Implementing Linda For Distributed and Parallel Processing

Jerrold S. Leichter
Department of Computer Science
Yale University
P.O. Box 2158 Yale Station
New Haven, Connecticut 06510
LEICHTER-JERRY@CS.YALE.EDU

Robert A. Whiteside
Scientific Computing Division
Sandia National Laboratories
Livermore, California 94550
WHITESIDE@SANDIA-2.ARPA

April 1989

ABSTRACT

In a recent paper [17], we described experiments using the VAX LINDA system. VAX LINDA allows a single application program to utilize many machines on a network simultaneously. Applications implemented on the network at Sandia National Laboratories have achieved speeds considerably greater than that of a Cray-1S.

In this paper, we discuss the implementation of the VAX LINDA system. The Linda language was originally conceived as a tool for programming parallel applications on multicomputers, and in fact VAX LINDA supports such programming on multiprocessor VAXes. We have demonstrated that, for suitable applications, we are able to treat an arbitrary collection of separate machines on a network as a "virtual multicomputer." Accomplishing this requires careful implementation. It also involves some effort to get around limitations in operating systems and network implementations which were not designed with this kind of usage in mind.[1]

1 INTRODUCTION

In the drive to obtain the ultimate in supercomputer performance, it has become clear that future machines will necessarily operate in parallel. Programming such machines has often proved difficult. The Linda language [1] is a language for explicit parallel programming. Unlike many such languages, Linda is *not* tied to any one parallel hardware architecture: It holds promise as a *portable* parallel programming language [6].

While it is clear what a *sequential* computer is, a *parallel* computer is essentially defined by negation: Any computer that does not limit itself to a single stream of operations on a single stream of data is a parallel computer. Even if we restrict ourselves to so-called MIMD (Multiple Instruction Multiple Data item) machines, the variety of machines is extremely broad. Linda has been implemented on bus-based disjoint memory machines [7, 9], shared memory multiprocessors [9], and disjoint memory hypercubes [5, 15]. Work on hardware specifically designed to support Linda has also been reported [2]. Each of these implementations has been successful; each has required a different class of implementation techniques.

The first author's dissertation [14] describes an implementation of Linda, VAX LINDA, which operates on two levels. At one level, it uses a shared memory for communication among multiple processes running on a single "node." Whether a node is a truly parallel multiprocessor, or a single processor shared among a number of processes, is unimportant. At the second level, the implementation views multiple nodes connected on an Ethernet local area network (LAN) as a novel kind of "virtual multiprocessor." As far as Linda programs can determine, this virtual multiprocessor is quite real. While communication among its components is, of necessity, much slower than communication among processors sharing memory, it still *looks* the same, in exactly the way that virtual memory *looks* the same to a program as real memory does.

When we used VAX LINDA to perform experiments at Sandia National Laboratories, we found that no one Ethernet had enough machines connected to it to satisfy our needs. So we implemented an application-specific third level of communication, which operated over DECnet links in a wide area network, or WAN.

As we will see, each level of this hierarchy raises interesting implementation issues.

2 A BRIEF INTRODUCTION TO THE LINDA LANGUAGE

The Linda language has been described elsewhere [11, 7], though the details of the language have changed since those papers were written. We will briefly describe the LINDA-C language used in VAX LINDA. More details can be found in [17]; the definitive reference is [13].

Linda was first defined by Gelernter [11]. As it has evolved, however, Linda is not a complete language. Rather, it is a set of objects and operations on those objects that are intended to be *injected* into an existing language, thus producing a new language intended for distributed programming.

2.1 FUNDAMENTAL OBJECTS

Linda is based on two fundamental objects: *tuples* and *tuple spaces*.

[1] Work at Sandia was supported by the United States Department of Energy. Jerrold Leichter's work is supported by a grant from the Digital Equipment Corporation's Graduate Engineering Education Program, and by National Science Foundation grants CCR-8601920 and CCR-8657615, and ONR N00014-86-K-0310.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1989 ACM 0-89791-309-4/89/006/0041 $1.50

Tuples are ordered collections of *fields*. Fields have fixed types associated with them; the types are drawn from the underlying language. Which types are supported is up to the implementation, but the intent is that any type that "makes sense" should be allowed. In particular, in LINDA-C a field can contain an entire array or structure. LINDA-C also allows the programmer to create new, unique types, providing additional control.

A field can be *formal* or *actual*. A formal field is a place-holder — it has a type, but no value. An actual field carries a value drawn from the set of possible values allowed for that type by the underlying language.

For example, suppose that the underlying language has types **int** (integer) and **float** (floating-point number). Then $\langle 1_{\mathbf{int}}, 1.5_{\mathbf{float}}, 2_{\mathbf{int}} \rangle$ is a tuple with three fields: Two integers, 1 and 2, and a floating point value, 1.5. It is different from the tuple $\langle 1_{\mathbf{int}}, 1.5_{\mathbf{float}}, 2_{\mathbf{float}} \rangle$, whose third field has a different type.

Tuple space is a collection of tuples. It may contain any number of copies of the same tuple: It is a *bag*, not a *set*.

Tuple space is the fundamental medium of communication in Linda. It is a *global, shared* object — all Linda processes that are part of the same Linda program have access to the same (logical) tuple space.

2.2 TUPLE SPACE OPERATIONS

The **out** operation inserts a tuple into tuple space. If *f* is a variable of type **float** with value 1.5, and *i* is an **int** with value 2, then **out**(1,*f*,*i*) would insert the tuple we saw earlier, $\langle 1_{\mathbf{int}}, 1.5_{\mathbf{float}}, 2_{\mathbf{int}} \rangle$, into tuple space.

The **in** operation extracts tuples from tuple space. It finds tuples that *match* its arguments. The tuple of the previous paragraph could be extracted by the operation **in**(1,1.5,2).

Formal fields are denoted by a question mark prefix. What should follow a question mark is language-dependent; the intent is that it should be "anything that can go to the left of an equal sign." Typically, it is just a variable.

When a formal is used in an **in**, any actual in the tuple will match. The operation **in**(1,?*f*,2) might extract the same tuple as our previous operation. In addition to removing the tuple, it would assign 1.5 to *f*. Note that the type of *f* is significant: For this match to be possible, *f* must have type **float**.

It is possible to place a tuple with a formal field into tuple space — that is, the question mark prefix is valid in **out** operations, too. This ability is rarely used; it is intended to support operations such as addressing any of a number of servers, while retaining the ability to address a particular one.

After an **in** operation, the tuple matched is removed from tuple space. The **rd** operation is similar to **in**, but leaves the matched tuple in tuple space unchanged. It is used for its side effects of bindings and synchronization.

2.3 TUPLE MATCHING

The **in** and **rd** operations are defined in terms of *matching*. Call the tuple defined by the fields of an **in** or **rd** the *template*. For a tuple to match the template, they must have the same number of fields; the corresponding fields must have the same types; fields that are actuals in both tuple and template must be equal; and no fields can be formals in both tuple and template. When a formal in a template matches an actual in a tuple, the value matched is assigned to the variable named by the formal.

If no matching tuple can be found in tuple space, **in** and **rd** block, and the process waits for a tuple to appear. If there is more than one matching tuple, **in** and **rd** choose one non-deterministically.

2.4 EXAMPLES

Some simple examples illustrate some of the uses of the Linda constructs. Simple message passing can be accomplished quite easily. Suppose that the sending process has executed:

```
out("Node 1",3.14159);
```

After the receiver process (`Node 1`) completes

```
in("Node 1",?val);
```

the value of variable *val* will be 3.14159, and the two operations will have effected the transfer of a floating-point value from the sender to `Node 1`.

Many parallel computations can be organized as a master and a group of workers. Suppose the master requires the results of one hundred computations, and these computations may be performed in parallel. The outline of the master process is then:

```
for (i = 0; i < 100; i++)
        out("Do this",i);
for (i = 0; i < 100; i++)
{       in("Result",?j,?result);
        result_vec[j] = result);
}
```

The corresponding workers have the form:

```
while(1)
{       in("Do this",?job);
        result = do_the_job(job);
        out("Result",job,result);
}
```

3 THE IMPLEMENTATION HIERARCHY

Tuple space is often best viewed as a memory. It differs from traditional memories in two significant ways: Its "word size" is not fixed, but is entirely determined by the tuples the programmer choses to place into it; and it is accessed associatively.

An important driving force behind all computer architectures has been the organization of memory. All systems today use a memory hierarchy of many levels, from registers to caches to main memories to disks and so on. VAX LINDA views the tuple space implementation problems as one of organizing a memory hierarchy. Figure 1 illustrates the lower two levels of this hierarchy. Each $Node_i$ box represents one or more VAX processors sharing main memory. We call this the *single node level*. These processors are, in turn, connected by an Ethernet. The entire figure comprises the *LAN level*. Finally, although we don't show it in the figure, multiple groups of machines at the LAN level could be linked into a *WAN level*.

The VAX LINDA implementation attempts to hide this hierarchy of levels from the Linda programmer, although there is naturally a difference in performance. In fact, tuple operations that can complete at the single node level are

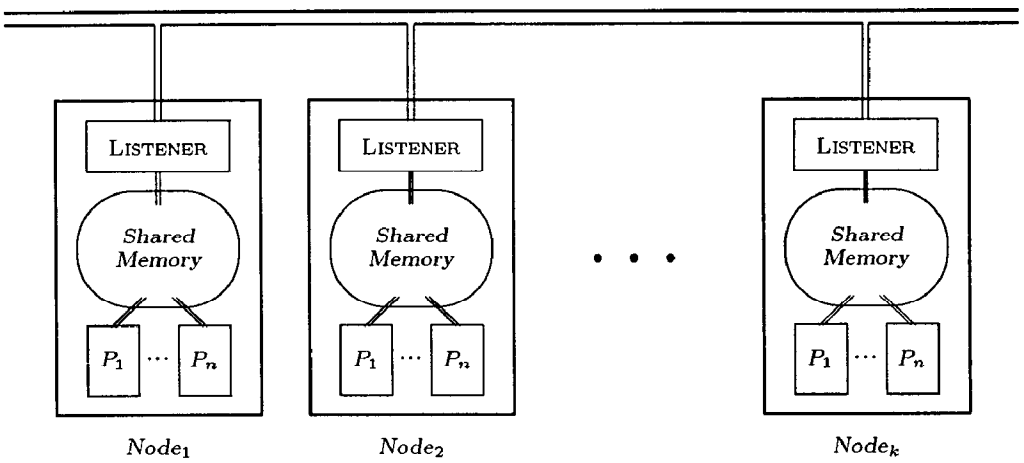

Figure 1: Idealized VAX LINDA Program on a Local Area Network

about order of magnitude faster than those which must be handled at the LAN level.[2] Depending on the configuration, operations at the WAN level would be up to an order of magnitude slower than those at the LAN level.

4 THE SINGLE NODE LEVEL

The fundamental implementation issue at the single node level is matching: Given a template, search tuple space for a matching tuple. Since any combination of formal and actual fields is possible in a template, we must be prepared to search tuple space using any subset of the fields as a key. In database terms, we must perform a multi-key retrieval.[3] It's a curiosity that the literature contains many algorithms for single-key retrieval, for both "in-core" (symbol tables) and disk (indexed files) use, as well as algorithms for multi-key disk (relational database) use; but very little work appears to have been done on the "in-core" multi-key problem. The multi-key retrieval problem for tuple space is made all the more difficult because the expected lifetime of a tuple is short: It is impossible to amortize the cost of a complex insertion procedure across a large number of lookups.

The approach taken in all existing implementations of which we are aware, VAX LINDA included, is to avoid the issue. VAX LINDA organizes tuple space as a hash table; it generates the hash code from the *type signature* — the list of types of all the fields — and the value of the first field of the tuple, which it requires to be an actual. While it is easy to construct examples for which this approach fails badly, it is surprising how many programs do well anyway: Most of their tuples are easily distinguished by the information that determines the hash value, and the cases where multiple fields are really used in matching have only a relatively few tuples with the same leading field.[4]

There is a fundamental point here: Just because a very general feature is present in a system doesn't mean any single program will use it in all its generality. Efficient implementations of systems rely on recognizing the cases the actually occur often enough to be worth optimizing. For example, the compiler described by Carriero [9] recognizes some very common patterns of tuple usage, such as those implementing semaphores or simple message passing.[5] It then replaces the general code with tuned special-case code. VAX LINDA doesn't currently include these optimizations, although we plan to add them. We have suggested a number of alternative organizations which support multi-key retrieval directly [14], and we plan to experiment with them in the future.

4.1 STRUCTURE OF A TUPLE SPACE

Figure 2 shows how VAX LINDA stores a tuple space. Everything shown in the figure is stored in shared memory, in a VMS global section. The Header contains control information and locks. Tuple data is stored in the Data Table, growing up from the bottom. The Hash Table region contains the heads of hash buckets, stored as linked lists of entries in a Chain Area which grows downward toward the tuple data. When the two meet, a garbage collection[6] is done, combining all free space in the Data Table into a single contiguous block.

Data Table entries hold the tuple data in a "normalized" format. A run-time type system identifies all types used in tuples, and all type signatures of tuples that actually occur in the program. A Data Table entry includes the type signature index, from which the number of fields, and their respective types, can be determined. At this level, fields come in two styles, fixed and variable length. The type system records the size of fixed-length fields; the sizes of variable length fields are stored with the fields. This representation is language-independent. Language-dependent information stored in the type system allows VAX LINDA to convert, say, a C null-terminated string into a normalized variable-length field when a tuple is created, then convert it back when the tuple is matched. While code to support multiple languages

[2] An **out-in** pair on a MicroVAX II takes about 2.1 ms. at the single node level, and about 22.8ms. at the LAN level. These are worst case values for code that has not been heavily optimized [14].

[3] The presence of formal fields in tuples, not just in templates, adds complexity. Fortunately, such constructs are rare in actual code, and we will not consider them further.

[4] Actually, a close analysis reveals that this fact should not be so surprising after all; it is the result of common tuple usage patterns and their interaction with such an implementation [14].

[5] It also includes the obvious optimization of allowing *any* field that is actual in all instances of tuples with the same number and types of fields to be used as a hash key, rather than insisting on using the first field.

[6] More accurately, a storage compaction — as we will see, the only pointers to tuple data are from Hash Table and Chain Area entries, so no "sweep" phase to follow pointers is necessary.

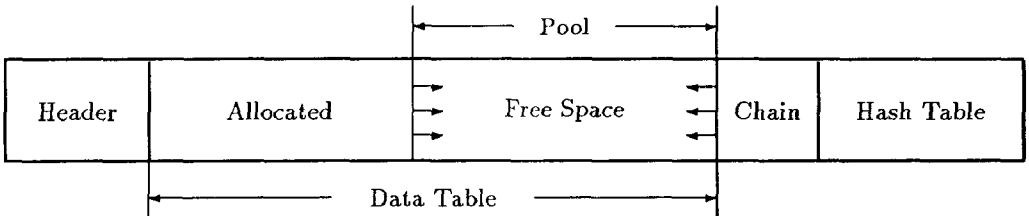

Figure 2: Layout of Single-Node Tuple Space

in VAX LINDA is not complete, the system is intended to be able to use different conversions at the time of tuple creation and matching, allowing tuple space to be used as a data representation translator.

4.2 SYNCHRONIZATION AND GARBAGE COLLECTION

A lock in the Header allows either shared or exclusive access to the tuple space. Most access is shared; garbage collection requires exclusive access. Since tuples in the Data Table may move during a garbage collection, no process can assume that any pointer it holds into the Data Table will remain valid if it releases the access it has been granted by the Header lock.

While Data Table entries move, Hash Table and Chain Area entries are guaranteed to remain fixed. This leads to an curious fact about the garbage collector: Rather than tracing the "pointers" (the Hash Table and Chain Area), it traces the "data" (the Data Table). Each allocated Data Table entry contains a back-pointer to the (unique) Hash Table or Chain Area entry that points to it. The garbage collector can thus sweep through the Data Table, updating Hash Table and Chain Area entries as it moves Data Table entries. One interesting programming technique grows out of this design: It is possible to have Data Table and Chain Area entries that point to each other, and use them across garbage collections, even though the Chain Area entry is not linked into any hash bucket. This technique turns out to be important when, for example, an attempt is being made to send a tuple to a remote node. The remote node may refuse the tuple, perhaps after a delay; the sending node must be prepared to re-insert it into tuple space. This is most easily accomplished if the tuple effectively remains in tuple space the whole time.

Single-bit locks, accessed using synchronized VAX hardware instructions, control access to each hash bucket. By design, Hash Table and Chain Area entries are small (6 bytes) so that we can allocate large numbers of them (typically several hundred). Hence, chains are usually short. Even when a chain gets long, it is often filled with tuples that match the same set of templates. Since the semantics of in and rd allows us to return *any* matching tuple, we can often just return the first member of a long chain. As a result, in our experience, neither contention for the bucket lock, nor time spent scanning long bucket chains, has been a significant factor.

Allocating memory from a contiguous region and relying on a garbage collector simplifies the code. In our experience, the cost of running the garbage collector is small — so small, in fact, that we have never successfully measured its impact even on test programs designed to stress it.

Writing low-level code to access garbage collected storage is tricky, as is writing code that uses a large number of locks for synchronization. Curiously, the discipline of meeting both sets of constraints seems to ease the difficulty. Pointers into the garbage collected Data Table only remain valid as long as the program retains the appropriate locks, and holding those locks ensures that no collections take place. Hence, the areas in which pointers are valid are clearly demarcated by lock and unlock calls. We know of no bugs that arose during the development of VAX LINDA as a result of violations of either set of constraints.

4.3 EFFICIENT SYNCHRONIZATION

Parallel programs generally have access to two levels of locking mechanisms: Whatever interlocked operations the hardware provides, and the synchronization mechanisms provided by the operating system. There is typically a two order of magnitude difference in the cost of the two levels. One thing that became clear during the development of VAX LINDA was the difficulty of combining the two levels. Low-cost hardware locks that are usually released quickly — the locks on tuple space hash buckets, for example — are best accessed using code that does "busy waiting," at least when there is a good chance that there is another processor available to run the code that will release the lock. When that code is likely (or certain) to run on the same processor, this approach guarantees poor performance: The process will "busy wait" until the end of its CPU quantum. Instead, it is better to release control to the operating system (and thus some other process).

Unfortunately, it's difficult or impossible to know how good the chance of another processor being available is, especially in a dynamic setting. One is reduced to code of the form:

1. Try test-and-set up to K times; if any attempt succeeds, done.
2. Release CPU.
3. Goto 1.

There is no obvious way to choose K: $K = 1$ is optimal when the process holding the lock will certainly use the same CPU, $K = \infty$ when it certainly will not. In the current version of VAX LINDA, we use $K = \infty$, following a folk wisdom of optimizing for the "best" (here, most parallel) hardware. We certainly pay a cost on real hardware, but we can't quantify it.

The underlying problem is that the operating system is simply unaware that the program is attempting to use the hardware synchronization primitives. Although it is possible to design appropriate primitives into operating systems [3], Version 4 of VMS, which we used, is typical of traditional operating systems in lacking them.[7]

[7]Version 5 of VMS has added library routines that, at least in intent, provide appropriate primitives. It appears, however, that they are simply shells built on the same inadequate support as our own code.

Traditional operating systems view their rôle as mediating among competing processes. Parallel programs, of course, are build of *coöperating* processes. The difference in viewpoint leads to conflicts. The lack of fast synchronization primitives is one result: Competing processes rarely synchronize with each other. Another interesting example, reminiscent of the *convoy phenomenon* [12], arises from a conflict between user-level locking and the operating system scheduler. Consider a situation in which several processes are waiting for an event, such as the arrival of a matching tuple. Since the wait may be long, the waiting processes use a VMS system service to stall themselves.

When the event occurs, the process which notices needs to walk the queue of waiting processes and awaken them. The straightforward implementation leads to very poor performance if there are more processes than processors, however. Since processes in VMS usually wait for external events, such as completion of input, VMS grants a process leaving a wait state a temporary priority boost.[8] The awakened process thus gains a priority advantage over its awakener and, assuming there are no free processors, preëmpts it. What usually happens next is that the awakened process needs to gain access to the data structures involved — the queue it was in, for example. However, since the awakening process was manipulating that structure it still holds a lock on it. Hence the awakened process stalls. This pattern repeats for each process in the queue; only when the queue is finally empty will the awakening process release its locks and (eventually) allow the awakened processes to proceed. In cases where only one of the processes will actually get the resource they were all waiting for, this situation quickly re-establishes itself.

A simple re-arrangement of the code eliminates this problem: The awakening process must first scan the queue and remove all processes it will awaken, then release its locks and begin awakening processes. It is easy to see that this results in an optimal pattern of execution, however many processes and processors there are.

This issue may seem trivial, but it produced a significant degradation in VAX LINDA's performance. It also appears to be a more common problem than one might expect. After we described it to Bjornson, he discovered a similar effect in his Intel iPSC implementation of Linda [4].

5 THE LAN LEVEL

The single node level builds a shared tuple memory on top of a shared byte memory. The LAN level must combine multiple segments of tuple memory into a unified single tuple space. The fundamental issue here is tuple and template *distribution*: Ensuring that matching tuples and templates find each other.

Existing implementations have used at least four different approaches. Carriero's original S/Net implementation [7] broadcast all tuples (**out**'s) so that all tuple space segments were identical. Templates (**in**'s and **rd**'s) could be handled locally, except for a "delete protocol" to ensure that only one **in** for a tuple succeeded and that all copies of the matched tuple were deleted. We call this scheme *positive broadcast*.

VAX LINDA uses *negative broadcast*, the logical converse of Carriero's S/Net scheme: Tuples are handled locally, while templates are broadcast. We discuss our implementation in more detail below.

[8] This helps I/O-bound processes finish quickly and get out of the way of CPU-bound ones.

Two independent implementations of Linda for hypercubes, Bjornson's for the Intel iPSC [5] and Lucco's for an experimental hypercube developed at Bell Laboratories [15], chose *distributed hash tables*. In this scheme, each node in the cube has responsibility for a segment of tuple space defined by some subset of the possible hash code values of tuples. Any **out** sends its tuple to the node responsible for it; any **in** or **rd** requests a matching tuple from that node. Many variations on and optimizations of this basic scheme are possible, and in fact the detailed organizations of Bjornson's and Lucco's implementations are quite different.

Finally, an implementation of the Linda primitives in hardware [2] uses a hybrid strategy: Organized as a 2-d mesh, it broadcasts tuples along rows and templates along columns, an idea originally proposed in Gelernter's dissertation [10].

5.1 CHOOSING A DISTRIBUTION STRATEGY

Distribution strategies differ along a number of axes. One important axis is the hardware support required. Carriero's choice was influenced by the S/Net's hardware support for reliable broadcast. While an Ethernet provides a broadcast facility, neither it nor any kind of Ethernet communication is immune from loss of packets. Experiments indicated that a tuple broadcast strategy without some kind of scheme to detect lost packets would quickly fail, even on Ethernet under fairly light load. One of the strongest points of a template broadcast strategy is that it can, in a natural way, be made immune to packet loss: While complex algorithms are needed to allow the broadcast of a tuple to be repeated safely, it is simple to allow the broadcast of a template to be repeated.

Referring back to Figure 1, notice the boxes labeled LISTENER. LISTENERs implement the LAN level of our hierarchy. Ideally, a LISTENER appears to other processes on its node to be just another Linda process, manipulating tuple space with **in**'s and **out**'s. The single node level is unaware of the existence of the LAN level: When it processes an **out**, it places a tuple in the local tuple space segment and awakens any local process — perhaps the LISTENER — which might be waiting for it. When it processes an **in** or **rd**, it searches the local tuple space segment. If it finds a matching tuple, it may extract it. If it fails to find a match, it registers its template in a shared data structure and blocks.

The LISTENER, of course, is not quite like user Linda processes. It scans the registry of templates and multicasts them to an address that all LISTENERs listen to. A LISTENER receiving a template searches its local tuple space segment. On failure, it does nothing; on success, it effectively does an **in** for the matched tuple. It then sends the matched tuple to the requesting LISTENER, using a virtual circuit maintained using standard techniques. Once successful transmission has been completed, the tuple is deleted from the sending tuple space segment.

At the other end of the virtual circuit, the LISTENER receiving the tuple merely **out**'s it into its local tuple space segment. If doing so satisfies a waiting template, some blocked **in** or **rd** will complete, with no indication that the matching tuple came from a remote node.

If a LISTENER receives no response to its multicast of a template, it is free to try again later. If it receives a single response, it can generally use that response to satisfy a pending **in** or **rd**. However, it may also receive multiple responses.

By definition, a tuple is in tuple space if it is in *any* tuple space segment. It makes no difference *where* the tuple is stored. Hence, no problems arise if a LISTENER's template draws multiple responses: The LISTENER **out**'s *all* the received tuples into its local tuple space segment. Their movement may have been unnecessary, but at worst it is not incorrect. For some patterns of tuple usage, the "extra" responses may be helpful — the **in** which requested them could be in a loop, and may be executed again in a short time. Later executions will find matching tuples available locally, and complete quickly.[9]

ACTUAL IMPLEMENTATION

The scheme we've just outlined works, but is essentially based on polling: The LISTENER must poll the registry of pending templates, and it must poll other LISTENERs for matches. Nothing happens between polls. This leads to unnecessary overhead and delays.

The actual implementation of VAX LINDA is, for efficiency, more complex. It also blurs the sharp inter-layer boundaries implied by Figure 1:

- While a registry of pending templates exists, it is not polled by the LISTENER. While a user process is waiting for a match for its template, it cannot run the user's code. So it is instead given the responsibility for sending its template in a multicast message to all remote LISTENERs. It continues to do so periodically[10] until it finds a match. User processes do not listen to the Ethernet, however; any responses from a remote LISTENER are directed to the LISTENER on the requestor's node.
- If a LISTENER cannot find a tuple to satisfy a template it receives, it registers the template as pending, just as a blocked **in** or **rd** registers its template. Any subsequent **out** that matches the template will cause the LISTENER to be informed. It will then send the tuple to the requesting node.

  A LISTENER discards the templates it has registered when they become too old. If a repeated request for a template arrives, it is "rejuvenated," and remains in the registry.

5.2 OTHER AXES OF COMPARISON

While the primitives supported by the hardware are important, there are other factors to consider. For example, the following are some axes along which protocols might be compared:

- The amount of message traffic they produce. This is complicated for real communications media, like Ethernets, which exhibit large fixed per-message costs: Algorithms that send relatively few long messages do better on an Ethernet than those that send many short ones.
- Parallelizability of expensive operations, particularly matching. All other things being equal, it is valuable to have multiple nodes execute a match at once.
- Memory usage.
- Adaptation to typical access and communication patterns. For example, some tuples form part of distributed data structures [8]; they will probably exhibit a pattern of access much like that to data structures in conventional memory. This pattern is very different from that seen for tuples used to simulate message passing. We have identified about five such patterns. [14]
- Use of locality. Since on-node communication is much cheaper than off-node, a protocol that can get by entirely without off-node communication in common cases is at an advantage.[11]
- Robustness in the face of failures of links or nodes.

No one technique is optimal along all of these axes. The negative broadcast technique we chose does reasonably well on all these measures, with the single exception of a "multicast" pattern, in which a tuple contains information needed by a large number of nodes.[12] Such tuples can often be identified, and access to them can be optimized by replication.

Negative broadcast's greatest strengths are in use of locality and robustness in the face of link failure.

6 THE WAN LEVEL

The original design of VAX LINDA did not forsee a need to support wide-area networks. It seemed to us that LAN communication would already be too slow for most problems; going an order of magnitude beyond that seemed pointless.

However, when we attempted to use large numbers of VAXes at Sandia National Laboratories, we found that there were very few on any one Ethernet segment. To go beyond small numbers of machines, we would have to support WAN links in some way. Adding this to the support code in full generality would have required a great deal of effort, and at the time we didn't know if it would pay off. So we created application-specific extensions.

6.1 APPLICATION-SPECIFIC EXTENSIONS

Two experiments were run on WAN-connected VAXes at Sandia: BOHR, a Monte Carlo particle simulation; and CHARM, a rocket plume simulation parameter study [17]. The two sets of computations were organized in different ways.

BOHR

The communication requirements of BOHR are very simple: A master process sends some initialization data to a number of simulation processes, which then run independently, periodically sending results back to the master. Eventually, the master gathers enough results and signals the workers to stop. Tuple space is thus used for two things: Communication and control.

VAX LINDA was extended to the WAN by defining a simple protocol for transferring tuples and templates over a WAN link to a node connected to a central Ethernet. In effect, the tuple space operations where "stretched" across a virtual node of two processors, one connected directly to tuple space, one potentially far away. Since there were only a few kinds of tuples and templates used in the entire system,

[9] The gain here is the same as that seen in caches which fill a line larger than the data item actually requested, on the assumption that the executing program will probably try to access "nearby" memory [14].

[10] In the current implementation, the multicast is repeated every two seconds.

[11] How often locality will actually be seen is clearly influenced by the typical access and communication patterns of the previous item.

[12] It also does not automatically handle node failures, an issue we will return to later.

it was no problem to write code that knew how to handle each of them.

CHARM

CHARM is a computer program modeling high-altitude rocket plumes. The model has a number of parameters, and a study under way at Sandia seeks to determine the sensitivity of the plume simulation results to the values of these parameters. A sophisticated technique is used to select parameter values for scanning the space of possibilities, but the computationally intensive portion of the study reduces to running CHARM many times with different input values. Each CHARM run requires from 15 minutes to two hours of execution time on a VAX 8700, and a full sensitivity analysis may requires hundreds of such runs.

The factor which determined the organization of CHARM was the large size of the input and output data sets of each simulation run. These were contained in files of significant size. While it would have been possible to read the files and pass their contents in tuple space, this did not seem to be an efficient approach, as there already exist file transfer protocols optimized for this use.

A first cut at a "WAN Linda" implementation of CHARM thus had the following structure: A central master process created input data sets in local files. It passed the names of those files through tuple space to remote worker processes, using a design similar to that used for BOHR to allow the remote processes to join the computation. The workers then used DECnet remote file access to read their data, produced output data sets locally, and in turn passed the names of those data sets back to the master. Finally, the master again used DECnet to read the results, which it summarizes.

Several problems made this approach fail. We quickly discovered that we could not maintain enough remote file links: Some of the intermediate router nodes were unable to support them. Instead we had the master copy the input data sets to the worker nodes, then copy the results back when they were complete. While this worked, it performed poorly, since a worker that had completed a simulation had to wait for the master to copy its results back, then send it a new data set, and finally tell it to continue. This took too long. Instead, it proved to be necessary to overlap both copy activities with the worker's computation.

In the current implementation, the master maintains a status tuple for each worker, listing its current and "on deck" data set. The "on deck" data set is copied to the worker's machine ahead of time, and similarly the results are copied back after the worker begins work on the next simulation. The resulting code is quite complex, and provides an illustration of the difficulties involved in trying to coax good performance from a distributed algorithm.

SURVIVING NODE FAILURES

VAX LINDA provides immunity to link failures, but is vulnerable to node failure. The BOHR and CHARM experiments revealed that some degree of resiliance in the face of node failure is essential in this kind of system. It's not so much that systems crashed, though with up to 14 systems in some experiments that did happen. Rather, most of the problems were actually not crashes but the result of system loading. The various Linda processes were run as guests, and so were given low priority. As long as the machines were "quiet," as was usually the case at night, the priority didn't matter. But sometimes competitive jobs would appear, and the Linda processes might be granted so little run time that they were effectively removed from the computation.

The designs of both BOHR and CHARM dealt with this problem on the application level. In the case of BOHR, if some workers simply never returned results, the master would still receive all the results it needed eventually — it would just have to wait longer. In the case of CHARM, the master created an initial task list and assigned tasks to workers. As workers reported their tasks complete, the master "checked them off" and assigned new tasks. Once all tasks had been assigned, the master would start assigning any that had been out "too long" to other workers. Hence, any work dedicated to a lost processor would eventually be re-assigned. In some cases, the lost processor would later report back, so two sets of results might be received for some simulations. The master would simply discard the extra results.

Table 1: Execution Times for BOHR

Configuration	CPU's	Elapsed min.	Crays
1×VAX 8700	1	152.0	.05
4×VAX 8000 (LIV)	9	42.0	.18
11×VAX 8000 (BIG)	11	15.0	.49
Cray-1S	1	7.4	1.00

6.2 SUMMARY OF RESULTS

Tables 1 and 2 summarize the results of our BOHR and CHARM experiments.[13] The BOHR results of Table 1 were generated by runs limited to 200 trajectories. The production runs of BOHR currently being made may require over 2000 trajectories.

The BOHR program vectorizes rather well on the Cray, and a Cray-1S runs BOHR about 20 times faster than the 8700. The configuration labeled LIV in the Table consists of a set of machines on the network located at Sandia National Laboratories in Livermore. There were nine machines in all, including three 8000-class VAXes, three MicroVAX II's, and three VAX 11/780's; we've expressed the computational power of these machines in units of VAX 8000's to allow for more meaningful comparisons.

The BIG configuration consists of eleven VAX 8700, 8650, and 8550 machines in Livermore, California and Albuquerque, New Mexico. The two sites are joined by a 56 Kbit/second link.

We've also run BOHR on more heterogeneous collections of machines. In one such experiment, we ran BOHR on five VAX systems, running VMS; three Sun/3 workstations, running Unix; and an Intel 310 personal computer running Xenix. Since there is only one process on each node, and it exchanges tuples only with the master process, a very limited local implementation of Linda is sufficient — tuples need only be transmitted over a link to the remote master running on a VMS VAX.

Table 2 shows the results for a sensitivity analysis that required 99 runs of the CHARM program. The analysis required about 1500 minutes on a single VAX 8700, or about 15 minutes per simulation. CHARM runs mostly in single precision on the VAX, and vectorizes very poorly on a Cray-1S, which manages to run only a factor of 4.4 times faster than

[13] These tables are taken from the first author's dissertation [14]. Some of the figures have been previously published [17].

Table 2: Execution Times for CHARM

Configuration	Elapsed min.	Crays
1×VAX 8700	1516	.23
3×VAX 8000	537	.65
10×VAX 8000	177	1.97
14×VAX 8000	143	2.40
Cray-1S	348	1.00
Cray XMP (1 processor)	238	1.46

the VAX; recall that the factor was 20 for BOHR. We've included timing figures for a more recent Cray, an XMP, which is apparently able to vectorize CHARM somewhat more effectively.

It's clear from the Table that five 8000-class VAXes would give about one Cray of performance on this application. The collection of fourteen 8000-class machines — like the BIG configuration of Table 1, consisting of machines at Sandia sites in California and New Mexico — runs well over twice as fast as a Cray. Thus, for this application at least, VAX LINDA-C running on a network can deliver true supercomputer performance.

The CHARM application requires a very large amount of memory. Our attempts to include smaller VAXes in the computation using a configuration like LIV of Table 1 were unsuccessful — none had enough memory to avoid excessive paging.

6.3 GENERIC IMPLEMENTATIONS

We have seen that even WAN implementations of Linda, slow as they might be, are still useful for some interesting problems. We've begun investigating how to construct such implementations.

Once again, the first problem is one of tuple distribution. Our LAN implementation assumes locality of reference for tuple operations, though it leaves an opening for efficient implementation of certain non-local reference patterns. We extend this assumption to the WAN setting: We will assume that, just as many operations can be carried out at the single node level, so many of the rest can be carried out at the LAN level. Only a small residual set of tuple operations should need access to the WAN level. Hence, our model is of a group of LAN's connected by WAN links, with most traffic limited to the individual LAN's. For simplicity, we view the WAN network as completely connected with links of equivalent cost.

The current design extends cleanly to this model. Consider what conclusions a LISTENER can reach by watching the templates it receives. If it receives a template from node $\mathcal{N}$, then it can conclude that $\mathcal{N}$ has no matching tuple.[14] If $\mathcal{N}$'s template is repeated — an event the LISTENER must already detect — then no other node on the LAN has a matching tuple either. Hence, if the LISTENER has a WAN link to a LISTENER on a remote node, it should now forward the template.

An additional requirement, as we noted at the end of the previous section, is robustness in the face of node failures. While this is not strictly a WAN problem, it becomes more important when nodes are geographically distributed and are under the control of independent organizations.

[14] More accurately, that it didn't have any matching tuple a short time in the past. We ignore this distinction.

True immunity from node failure is difficult to attain efficiently. There are a number of possible "approximations," however. If the programmer can specify that some nodes are "crash resistant" — for example, they are reliable nodes over which the programmer has direct control — it is straightforward to have non-resistant nodes refuse to cache tuples locally, but simply forward them to resistant nodes. A crash is then less likely to cause significant damage. This is the approach we took in the BOHR and CHARM experiments: The master processes ran on crash resistant nodes, and the application specific WAN implementation ensured that no tuples were cached on other nodes.

If, as we observed, most "crashes" were really signs of locally-generated work on borrowed machines, then an approach such as that described by Nichols [16] may be appropriate: A so-called "butler" process determines whether a node is to be made accessible for remote use. If a node becomes inaccessible, the butler informs the local guest processes, which clean up and exit, and the local LISTENER, which forwards all tuples in the local tuple space segment to other nodes.

Some additional techniques have been proposed [14], though further research is needed to determine which will prove effective.

7 CONCLUSIONS

We have examined some of the implementations issues that arose in developing VAX LINDA and using it for some large applications. Our experience has been consistent with that of several other Linda implementation efforts: Producing an efficient Linda implementation raises a number of challenging problems, but those problems are solvable.

Getting really good performance requires some effort at optimization of the Linda program's communication. The Linda tuple space model seems to provide many opportunities for such optimizations, most of which we have just barely begun to explore.

REFERENCES

[1] AHUJA, S., CARRIERO, N., AND GELERNTER, D. Linda and friends. *IEEE Computer* (Aug. 1986), 26–34.

[2] AHUJA, S., CARRIERO, N., GELERNTER, D., AND KRISHNASWAMY, V. Matching language and hardware for parallel computation in the Linda machine. *IEEE Trans. Comput. C-37*, 8 (Aug. 1988).

[3] BIRRELL, A. D., GUTTAG, J. V., HORNING, J. J., AND LEVIN, R. Synchronization primitives for a multiprocessor: A formal specification. *Operating Systems Review 21*, 5 (1987). Also published as Digital Systems Research Center Technical Report 20, Aug. 1987.

[4] BJORNSON, R. Personal communication.

[5] BJORNSON, R., CARRIERO, N., AND GELERNTER, D. The implementation and performance of hypercube Linda. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications* (1989). Also an unnumbered March 1989 Yale University Department of Computer Sciencetechnical report.

[6] BJORNSON, R., CARRIERO, N., GELERNTER, D., AND LEICHTER, J. Linda, the portable parallel. Tech. Rep. TR-520, Yale University Department of Computer Science, Feb. 1987.

[7] CARRIERO, N., AND GELERNTER, D. The S/Net's Linda kernel. *ACM Trans. Comput. Syst. 4*, 2 (May 1986).

[8] CARRIERO, N., GELERNTER, D., AND LEICHTER, J. Distributed data structures in Linda. Tech. Rep. TR-438, Yale University Department of Computer Science, Nov. 1985.

[9] CARRIERO, JR., N. J. Implementation of tuple space machines. Tech. Rep. RR-567, Yale University Department of Computer Science, Dec. 1987. Also a 1987 Yale University PhD thesis.

[10] GELERNTER, D. H. *An Integrated Microcomputer Network for Experiments in Distributed Computing.* PhD thesis, State University of New York at Stony Brook, 1982.

[11] GELERNTER, D. H. Generative communication in Linda. *ACM Trans. Program. Lang. Syst. 7*, 1 (Jan. 1985).

[12] GRAY, J. N. Notes on database operating systems. In *Operating Systems — An Advanced Course*, R. Bayer, R. Graham, and G. Seegmüller, Eds., no. 60 in Lecture Notes in Computer Science. Springer-Verlag, 1978.

[13] LEICHTER, J. S. The VAX LINDA-C user's guide. Tech. Rep. TR-615, Yale University Department of Computer Science, Mar. 1988.

[14] LEICHTER, J. S. *Shared Tuple Memories, Shared Memories, Buses and LAN's — Linda Implementations Across the Spectrum of Connectivity.* PhD thesis, Yale University, 1989. In preparation.

[15] LUCCO, S. E. A heuristic Linda kernel for hypercube multiprocessors. In *Hypercube Multiprocessors 1987*, M. T. Heath, Ed. SIAM, 1987. Proceedings of the Second Conference on Hypercube Multiprocessors, 1986.

[16] NICHOLS, D. A. Using idle workstations in a shared computing environment. *Operating Systems Review 21*, 5 (1987).

[17] WHITESIDE, R. A., AND LEICHTER, J. S. Using Linda for supercomputing on a local area network. In *Proceedings, Supercomputing '88* (Nov. 1988). Also Yale University Department of Computer Science Technical Report TR-638 and Sandia National Laboratories Report SAND88-8818, both June 1988.