Ersch. in: Proceedings of the Eleventh IASTED International Conference Parallel and Distributed Computing and Systems : November 3 - 6, 1999, MIT, Cambridge, Massachusetts, USA / a publ. of The International Association of Science and Technology for Development - IASTED. Ed.: Si-Qing Zheng. - Anaheim ; Zurich [u.a.] : Acta Press, 1999. - S. 683-692. - ISBN 0-88986-275-3

JaCaL: AN IMPLEMENTATION OF LINDA IN JAVA

PIOTR TYSOWSKI
University of Waterloo
Dept. of Electrical and Computer Engineering
Waterloo, ON N2L 3G1
Canada
pktysows@uwaterloo.ca

MOHAMMAD ZULKERNINE
University of Waterloo
Dept. of Electrical and Computer Engineering
Waterloo, ON N2L 3G1
Canada
mzulker@uwaterloo.ca

STEFAN LEUE
University of Waterloo
Dept. of Electrical and Computer Engineering
Waterloo, ON N2L 3G1
Canada
sleue@uwaterloo.ca

ABSTRACT

Java is an object-oriented programming language with built-in features for creating distributed programs. A key feature-set that is missing, however, is an easy-to-use, reliable, and scaleable tool for writing truly parallel programs. The Linda parallel programming model defines a client-server approach where concurrent execution requests are serviced and results are stored to a shared data repository called a Tuple Space. Tuples consist of heterogeneous collections of data of various types. An interpretation of the model allows the programmer to create active tuples, in which user-defined functions are automatically and transparently launched in concurrently executing processes distributed on remote workstations. This paper presents an implementation of Linda in Java called JaCaL, a library of classes and interfaces easily integrated into a client application. The Tuple Space has been implemented as a data store residing on a single machine, whilst being accessible by clients distributed on other machines. Clients can create active tuples, which cause processes on distributed machines to be transparently invoked to execute user-defined functions, employing a load-balanced worker process model. No pre-processor is required to parse the client program before compilation. The implementation allows a Java application developer to create more efficient coarse-grained parallel programs with minimal effort. Performance measurements have been made and compared to those of another, similar implementation.

KEY WORDS

Linda, Java, concurrency, parallel programming, distributed computing.

1. INTRODUCTION

Java is a popular object-oriented programming language that is well suited to the distributed computing domain. It offers dynamic code loading and linking, abstraction through interfaces, and remote method invocation (RMI) for seamless distributed communication. However, one noticeably missing and useful feature is an easy-to-use class library for communicating through a shared memory and transparently processing user tasks in parallel.

Linda is a generic model that can be implemented in the most common programming languages to support parallel programming features[1,2]. It is designed to allow co-ordination and data sharing between multiple processes or threads. It is flexible in that it supports three major parallel programming approaches[2]:

Result parallelism	Workers, executing in parallel, each produce a different piece of the result.
Specialist parallelism	Each worker is assigned to perform one specific kind of work only.
Agenda parallelism	Each worker is assigned to help out with the current item on an agenda.

In the past, Linda has been implemented as a dialect in structured programming languages such as C, C++, and Fortran[1]. It is widely used in parallel programming experiments, some of which have become commercially available. In this paper, we describe the design and implementation of Linda in Java, called *JaCaL* (an acronym for Java and Concurrency and Linda), a research idea that has only very recently begun to be studied[3,4,5,6]. Most of the current implementations are overly complex in machine-to-machine communication and are awkward in their interface. From the beginning, our implementation has been designed to reduce complexity on the user end by exploiting well Java's inherent object-oriented and inter-process communication capabilities.

JaCaL is nearly semantically equivalent to the original Linda specification, with a minor noted exception required to support parallelism. The necessary criteria for a faithful implementation include client attachment to a shared memory, a tuple search mechanism, guaranteed mutual exclusion of data, inter-process communication and co-ordination, and execution of user-defined functions on distributed worker processes. JaCaL succeeds in satisfying these constraints while hiding as much complexity as possible in a simplified programmer's interface. It is slanted towards high efficiency for coarse-grained applications, due to the expense of accesses to the shared memory*.

Thus far, we have begun with a brief introduction to Linda and Java, and discussed the purpose of and motivation behind our implementation. In section 2, we have defined the standard Linda co-ordination model that constituted the theoretical basis for JaCaL. Then, in section 3 we have briefly analysed the various features of some of the currently available Linda implementations in Java, and made a comparison to our own system in each case. Section 4 elaborates on the specifics of the approach taken in the design and subsequent implementation of Linda in Java. A sample client application was used to test this proof-of-concept system, and it is described in section 5. Performance measurements were made on both our system and a similar, existing implementation, and the results appear in section 6. Conclusions are drawn in section 7 with a brief summary of achievements. Finally, directions for future research appear in section 8.

2. THE LINDA MODEL

Linda is a programming model proposed by Gelernter for incorporating distributed processing and intercommunication in almost any high-level programming language.[2] It provides for concurrent execution through access to a shared data space called a Tuple Space. Tuples, fundamental data structures consisting of fields of various data types, are stored in a collection called a Tuple Space and accessed by matching the contents of data fields rather than a specific memory address. Tuples are identified by their content alone. There exist two kinds of tuples in Linda: data tuples and process (active) tuples. Process tuples execute in parallel and exchange data by generating, reading and consuming data tuples, static data that is always passive. An active process tuple becomes a passive data tuple after it has finished its execution, and it is then recognised as being no different from other data tuples. Process tuples may execute either on the client side or elsewhere in the system.

Parallelism in Linda is achieved by a very small number of operations[7] invoked by the user application, as described below:

Primitive	Description
out	Store a tuple, consisting of several data fields, in the Tuple Space. Control is then returned to the program immediately.
in	Match a tuple, based on a specified subset of the tuple fields, and remove it from the Tuple Space so that it is no longer available for other retrievals. If no matching tuple is found, the command blocks until the appearance of at least one matching tuple in the Tuple Space.
rd	Match a tuple and return a copy of it. The original is not removed. If no match is found, this command also blocks until the appearance of at least one matching tuple in the Tuple Space.
eval	Create a tuple by first instantiating a new process. A function is provided by the user and then executed in this process. The result of the function is resolved into a data tuple that is stored in the Tuple Space. Control is then returned to the invoking program.

In addition, there are two operations, *inp* and *rdp* that function much like *in* and *out*, except that they return immediately to the calling process by returning a null value if no tuple can be matched, instead of blocking. The functionality of the Tuple Space access operations has been implemented in JaCaL in accordance with the above definitions, with the exception of the *eval* operation, which returns control to the user process immediately after being called. This allows the user to queue up a number of tasks that will then be executed in parallel in the JaCaL system. The details of this approach are described later.

The Tuple Space is the heart of the Linda system. It must have the following four important characteristics in order to ensure efficient communication and co-ordination[8,5], all of which are satisfied by the JaCaL implementation:

* The shared memory definition used throughout this paper refers to a non-partitioned memory space residing on a single computer and accessible only through the interface routines provided by the JaCaL class library by any number of other machines.

Characteristic	Description
Dynamic tasking	The master process assigns tasks for processing tuples, and worker processes then retrieve tuple data and perform work. After that, the workers write their results to the Tuple Space. Other workers can then perform tasks based on the content of these tuples. This results in parallelism of data manipulation.
Distributed data structures	The Tuple Space is capable of storing arbitrary data structures and sharing them among several processes. All modifications to tuples are done outside of its domain. The ability to share data of arbitrary complexity among multiple processes and even machines enables practical parallelism.
Time de-coupled communication	The Tuple Space stores user data in persistent objects, remaining on the Tuple Space server until it is retrieved and deleted. The creation time of tuples is of no concern, unlike messages in general that are transient events. There exists no inherent bond between data and any particular process, and thus the data stored in the Tuple Space is made available for other processes even after the creating process's termination.
Anonymous communication	All communication is performed through the Tuple Space, and processes cannot identify the creators of the tuples that they are making use of. This eliminates the overhead of process identification. Processes need only to understand the structure of the data in order to retrieve it, and do not require the overhead of keeping track of its ownership or history. Of course, no security is present within this model.

3. RELATED WORKS

Generally speaking, the implementation of Linda in Java is still very much in an initial, experimental stage. There are only a handful of other known implementations of Linda-like co-ordination languages in Java. The principal aim of all these implementations is to exploit the built-in distributed computing features of the Java language to create an easy-to-use, reliable, and high-performance parallel programming model, although no known implementation to date has managed to effectively satisfy all of these constraints. However, we believe that JaCaL comes closest to doing so. In the following paragraphs, the most important features of recent Linda implementations in Java are described, and a comparison to our own design is made in each case.

Javelin[5]: Tuples are implemented as classes so that the Tuple Space is simply a collection of class instances. To create a class that can be stored in the Tuple Space, the programmer creates a subclass of Tuple, and in it defines all of the fields and methods that will comprise the tuple. The class Tuple provides the six Tuple Space operations. Javelin implements the *eval* primitive by transforming the user-defined classes using a pre-processor that employs Java threads. This increases the complexity of the system through the use of a separate component. In our design, no pre-processor is necessary, as all of the JaCaL primitives are directly compiled. This approach also greatly aids in debugging the user code.

Java-Linda[4]: Here, tuple sets are simply classes defined by the users and actual tuples are simply instantiated objects of those classes. The fields of the tuples are objects rather than primitive data types. A tuple hierarchy tree is used to store tuples based on the inheritance mechanism of Java. Tuples are stored in a linked list and hashing is used to increase the runtime tuple matching efficiency. Linda-Java provides all the basic tuple operations including *eval*. *Eval* has been implemented using threads in Java. However, in addition to threads, our design allows for *eval* tasks to be executed on distributed processes, communicating using the Java RMI mechanism, as explained later. This results in true parallelism across multiple machines, something that cannot be attained on a single machine using only threads.

Jada[6]: The shared data spaces in Jada are called ObjectSpaces as they contain Java objects, instead of tuples as in standard Linda. Only the Java objects that implement an interface called JadaItem can be stored or retrieved from the Jada ObjectSpaces. Jada's ObjectSpaces provide all of the Linda-equivalent primitives except *eval*. The behaviour of *eval* can be simulated by explicitly starting new Java threads, computing a result, and writing its result into the object space. Unlike Jada, our design has full, integrated, and transparent support for *eval*, and is not restricted to threads. Also, Jada utilises sockets for connections to its object server. RMI does not appear to be utilised at all, as in our system. RMI makes the user programmer's job considerably easier, as it abstracts from sockets and presents a simplified interface for process intercommunication.

JavaSpace[9,3]: This is Sun's implementation of Linda-like primitives in Java at a higher level of abstraction than RMI and object serialisation. The Tuple Space holds "entries," which are typed groups of objects expressed by a class in Java that implements the interface space. The following four operations can be invoked on the space:

write(): add the given entry into the space.
read(): read an entry that matches the given template.
take(): remove the entry from the space that matches the given template.
notify(): notify a specified object when entries that match the given template are available in the space.

The *read* in JavaSpace appears to be a non-blocking version of *rd* in standard Linda, and that to lead into a transaction equivalent to a blocking *in* would require packaging the functionality of *notify*. In our design, the

programmer may easily choose between a blocking and a non-blocking version of *rd* by calling the corresponding method name variant. JavaSpace also provides two additional methods to control previously sent notification requests: *renew()* and *cancel()*, which allows a client to request an extension or cancellation, respectively, of a previous notification request. Interestingly enough, it does not offer the equivalent of an *eval* operation as in standard Linda, a feature that has been implemented in our own design. However, it does allow for multiple clients and JavaSpaces within a single distributed application.

In comparison to the various implementations of Linda in Java that are currently available, we conclude that JaCaL is novel in terms of the functionality it provides as a native Java application, and addresses several aspects not covered by other current implementations, as demonstrated above. We believe that only JaCaL offers a truly distributed and concurrent processing model completely transparent to the user whilst providing great ease of use and reliability through functionality consistent with the Linda model.

4. IMPLEMENTATION

ARCHITECTURAL OVERVIEW

A class library has been written in Java that provides all of the basic tuple operations. The shared tuple data space is implemented as a shared memory with synchronised access, residing in a single process on a server workstation. This memory is accessible by distributed client processes through the provided interface. The interface takes the form of a library of primitives. The server in turn controls a pool of distributed worker processes to compute active tuples. All inter-process communication is performed through the RMI mechanism† [10] to take advantage of its programming simplicity and its reliability. The implementation allows for true concurrency in application processing. A unique challenge solved by our design is a true object-oriented architecture benefiting from a full use of Java's features. A basic architectural view is presented in Figure 1.

† RMI (Remote Method Invocation) is a communications tool provided by the JDK (Java Development Kit) version 1.1.5 and higher. It allows processes to create socket connections between themselves and uses TCP/IP as the underlying protocol for data transport. Nearly any user-defined object may be "serialised," or defined as being convertible to a bit stream, and transmitted seamlessly using an open RMI connection. A server application implementing a callable routine registers itself in a public registry, and the calling client machine invokes a remote method on it by addressing it through a so-called "stub." The fact that both member data in user-defined classes and even the methods that they implement may be transmitted was put to good use in our novel implementation of the *eval* primitive, as explained later.

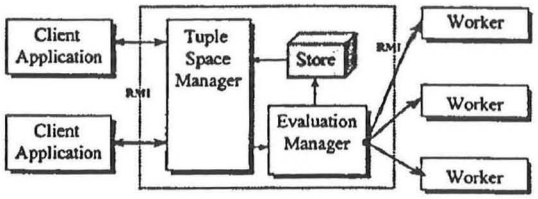


Figure 1. A basic architectural view of the JaCaL system.

TUPLE CREATION

The creation of tuple data requires only a minimum of effort on the part of the application programmer. A tuple is an instance of the *Tuple* class, which is an array of a requested length containing a heterogeneous collection of tuple type classes derived from a common *TupleField* class. A number of common types are supplied by JaCaL, including *TupleInt* (integer) and *TupleStr* (string) types. Each of these derived classes defines its own comparison algorithm so that it is possible to meaningfully compare the object's contents to another instance of the same class type by calling a standard method, a function utilised by the Tuple Space when matching against anti-tuples (described later). The *TupleField* base class defines an object container to store the tuple field contents, and initialises it. In addition, it implements the *TupleFieldIF* interface, which defines a number of methods for comparing, evaluating, and printing the contents of the tuple field as an aid in debugging. The use of generic types and casting, as well as interfaces, results in a great deal of flexibility and as little work as possible for the programmer in defining new and complex data types. The contents of a tuple field could, for instance, contain an arbitrary linked list or tree structure, and would easily be integrated by defining an appropriate comparison algorithm and output function, if so desired. For example, the comparison method for a linked list data type, (invoked by the Tuple Space when matching against anti-tuples), would implement an algorithm for traversing and inspecting the contents of each of the nodes of the linked list.

The user proceeds by instantiating the fields, or elements, of a tuple and constructs a tuple container. The only parameter to all Linda primitives is a *Tuple*, thus taking advantage of Java's strict type checking to detect errors at compile-time, rather than requiring an additional pre-processing tool. The basic structure of tuples and their fields is shown in Figure 2.

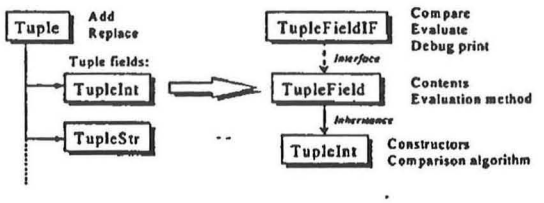


Figure 2. The structure of tuples and fields, and the methods supported.

Each tuple is passed by value rather than by reference from the client application to the Tuple Space using an existing RMI connection.

TUPLE SPACE STORAGE

The standard Java vector container is used for storage and retrieval of tuples in the Tuple Space. It implicitly supports automatic growth and indexed access, and thus a linear search is performed for matching operations when retrieving tuples, namely the *in* and *rd* primitives. For the initial version of our implementation, the vector container was used in favour of a more complicated, if more efficient, hashing or tree routine whose impact on the projected performance of our system is expected in most cases to be negligible. Our approach favours applications with minimal interaction with the Tuple Space storage, due to the higher efficiency that results from coarse-grained processing. In our case, coarse-grained processing involves evaluation programs with longer periods of computation between Tuple Space input/output access, thus minimising the relatively high cost of RMI communication. This is a programming style that cannot be enforced by our system, but the guidelines are made clear. The initial goal was an implementation of a proof-of-concept distributed system rather than an attempt to find the optimal data access algorithm for Tuples, although the current implementation can easily be extended. However, one simple optimisation was added whereupon the scanning of tuples for retrieval begins from the end of the container, as the user is likely to store an active tuple, wait for it to be processed, then read it back immediately, from the end. Thus, spatial locality is utilised for faster access. In addition, an output mechanism has been built to display the contents of all tuples stored in the Tuple Space, and it has proven to be useful during integration testing.

TUPLE SPACE ACCESS

The primitives *in, out,* and *rd* invoke methods of the Tuple Space manager process, encapsulated into a single *TupleSpace* class. Consistency of the tuple data is ensured by the data synchronisation mechanisms available in Java, so that the same tuple field in the storage cannot fulfil a simultaneous read and write request. The contention problem is resolved by Java's ability to queue methods waiting on the Tuple Space storage object through the *synchronized* feature.

As noted previously, a minor semantic deviation from the original Linda model[2] occurs when storing a result in the Tuple Space. As explained in the later subsection discussing the *eval* primitive, the *out* method necessarily spawns a thread to complete this task, while returning control to the user application immediately. Thus, the store operation is non-blocking. However, the same is not true for the *in* input operation, and thus this is not considered to be a great restriction as long as the centre of process co-ordination remains within the Tuple Space through the data that it stores.

ANTI-TUPLES AND QUERIES

To retrieve a tuple from the Tuple Space, the application programmer constructs a so-called anti-tuple, and supplies it through an *in* or *rd* method invocation. This procedure is similar to that described for storing tuples. However, the anti-tuple is meant to represent a key through which associative matching against its content can be performed. Specifically, a subset of the tuple fields is not initialised by the user application and will be filled with data once a match occurs in the Tuple Space against the other fields initialised with content, when a matching tuple in the Space is found. Two tuples are considered to match if they contain the same number of fields and the i^{th} field of the anti-tuple matches (in data type and value) the i^{th} field of the tuple currently being examined in the Tuple Space, where i is each non-initialised field of the anti-tuple. Once a match occurs, the null fields are replaced with copies of the fields from the tuple in the Tuple Space constituting a match. The tuple is then returned to the application. By the Linda definition, it is possible for duplicates to occur in the Tuple Space, and the first tuple to be matched is simply returned.

For example, consider an anti-tuple constructed to retrieve a prime or non-prime condition calculated on the number 119, stored in the Tuple Space, earlier. In the following code sample, the anti-tuple is first allocated, and the first two fields are programmed to contain the key used by the Tuple Space server to locate the desired tuple, namely the "Primes" identifier string and the number 119. To the third and last field, an empty integer object is added, which will contain the result of the data retrieval from the Tuple Space. The anti-tuple is then submitted to the Tuple Space server. If it finds a tuple that matches the first two non-empty fields, it will fill in the contents of the third field of the anti-tuple, and return it to the client application.

```
Tuple antituple = new Tuple(3);
antituple.addField("Primes");
antituple.addField((TupleField) new TupleInt(119));
antituple.addField((TupleField) new TupleInt());
Tuple tuple_found = _ts.rd(antituple);
```

The non-initialised tuple fields, which contain null content, replace the need for the *"?"* qualifier in other implementations of Linda, and thus make a pre-processor obsolete. Our design results in a fully integrated library tool simplifying the task of debugging for the programmer, as direct compilation is possible.

If a match cannot be found on a blocking *in* operation, then the method blocks waiting on the Tuple Space container using the Java *wait* mechanism, similar to a monitor mechanism but without a FIFO ordering

implied[11]. The method resumes its search upon being awakened by a *notify* call on the completion of an external *out* operation. Due to the nature of the synchronisation mechanism in Java, all *in* methods compete for being awakened, regardless of which operation is waiting on what data. In addition, even though one *in* operation may complete by finding the new data, all other *in* calls will still need to be unnecessarily processed for the same notify event. This behaviour is inefficient but unavoidable in Java.

EVALUATION PRIMITIVE

One of the key components of the JaCaL programming library is the implementation of the *eval* method evaluation mechanism. A novel modification was made to the semantics of the standard Linda primitives. The *out* call was combined with *eval* functionality, thus simplifying the programmer's understanding of the interface. In addition, the *TupleField* class was extended to accommodate new functionality. The result is that tuples can easily be made active by attaching evaluation methods to tuple field data that are then invoked by the Tuple Space manager and run concurrently in a worker pool.

Specifically, the user instantiates a class adhering to the *Eval* interface, and encapsulates the evaluation method within the object, accessible to the Tuple Space manager by a known method defined in *Eval*. One example is *EvalPrime*, which calculates all of the prime numbers up to a specified limit. This limit constitutes the initial data for the evaluation object, and is specified in its constructor and stored within the object as member data. The user normally defines the evaluation objects required by the application, as well as the initial data storage in the derived class. The *TupleField* class in turn defines a reference to an evaluation object that can be attached to the field. This reference will be null if no method is attached. Calling one of the tuple field object's methods performs the process of attaching an evaluation object.

Next, when the now-active tuple is sent to the Tuple Space Manager residing on the server, an Evaluation Manager spawns a new thread to handle the *out* operation all by itself. The actual code for these methods is transported to the Tuple Space, as per the defined RMI behaviour. All fields within the tuple are sequentially scanned to detect the presence of an evaluation method attachment. If one is found, then the thread proceeds to assign the task to a worker process. The basic structure of an active tuple is displayed in Figure 3.

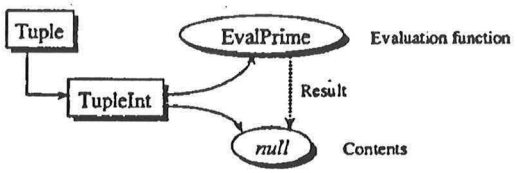


Figure 3. An example of the basic structure of an active tuple.

A pool of worker processes is maintained, and their server names are recorded within a worker registry local to the Tuple Space manager. A number of workers would normally co-exist in the system, with one worker process assigned to each workstation in use. Load balancing ensures an equal workload for all processors. Load statistics are maintained for each processor, and a request is always forwarded to a processor with the lightest load. Each worker then runs the evaluation routine, and returns the result to the Tuple Space Manager. The load balancing technique used is naïve in that the load statistic is only a measure of the number of *eval* tasks that a processor is currently executing. It does not take into consideration any differences in communication delays, processing power, or the current load on a machine in terms of all of the processes that it is running. The load-balancing algorithm could be extended to support these additional characteristics with appropriate weights. But the important point is that load statistics are dynamic and always taken into consideration when assigning tasks.

Once all fields of the tuple being stored are sequentially processed, the result is immediately stored in the Tuple Space. Therefore, no distinction is made between the *out* and the *eval* methods, as evaluation is done purely as a result of the content of tuple fields. This reduces the number of primitives required to be understood and utilised by the client application programmer, by simplifying the interface. It should be noted that an important component of this work was the creation of a very intuitive and easy-to-use interface. A basic architectural view of the worker model is illustrated in Figure 4.

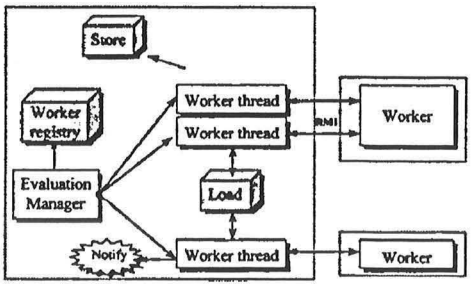


Figure 4. A basic architectural view of the worker model.

The fact that the *out* method spawns a new thread and immediately returns allows the client application to quickly queue up multiple requests and continue with other processing, then perform a blocking read, using the *in* primitive, to wait for the results at a later point in the code. Thus, the client can continue to work in parallel with the worker pool, maximising the workload on each workstation. An option is to assign workers to different partitions of a problem space, in a divide-and-conquer manner. In terms of initialisation, worker process agents are manually created on distinct workstations and added to the worker registry residing in the Tuple Space manager.

ARCHITECTURAL NOTES

One potential difficulty encountered when mapping the worker model to an implementation was a statement found in Sun's specification notes on RMI in Java.[3] Remote methods are not guaranteed to run in separate threads if the calls are invoked from a single process, although the methods will indeed run in separate threads if the callers originate from different Java virtual machines. Therefore, multiple evaluation methods may not run in multiple concurrent threads on worker processes, but could be queued up instead. However, the load balancing mechanism for determining the load on a worker, as explained earlier, would still be useful even in the event of *eval* tasks being queued on a single worker. Instead of the load number representing the total number of concurrent threads being executed on a worker, it would represent the number of requests on its input queue, likewise a satisfactory measure of load. The simulation runs of JaCaL on our Unix environment indeed showed that RMI requests were being queued up. However, since *eval* requests originating from a single client application result in *eval* threads being spawned by the Evaluation Manager and an immediate return occurs from each RMI call, this restriction did not invalidate our parallel-processing model in any way.

It must be observed that if the worker processes cannot initiate communication with the Tuple Space Manager while performing their evaluation tasks, then the input and output of the evaluation method is restricted to the entry and exit points of the specified evaluation method. This restricts the use of the system in an agenda-based parallelism approach, in which synchronisation of events would depend on the contents of the Tuple Space. So, JaCaL was extended to support communication between a worker and the Tuple Space server at any time (with the connection again being transparently set up), with full access to all primitives. The burden of creating applications interactive to the Tuple Space is placed on the user, but it does allow for greater flexibility in writing parallel applications. Although this can potentially increase the bottleneck in accessing the Tuple Space and impose greater penalties on access time for clients, this model shifts as much processing as possible to the worker pool, improving the overall efficiency. The user is afforded a great deal of flexibility in defining custom evaluation methods and partitioning the work appropriately.

Overall, the architecture of JaCaL provides effective parallel computation for client tasks that mainly involve intensive, raw computation with minimal inter-task synchronisation. Thus, the implementation is well suited for coarse granularity applications. It should be noted that initially, a multithreaded approach was taken for workers, in that each worker constituted a single thread rather than a process. However, this approach was found to be unsatisfactory. Since current implementations of the Java Virtual Machine do not allow multiple threads to be distributed or even run concurrently on multiple processors of a single machine, the normal thread model of Java was found to be insufficient for a truly concurrent execution model. That is, processing evaluation methods in multiple threads will result in no advantage in performance, as completing ten threads would require roughly ten times longer than running a single thread, as all processing would be done in a time-sharing fashion. In addition, no input/output operations to disk are possible in evaluation methods, and thus full CPU utilisation occurs even with only one thread in the system. Thus, it was decided that a client-server model with a distributed worker pool would best fit the requirements, and so the approach described in 0 above was taken.

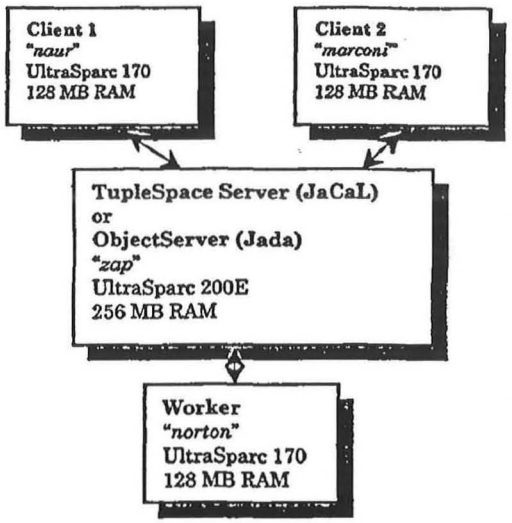


Figure 5. Configuration for the Tuple Space access performance test.

5. CLIENT APPLICATION EXAMPLE

A standard serial routine has been implemented in a client application that addresses the problem of finding all of the primes within a given range. The prime finding problem has become a benchmark example been used by many researches in the realm of parallel programming literature especially for verifying the correctness and efficiency of implemented Linda primitives.[1,2] The algorithm stores a number in the Tuple Space with a flag indicating whether it is prime or not after the determination is made. Consecutive numbers are tested for division by prime numbers by retrieving previous results from the Tuple Space. The high degree of interaction with the Tuple Space allowed the JaCaL library to be comprehensively tested for correctness of functionality and usability.

6. PERFORMANCE MEASUREMENTS

A number of performance benchmark tests were run on the JaCaL system, as well as Jada, an implementation described earlier, in section 3, one of the very few implementations publicly available for testing purposes. Both systems were tested in the same computing environment for minimal bias. The following is a description of the test configurations set up for the performance tests of the simulations.

In the first test scenario, a rough measurement was made of the performance of the RMI server access routines. The configuration is shown in Figure 5, where each block constitutes a machine, and the arrows denote Ethernet LAN connections. Client 1 recorded a tuple consisting of an identifier and an integer value in the Tuple Space by invoking the *out* primitive on the server. The other client, Client 2, blocked waiting for this value to appear by calling the *in* operation. Once this value become available, client 2 retrieved it and then immediately wrote a similar tuple to the Tuple Space, while client 1 blocked waiting for it. This "Ping-Pong" effect was repeated for 1000 iterations and the average duration of each *in* and *out* pair was calculated. This experiment was repeated with the Jada system[6] by running the ObjectServer on the machine previously occupied by the TupleSpace in the earlier JaCaL test. Finally, a test run of the JaCaL implementation was made in which each tuple written was active (an empty evaluation method was attached) which caused the Tuple Space to invoke a worker process, thus adding a higher communication cost to each transaction. The following results were obtained:

Implementation	Active tuple (evaluation performed)	Average out & in access time
JaCaL	No	137 ms
JaCaL	Yes	192 ms
Jada	No	193 ms
Java-Linda	No	400 ms

It is clear that running within the same environment, JaCaL is approximately 29% faster than Jada when comparing simple Tuple Space accesses, while Tuple Space accesses invoking workers in JaCaL cost as much as the basic accesses in Jada (where no workers are supported at all). Although the Java-Linda implementation could not be acquired for testing, the published results[4] indicate access times of 400 ms for combined out & in accesses on a similar network, also considerably slower than that of JaCaL. This proves that JaCaL is a reasonably efficient implementation.

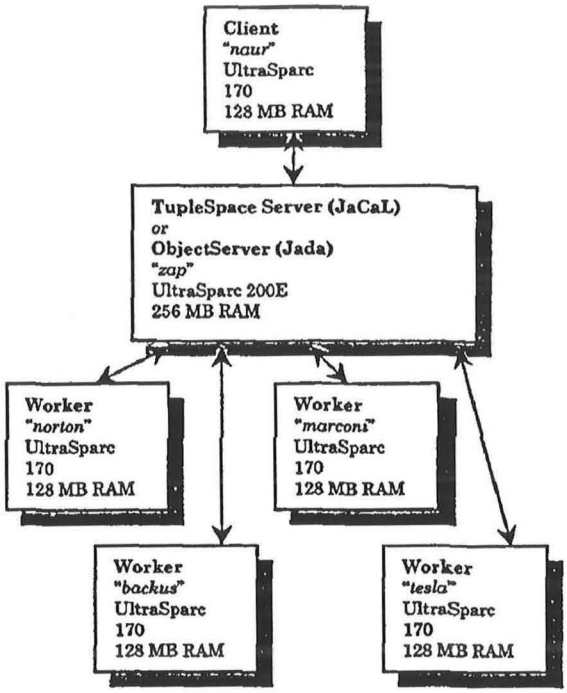
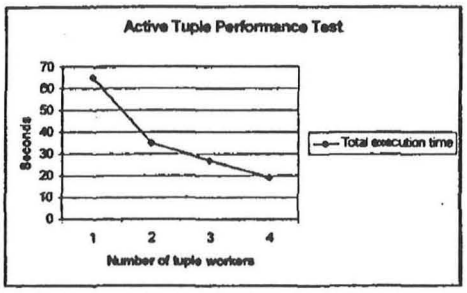


Figure 6. Configuration for the active tuple evaluation by multiple worker performance test.

In the second series of performance tests made on the JaCaL system, the improvement in processing speed due to parallel execution of active tuples was measured. The test configuration is shown in Figure 6. A single client application initiated 12 consecutive tasks (for equal division among workers) by writing that number of active tuples to the Tuple Space, with each tuple having an evaluation method attached, which performed a repeated arithmetic operation. In each of the four scenarios run, the number of workers available for processing was modified. Load balancing ensured that each worker was assigned an equal number of tasks, which was reasonable in light of the almost identical capabilities and loads on each of the worker machines. The results are as follows:

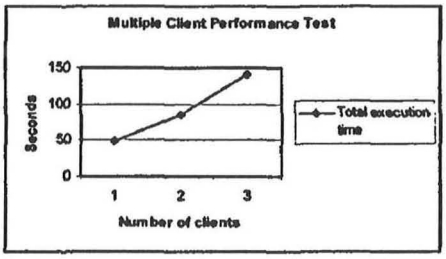
No. of workers	Time to finish all jobs	Speedup over 1-worker case
1	65.1 s	—
2	35.2 s	1.85
3	26.7 s	2.44
4	19.3 s	3.37

It is clear that as the number of workers is increased, the amount of time required to complete all client tasks decreases proportionally. This proves that the JaCaL implementation is scaleable to the number of tasks that it serves. In the third test scenario, the test configuration of Figure 5 was again utilised. First, Client 1 initiated 10 lengthy consecutive evaluation operations, then read all of the results back from the Tuple Space. Next, clients 1 and 2 each simultaneously requested 10 disparate evaluation operations of the same Tuple Space, and the longest



execution time was observed. A test with three clients, where the third client machine was equivalent to the two others, was also executed. The results are as follows:

No. of clients	Average execution time	Slowdown over 1-client case
1	48.2 s	—
2	85.3 s	77%
3	141.4 s	193%



It is therefore evident that the execution time of the tasks on the server is proportional to the number of clients being serviced by the system, and that JaCaL is scalable to the number of clients making simultaneous requests.

7. CONCLUSIONS

The top priority of the JaCaL implementation was a useful exercise in the design process, implementation, and simulation of a feasible distributed programming model for Java, and a successful attempt at a faithful, easy-to-use and working implementation of the same. In order to achieve this, the implementation was designed so as to satisfy near semantic equivalence for the Linda primitives. We strongly believe that these goals were achieved.

It is widely recognised that Java's RMI mechanism makes a trade-off in terms of communication cost versus complexity of programming, although the access times on a local area network were not found to be prohibitive at all during simulation. The usefulness and correctness of the JaCaL Linda primitives have been thoroughly verified using the benchmark prime finding problem and early test harnesses, in addition to the simulations. Much of the work was done in a prototypical approach so as to test the feasibility of the architecture at different stages, which led to an examination of different alternatives, with the final result being the best, in our opinion.

Also, the system was designed to make logical, clean, and flexible use of existing Java distributed features, rather than striving for maximum efficiency. The general aim is only for the user to have at his disposal an easy-to-use distributed programming library, one that will allow a reasonable level of parallel computation of large tasks. The performance tests indicate the high efficiency and scalability of our architecture. The combination of the *out* and *eval* operations into an active data tuple is, to our knowledge, a new addition to the work on Linda and makes effective use of Java's object-oriented paradigm.

A distributed Tuple Space was not found to be particularly advantageous in this model, as only the results of long computations would satisfy the criteria for coarse-grained processing, especially in light of RMI's relatively high communication cost. Assigning micro Tuple Spaces local to workers would necessitate increased communication cost in routing matching requests from clients. However, less contention for the central Tuple Space manager would result in faster access times. Tuples could be distributed across multiple participating servers through a hashing mechanism, a viable option taking into account that all objects produce hash codes in Java.

8. FUTURE WORK

Throughout the design process, a number of possible enhancements were considered, and these are summarised here. For instance, one possible improvement would be to use a data type tree for the storage, in which each node would constitute a particular field data type, while levels would signify consecutive fields in the tuple. This would result in a quicker search through the Tuple Space when matching anti-tuples. In addition, the nesting of tuples such that individual fields could contain entire tuples themselves would lessen the number of RMI calls required to store and retrieve data. Additional constructors could easily be added to the *Tuple* class to support this feature. A third improvement would be to allow concurrent reads of the tuple store, by synchronising on individual tuples rather than on the container as a whole.

A number of improvements to the evaluation mechanism could also be made. For example, multi-threading could be added to individual worker processes, and the user could optionally specify higher priorities for certain evaluation objects. Java's support for thread priorities would be used to advantage in this case. Also, to improve the reliability of the system, a watchdog timer could be added to monitor the status of worker processes and stop forwarding evaluation requests in case of failure. Processes could be dynamically spawned on new workstations to handle greater workloads, and load balancing could be skewed to account for differences in

workstation speeds and loads. Lastly, timeout exceptions on blocking receives would increase user-friendliness by preventing unanticipated deadlock situations.

9. REFERENCES

[1] N. Carriero and D. Gelernter, How to Write Parallel Programs: A First Course, Cambridge, MA, MIT Press, 1990.

[2] N. Carriero and D. Gelernter, How to Write Parallel Programs: A Guide to the Perplexed, *ACM Computing Surveys*, Vol. 21, No. 3, September 1989.

[3] JavaSpace Specification, Sun Microsystems Inc., http://java.sun.com/products/javaspaces.

[4] Andrew Smith, Towards Wide-Area Network Piranha: Implementing Java-Linda, http://www.cs.yale.edu/users/asmith/cs690/cs690.html.

[5] Robert Greig, Javelin: A distributed Linda System, http://www.dcs.gla.ac.uk/~greigrg/javelin/report.html.

[6] Davis Rosi, Jada: multiple object spaces for Java, http://www.cs.unibo.it/~rossi/jada/.

[7] Linda Introduction, Scientific Computing Associates, Inc. http://www.sca.com/ltutorial.html.

[8] W. Leler, Linda meets Unix, *IEEE Computer*, Vol. 23, No. 2, pp. 323-357, February 1990.

[9] Jim Farley, *Java Distributed Computing* (O'Reilly & Associates Inc., 1998).

[10] J. P. Morgenthal, The Distributed Java Platform, *Information Systems Management*, Winter 1998.

[11] Scott Oaks, Henry Wong, *Java Threads* (O'Reilly & Associates Inc., 1997).