform complex matching operations based on this (providing an equivalent to the XML support in TSpaces and XMLSpaces).

APPLICATIONS OF ELINDA

The eLinda system has been used for a number of applications, including a distributed video-on-demand demonstration and ray-tracing. As a particular example, which highlights the flexibility and power of the Programmable Matching Engine, the eLinda system has been used to parallelise a graphical parsing algorithm. Visual languages are used in many fields to

[‖]While it is not required of a Linda system, the eLinda system stores tuples using a localised FIFO queuing technique for fairness.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

depict situations or activities in a pictorial or diagrammatic form, which is often easier for human beings to comprehend than an equivalent textual form. Examples of such languages abound, not least in the field of Computer Science, where notations such as flowcharts, state transition diagrams, entity-relationship diagrams, etc. are widely used.

For such graphical models to be used by computer systems there is a requirement for parsing them in order to analyse their structure. This process is directly analogous to the parsing of textual computer programming languages. However visual parsing is distinguished by the increased complexity of the relationships between the components. In a textual language there is a simple, positional sequence relating the components of the language (i.e. the keywords and other tokens). In the case of a visual language there is far more scope for different relationships to exist between tokens in two dimensions (or, more generally, in three or even more dimensions). For example, tokens may be related by inclusion, by contact, by position (e.g. one above another), and so on.

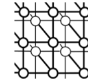
There are many different methods that may be used for specifying and for parsing visual languages. A classification of visual languages that highlights some of these differences can be found in [30]. The specific method that was parallelised using eLinda is *picture layout grammars*, as developed by Golin[31]. Picture layout grammars provide a particularly flexible and powerful means of expressing the syntax of visual languages. However their parsing is a complex, multi-stage process with an algorithmic complexity of $O(n^9)$ in the worst case. Much of the parsing process can be parallelised as the graphical tokens can be reduced largely independently of each other. Similarly, the final stage of parsing using Golin's technique consists of checking the parse tree structures to remove redundant and invalid paths. This can also easily be done in parallel. The parallelisation makes use of the common replicated-worker pattern[12]. More details of the parallel parsing algorithm are given in [32].

It was found that the enhanced matching mechanisms in eLinda were very useful during the development of this application. At a number of points there is a need to use complex criteria to specify the tuples that are to be retrieved from tuple space. Four customised matchers were written for the visual language parser. Of these matchers, two are general-purpose, and may be useful in other applications. The number of lines of code, while not an accurate metric, does give an approximate indication of the complexity of a matcher. Accordingly, the total number of lines of commented Java code are given below for each of the matchers written for the visual parsing algorithm, together with a brief description of the purpose of the matcher. Better complexity measures are presented further below.

RHSMatcher (130 lines of code) This matcher is the most complex of those used in the visual language parsing application. It is used to search the tuple space containing the grammar rules, looking for a rule that could be applied to reduce a given symbol. This requires searching through the rules looking for the given symbol in the right hand side of all the rules. If the rule is a simple one then the matcher additionally checks the constraints specified by the rule.

ConstraintMatcher (114 lines of code) This matcher is used when applying more complex rules to locate suitable symbols (Y) to reduce with the current symbol (X). In order to do this it has to check the constraints of the attributes of the available Y symbols (usually

Copyright © 2003 John Wiley & Sons, Ltd.



in conjunction with the attributes of the symbol X). It returns multiple matching tuples, using a Java `Vector`.

SetMatcher (111 lines of code) This matcher is used by the worker processes to retrieve a symbol from the tuple space for reduction. Due to a small modification required for the parallelisation of the parsing algorithm, this symbol needs to be chosen from a set of symbols to be considered at the current parsing level.

This matcher could be used by any application that had a similar requirement to match tuples where a field has one of a set of defined values. The sets are handled using the `java.util.Set` interface within the matcher, allowing considerable flexibility.

AllMatcher (67 lines of code) This matcher can be used to retrieve all the tuples matching a given anti-tuple (in the same way as the `scan` and `consumingScan` operations provided by TSpaces). It is employed during the checking phase of the visual parsing application to retrieve all entries in the tuple space for a given symbol (i.e. all of the possible subtrees).

Again, this matcher could be used by any application that needed to retrieve the set of all tuples meeting some criterion. It returns the multiple tuples in a Java `Vector`.

In order to try to convey a better idea of the complexity of these matchers, the Halstead metrics[33] for the main methods used in the matchers above are presented in Table II. This shows LoC, the actual number of lines of code (excluding whitespace and comments), D, the Halstead program difficulty (a measure of how difficult a method's code is to understand), and E, the Halstead program effort (a measure of difficult a method is to write). For comparative purposes, these metrics are also given for the total matcher, discussed earlier, and for the common Bubble Sort algorithm.

PERFORMANCE MEASUREMENTS

Performance testing results were obtained for a simple ray-tracing program. These were used to compare the performance of eLinda with the other commercially developed Linda systems, and also to examine the general behaviour of eLinda with regard to varying the communication:computation ratio.

All the results presented in this paper were obtained using an undergraduate teaching laboratory in the Department of Computer at Rhodes University. The computers in this facility were at that time equipped with 133MHz Pentium I processors and 32MB of main memory, and networked using standard 10Mb/s shared Ethernet. The operating system used was Windows NT 4.0. The version of Java used for the testing was 1.3.0.

At the outset, it should be noted that this hardware specification is barely adequate for the execution of Java applications. Sun's minimum recommended memory allowance for Java is 32MB, and the minimum processor recommendation is a 166MHz Pentium processor[34]. For some of the testing the limitations of the hardware became apparent, particularly in the form of excessive virtual memory paging and, in some extreme cases, complete system failures due to insufficient memory. These occurrences will be noted where relevant in the discussion below.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls

Table II. Complexity Metrics for Matchers

Method	LoC	D	E
AllMatcher			
match	3	2.5	60
matchList	42	20.429	25 862.58
ConstraintMatcher			
match	16	16.056	7 411.816
matchList	73	35.164	95 598.977
RHSMatcher			
match	20	15.417	9 005.79
matchList	73	38.644	105 855.219
SetMatcher			
match	12	7.438	1 934.235
matchList	61	34.036	70 393.508
TotalMatcher			
match	3	1	4.755
matchList	97	39.741	127 187.461
Bubble Sort			
bubbleSort	13	15.619	7 241.638

Ray-tracing performance

One of the demonstration programs provided with JavaSpaces is a simple ray-tracing application, written using a replicated-worker pattern[12]. This application was extended to produce timing results, and then ported to eLinda, TSpaces and GigaSpaces. The ray-tracing program did not make use of any of the new or extended features of any of these Linda systems. Only eLinda1 and eLinda2 were used as this application did not make use of the broadcast communication used in eLinda3. It should also be noted that this application does not utilise the extended matching facilities of the Programmable Matching Engine.

While the results found for this application give a good indication of the relative efficiency of the Linda implementations it should be noted that they are not very impressive in terms of ray-tracing programs in general.

The timing results for this application are shown in Table III. Figure 1 shows these results in the form of the speedup of the ray-tracing program as the number of worker processes is increased, relative to the time taken by a single worker. A maximum of only eight worker processes could be used with JavaSpaces—over this limit the system became unstable. The performance of GigaSpaces was erratic as the server experienced serious virtual memory problems in this case.

Copyright © 2003 John Wiley & Sons, Ltd.

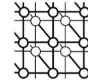


Table III. Detailed Results for the Ray-Tracing Application

Number of Workers	Linda System eLinda1	eLinda2	TSpaces	JavaSpaces	GigaSpaces
1	60.66	57.85	58.48	63.48	62.46
2	34.86	31.89	31.78	33.65	35.82
3	32.60	23.63	24.37	25.32	25.09
4	25.70	20.59	19.36	22.69	21.38
5	22.62	18.20	17.34	23.71	23.95
6	21.09	17.66	16.89	21.80	23.56
7	20.87	16.78	16.96	21.88	17.32
8	23.99	16.60	17.70	24.33	17.74
9	29.01	17.04	17.58	—	19.88

Note: Times are all in seconds.

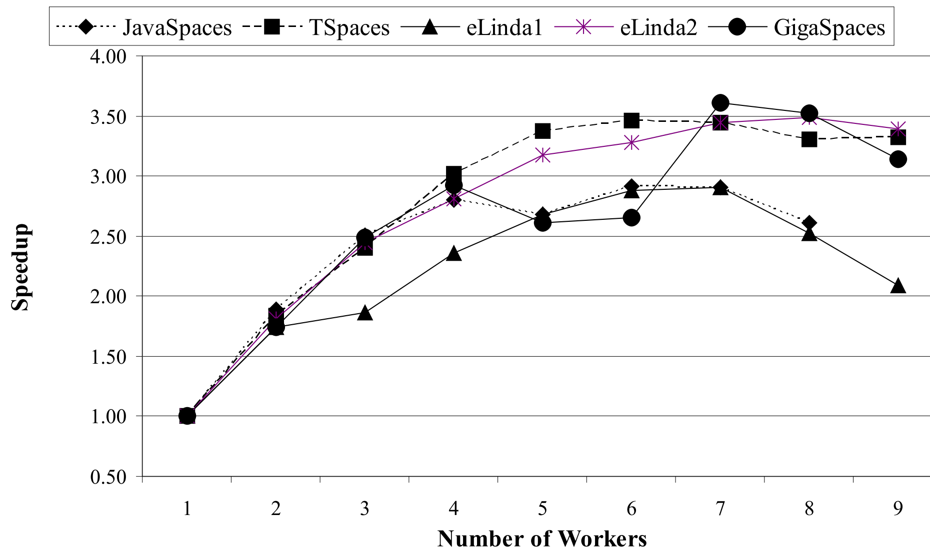


Figure 1. Speedup for the Ray-Tracing Application

Copyright © 2003 John Wiley & Sons, Ltd.
Prepared using cpeauth.cls

Table IV. Ray-Tracing Results for Differing Image Segment Sizes for eLinda1

Workers	Segment Size		
	50 × 50	**100 × 100**	**200 × 200**
1	125.82	70.87	60.66
2	66.34	44.55	34.86
3	53.30	59.72	32.60
4	30.65	23.61	25.70
5	28.32	25.62	22.62
6	23.09	15.65	21.09
7	21.15	18.94	20.87
8	22.31	17.31	23.99
9	19.52	24.45	29.01

Note: Times are all in seconds.

These results clearly demonstrate the similarity between the implementations: all exhibit very similar behaviour, with the speedup levelling off or, in some cases, decreasing from about seven worker processes. The maximum speedup obtained was approximately 3 to 3.5 times, using six to eight processors, as shown in Figure 1. TSpaces showed the best speedup overall. The results for eLinda are very close to those for JavaSpaces, and fairly close to those for TSpaces. From Table III we note that eLinda2 with eight worker processes produced the best result overall.

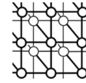
The ray-tracing application allows the size of the segments of the image that are distributed to the workers to be changed. The results above were all measured for an image segment size of 200 × 200 pixels (the full image is 640 × 480 pixels).

To assess the impact of different image segment sizes, the measurements were repeated for segments of 50 × 50 pixels, 100 × 100 pixels and 200 × 200 pixels. The best results for all three systems were generally found for larger segment sizes. The results for eLinda1 showing the impact of the segment size are summarised in Table IV. This indicates that, in this case, the optimum performance for the ray-tracing application is to be found using eight worker processes and an intermediate image size (100 × 100).

SUMMARY AND CONCLUSIONS

This paper presented a survey of selected Linda implementations in Java, concentrating on the commercial offerings and on systems that provide some form of alternative mechanisms for matching. The eLinda system was described, focussing on the Programmable Matching Engine, intended to help solve the problems inherent in simple associative matching. In addition the results of some performance measurements were reported for eLinda and the commercial Linda implementations.

Copyright © 2003 John Wiley & Sons, Ltd.



The implementation of the visual language parsing application using the Programmable Matching engine facilities was relatively simple. This points to the desirability of more flexible matching mechanisms in Linda. Additionally, the performance results shown above (see Table III) indicate that the eLinda system is generally as efficient as the commercial Linda implementations (eLinda2 produces the best result overall). This is very satisfying for a small-scale academic project, embodying a number of ambitious extensions. However, it must also be noted that the results indicate that none of these Java-based Linda systems is particularly suitable for fine-grained parallel processing applications.

Future work on the eLinda system will be focussed on the further development of the multimedia extensions, and on the application of the PME facilities for distributed Internet applications.

ACKNOWLEDGEMENTS

This work was supported by the Distributed Multimedia Centre of Excellence in the Department of Computer Science at Rhodes University, with funding from Telkom SA, Comparex Africa, Letlapa Mobile Solutions and THRIP.

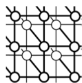
The constructive criticism provided by the anonymous referees, which helped to substantially improve the original draft of this paper, is also gratefully acknowledged.

REFERENCES

1. D. Gelernter. Generative communication in Linda. *ACM Trans. Programming Languages and Systems*, 7(1):80–112, January 1985.
2. S. Hupfer, D. Kaminsky, N. Carriero, and D. Gelernter. Coordination applications of Linda. In Banâtre and Métayer [35], pages 187–194.
3. S. Ahuja, N. Carriero, and D. Gelernter. Linda and friends. *IEEE Computer*, 19(8):26–34, August 1986.
4. N. Carriero and D. Gelernter. The S/Net's Linda kernel. *Operating Systems Review*, 19(5):54–71, March 1985.
5. S.E. Zenith. A rationale for programming with Ease. In Banâtre and Métayer [35], pages 147–156.
6. N. Carriero and D. Gelernter. *How to Write Parallel Programs: A First Course*. The MIT Press, 1990.
7. Sun Microsystems. Jini connection technology. URL: `http://www.sun.com/jini`.
8. IBM. The TSpaces vision. URL: `http://www.almaden.ibm.com/cs/TSpaces/html/Vision.html`.
9. P. Wyckoff, S. W. McLaughry, T. J. Lehman, and D. A. Ford. T Spaces. *IBM Systems Journal*, 37(3):454–474, 1998.
10. World Wide Web Consortium. Extensible markup language (XML). URL: `http://www.w3.org/XML`.
11. M. Foster, N. Matloff, R. Pandey, D. Standring, and R. Sweeney. I-Tuples: A programmer-controllable performance enhancement for the Linda environment. In H.R. Arabnia, editor, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 357–361. CSREA Press, June 2001.
12. E. Freeman, S. Hupfer, and K. Arnold. *JavaSpaces Principles, Patterns, and Practice*. Addison-Wesley, 1999.
13. C. Austin and M. Pawlan. *Advanced Programming for the Java 2 Platform*. Addison-Wesley, September 2000.
14. GigaSpaces Technologies Ltd. Gigaspaces. URL: `http://www.gigaspaces.com/index.htm`, 2001.
15. G.C. Wells. *A Programmable Matching Engine for Application Development in Linda*. PhD thesis, University of Bristol, U.K., 2001.
16. R. Tolksdorf and D. Glaubitz. Coordinating web-based systems with documents in XMLSpaces. URL: `http://flp.cs.tu-berlin.de/~tolk/xmlspaces/webxmlspaces.pdf`, 2001.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls



17. J. Robie, J. Lapp, and D. Schach. XML Query Language (XQL). URL: http://www.w3.org/TandS/QL/-QL98/pp/xql.html, September 1998.
18. World Wide Web Consortium. XML Path language (XPath) version 1.0. W3C Recommendation, URL: http://www.w3.org/TR/xpath.html, November 1999.
19. T. Holvoet. *An Approach for Open Concurrent Software Development.* PhD thesis, Department of Computer Science, K.U.Leuven, December 1997.
20. T. Holvoet and Y. Berbers. Reflective programmable coordination media. In H.R. Arabnia, editor, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 1236–1242. CSREA Press, June 2001.
21. T. Kielmann. Object-Oriented Distributed Programming with Objective Linda. In *First International Workshop on High Speed Networks and Open Distributed Platforms*, St. Petersburg, Russia, June 1995.
22. T. Kielmann. *Objective Linda: A Coordination Model for Object-Oriented Parallel Programming.* PhD thesis, University of Siegen, Germany, 1997.
23. A. Smith. Towards wide-area network Piranha: Implementing Java-Linda. URL: http://-www.cs.yale.edu/homes/asmith/cs690/cs690.html.
24. A. Rowstron. Mobile co-ordination: Providing fault tolerance in tuple space based co-ordination languages. URL: http://www.research.microsoft.com/~antr/papers/mobile.ps.gz, 1999.
25. G.C. Wells, A.G. Chalmers, and P.G. Clayton. Extending the matching facilities of Linda. In F. Arbab and C. Talcott, editors, *Proc. 5th International Conference on Coordination Models and Languages (COORDINATION 2002)*, volume 2315 of *Lecture Notes in Computer Science*, pages 380–388. Springer, April 2002.
26. P. Butcher, A. Wood, and M. Atkins. Global synchronisation in Linda. *Concurrency: Practice and Experience*, 6(6):505–516, September 1994.
27. A. Rowstron and A. Wood. Solving the Linda multiple rd problem. In P. Ciancarini and C. Hankin, editors, *Coordination Languages and Models, Proc. Coordination '96*, volume 1061 of *Lecture Notes in Computer Science*, pages 357–367. Springer-Verlag, 1996.
28. D.K.G. Campbell. Constraint matching retrieval in Linda: extending retrieval functionality and distributing query processing. Technical Report YCS 285, University of York, 1997.
29. P. Broadbery and K. Playford. Using object-oriented mechanisms to describe Linda. In G. Wilson, editor, *Linda-Like Systems and Their Implementation*, Technical Report 91-13, pages 14–26. Edinburgh Parallel Computing Centre, June 1991.
30. K. Marriott and B. Meyer. The classification of visual languages by grammar hierarchies. *Journal of Visual Languages and Computing*, 8(4):375–402, August 1997.
31. E.J. Golin. *A Method for the Specification and Parsing of Visual Languages.* PhD thesis, Brown University, 1991.
32. G.C. Wells, A.G. Chalmers, and P.G. Clayton. Extending Linda to simplify application development. In H.R. Arabnia, editor, *Proc. International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA'2001)*, pages 108–114. CSREA Press, June 2001.
33. M.H. Halstead. *Elements of Software Science.* Elsevier, 1977.
34. Sun Microsystems. System requirements. Java 2 SDK, Standard Edition Version 1.3.0, README File, 2000.
35. 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, 1992.

Copyright © 2003 John Wiley & Sons, Ltd.

Prepared using cpeauth.cls